March 11, 2024

(s) Fiona Kissi

F223966

AI METHODS REPORT

MLP TRAINING

# Introduction

## Overview of the data set and its characteristics

The dataset comprises information on 597 river catchments in the United Kingdom, with the primary goal of predicting the Index Flood based on eight catchment descriptors.

Each catchment is characterized by the following variables:

* Area (km^2)
* Represents the geographical extent of the catchment in square kilometres.
* BFIHOST
* Indicates the proportion of streamflow sustained by baseflow, providing insights into groundwater contributions.
* FARL (Flood Attenuation due to Reservoirs and Lakes)
* Describes the impact of reservoirs and lakes on flood attenuation
* FPEXT (Flood Plain Extent)
* Provides information on the extent of flood plains within the catchment, influencing flood propagation.
* LDP (Longest Drainage Path)
* Represents the longest path within the catchment that a water droplet would follow before reaching the river network.
* PROPWET (Proportion of Wet Days)
* Indicates the fraction of days in a year with measurable precipitation
* RMED-1D (Median Annual Maximum 1-Day Rainfall)
* Reflects the median of the annual maximum 1-day rainfall
* SAAR (Standard Annual Average Rainfall)
* Represents the standard annual average rainfall in the catchment area.
* Index Flood (m^3/s)
* The target variable to be predicted, representing the median of the annual maximum series of catchment flow. This serves as an essential indicator of the catchment's flood potential.

# Data Pre-processing

## Dataset Characteristics

**Number of Instances**: 597 river catchments

**Problem**: Regression, as the goal is to predict a continuous variable (Index Flood) based on multiple features.

**Predictors**: The eight catchment descriptors

Target Variable: Index Flood (Median of the Annual Maximum Series of Catchment Flow).

**Geographical Scope**: United Kingdom.

**Use Case**: Predicting the Index Flood to enhance understanding and management of flood risk in river catchments.

Mention of the programming language chosen for implementation.

## Data Cleansing

KEY

WDV: wrong data value

OUT: outliers

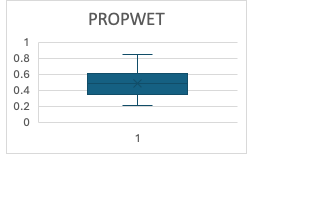
MCD: missing cell data

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Area | BFIHOST | FARL | FPEXT | LDP | PROPWET | RMED- 1D | SAAR | INDEX FLODD | REASON |
| -999 | 0.483 | 0.979 | 0.0339 | 40.88 | 0.58 | 42.8 | 1145 | 63.141 | WDV |
| 235.21 | a | 0.979 | 0.034 | 34.4 | 0.32 | 34.3 | 736 | 3.545 | WDV |
| 53.88 | 0.461 | -999 | -999 | 14.1 | 0.54 | 43.8 | 1316 | 19.135 | WDV |
| 173.32 | 0.466 | 0.996 | 0.0309 | bbb | 0.61 | 44.8 | 1394 | 119.551 | WDV |
| 87.41 | 0.499 | 0.979 | 0.0837 | 17.88 | a | 39.1 | 891 | 26.158 | WDV |
| 58.16 | 0.523 | 0.996 | 0.0789 | 13.64 | 0.25 | -999 | 560 | 4.628 | WDV |
| 164.46 | 0.419 | 0.997 | 0.0414 | 36.96 | 0.59 | 53.1 |  | 114.109 | MCD |
| 56.18 | 0.968 | 0.983 | 0.1161 | 21.65 | 0.24 | 30.6 | 688 | -999 | WDV |
| 79.86 | 0.561 | 0.975 | 0.0316 | 21.35 |  | 38.3 | 949 | 19.489 | MCD |
| 1380.04 | 0.506 | 0.976 | 0.0468 | 107.51 | 1 .74 | 38 | 1108 | 436.809 | OUT |

Investigating the correlations between each catchment descriptor and the Index Flood revealed varying degrees of association. Scatter plots were generated to visually assess these relationships, providing insights into potential predictors for the MLP model. The Python code utilized for creating these scatter plots is detailed in Appendix A – outliers.py, including the use of libraries such as Pandas for data handling and Matplotlib for plotting. The scatter graph also helped visualise outliers which were further confirmed using boxplots of those specific catchments using Excel.

A diagram of a graph

Description automatically generated with medium confidence



A diagram of a scatter plot

Description automatically generatedA diagram of a scatter plot

Description automatically generated

A diagram of a scatter plot

Description automatically generatedA graph of blue dots

Description automatically generated

A diagram of a scatter plot

Description automatically generated

A diagram of a scatter plot

Description automatically generated

A diagram of a scatter plot

Description automatically generated

## Data Splitting

Explanation of how the data is split into subsets (training, validation, testing).

**Identifying Predictors**

I am using the eight catchments to predict the index flood. I used the =correl(array1, array2) function in excel to the correlation coefficient values. From calculating the correlation coefficient values for each catchment Area and LDP had noticeably large positive correlation whereas BFI Host and FPEXT had the highest negative correlation which could reflect how the weight distribution needs to level between the eight catchments.

|  |  |
| --- | --- |
| Index Flood Against Catchment | Correlation coefficient (5s.f) |
| Area | 0.758956 |
| BFI Host | -0.27143 |
| FARL | -0.07035 |
| FPEXT | -0.09715 |
| LDP | 0.693782 |
| PROPWET | 0.393917 |
| RMED-1D | 0.180823 |
| SAAR | 0.240355 |

## Standardisation – Minmax Method

The minimum and maximum for each column came from the training set and validation set only to prevent data leakage and ensure the MLP generalises well to unseen data. The Minmax method is suitable for non-normally distributed data.

A math equation with black text

Description automatically generated

Possible Problem (Minmax method)

* The test set may include a minimum or maximum that is smaller than the minimum of the training/validation set or greater than the maximum of the training/validation set leading to values that standardise to outside the range.

# Implementation of the MLP Algorithm

## Programming Language Choice

I chose Python as my language of choice for designing my Multi-Layer Perceptron (MLP), leveraging libraries like NumPy, Pandas, and Matplotlib.

**NumPy for Matrix Manipulation**

NumPy's efficient array operations were instrumental in handling the matrix computations inherent in MLPs. With NumPy, I could perform essential mathematical operations like matrix multiplications and element-wise operations with ease.

**Pandas for Data Management**

Although I didn't utilize Pandas for standardization, it played a crucial role in data handling and pre-processing tasks. By reading my Excel data into a Pandas Data Frame, I could manipulate and prepare the data for training and evaluation.

**Matplotlib for Data Visualization**

Matplotlib allowed me to create insightful visualizations to analyse the performance of my MLP. I used Matplotlib to draw line graphs and scatter graphs comparing predicted values against actual values. This visualization provided a clear understanding of how well my model was performing.

## A whiteboard with text and diagrams Description automatically generatedCode structure (diagram)

### Forward Pass

Input Data:

I receive input data stored in the selected\_data\_values array.

#### Input Matrix

Before processing the input data, I organize it into a matrix format to make computations easier. I reshape the input data vector into a 2D array using the generate\_input\_matrix function. It has one row, and each column represents a weight from the input to each hidden node. This is done for every row in the dataset. Additionally, I add a column of ones as the first column, which acts as a bias term. This resulting matrix is what I'll be working with and is called input\_matrix.

**Appendix B – mlp.py**

**def** generate\_input\_matrix(selected\_data\_values):

# Reshape the 1D array to a 2D array (assuming it's a row vector)

selected\_data\_values\_2d = selected\_data\_values.reshape(1, -1)

# New column to add

new\_column = np.ones((selected\_data\_values\_2d.shape[0], 1))

# Concatenate the new column to the existing matrix

input\_matrix=np.concatenate((new\_column,selected\_data\_values\_2d),axis=1)

**return** input\_matrix

#### Hidden Layer Weights and Biases:

Next, I need to initialize the weights and biases connecting the input layer to the hidden layer. I do this using the generate\_hidden\_matrix function, which generates random weights and biases for these connections. These are stored in the hidden\_matrix.

**Appendix B – mlp.py**

**def** generate\_hidden\_matrix(input\_size):

# Use NumPy to generate the entire matrix in one go

**return** np.random.uniform((-2/input\_size), (2/input\_size), (input\_size+1, hidden\_nodes))

#### Weighted Sum Calculation

I calculate the weighted sum for each neuron in the hidden layer by performing a dot product between the input\_matrix and the hidden\_matrix using the get\_weighted\_sum\_matrix function.

**Appendix B – mlp.py**

#use this function to work out weighted sum using dot product of two matrices

**def** get\_weighted\_sum\_matrix(matrix\_1,matrix\_2):

result = np.dot(matrix\_1,matrix\_2)

**return** result

#### Activation Function (Sigmoid)

To introduce non-linearity into the network, I pass the weighted sums through an activation function. In my case, I use the sigmoid activation function, which I compute using the get\_sigmoid\_matrix function for each neuron in the hidden layer.

**Appendix B – mlp.py**

#function gets weighted sum of each hidden node and then works out sigmoid function on each to get 1\*(hidden\_nodes+1) matrix

**def** get\_sigmoid\_matrix(weighted\_sum\_matrix):

sigmoid\_matrix = np.zeros(hidden\_nodes+1)

# New column to add

sigmoid\_matrix[0] = 1

**for** i **in** range(1,hidden\_nodes+1):

#sigmoid matrix is updated with values from the 2nd column onwards

#weighted sum matrix uses zero indexing therefore need to use i-1 to make sure i matches to right index

sigmoid\_matrix[i] = 1 / (1 + np.exp(-weighted\_sum\_matrix[0][i-1]))

**return** sigmoid\_matrix

#### Output Layer Weights and Biases

Once the hidden layer activations are obtained, I initialize the weights and biases for the connections between the hidden layer and the output layer using the generate\_output\_matrix function.

**Appendix B – mlp.py**

**def** generate\_output\_matrix():

output\_matrix = np.zeros((hidden\_nodes+1, 1))

**for** j **in** range(hidden\_nodes+1):

output\_matrix[j][0] = get\_bias\_or\_get\_weight(input\_size)

**return** output\_matrix

#### Final Weighted Sum and Output

Finally, I calculate the weighted sum for the output layer by performing a dot product between the hidden layer activations and the output layer weights. This gives me the final output, which I pass through the sigmoid activation function to obtain the predicted output. This predicted output represents the predicted index flood value.

**Appendix B – mlp.py**

#apply sigmoid function to final weighted sum to get predicted index flood

**def** get\_index\_flood\_predicted(x):

index\_flood\_predicted = 1 / (1 + np.exp(-x))

**return** index\_flood\_predicted

### Backpropagation

#### Output Delta Calculation

The first step in backpropagation is to compute the error at the output layer. This is done by taking the difference between the actual target value (index\_flood\_actual) and the predicted output value (x). This difference is then multiplied by the derivative of the sigmoid activation function applied to the predicted output. This process is represented by the get\_output\_delta function.

**Appendix B – mlp.py**

#gets the output delta needed to update the weight values

**def** get\_output\_delta(index\_flood\_actual,x):

#x is the predicted index flood

output\_delta = (index\_flood\_actual - x)\*(x\*(1-x))

**return** output\_delta[0]

#### Hidden Delta Calculation

Once the error at the output layer is calculated, we propagate it backward to the hidden layers. Hidden delta represents the contribution of each hidden node to the output error. It is computed by multiplying the weights connecting the hidden layer to the output layer by the output delta and then further multiplied by the derivative of the sigmoid activation function applied to the activations of the hidden layer. This process is represented by the get\_hidden\_delta function.

**Appendix B – mlp.py**

#work out the delta values for each hidden node and use to update weights later on

**def** get\_hidden\_delta(output\_delta, hidden\_sigmoid\_matrix, output\_matrix):

# Extract weights from hidden layer to output node

hidden\_weights = output\_matrix[1:, 0]

# Calculate hidden delta in a vectorized way

#∂j=Wj,o\*∂o\*Ui(1-Ui)

hidden\_delta = hidden\_weights \* output\_delta \* hidden\_sigmoid\_matrix[1:] \* (1 - hidden\_sigmoid\_matrix[1:])

**return** hidden\_delta

A white background with black text

Description automatically generated

## Modifications/Improvements

### Momentum

It accelerates the optimisation process. It builds upon the basic backpropagation algorithm by adding a momentum term to the weight update rule.

#### Weight Update with Momentum

Instead of updating the weights and biases directly using the calculated deltas, I introduce momentum terms that accumulate the gradients over time. These momentum terms influence the direction and magnitude of weight updates. The momentum term is a fraction of the previous weight update added to the current weight update denoted by alpha.

**Appendix B – mlp.py**

#initialise momentum matrices to store the change in weights and biases for momentum

output\_momentum = np.zeros\_like(output\_matrix)

hidden\_momentum = np.zeros\_like(hidden\_matrix)

# Momentum coefficient

alpha = 0.9

#updates all the weights and biases using the delta values calculated

**def** update\_weights\_and\_biases(output\_matrix,hidden\_matrix,output\_delta,hidden\_delta,output\_momentum,hidden\_momentum):

# Update bias for output layer with momentum

output\_bias\_update = p \* output\_delta + alpha \* output\_momentum[0,0]

output\_matrix[0,0] += output\_bias\_update

output\_momentum[0,0] = output\_bias\_update

#update the weights from hidden nodes to output

**for** i **in** range(1,hidden\_nodes+1):

weight\_update\_output = p \* output\_delta \* hidden\_sigmoid\_matrix[i] + alpha \* output\_momentum[i,0]

output\_matrix[i,0] += weight\_update\_output

output\_momentum[i,0] = weight\_update\_output

**for** i **in** range(hidden\_nodes):

bias\_update\_hidden = p \* hidden\_delta[i] + alpha \* hidden\_momentum[0,i]

hidden\_matrix[0,i] += bias\_update\_hidden

hidden\_momentum[0,i] = bias\_update\_hidden

**for** j **in** range(1,input\_size+1):

weight\_update\_hidden = p \* hidden\_delta[i] \* input\_matrix[0,j] + alpha \* hidden\_momentum[j,i]

hidden\_matrix[j,i] += weight\_update\_hidden

hidden\_momentum[j,i] = weight\_update\_hidden

**return** hidden\_matrix, output\_matrix

#### Model Improvement

By adding momentum, the optimization process becomes more stable and less likely to get stuck in local minima. It helps to smooth out fluctuations in the gradient descent trajectory and can lead to faster convergence.

A math equations and a blue arrow

Description automatically generatedTop of FormBottom of Form

### Annealing

#### **Implementation**

The function computes the learning rate (p) based on the current epoch (x) and the total number of epochs (epochs) provided as parameters.

It uses a logistic function to smoothly transition the learning rate from the initial value (p\_start) to the final value (p\_end) over the course of training.

The logistic function used in this context is:

A graph of a function

Description automatically generated

#### **Learning Rate Calculation**

The learning rate p is computed as a linear interpolation between p\_start and p\_end, where the interpolation factor is determined by the logistic function.

Initially, the logistic function evaluates to approximately 0, resulting in p being close to p\_start.

As training progresses and x increase towards epochs, the logistic function approaches 1, causing p to approach p\_end.

The transition from p\_start to p\_end is smooth and gradual, controlled by the logistic function.

**Appendix B – mlp\_anneal.py**

**def** anneal(p\_start, p\_end, x, epochs):

p = p\_start + (p\_end - p\_start) \* (1 / (1 + np.exp(10 - 20 \* x / epochs)))

**return** p

#### Model Improvement

**Adaptive Learning Rate**

The purpose of using this annealing function is to adaptively adjust the learning rate during training based on the progress of the optimisation process.

Initially, a higher learning rate (p\_start) may allow for faster convergence in the early stages of training when the model parameters are far from optimal.

As training progresses and the model approaches convergence, a lower learning rate (p\_end) can help fine-tune the model parameters more accurately and avoid overshooting the optimal solution.

**Stabilizing Training**

By smoothly annealing the learning rate, the optimisation process becomes more stable and less prone to oscillations or divergent behaviour.

It helps prevent the model from getting stuck in local minima or overshooting the optimal solution.

**Enhanced Convergence**

By adjusting the learning rate according to the progress of training, the model can effectively navigate the optimisation landscape and converge to a better solution more efficiently.

### Bold Driver

After each epoch, I compute the difference in MSE between the current epoch and the previous one. This difference, termed mse\_diff, indicates whether the model performance has improved, worsened, or remained the same compared to the previous epoch.

#### Adapting the Learning Rate

If the MSE has decreased (indicating improvement), I consider it a positive sign that I'm moving towards a better solution. In response, I decide to increase the learning rate to accelerate the learning process.

To achieve this, I multiply the current learning rate (p) by a factor of 1.05, promoting a more aggressive update of the model parameters.

#### Reacting to Worsening MSE

Conversely, if the MSE has increased (indicating a worsening of my model’s performance), I interpret it as a sign that the learning rate might be too high, causing oscillations or overshooting of the optimal solution. In such cases, I decide to be more conservative and decrease the learning rate to stabilise the training process.

I achieve this by multiplying the current learning rate (p) by a factor of 0.7, reducing the magnitude of parameter updates.

#### Maintaining the Learning Rate

If the MSE remains unchanged between epochs, I maintain the current learning rate without any adjustments. This helps in situations where the model is close to convergence or has reached a plateau, and drastic changes in the learning rate may not be beneficial.

**Appendix B – mlp\_bold\_driver.py**

# Ensure there are at least two previous epochs to compare MSEs

**if** epoch >= 2000 **and** epoch % 2000 == 0:

# Calculate the difference in MSE between the last two recorded epochs

mse\_diff = mse\_history\_training[-2] - mse\_history\_training[-1] # Use -2 and -1 for the last two items in list

# If the MSE decreased (improvement), consider increasing the learning rate

**if** mse\_diff > 0:

p \*= 1.05

print(f"Increasing learning rate to {p} at epoch {epoch}")

# If the MSE increased (worsening), decrease the learning rate

**elif** mse\_diff < 0:

p \*= 0.7

print(f"Decreasing learning rate to {p} at epoch {epoch}")

**else**:

### p = p

#### Model Improvement

Faster Convergence

By increasing the learning rate when the MSE decreases, the model makes more substantial updates to its parameters. This aggressive adjustment helps the model converge faster towards an optimal solution, reducing the number of epochs needed for training.

Efficient Exploration of the Optimization Landscape:

The adaptive nature of the Bold Driver technique allows the model to explore the optimisation landscape more efficiently. When the MSE decreases, indicating a favourable direction in the parameter space, the technique amplifies the learning rate, enabling the model to explore promising regions more rapidly. By dynamically adjusting the learning rate based on the direction of MSE changes, the Bold Driver technique helps the model navigate away from local minima more effectively. It encourages the exploration of alternative paths in the optimization landscape, increasing the likelihood of finding a globally optimal solution.

Stability and Robustness

When the MSE increases, suggesting that the model might be overshooting or encountering oscillations, the Bold Driver technique responds by reducing the learning rate. This stabilizes the training process, preventing drastic parameter updates that could lead to divergence or instability.

### Weight Decay

**Weight Decay** is a regularisation technique commonly used in machine learning, including neural networks like Multilayer Perceptrons (MLPs). The goal of weight decay is to prevent overfitting, which is when a model learns to memorize the training data rather than generalize well to unseen data.

#### Explanation

Concept

**Penalizing Large Weights**

Weight decay adds a penalty term to the loss function based on the magnitude of the weights in the model. It discourages large weights by imposing a cost for their presence in the model.

Implementation

**Regularisation Term**

Weight decay modifies the optimisation objective by adding a regularisation term to the loss function. This term penalises large weights and biases.

A black and yellow text

Description automatically generated with medium confidence

**Mechanism**

**Bias-Variance Trade-off**

Weight decay addresses the bias-variance trade-off in machine learning. It helps in reducing the model's variance (overfitting) by sacrificing a small amount of bias (underfitting). This trade-off leads to better generalization performance on unseen data.

**Preventing Overfitting**

Large weights in a model can lead to overfitting, where the model fits the noise in the training data rather than the underlying patterns. Weight decay encourages the model to prefer simpler solutions with smaller weights, reducing overfitting.

Impact on MLP Models

**Improved Generalisation**

By penalising large weights, weight decay encourages the MLP model to generalize better to unseen data. It helps in producing a model that captures the underlying patterns in the data without memorising noise.

**Stability during Training**

Weight decay can improve the stability of training by preventing weights from growing too large, which can lead to numerical instability and convergence issues.

**Reduced Sensitivity to Data**

MLP models trained with weight decay are less sensitive to small changes in the training data, making them more robust and reliable in real-world applications.

#### Omega

**Definition**: Omega is a parameter representing the regularization term used in weight decay. It quantifies the magnitude of the penalty applied to the neural network's weights to prevent overfitting.

Omega is calculated using the formula:

**Appendix B – mlp\_anneal\_weight\_decay.py (note: code snippet also found in mlp\_bold\_driver\_weight.decay.py)**

A mathematical equation with numbers and symbols

Description automatically generated

where n represents the total number of weights (including both hidden and output layers), and sum\_squares is the sum of squares of all weights excluding biases.

**def** calculate\_omega(hidden\_matrix, output\_matrix):

sum\_squares = 0

n = hidden\_matrix.size + output\_matrix.size

sum\_squares = np.sum(np.square(hidden\_matrix[1:])) + np.sum(np.square(output\_matrix[1:]))

omega = (1/(2\*n))\* sum\_squares

**return** omega

**Purpose**: Omega controls the strength of weight decay regularisation. A higher value of omega implies a stronger penalty on the weights, encouraging them to remain small and preventing them from fitting the training data too closely. This helps in improving the model's generalization performance by reducing overfitting.

#### Upsilon

**Definition**: Upsilon is a parameter used to adjust the influence of weight decay during training. It modifies the weight update process based on the learning rate (p) and the current epoch.

**U**psilon is computed as:

A mathematical equation with black text

Description automatically generatedwhere p represents the learning rate, and epoch is the current epoch number during training.

**Appendix B – mlp\_anneal\_weight\_decay.py (note: code snippet also found in mlp\_bold\_driver\_weight.decay.py)**

**def** calculate\_upsilon(p,epoch):

upsilon = 1/(p\*epoch)

**return** upsilon

**Role**

Upsilon dynamically scales the effect of weight decay over the course of training. As the number of epochs increases, the value of upsilon decreases, resulting in a reduced contribution of weight decay to the weight updates. This adaptive adjustment helps in fine-tuning the balance between regularisation and optimisation, facilitating better convergence and model performance.

Integration into Weight Update

Both omega and upsilon are utilised in the calculation of the output delta, which represents the error gradient with respect to the output layer.

Omega is directly multiplied by upsilon and incorporated into the output delta term to modulate the effect of weight decay on the weight updates.

By adjusting the output delta using omega and upsilon, the weight updates are influenced not only by the error but also by the regularisation term, promoting stable convergence and preventing overfitting.

**Appendix B – mlp\_anneal\_weight\_decay.py (note: code snippet also found in mlp\_bold\_driver\_weight.decay.py)**

# Update weights and biases based on backpropagation and weight decay

omega = calculate\_omega(hidden\_matrix,output\_matrix)

upsilon = calculate\_upsilon(p,epoch+1)

output\_delta = get\_output\_delta(index\_flood\_actual,index\_flood\_predicted,omega,upsilon)

hidden\_delta = get\_hidden\_delta(output\_delta,hidden\_sigmoid\_matrix,output\_matrix)

update\_weights\_and\_biases(output\_matrix,hidden\_matrix,output\_delta,hidden\_delta,output\_momentum,hidden\_momentum)

# Training and Network Selection

## MLP with momentum

I decided to use 10 hidden nodes to test which number of epochs was optimal as it was halfway

|  |  |  |
| --- | --- | --- |
| Epochs | Minimum RMSE | Epoch minimum occurred at (rough estimation based on graph) |
| 1000 | 0.03990416923839261 | 1000 |
| 2500 | 0.038513523286480396 | 2500 |
| 5000 | 0.03823327179811419 | 1200 |
| 10000 | 0.03684704808307157 | 5115 |
| 20000 | 0.03745146774399974 | 2200 |
| 40000 | 0.0364383915878707 | 3350 |

1000 Epochs

A graph with red and blue lines

Description automatically generatedA graph with a line

Description automatically generated

2500 EpochsA graph with red and blue lines

Description automatically generatedA graph with a line

Description automatically generated

A graph of a graph

Description automatically generated with medium confidenceA graph with a line

Description automatically generated5000 Epochs

A graph with blue lines

Description automatically generated10000 Epochs

A graph with red and blue lines

Description automatically generated

20000 Epochs

A graph with red and blue lines

Description automatically generatedA graph with a line

Description automatically generated

A graph with a line

Description automatically generatedA graph of a graph

Description automatically generated with medium confidence40000 Epochs

To determine the best number of epochs I did a rerun for 7000 epochs as the minimum MSE is between 2000-7000 epochs and then got the program to give me a more accurate estimate on the epoch number that gives you the minimum. This value falls between 2800 -3500 epochs after several test runs with a RMSE of 0.036928438804733746 - 0.037720290870464544. I will use the upper bound of the range to further test the model under different number of hidden nodes so I can at least have some assurance the minimum is within the number of epochs I am using.

|  |  |  |  |
| --- | --- | --- | --- |
| Hidden nodes | Minimum RMSE | Epochs at minimum (from program) | Correlation coefficient between modelled and observed |
| 4 | 0.037082039781579104 | 1600 | 0.9769831712938067 |
| 8 | 0.03688989351995727 | 3400 | 0.9695634841040888 |
| 12 | 0.03877727311762301 | 2500 | 0.9769252078879032 |
| 16 | 0.03771956636486544 | 2300 | 0.9765512267811365 |

8 Hidden Nodes

A graph with a line

Description automatically generatedA graph of a graph

Description automatically generated with medium confidenceA graph with blue dots

Description automatically generated

This is the optimised number of hidden nodes for my mlp with momentum it has the smallest root mean squared error and the correlation can be further improved by increasing the number of epochs.

Starting with the optimal epoch found in my base model of the mlp with just momentum and then experimenting with increasing epochs. By gradually adjusting one parameter at a time while keeping others fixed, I can systematically explore the effects of each change and determine whether it leads to further improvements in model performance. Hence, I start with 3400 epochs and 8 hidden nodes based on the basic model.

## MLP with momentum and annealing

|  |  |  |
| --- | --- | --- |
| Epochs | Minimum RMSE | Epoch minimum occurs (from program) |
| 3400 | 0.037891190194672894 | 1500 |
| 5000 | 0.03688785437052151 | 2400 |
| 10000 | 0.03874977959617367 | 3900 |

A graph with a line

Description automatically generatedA graph with a line

Description automatically generated3400 Epochs 5000 Epochs

10000 Epochs

A graph with a line

Description automatically generated

The lowest rmse was when the epochs were 2400 so to test the different hidden nodes for this mlp I will use 3500 to make sure it converges to the right minimum

|  |  |  |
| --- | --- | --- |
| Hidden nodes | Minimum RMSE | Epoch minimum occurs (from program) |
| 4 | 0.037419370633704506 | 1800 |
| 8 | 0.03696297135253719 | 1700 |
| 12 | 0.038844793304269876 | 1700 |
| 16 | 0.03753817705757064 | 1700 |

A graph with a line

Description automatically generated

## MLP with momentum and annealing and weight decay. (5000 epochs)

|  |  |  |  |
| --- | --- | --- | --- |
| Hidden nodes | Minimum RMSE | Epoch minimum occurs (from program) | Correlation coefficient |
| 8 | 0.03818325897780245 | 1800 | 0.9768933383628015 |

## MLP with momentum and bold driver

|  |  |  |
| --- | --- | --- |
| Epochs | Minimum RMSE | Epoch minimum occurs (from program) |
| 3400 | 0.03577825856352287 | 3300 |
| 5000 | 0.03556222339806307 | 3800 |
| 10000 | 0.03716977927750789 | 1700 |

Although graphs not shown bold driver had close to 0 positive correlation between modelled and observed suggesting overfitting

3400 Epochs 5000 Epochs

A graph with a red line

Description automatically generatedA graph with a red line

Description automatically generated

|  |  |  |
| --- | --- | --- |
| Hidden nodes | Minimum RMSE | Epoch minimum occurs (from program) |
| 8 | 0.03601982441623533 | 3900 |

## A graph of a test Description automatically generated with medium confidence

A graph with a red line

Description automatically generated

## MLP with momentum and bold driver and weight decay (tested 2500)

|  |  |  |
| --- | --- | --- |
| Hidden nodes | Minimum RMSE | Epoch minimum occurs (from program) |
| 8 | 1900 | 0.039614764655765154 |

# Evaluation of Final Model

Comparison of Algorithm Modifications

|  |  |  |  |
| --- | --- | --- | --- |
| MLP MODEL BEST VERSIONS | NUMBER OF EPOCHS | NUMBER OF HIDDEN NODES | MINIMUM RMSE |
| Momentum only | 3400 | 8 | 0.03688989351995727 |
| Momentum with annealing | 1700 | 8 | 0.03696297135253719 |
| Momentum with bold driver | 5000(minimum occurs in 1800 epochs but the correlation coefficient is higher for slightly larger epochs) | 8 | 0.03556222339806307 |
| Momentum with annealing and weight decay | 5000(minimum occurs in 1800 epochs but the correlation coefficient is higher for slightly larger epochs) | 8 | 0.03818325897780245 |
| Momentum with bold driver and weight decay | 5000 | 8 | 0.02737848254804788 |

# Comparison with Another Model or Baseline

## Baseline Model

This command generates the necessary coefficients using the training data set

=LINEST(R2:R353, J2:Q353, TRUE, TRUE)

Find code for simple linear regression in Appendix C

## Performance Comparison

### Comparison with the neural network model

|  |  |  |
| --- | --- | --- |
| MLP MODEL | RMSE (validation) | RMSE (test) |
| Multiple linear regression (LINEST) | 0.36511972505982954 | 0.3508664501833008 |

Simple linear regression model has a very high rmse with respect to the mlp models which even without the improvements yield a much lower rmse

# Conclusion

## Summary of key findings and outcomes

|  |  |  |  |
| --- | --- | --- | --- |
| BEST MODEL | NUMBER OF HIDDEN NODES | NUMBER OF EPOCHS FOR MINIMUM | MINIMUM RMSE |
| Momentum with bold driver and weight decay | 8 | 5000 | 0.03737848254804788 |

## Reflection on the challenges faced and lessons learned

My main challenge was running large epoch values on my laptop. My laptop’s CPU had to use too much capacity to run the models at large epochs the more improvements were used on them. This meant I was not able to properly explore the models and judge how well could potentially predict the index flood.

Throughout the development and implementation of the mlp, I encountered several challenges that provided valuable insights and lessons for future endeavours. One significant challenge revolved around keeping track of the dimensions of matrices and ensuring their correct application in the backpropagation process. As neural networks involve numerous matrix operations, accurately managing the dimensions of these matrices is crucial for proper functioning. I learned the importance of maintaining meticulous attention to detail, especially when implementing recursive functions for backpropagation. By carefully verifying the dimensions of input and output matrices at each step, I gained a deeper understanding of the underlying mathematics of neural networks and improved my ability to troubleshoot errors effectively.

Another challenge I faced was related to data pre-processing, particularly when manually cleaning data in Excel. While Excel offers flexibility and familiarity, it also presents risks such as inadvertently losing data values or corrupting the dataset structure. Through this experience, I realized the importance of implementing robust data processing pipelines. In the future, I intend to automate data cleaning, splitting, and standardization processes using programming languages like Python. By scripting these tasks, I can ensure reproducibility, minimize human error, and streamline the workflow. Top of Form

Bottom of Form

## Further possible improvements to MLP model

Firstly, adopting batch learning or parallelism techniques could significantly accelerate the training process. By processing multiple data samples concurrently or in batches. This approach not only reduces training time but also enhances efficiency, particularly when dealing with large datasets.

Moreover, further leveraging vectorisation techniques with my matrices will present another avenue for improving computational efficiency. Vectorisation allows for the simultaneous execution of operations on arrays or matrices, exploiting the inherent parallelism offered by modern processors. By maximising the use of vectorised operations in my algorithms, I can achieve substantial speed improvements, especially in numerical computations.

# References

Dawson, Christian. (2024). Introduction to ANNS. AI Methods/23COB107. Loughborough University

Dawson, Christian. (2024). Backpropagation. AI Methods/23COB107. Loughborough University

# Program Listing

## Appendix

### Appendix A: Code for generating the scatter plots for each catchment against Index flood

#### Outliers.py

* Used to draw scatter graphs to visualise correlation

**import** pandas **as** pd

**import** matplotlib.pyplot **as** plt

**import** seaborn **as** sns

# Load your dataset

df = pd.read\_excel('AI\_Dataset.xlsx', engine='openpyxl')

# Define the list of descriptors to plot against the Index flood.

# Exclude 'Index flood' from the plotting list if it's included in the columns.

descriptors = df.columns[:-1] # Assuming 'Index flood' is the last column

# Iterate through each descriptor and create a scatter plot against the Index flood

**for** descriptor **in** descriptors:

plt.figure(figsize=(10, 6)) # Adjust the figure size as needed

sns.scatterplot(data=df, x=descriptor, y='Index flood', color='blue', alpha=0.6)

# Adding title and labels

plt.title(f'Scatter Plot of {descriptor} vs. Index Flood')

plt.xlabel(descriptor)

plt.ylabel('Index Flood (m³/s)')

# Show the plot

plt.show()

### Appendix B: Code used to train multi-layer perceptron with different modifications and improvements

#### Mlp.py

* Used to show how momentum is applied when updating weights and biases.

**import** numpy **as** np

**import** pandas **as** pd

**from** numpy.random **import** default\_rng

**import** matplotlib.pyplot **as** plt

rng = np.random.default\_rng()

#generate biases for each node in hidden layer

**def** get\_bias\_or\_get\_weight(input\_size):

**return** rng.uniform((-2/input\_size), (2/input\_size))

# Define the neural network architecture parameters

input\_size = int(input("how many predictors?")) # Number of input features

# min\_hidden\_nodes = input\_size // 2

# max\_hidden\_nodes = 2 \* input\_size

# Generate a random number between min\_hidden\_nodes and max\_hidden\_nodes

hidden\_nodes = int(input("how many hidden nodes?"))

# Round the value to an integer (if needed)

hidden\_nodes = round(hidden\_nodes)

output\_nodes = 1 # Number of outputs

# Learning rate

p = 0.1

#generate bias for output layer

outputBias = get\_bias\_or\_get\_weight(input\_size)

'''

creates the input\_ matrix where it gets all the input catchments and generates in the right dimensions for dot product so places a 1 in the first column to get a 1\*9 input matrix

'''

**def** generate\_input\_matrix(selected\_data\_values):

# Reshape the 1D array to a 2D array (assuming it's a row vector)

selected\_data\_values\_2d = selected\_data\_values.reshape(1, -1)

# New column to add

new\_column = np.ones((selected\_data\_values\_2d.shape[0], 1))

# Concatenate the new column to the existing matrix

input\_matrix=np.concatenate((new\_column,selected\_data\_values\_2d),axis=1)

**return** input\_matrix

'''

creates the hidden layer matrix by making the first row the bias for each node with each column representing

a node so a column's first row is the node's bias and the subsequent rows in the column is the weight assigned from input to the hidden node in question

'''

**def** generate\_hidden\_matrix(input\_size):

# Use NumPy to generate the entire matrix in one go

**return** np.random.uniform((-2/input\_size), (2/input\_size), (input\_size+1, hidden\_nodes))

# Initialize weights and biases

hidden\_matrix = generate\_hidden\_matrix(input\_size)

#use this function to work out weighted sum using dot product of two matrices

**def** get\_weighted\_sum\_matrix(matrix\_1,matrix\_2):

result = np.dot(matrix\_1,matrix\_2)

**return** result

#function gets weighted sum of each hidden node and then works out sigmoid function on each to get 1\*(hidden\_nodes+1) matrix

**def** get\_sigmoid\_matrix(weighted\_sum\_matrix):

sigmoid\_matrix = np.zeros(hidden\_nodes+1)

# New column to add

sigmoid\_matrix[0] = 1

**for** i **in** range(1,hidden\_nodes+1):

#sigmoid matrix is updated with values from the 2nd column onwards

#weighted sum matrix uses zero indexing therefore need to use i-1 to make sure i matches to right index

sigmoid\_matrix[i] = 1 / (1 + np.exp(-weighted\_sum\_matrix[0][i-1]))

**return** sigmoid\_matrix

'''

creates the output matrix which is the bias on output node (row 1) and weights from each hidden node to output

so you get a (hidden\_nodes+1 \* 1) matrix

'''

**def** generate\_output\_matrix():

output\_matrix = np.zeros((hidden\_nodes+1, 1))

**for** j **in** range(hidden\_nodes+1):

output\_matrix[j][0] = get\_bias\_or\_get\_weight(input\_size)

**return** output\_matrix

output\_matrix = generate\_output\_matrix()

#apply sigmoid function to final weighted sum to get predicted index flood

**def** get\_index\_flood\_predicted(x):

index\_flood\_predicted = 1 / (1 + np.exp(-x))

**return** index\_flood\_predicted

#gets the output delta needed to update the weight values

**def** get\_output\_delta(index\_flood\_actual,x):

#x is the predicted index flood

output\_delta = (index\_flood\_actual - x)\*(x\*(1-x))

**return** output\_delta[0]

#work out the delta values for each hidden node and use to update weights later on

**def** get\_hidden\_delta(output\_delta, hidden\_sigmoid\_matrix, output\_matrix):

# Extract weights from hidden layer to output node

hidden\_weights = output\_matrix[1:, 0]

# Calculate hidden delta in a vectorized way

#∂j=Wj,o\*∂o\*Ui(1-Ui)

hidden\_delta = hidden\_weights \* output\_delta \* hidden\_sigmoid\_matrix[1:] \* (1 - hidden\_sigmoid\_matrix[1:])

**return** hidden\_delta

#initialise momentum matrices to store the change in weights and biases for momentum

output\_momentum = np.zeros\_like(output\_matrix)

hidden\_momentum = np.zeros\_like(hidden\_matrix)

# Momentum coefficient

alpha = 0.9

#updates all the weights and biases using the delta values calculated

**def** update\_weights\_and\_biases(output\_matrix,hidden\_matrix,output\_delta,hidden\_delta,output\_momentum,hidden\_momentum):

# Update bias for output layer with momentum

output\_bias\_update = p \* output\_delta + alpha \* output\_momentum[0,0]

output\_matrix[0,0] += output\_bias\_update

output\_momentum[0,0] = output\_bias\_update

#update the weights from hidden nodes to output

**for** i **in** range(1,hidden\_nodes+1):

weight\_update\_output = p \* output\_delta \* hidden\_sigmoid\_matrix[i] + alpha \* output\_momentum[i,0]

output\_matrix[i,0] += weight\_update\_output

output\_momentum[i,0] = weight\_update\_output

**for** i **in** range(hidden\_nodes):

bias\_update\_hidden = p \* hidden\_delta[i] + alpha \* hidden\_momentum[0,i]

hidden\_matrix[0,i] += bias\_update\_hidden

hidden\_momentum[0,i] = bias\_update\_hidden

**for** j **in** range(1,input\_size+1):

weight\_update\_hidden = p \* hidden\_delta[i] \* input\_matrix[0,j] + alpha \* hidden\_momentum[j,i]

hidden\_matrix[j,i] += weight\_update\_hidden

hidden\_momentum[j,i] = weight\_update\_hidden

**return** hidden\_matrix, output\_matrix

# Mean squared error function

**def** mean\_squared\_error(predicted, actual):

**return** np.mean((actual - predicted) \*\* 2)

# MSE history

mse\_history\_validation = []

# Number of epochs

epochs = int(input("Enter the number of epochs: "))

epochs\_validation = []

# Load training dataset

df\_training = pd.read\_excel("Training\_Set\_MinMax.xlsx")

columns\_of\_interest\_training = df\_training.columns[9:17]

index\_flood\_column\_training = df\_training.columns[17]

# Load validation dataset

df\_validation = pd.read\_excel("Validation\_Set\_MinMax.xlsx")

columns\_of\_interest\_validation = df\_validation.columns[9:17]

index\_flood\_column\_validation = df\_validation.columns[17]

**for** epoch **in** range(epochs):

total\_error\_validation = 0

**if** epoch >= 100 **and** epoch % 100 == 0:

df = df\_validation

columns\_of\_interest = columns\_of\_interest\_validation

index\_flood\_column = index\_flood\_column\_validation

epochs\_validation.append(epoch)

**else**:

df = df\_training

columns\_of\_interest = columns\_of\_interest\_training

index\_flood\_column = index\_flood\_column\_training

**for** i **in** range(len(df)):

# Extracting row-wise data

selected\_data\_values = np.array(df.loc[i, columns\_of\_interest])

index\_flood\_actual = df.loc[i, index\_flood\_column]

# Generate matrices for input and perform forward propagation

input\_matrix = generate\_input\_matrix(selected\_data\_values)

hidden\_sigmoid\_matrix = get\_sigmoid\_matrix(get\_weighted\_sum\_matrix(input\_matrix, hidden\_matrix))

output\_weighted\_sum = get\_weighted\_sum\_matrix(hidden\_sigmoid\_matrix, output\_matrix)

index\_flood\_predicted = get\_index\_flood\_predicted(output\_weighted\_sum)

# Update weights and biases based on backpropagation

output\_delta = get\_output\_delta(index\_flood\_actual, index\_flood\_predicted)

hidden\_delta = get\_hidden\_delta(output\_delta, hidden\_sigmoid\_matrix, output\_matrix)

update\_weights\_and\_biases(output\_matrix, hidden\_matrix, output\_delta, hidden\_delta, output\_momentum, hidden\_momentum)

# Average error for the epoch

**if** df.equals(df\_validation):

# Calculate error

error\_training = mean\_squared\_error(index\_flood\_predicted, index\_flood\_actual)

total\_error\_validation += error\_training

**if** df.equals(df\_validation):

avg\_epoch\_error = total\_error\_validation/ len(df)

mse\_history\_validation.append(avg\_epoch\_error)

print(f"Epoch {epoch+1}: Validation MSE = {avg\_epoch\_error}, {hidden\_nodes} hidden nodes")

# Plot MSE history for validation data

# Assuming validation MSE is recorded every 100 epochs

plt.figure(figsize=(10, 6))

plt.plot(epochs\_validation, mse\_history\_validation, label='MSE Validation')

plt.xlabel('Epoch')

plt.ylabel('MSE')

plt.title('MSE over Epochs - Validation Data MLP with momentum')

plt.legend()

plt.show()

# Convert MSE values to RMSE values

rmse\_array = np.sqrt(mse\_history\_validation)

# Print or use the RMSE values as needed

min\_rmse = np.min(rmse\_array)

index = np.where(rmse\_array == min\_rmse)[0]

optimal\_epoch = epochs\_validation[index[0]]

print("RMSE:",min\_rmse, optimal\_epoch)

# Load test dataset

df\_test = pd.read\_excel("Test\_Set\_MinMax.xlsx")

columns\_of\_interest\_test = df\_test.columns[9:17]

index\_flood\_column\_test = df\_test.columns[17]

# Lists to store predicted and actual values

modelled\_test = []

observed\_test = []

# Iterate over the test data

**for** i **in** range(len(df\_test)):

# Extracting row-wise data

selected\_data\_values = np.array(df\_test.loc[i, columns\_of\_interest\_test])

index\_flood\_actual = df\_test.loc[i, index\_flood\_column\_test]

# Generate matrices for input and perform forward propagation

input\_matrix = generate\_input\_matrix(selected\_data\_values)

hidden\_sigmoid\_matrix = get\_sigmoid\_matrix(get\_weighted\_sum\_matrix(input\_matrix, hidden\_matrix))

output\_weighted\_sum = get\_weighted\_sum\_matrix(hidden\_sigmoid\_matrix, output\_matrix)

index\_flood\_predicted = get\_index\_flood\_predicted(output\_weighted\_sum)

# Store predicted and actual values

modelled\_test.append(index\_flood\_predicted.item())

observed\_test.append(index\_flood\_actual)

# Plot predicted and actual values

plt.figure(figsize=(10, 6))

plt.plot(range(len(df\_test)), modelled\_test, label='Predicted', color='blue')

plt.plot(range(len(df\_test)), observed\_test, label='Actual', color='red')

plt.xlabel('Sample')

plt.ylabel('Index Flood Value')

plt.title('Predicted vs Actual - Test Data MLP with momentum')

plt.legend()

plt.show()

# Create scatter plot

plt.scatter(observed\_test,modelled\_test)

plt.title('MLP Scatter Plot')

plt.xlabel('Observed')

plt.ylabel('Modelled')

plt.grid(**True**)

plt.show()

**def** correlation\_coefficient(x, y):

"""

Calculate the correlation coefficient between two arrays of data.

Parameters:

x (array): First array of data.

y (array): Second array of data.

Returns:

float: Correlation coefficient between the two arrays.

"""

# Convert input arrays to numpy arrays if they are not already

x = np.array(x)

y = np.array(y)

# Calculate means of x and y

mean\_x = np.mean(x)

mean\_y = np.mean(y)

# Calculate covariance and standard deviations

covariance = np.mean((x - mean\_x) \* (y - mean\_y))

std\_dev\_x = np.std(x)

std\_dev\_y = np.std(y)

# Calculate correlation coefficient

correlation = covariance / (std\_dev\_x \* std\_dev\_y)

**return** correlation

print("Correlation coefficient:", correlation\_coefficient(observed\_test, modelled\_test))

#### Mlp\_anneal.py

**import** numpy **as** np

**import** pandas **as** pd

**from** numpy.random **import** default\_rng

**import** matplotlib.pyplot **as** plt

rng = np.random.default\_rng()

#generate biases for each node in hidden layer

**def** get\_bias\_or\_get\_weight(input\_size):

**return** rng.uniform((-2/input\_size), (2/input\_size))

# Define the neural network architecture parameters

input\_size = int(input("how many predictors?")) # Number of input features

# min\_hidden\_nodes = input\_size // 2

# max\_hidden\_nodes = 2 \* input\_size

# Generate a random number between min\_hidden\_nodes and max\_hidden\_nodes

hidden\_nodes = int(input("how many hidden nodes?"))

# Round the value to an integer (if needed)

hidden\_nodes = round(hidden\_nodes)

output\_nodes = 1 # Number of outputs

# Learning rate

p\_start = 0.1

p\_end = 0.01

#generate bias for output layer

outputBias = get\_bias\_or\_get\_weight(input\_size)

'''

creates the hidden layer matrix by making the first row the bias for each node with each column representing

a node so a column's first row is the node's bias and the subsequent rows in the column is the weight assigned from input to the hidden node in question

'''

**def** generate\_hidden\_matrix(input\_size):

# Use NumPy to generate the entire matrix in one go

**return** np.random.uniform((-2/input\_size), (2/input\_size), (9, hidden\_nodes))

# Initialize weights and biases

hidden\_matrix = generate\_hidden\_matrix(input\_size)

'''

creates the input\_ matrix where it gets all the input catchments and generates in the right dimensions for dot product

'''

**def** generate\_input\_matrix(selected\_data\_values):

# Reshape the 1D array to a 2D array (assuming it's a row vector)

selected\_data\_values\_2d = selected\_data\_values.reshape(1, -1)

# New column to add

new\_column = np.ones((selected\_data\_values\_2d.shape[0], 1))

# Concatenate the new column to the existing matrix

input\_matrix=np.concatenate((new\_column,selected\_data\_values\_2d),axis=1)

**return** input\_matrix

#use this function to work out weighted sum using dot product of two matrixes

**def** get\_weighted\_sum\_matrix(matrix\_1,matrix\_2):

result = np.dot(matrix\_1,matrix\_2)

**return** result

#function gets weighted sum of each hidden node and then works out sigmoid function on each to get 1\*4 matrix

**def** get\_sigmoid\_matrix(weighted\_sum\_matrix):

sigmoid\_matrix = np.zeros(hidden\_nodes+1)

# New column to add

sigmoid\_matrix[0] = 1

**for** i **in** range(1,hidden\_nodes+1):

#sigmoid matrix is updated with values from the 2nd column onwards

#weighted sum matrix uses zero indexing therefore need to use i-1 to make sure i matches to right index

sigmoid\_matrix[i] = 1 / (1 + np.exp(-weighted\_sum\_matrix[0][i-1]))

**return** sigmoid\_matrix

'''

creates the output matrix which is the bias on output node (row 1) and weights from each hidden node to output

so you get a (hidden\_nodes+1 \* 1) matrix

'''

**def** generate\_output\_matrix():

output\_matrix = np.zeros((hidden\_nodes+1, 1))

**for** j **in** range(hidden\_nodes+1):

output\_matrix[j][0] = get\_bias\_or\_get\_weight(input\_size)

**return** output\_matrix

output\_matrix = generate\_output\_matrix()

#apply sigmoid function to final weighted sum to get predicted index flood

**def** get\_index\_flood\_predicted(x):

index\_flood\_predicted = 1 / (1 + np.exp(-x))

**return** index\_flood\_predicted

#gets the output delta needed to update the weight values

**def** get\_output\_delta(index\_flood\_actual,x):

#x is the predicted index flood

output\_delta = (index\_flood\_actual - x)\*(x\*(1-x))

**return** output\_delta[0]

#work out the delta values for each hidden node and use to update weights later on

**def** get\_hidden\_delta(output\_delta, hidden\_sigmoid\_matrix, output\_matrix):

# Extract weights corresponding to hidden nodes

hidden\_weights = output\_matrix[1:, 0]

# Calculate hidden delta in a vectorized way

hidden\_delta = hidden\_weights \* output\_delta \* hidden\_sigmoid\_matrix[1:] \* (1 - hidden\_sigmoid\_matrix[1:])

**return** hidden\_delta

#initialise momentum matrices to store the change in weights and biases for momentum

output\_momentum = np.zeros\_like(output\_matrix)

hidden\_momentum = np.zeros\_like(hidden\_matrix)

# Momentum coefficient

alpha = 0.9

#updates all the weights and biases using the delta values calculated

**def** update\_weights\_and\_biases(output\_matrix,hidden\_matrix,output\_delta,hidden\_delta,output\_momentum,hidden\_momentum):

# Update bias for output layer with momentum

output\_bias\_update = p \* output\_delta + alpha \* output\_momentum[0,0]

output\_matrix[0,0] += output\_bias\_update

output\_momentum[0,0] = output\_bias\_update

#update the weights from hidden nodes to output

**for** i **in** range(1,hidden\_nodes+1):

weight\_update\_output = p \* output\_delta \* hidden\_sigmoid\_matrix[i] + alpha \* output\_momentum[i,0]

output\_matrix[i,0] += weight\_update\_output

output\_momentum[i,0] = weight\_update\_output

**for** i **in** range(hidden\_nodes):

bias\_update\_hidden = p \* hidden\_delta[i] + alpha \* hidden\_momentum[0,i]

hidden\_matrix[0,i] += bias\_update\_hidden

hidden\_momentum[0,i] = bias\_update\_hidden

**for** j **in** range(1,input\_size+1):

weight\_update\_hidden = p \* hidden\_delta[i] \* input\_matrix[0,j] + alpha \* hidden\_momentum[j,i]

hidden\_matrix[j,i] += weight\_update\_hidden

hidden\_momentum[j,i] = weight\_update\_hidden

**return** hidden\_matrix, output\_matrix

# Mean squared error function

**def** mean\_squared\_error(predicted, actual):

**return** np.mean((actual - predicted) \*\* 2)

**def** anneal(p\_start, p\_end, x, epochs):

p = p\_start + (p\_end - p\_start) \* (1 / (1 + np.exp(10 - 20 \* x / epochs)))

**return** p

# MSE history

mse\_history\_validation = []

#learning parameter history

p\_history = []

# Number of epochs

epochs = int(input("Enter the number of epochs: "))

epochs\_validation =[]

# Load training dataset

df\_training = pd.read\_excel("Training\_Set\_MinMax.xlsx")

columns\_of\_interest\_training = df\_training.columns[9:17]

index\_flood\_column\_training = df\_training.columns[17]

# Load validation dataset

df\_validation = pd.read\_excel("Validation\_Set\_MinMax.xlsx")

columns\_of\_interest\_validation = df\_validation.columns[9:17]

index\_flood\_column\_validation = df\_validation.columns[17]

**for** epoch **in** range(epochs):

total\_error\_validation = 0

**if** epoch >= 100 **and** epoch % 100 == 0:

df = df\_validation

columns\_of\_interest = columns\_of\_interest\_validation

index\_flood\_column = index\_flood\_column\_validation

epochs\_validation.append(epoch)

**else**:

df = df\_training

columns\_of\_interest = columns\_of\_interest\_training

index\_flood\_column = index\_flood\_column\_training

p = anneal(p\_start, p\_end, epoch, epochs)

p\_history.append(p)

print(f"Epoch {epoch + 1}: Learning Rate = {p}")

**for** i **in** range(len(df)):

# Extracting row-wise data

selected\_data\_values = np.array(df.loc[i, columns\_of\_interest])

index\_flood\_actual = df.loc[i, index\_flood\_column]

# Generate matrices for input and perform forward propagation

input\_matrix = generate\_input\_matrix(selected\_data\_values)

hidden\_sigmoid\_matrix = get\_sigmoid\_matrix(get\_weighted\_sum\_matrix(input\_matrix, hidden\_matrix))

output\_weighted\_sum = get\_weighted\_sum\_matrix(hidden\_sigmoid\_matrix, output\_matrix)

index\_flood\_predicted = get\_index\_flood\_predicted(output\_weighted\_sum)

# Update weights and biases based on backpropagation

output\_delta = get\_output\_delta(index\_flood\_actual,index\_flood\_predicted)

hidden\_delta = get\_hidden\_delta(output\_delta,hidden\_sigmoid\_matrix,output\_matrix)

update\_weights\_and\_biases(output\_matrix,hidden\_matrix,output\_delta,hidden\_delta,output\_momentum,hidden\_momentum)

# Average error for the epoch

**if** df.equals(df\_validation):

# Calculate error

error\_training = mean\_squared\_error(index\_flood\_predicted, index\_flood\_actual)

total\_error\_validation += error\_training

**if** df.equals(df\_validation):

avg\_epoch\_error = total\_error\_validation/ len(df)

mse\_history\_validation.append(avg\_epoch\_error)

print(f"Epoch {epoch+1}: Validation MSE = {avg\_epoch\_error}, {hidden\_nodes} hidden nodes")

# Plot MSE history for validation data

# Assuming validation MSE is recorded every 100 epochs

plt.figure(figsize=(10, 6))

plt.plot(epochs\_validation, mse\_history\_validation, label='MSE Validation')

plt.xlabel('Epoch')

plt.ylabel('MSE')

plt.title('MSE over Epochs - Validation Data Anneal')

plt.legend()

plt.show()

# Plot learning parameter history

plt.figure(figsize=(10, 6))

plt.plot(range(1, epochs + 1), p\_history, label='Learning parameter')

plt.xlabel('Epoch')

plt.ylabel('Learning parameter')

plt.title('Learning parameter over Epochs')

plt.legend()

plt.show()

# Convert MSE values to RMSE values

rmse\_array = np.sqrt(mse\_history\_validation)

# Print or use the RMSE values as needed

min\_rmse = np.min(rmse\_array)

index = np.where(rmse\_array == min\_rmse)[0]

optimal\_epoch = epochs\_validation[index[0]]

print("RMSE:",min\_rmse, optimal\_epoch)

#need a graph of learning parameter against mse to see which has the minimum mse

# Load test dataset

df\_test = pd.read\_excel("Test\_Set\_MinMax.xlsx")

columns\_of\_interest\_test = df\_test.columns[9:17]

index\_flood\_column\_test = df\_test.columns[17]

# Lists to store predicted and actual values

modelled\_test = []

observed\_test = []

# Iterate over the test data

**for** i **in** range(len(df\_test)):

# Extracting row-wise data

selected\_data\_values = np.array(df\_test.loc[i, columns\_of\_interest\_test])

index\_flood\_actual = df\_test.loc[i, index\_flood\_column\_test]

# Generate matrices for input and perform forward propagation

input\_matrix = generate\_input\_matrix(selected\_data\_values)

hidden\_sigmoid\_matrix = get\_sigmoid\_matrix(get\_weighted\_sum\_matrix(input\_matrix, hidden\_matrix))

output\_weighted\_sum = get\_weighted\_sum\_matrix(hidden\_sigmoid\_matrix, output\_matrix)

index\_flood\_predicted = get\_index\_flood\_predicted(output\_weighted\_sum)

# Store predicted and actual values

modelled\_test.append(index\_flood\_predicted)

observed\_test.append(index\_flood\_actual)

# Plot predicted and actual values

plt.figure(figsize=(10, 6))

plt.plot(range(len(df\_test)), modelled\_test, label='Predicted', color='blue')

plt.plot(range(len(df\_test)), observed\_test, label='Actual', color='red')

plt.xlabel('Sample')

plt.ylabel('Index Flood Value')

plt.title('Predicted vs Actual - Test Data Anneal')

plt.legend()

plt.show()

# Create scatter plot

plt.scatter(observed\_test, modelled\_test)

plt.title('MLP Scatter Plot')

plt.xlabel('Observed')

plt.ylabel('Modelled')

plt.grid(**True**)

plt.show()

**def** correlation\_coefficient(x, y):

"""

Calculate the correlation coefficient between two arrays of data.

Parameters:

x (array): First array of data.

y (array): Second array of data.

Returns:

float: Correlation coefficient between the two arrays.

"""

# Convert input arrays to numpy arrays if they are not already

x = np.array(x)

y = np.array(y)

# Calculate means of x and y

mean\_x = np.mean(x)

mean\_y = np.mean(y)

# Calculate covariance and standard deviations

covariance = np.mean((x - mean\_x) \* (y - mean\_y))

std\_dev\_x = np.std(x)

std\_dev\_y = np.std(y)

# Calculate correlation coefficient

correlation = covariance / (std\_dev\_x \* std\_dev\_y)

**return** correlation

print("Correlation coefficient:", correlation\_coefficient(observed\_test, modelled\_test))

#### Mlp\_anneal\_weight\_decay.py

**import** numpy **as** np

**import** pandas **as** pd

**from** numpy.random **import** default\_rng

**import** matplotlib.pyplot **as** plt

rng = np.random.default\_rng()

#generate biases for each node in hidden layer

**def** get\_bias\_or\_get\_weight(input\_size):

**return** rng.uniform((-2/input\_size), (2/input\_size))

# Define the neural network architecture parameters

input\_size = int(input("how many predictors?")) # Number of input features

# min\_hidden\_nodes = input\_size // 2

# max\_hidden\_nodes = 2 \* input\_size

# Generate a random number between min\_hidden\_nodes and max\_hidden\_nodes

hidden\_nodes = int(input("how many hidden nodes?"))

# Round the value to an integer (if needed)

hidden\_nodes = round(hidden\_nodes)

output\_nodes = 1 # Number of outputs

# Learning rate

p\_start = 0.1

p\_end = 0.01

#generate bias for output layer

outputBias = get\_bias\_or\_get\_weight(input\_size)

'''

creates the hidden layer matrix by making the first row the bias for each node with each column representing

a node so a column's first row is the node's bias and the subsequent rows in the column is the weight assigned from input to the hidden node in question

'''

**def** generate\_hidden\_matrix(input\_size):

# Use NumPy to generate the entire matrix in one go

**return** np.random.uniform((-2/input\_size), (2/input\_size),(9, hidden\_nodes))

# Initialize weights and biases

hidden\_matrix = generate\_hidden\_matrix(input\_size)

'''

creates the input\_ matrix where it gets all the input catchments and generates in the right dimensions for dot product

'''

**def** generate\_input\_matrix(selected\_data\_values):

# Reshape the 1D array to a 2D array (assuming it's a row vector)

selected\_data\_values\_2d = selected\_data\_values.reshape(1, -1)

# New column to add

new\_column = np.ones((selected\_data\_values\_2d.shape[0], 1))

# Concatenate the new column to the existing matrix

input\_matrix=np.concatenate((new\_column,selected\_data\_values\_2d),axis=1)

**return** input\_matrix

#use this function to work out weighted sum using dot product of two matrixes

**def** get\_weighted\_sum\_matrix(matrix\_1,matrix\_2):

result = np.dot(matrix\_1,matrix\_2)

**return** result

#function gets weighted sum of each hidden node and then works out sigmoid function on each to get 1\*4 matrix

**def** get\_sigmoid\_matrix(weighted\_sum\_matrix):

sigmoid\_matrix = np.zeros(hidden\_nodes+1)

# New column to add

sigmoid\_matrix[0] = 1

**for** i **in** range(1,hidden\_nodes+1):

#sigmoid matrix is updated with values from the 2nd column onwards

#weighted sum matrix uses zero indexing therefore need to use i-1 to make sure i matches to right index

sigmoid\_matrix[i] = 1 / (1 + np.exp(-weighted\_sum\_matrix[0][i-1]))

**return** sigmoid\_matrix

'''

creates the output matrix which is the bias on output node (row 1) and weights from each hidden node to output

'''

**def** generate\_output\_matrix():

output\_matrix = np.zeros((hidden\_nodes+1, 1))

**for** j **in** range(hidden\_nodes+1):

output\_matrix[j][0] = get\_bias\_or\_get\_weight(input\_size)

**return** output\_matrix

output\_matrix = generate\_output\_matrix()

#apply sigmoid function to final weighted sum to get predicted index flood

**def** get\_index\_flood\_predicted(x):

index\_flood\_predicted = 1 / (1 + np.exp(-x))

**return** index\_flood\_predicted.item()

#gets the output delta needed to update the weight values

**def** get\_output\_delta(index\_flood\_actual,x,omega,upsilon):

#x is the predicted index flood

output\_delta = ((index\_flood\_actual - x)+(omega\*upsilon))\*(x\*(1-x))

**return** output\_delta

#work out the delta values for each hidden node and use to update weights later on

**def** get\_hidden\_delta(output\_delta, hidden\_sigmoid\_matrix, output\_matrix):

# Extract weights corresponding to hidden nodes

hidden\_weights = output\_matrix[1:, 0]

# Calculate hidden delta in a vectorized way

hidden\_delta = hidden\_weights \* output\_delta \* hidden\_sigmoid\_matrix[1:] \* (1 - hidden\_sigmoid\_matrix[1:])

**return** hidden\_delta

#initialise momentum matrices to store the change in weights and biases for momentum

output\_momentum = np.zeros\_like(output\_matrix)

hidden\_momentum = np.zeros\_like(hidden\_matrix)

# Momentum coefficient

alpha = 0.9

#updates all the weights and biases using the delta values calculated

**def** update\_weights\_and\_biases(output\_matrix,hidden\_matrix,output\_delta,hidden\_delta,output\_momentum,hidden\_momentum):

# Update bias for output layer with momentum

output\_bias\_update = p \* output\_delta + alpha \* output\_momentum[0,0]

output\_matrix[0,0] += output\_bias\_update

output\_momentum[0,0] = output\_bias\_update

#update the weights from hidden nodes to output

**for** i **in** range(1,hidden\_nodes+1):

weight\_update\_output = p \* output\_delta \* hidden\_sigmoid\_matrix[i] + alpha \* output\_momentum[i,0]

output\_matrix[i,0] += weight\_update\_output

output\_momentum[i,0] = weight\_update\_output

**for** i **in** range(hidden\_nodes):

bias\_update\_hidden = p \* hidden\_delta[i] + alpha \* hidden\_momentum[0,i]

hidden\_matrix[0,i] += bias\_update\_hidden

hidden\_momentum[0,i] = bias\_update\_hidden

**for** j **in** range(1,input\_size+1):

weight\_update\_hidden = p \* hidden\_delta[i] \* input\_matrix[0,j] + alpha \* hidden\_momentum[j,i]

hidden\_matrix[j,i] += weight\_update\_hidden

hidden\_momentum[j,i] = weight\_update\_hidden

**return** hidden\_matrix, output\_matrix

# Mean squared error function

**def** mean\_squared\_error(predicted, actual):

**return** np.mean((actual - predicted) \*\* 2)

**def** anneal(p\_start, p\_end, x, epochs):

p = p\_start + (p\_end - p\_start) \* (1 / (1 + np.exp(10 - 20 \* x / epochs)))

**return** p

# MSE history

mse\_history\_validation = []

#learning parameter history

p\_history = []

# Number of epochs

epochs = int(input("Enter the number of epochs: "))

epochs\_validation = []

**def** calculate\_omega(hidden\_matrix, output\_matrix):

sum\_squares = 0

n = hidden\_matrix.size + output\_matrix.size

sum\_squares = np.sum(np.square(hidden\_matrix[1:])) + np.sum(np.square(output\_matrix[1:]))

omega = (1/(2\*n))\* sum\_squares

**return** omega

**def** calculate\_upsilon(p,epoch):

upsilon = 1/(p\*epoch)

**return** upsilon

# Load training dataset

df\_training = pd.read\_excel("Training\_Set\_MinMax.xlsx")

columns\_of\_interest\_training = df\_training.columns[9:17]

index\_flood\_column\_training = df\_training.columns[17]

# Load validation dataset

df\_validation = pd.read\_excel("Validation\_Set\_MinMax.xlsx")

columns\_of\_interest\_validation = df\_validation.columns[9:17]

index\_flood\_column\_validation = df\_validation.columns[17]

**for** epoch **in** range(epochs):

total\_error\_validation = 0

**if** epoch >= 100 **and** epoch % 100 == 0:

df = df\_validation

columns\_of\_interest = columns\_of\_interest\_validation

index\_flood\_column = index\_flood\_column\_validation

epochs\_validation.append(epoch)

**else**:

df = df\_training

columns\_of\_interest = columns\_of\_interest\_training

index\_flood\_column = index\_flood\_column\_training

p = anneal(p\_start, p\_end, epoch, epochs)

p\_history.append(p)

print(f"Epoch {epoch + 1}: Learning Rate = {p}")

**for** i **in** range(len(df)):

# Extracting row-wise data

selected\_data\_values = np.array(df.loc[i, columns\_of\_interest])

index\_flood\_actual = df.loc[i, index\_flood\_column]

# Generate matrices for input and perform forward propagation

input\_matrix = generate\_input\_matrix(selected\_data\_values)

hidden\_sigmoid\_matrix = get\_sigmoid\_matrix(get\_weighted\_sum\_matrix(input\_matrix, hidden\_matrix))

output\_weighted\_sum = get\_weighted\_sum\_matrix(hidden\_sigmoid\_matrix, output\_matrix)

index\_flood\_predicted = get\_index\_flood\_predicted(output\_weighted\_sum)

# Update weights and biases based on backpropagation

omega = calculate\_omega(hidden\_matrix,output\_matrix)

upsilon = calculate\_upsilon(p,epoch+1)

output\_delta = get\_output\_delta(index\_flood\_actual,index\_flood\_predicted,omega,upsilon)

hidden\_delta = get\_hidden\_delta(output\_delta,hidden\_sigmoid\_matrix,output\_matrix)

update\_weights\_and\_biases(output\_matrix,hidden\_matrix,output\_delta,hidden\_delta,output\_momentum,hidden\_momentum)

# Average error for the epoch

**if** df.equals(df\_validation):

# Calculate error

error\_training = mean\_squared\_error(index\_flood\_predicted, index\_flood\_actual)

total\_error\_validation += error\_training

**if** df.equals(df\_validation):

avg\_epoch\_error = total\_error\_validation/ len(df)

mse\_history\_validation.append(avg\_epoch\_error)

print(f"Epoch {epoch+1}: Validation MSE = {avg\_epoch\_error}, {hidden\_nodes} hidden nodes")

# Plot MSE history for validation data

# Assuming validation MSE is recorded every 100 epochs

plt.figure(figsize=(10, 6))

plt.plot(epochs\_validation, mse\_history\_validation, label='MSE Validation')

plt.xlabel('Epoch')

plt.ylabel('MSE')

plt.title('MSE over Epochs - Validation Data Anneal with weight decay')

plt.legend()

plt.show()

# Plot learning parameter history

plt.figure(figsize=(10, 6))

plt.plot(range(1, epochs + 1), p\_history, label='Learning parameter')

plt.xlabel('Epoch')

plt.ylabel('Learning parameter')

plt.title('Learning parameter over Epochs')

plt.legend()

plt.show()

# Convert MSE values to RMSE values

rmse\_array = np.sqrt(mse\_history\_validation)

# Print or use the RMSE values as needed

min\_rmse = np.min(rmse\_array)

index = np.where(rmse\_array == min\_rmse)[0]

optimal\_epoch = epochs\_validation[index[0]]

print("RMSE:",min\_rmse, optimal\_epoch)

# Load test dataset

df\_test = pd.read\_excel("Test\_Set\_MinMax.xlsx")

columns\_of\_interest\_test = df\_test.columns[9:17]

index\_flood\_column\_test = df\_test.columns[17]

# Lists to store predicted and actual values

modelled\_test = []

observed\_test = []

# Iterate over the test data

**for** i **in** range(len(df\_test)):

# Extracting row-wise data

selected\_data\_values = np.array(df\_test.loc[i, columns\_of\_interest\_test])

index\_flood\_actual = df\_test.loc[i, index\_flood\_column\_test]

# Generate matrices for input and perform forward propagation

input\_matrix = generate\_input\_matrix(selected\_data\_values)

hidden\_sigmoid\_matrix = get\_sigmoid\_matrix(get\_weighted\_sum\_matrix(input\_matrix, hidden\_matrix))

output\_weighted\_sum = get\_weighted\_sum\_matrix(hidden\_sigmoid\_matrix, output\_matrix)

index\_flood\_predicted = get\_index\_flood\_predicted(output\_weighted\_sum)

# Store predicted and actual values

modelled\_test.append(index\_flood\_predicted)

observed\_test.append(index\_flood\_actual)

# Plot predicted and actual values

plt.figure(figsize=(10, 6))

plt.plot(range(len(df\_test)), modelled\_test, label='Predicted', color='blue')

plt.plot(range(len(df\_test)), observed\_test, label='Actual', color='red')

plt.xlabel('Sample')

plt.ylabel('Index Flood Value')

plt.title('Predicted vs Actual - Test Data Anneal with weight decay')

plt.legend()

plt.show()

# Create scatter plot

plt.scatter(observed\_test, modelled\_test)

plt.title('MLP Scatter Plot')

plt.xlabel('Observed')

plt.ylabel('Modelled')

plt.grid(**True**)

plt.show()

**def** correlation\_coefficient(x, y):

"""

Calculate the correlation coefficient between two arrays of data.

Parameters:

x (array): First array of data.

y (array): Second array of data.

Returns:

float: Correlation coefficient between the two arrays.

"""

# Convert input arrays to numpy arrays if they are not already

x = np.array(x)

y = np.array(y)

# Calculate means of x and y

mean\_x = np.mean(x)

mean\_y = np.mean(y)

# Calculate covariance and standard deviations

covariance = np.mean((x - mean\_x) \* (y - mean\_y))

std\_dev\_x = np.std(x)

std\_dev\_y = np.std(y)

# Calculate correlation coefficient

correlation = covariance / (std\_dev\_x \* std\_dev\_y)

**return** correlation

print("Correlation coefficient:", correlation\_coefficient(observed\_test, modelled\_test))

#### Mlp\_bold\_driver.py

**import** numpy **as** np

**import** pandas **as** pd

**from** numpy.random **import** default\_rng

**import** matplotlib.pyplot **as** plt

rng = np.random.default\_rng()

#generate biases for each node in hidden layer

**def** get\_bias\_or\_get\_weight(input\_size):

**return** rng.uniform((-2/input\_size), (2/input\_size))

# Define the neural network architecture parameters

input\_size = int(input("how many predictors?")) # Number of input features

# min\_hidden\_nodes = input\_size // 2

# max\_hidden\_nodes = 2 \* input\_size

# Generate a random number between min\_hidden\_nodes and max\_hidden\_nodes

hidden\_nodes = int(input("how many hidden nodes?"))

# Round the value to an integer (if needed)

hidden\_nodes = round(hidden\_nodes)

output\_nodes = 1 # Number of outputs

# Learning rate

p = 0.1

#generate bias for output layer

outputBias = get\_bias\_or\_get\_weight(input\_size)

'''

creates the input\_ matrix where it gets all the input catchments and generates in the right dimensions for dot product so places a 1 in the first column to get a 1\*9 input matrix

'''

**def** generate\_input\_matrix(selected\_data\_values):

# Reshape the 1D array to a 2D array (assuming it's a row vector)

selected\_data\_values\_2d = selected\_data\_values.reshape(1, -1)

# New column to add

new\_column = np.ones((selected\_data\_values\_2d.shape[0], 1))

# Concatenate the new column to the existing matrix

input\_matrix=np.concatenate((new\_column,selected\_data\_values\_2d),axis=1)

**return** input\_matrix

'''

creates the hidden layer matrix by making the first row the bias for each node with each column representing

a node so a column's first row is the node's bias and the subsequent rows in the column is the weight assigned from input to the hidden node in question

'''

**def** generate\_hidden\_matrix(input\_size):

# Use NumPy to generate the entire matrix in one go

**return** np.random.uniform((-2/input\_size), (2/input\_size), (input\_size+1, hidden\_nodes))

# Initialize weights and biases

hidden\_matrix = generate\_hidden\_matrix(input\_size)

#use this function to work out weighted sum using dot product of two matrices

**def** get\_weighted\_sum\_matrix(matrix\_1,matrix\_2):

result = np.dot(matrix\_1,matrix\_2)

**return** result

#function gets weighted sum of each hidden node and then works out sigmoid function on each to get 1\*(hidden\_nodes+1) matrix

**def** get\_sigmoid\_matrix(weighted\_sum\_matrix):

sigmoid\_matrix = np.zeros(hidden\_nodes+1)

# New column to add

sigmoid\_matrix[0] = 1

**for** i **in** range(1,hidden\_nodes+1):

#sigmoid matrix is updated with values from the 2nd column onwards

#weighted sum matrix uses zero indexing therefore need to use i-1 to make sure i matches to right index

sigmoid\_matrix[i] = 1 / (1 + np.exp(-weighted\_sum\_matrix[0][i-1]))

**return** sigmoid\_matrix

'''

creates the output matrix which is the bias on output node (row 1) and weights from each hidden node to output

so you get a (hidden\_nodes+1 \* 1) matrix

'''

**def** generate\_output\_matrix():

output\_matrix = np.zeros((hidden\_nodes+1, 1))

**for** j **in** range(hidden\_nodes+1):

output\_matrix[j][0] = get\_bias\_or\_get\_weight(input\_size)

**return** output\_matrix

output\_matrix = generate\_output\_matrix()

#apply sigmoid function to final weighted sum to get predicted index flood

**def** get\_index\_flood\_predicted(x):

index\_flood\_predicted = 1 / (1 + np.exp(-x))

**return** index\_flood\_predicted

#gets the output delta needed to update the weight values

**def** get\_output\_delta(index\_flood\_actual,x):

#x is the predicted index flood

output\_delta = (index\_flood\_actual - x)\*(x\*(1-x))

**return** output\_delta[0]

#work out the delta values for each hidden node and use to update weights later on

**def** get\_hidden\_delta(output\_delta, hidden\_sigmoid\_matrix, output\_matrix):

# Extract weights from hidden layer to output node

hidden\_weights = output\_matrix[1:, 0]

# Calculate hidden delta in a vectorized way

#∂j=Wj,o\*∂o\*Ui(1-Ui)

hidden\_delta = hidden\_weights \* output\_delta \* hidden\_sigmoid\_matrix[1:] \* (1 - hidden\_sigmoid\_matrix[1:])

**return** hidden\_delta

#initialise momentum matrices to store the change in weights and biases for momentum

output\_momentum = np.zeros\_like(output\_matrix)

hidden\_momentum = np.zeros\_like(hidden\_matrix)

# Momentum coefficient

alpha = 0.9

#updates all the weights and biases using the delta values calculated

**def** update\_weights\_and\_biases(output\_matrix,hidden\_matrix,output\_delta,hidden\_delta,output\_momentum,hidden\_momentum):

# Update bias for output layer with momentum

output\_bias\_update = p \* output\_delta + alpha \* output\_momentum[0,0]

output\_matrix[0,0] += output\_bias\_update

output\_momentum[0,0] = output\_bias\_update

#update the weights from hidden nodes to output

**for** i **in** range(1,hidden\_nodes+1):

weight\_update\_output = p \* output\_delta \* hidden\_sigmoid\_matrix[i] + alpha \* output\_momentum[i,0]

output\_matrix[i,0] += weight\_update\_output

output\_momentum[i,0] = weight\_update\_output

**for** i **in** range(hidden\_nodes):

bias\_update\_hidden = p \* hidden\_delta[i] + alpha \* hidden\_momentum[0,i]

hidden\_matrix[0,i] += bias\_update\_hidden

hidden\_momentum[0,i] = bias\_update\_hidden

**for** j **in** range(1,input\_size+1):

weight\_update\_hidden = p \* hidden\_delta[i] \* input\_matrix[0,j] + alpha \* hidden\_momentum[j,i]

hidden\_matrix[j,i] += weight\_update\_hidden

hidden\_momentum[j,i] = weight\_update\_hidden

**return** hidden\_matrix, output\_matrix

# Mean squared error function

**def** mean\_squared\_error(predicted, actual):

**return** np.mean((actual - predicted) \*\* 2)

# MSE history

mse\_history\_training = []

mse\_history\_validation = []

# Number of epochs

epochs = int(input("Enter the number of epochs: "))

epochs\_validation = []

# Load training dataset

df\_training = pd.read\_excel("Training\_Set\_MinMax.xlsx")

columns\_of\_interest\_training = df\_training.columns[9:17]

index\_flood\_column\_training = df\_training.columns[17]

# Load validation dataset

df\_validation = pd.read\_excel("Validation\_Set\_MinMax.xlsx")

columns\_of\_interest\_validation = df\_validation.columns[9:17]

index\_flood\_column\_validation = df\_validation.columns[17]

**for** epoch **in** range(epochs):

total\_error\_training = 0

total\_error\_validation = 0

# Ensure there are at least two previous epochs to compare MSEs

**if** epoch >= 2000 **and** epoch % 2000 == 0:

# Calculate the difference in MSE between the last two recorded epochs

mse\_diff = mse\_history\_training[-2] - mse\_history\_training[-1] # Use -2 and -1 for the last two items in list

# If the MSE decreased (improvement), consider increasing the learning rate

**if** mse\_diff > 0:

p \*= 1.05

print(f"Increasing learning rate to {p} at epoch {epoch}")

# If the MSE increased (worsening), decrease the learning rate

**elif** mse\_diff < 0:

p \*= 0.7

print(f"Decreasing learning rate to {p} at epoch {epoch}")

**else**:

p = p

**if** epoch >= 100 **and** epoch % 100 == 0:

df = df\_validation

columns\_of\_interest = columns\_of\_interest\_validation

index\_flood\_column = index\_flood\_column\_validation

epochs\_validation.append(epoch)

**else**:

df = df\_training

columns\_of\_interest = columns\_of\_interest\_training

index\_flood\_column = index\_flood\_column\_training

**for** i **in** range(len(df)):

# Extracting row-wise data

selected\_data\_values = np.array(df.loc[i, columns\_of\_interest])

index\_flood\_actual = df.loc[i, index\_flood\_column]

# Generate matrices for input and perform forward propagation

input\_matrix = generate\_input\_matrix(selected\_data\_values)

hidden\_sigmoid\_matrix = get\_sigmoid\_matrix(get\_weighted\_sum\_matrix(input\_matrix, hidden\_matrix))

output\_weighted\_sum = get\_weighted\_sum\_matrix(hidden\_sigmoid\_matrix, output\_matrix)

index\_flood\_predicted = get\_index\_flood\_predicted(output\_weighted\_sum)

# Calculate error

error\_training = mean\_squared\_error(index\_flood\_predicted, index\_flood\_actual)

total\_error\_training += error\_training

# Update weights and biases based on backpropagation

output\_delta = get\_output\_delta(index\_flood\_actual,index\_flood\_predicted)

hidden\_delta = get\_hidden\_delta(output\_delta,hidden\_sigmoid\_matrix,output\_matrix)

update\_weights\_and\_biases(output\_matrix,hidden\_matrix,output\_delta,hidden\_delta,output\_momentum,hidden\_momentum)

**if** df.equals(df\_training):

avg\_epoch\_error = total\_error\_training / len(df)

mse\_history\_training.append(avg\_epoch\_error)

# Average error for the epoch

**if** df.equals(df\_validation):

# Calculate error

error\_training = mean\_squared\_error(index\_flood\_predicted, index\_flood\_actual)

total\_error\_validation += error\_training

**if** df.equals(df\_validation):

avg\_epoch\_error = total\_error\_validation/ len(df)

mse\_history\_validation.append(avg\_epoch\_error)

print(f"Epoch {epoch+1}: Validation MSE = {avg\_epoch\_error}, {hidden\_nodes} hidden nodes")

# Plot MSE history for validation data

# Assuming validation MSE is recorded every 100 epochs

plt.figure(figsize=(10, 6))

plt.plot(epochs\_validation, mse\_history\_validation, label='MSE Validation', color='red')

plt.xlabel('Epoch')

plt.ylabel('MSE')

plt.title('MSE over Epochs - Validation Data Bold Driver')

plt.legend()

plt.show()

# Convert MSE values to RMSE values

rmse\_array = np.sqrt(mse\_history\_validation)

# Print or use the RMSE values as needed

min\_rmse = np.min(rmse\_array)

index = np.where(rmse\_array == min\_rmse)[0]

optimal\_epoch = epochs\_validation[index[0]]

print("RMSE:",min\_rmse, optimal\_epoch)

# Load test dataset

df\_test = pd.read\_excel("Test\_Set\_MinMax.xlsx")

columns\_of\_interest\_test = df\_test.columns[9:17]

index\_flood\_column\_test = df\_test.columns[17]

# Lists to store predicted and actual values

modelled\_test = []

observed\_test = []

# Iterate over the test data

**for** i **in** range(len(df\_test)):

# Extracting row-wise data

selected\_data\_values = np.array(df\_test.loc[i, columns\_of\_interest\_test])

index\_flood\_actual = df\_test.loc[i, index\_flood\_column\_test]

# Generate matrices for input and perform forward propagation

input\_matrix = generate\_input\_matrix(selected\_data\_values)

hidden\_sigmoid\_matrix = get\_sigmoid\_matrix(get\_weighted\_sum\_matrix(input\_matrix, hidden\_matrix))

output\_weighted\_sum = get\_weighted\_sum\_matrix(hidden\_sigmoid\_matrix, output\_matrix)

index\_flood\_predicted = get\_index\_flood\_predicted(output\_weighted\_sum)

# Store predicted and actual values

modelled\_test.append(index\_flood\_predicted)

observed\_test.append(index\_flood\_actual)

# Plot predicted and actual values

plt.figure(figsize=(10, 6))

plt.plot(range(len(df\_test)), modelled\_test, label='Predicted', color='blue')

plt.plot(range(len(df\_test)), observed\_test, label='Actual', color='red')

plt.xlabel('Sample')

plt.ylabel('Index Flood Value')

plt.title('Predicted vs Actual - Test Data Bold Driver ')

plt.legend()

plt.show()

# Create scatter plot

plt.scatter(observed\_test, modelled\_test)

plt.title('MLP Scatter Plot')

plt.xlabel('Observed')

plt.ylabel('Modelled')

plt.grid(**True**)

plt.show()

**def** correlation\_coefficient(x, y):

"""

Calculate the correlation coefficient between two arrays of data.

Parameters:

x (array): First array of data.

y (array): Second array of data.

Returns:

float: Correlation coefficient between the two arrays.

"""

# Convert input arrays to numpy arrays if they are not already

x = np.array(x)

y = np.array(y)

# Calculate means of x and y

mean\_x = np.mean(x)

mean\_y = np.mean(y)

# Calculate covariance and standard deviations

covariance = np.mean((x - mean\_x) \* (y - mean\_y))

std\_dev\_x = np.std(x)

std\_dev\_y = np.std(y)

# Calculate correlation coefficient

correlation = covariance / (std\_dev\_x \* std\_dev\_y)

**return** correlation

print("Correlation coefficient:", correlation\_coefficient(observed\_test, modelled\_test))

#### Mlp\_bold\_driver\_weight\_decay.py

**import** numpy **as** np

**import** pandas **as** pd

**from** numpy.random **import** default\_rng

**import** matplotlib.pyplot **as** plt

rng = np.random.default\_rng()

#generate biases for each node in hidden layer

**def** get\_bias\_or\_get\_weight(input\_size):

**return** rng.uniform((-2/input\_size), (2/input\_size))

# Define the neural network architecture parameters

input\_size = int(input("how many predictors?")) # Number of input features

# min\_hidden\_nodes = input\_size // 2

# max\_hidden\_nodes = 2 \* input\_size

# Generate a random number between min\_hidden\_nodes and max\_hidden\_nodes

hidden\_nodes = int(input("how many hidden nodes?"))

# Round the value to an integer (if needed)

hidden\_nodes = round(hidden\_nodes)

output\_nodes = 1 # Number of outputs

# Learning rate

p = 0.1

#generate bias for output layer

outputBias = get\_bias\_or\_get\_weight(input\_size)

'''

creates the input\_ matrix where it gets all the input catchments and generates in the right dimensions for dot product so places a 1 in the first column to get a 1\*9 input matrix

'''

**def** generate\_input\_matrix(selected\_data\_values):

# Reshape the 1D array to a 2D array (assuming it's a row vector)

selected\_data\_values\_2d = selected\_data\_values.reshape(1, -1)

# New column to add

new\_column = np.ones((selected\_data\_values\_2d.shape[0], 1))

# Concatenate the new column to the existing matrix

input\_matrix=np.concatenate((new\_column,selected\_data\_values\_2d),axis=1)

**return** input\_matrix

'''

creates the hidden layer matrix by making the first row the bias for each node with each column representing

a node so a column's first row is the node's bias and the subsequent rows in the column is the weight assigned from input to the hidden node in question

'''

**def** generate\_hidden\_matrix(input\_size):

# Use NumPy to generate the entire matrix in one go

**return** np.random.uniform((-2/input\_size), (2/input\_size),(input\_size+1, hidden\_nodes))

# Initialize weights and biases

hidden\_matrix = generate\_hidden\_matrix(input\_size)

#use this function to work out weighted sum using dot product of two matrices

**def** get\_weighted\_sum\_matrix(matrix\_1,matrix\_2):

result = np.dot(matrix\_1,matrix\_2)

**return** result

#function gets weighted sum of each hidden node and then works out sigmoid function on each to get 1\*(hidden\_nodes+1) matrix

**def** get\_sigmoid\_matrix(weighted\_sum\_matrix):

sigmoid\_matrix = np.zeros(hidden\_nodes+1)

# New column to add

sigmoid\_matrix[0] = 1

**for** i **in** range(1,hidden\_nodes+1):

#sigmoid matrix is updated with values from the 2nd column onwards

#weighted sum matrix uses zero indexing therefore need to use i-1 to make sure i matches to right index

sigmoid\_matrix[i] = 1 / (1 + np.exp(-weighted\_sum\_matrix[0][i-1]))

**return** sigmoid\_matrix

'''

creates the output matrix which is the bias on output node (row 1) and weights from each hidden node to output

so you get a (hidden\_nodes+1 \* 1) matrix

'''

**def** generate\_output\_matrix():

output\_matrix = np.zeros((hidden\_nodes+1, 1))

**for** j **in** range(hidden\_nodes+1):

output\_matrix[j][0] = get\_bias\_or\_get\_weight(input\_size)

**return** output\_matrix

output\_matrix = generate\_output\_matrix()

#apply sigmoid function to final weighted sum to get predicted index flood

**def** get\_index\_flood\_predicted(x):

index\_flood\_predicted = 1 / (1 + np.exp(-x))

**return** index\_flood\_predicted

#gets the output delta needed to update the weight values

**def** get\_output\_delta(index\_flood\_actual,x,omega,upsilon):

#x is the predicted index flood

output\_delta = ((index\_flood\_actual - x)+(omega\*upsilon))\*(x\*(1-x))

**return** output\_delta[0]

#work out the delta values for each hidden node and use to update weights later on

**def** get\_hidden\_delta(output\_delta, hidden\_sigmoid\_matrix, output\_matrix):

# Extract weights from hidden layer to output node

hidden\_weights = output\_matrix[1:, 0]

# Calculate hidden delta in a vectorized way

#∂j=Wj,o\*∂o\*Ui(1-Ui)

hidden\_delta = hidden\_weights \* output\_delta \* hidden\_sigmoid\_matrix[1:] \* (1 - hidden\_sigmoid\_matrix[1:])

**return** hidden\_delta

#initialise momentum matrices to store the change in weights and biases for momentum

output\_momentum = np.zeros\_like(output\_matrix)

hidden\_momentum = np.zeros\_like(hidden\_matrix)

# Momentum coefficient

alpha = 0.9

#updates all the weights and biases using the delta values calculated

**def** update\_weights\_and\_biases(output\_matrix,hidden\_matrix,output\_delta,hidden\_delta,output\_momentum,hidden\_momentum):

# Update bias for output layer with momentum

output\_bias\_update = p \* output\_delta + alpha \* output\_momentum[0,0]

output\_matrix[0,0] += output\_bias\_update

output\_momentum[0,0] = output\_bias\_update

#update the weights from hidden nodes to output

**for** i **in** range(1,hidden\_nodes+1):

weight\_update\_output = p \* output\_delta \* hidden\_sigmoid\_matrix[i] + alpha \* output\_momentum[i,0]

output\_matrix[i,0] += weight\_update\_output

output\_momentum[i,0] = weight\_update\_output

**for** i **in** range(hidden\_nodes):

bias\_update\_hidden = p \* hidden\_delta[i] + alpha \* hidden\_momentum[0,i]

hidden\_matrix[0,i] += bias\_update\_hidden

hidden\_momentum[0,i] = bias\_update\_hidden

**for** j **in** range(1,input\_size+1):

weight\_update\_hidden = p \* hidden\_delta[i] \* input\_matrix[0,j] + alpha \* hidden\_momentum[j,i]

hidden\_matrix[j,i] += weight\_update\_hidden

hidden\_momentum[j,i] = weight\_update\_hidden

**return** hidden\_matrix, output\_matrix

# Mean squared error function

**def** mean\_squared\_error(predicted, actual):

**return** np.mean((actual - predicted) \*\* 2)

# MSE history

mse\_history\_training = []

mse\_history\_validation = []

# Number of epochs

epochs = int(input("Enter the number of epochs: "))

epochs\_validation=[]

# Load training dataset

df\_training = pd.read\_excel("Training\_Set\_MinMax.xlsx")

columns\_of\_interest\_training = df\_training.columns[9:17]

index\_flood\_column\_training = df\_training.columns[17]

# Load validation dataset

df\_validation = pd.read\_excel("Validation\_Set\_MinMax.xlsx")

columns\_of\_interest\_validation = df\_validation.columns[9:17]

index\_flood\_column\_validation = df\_validation.columns[17]

**def** calculate\_omega(hidden\_matrix, output\_matrix):

sum\_squares = 0

n = hidden\_matrix.size + output\_matrix.size

sum\_squares = np.sum(np.square(hidden\_matrix[1:])) + np.sum(np.square(output\_matrix[1:]))

omega = (1/(2\*n))\* sum\_squares

**return** omega

**def** calculate\_upsilon(p,epoch):

upsilon = 1/(p\*epoch)

**return** upsilon

**for** epoch **in** range(epochs):

# Ensure there are at least two previous epochs to compare MSEs

**if** epoch >= 2000 **and** epoch % 2000 == 0:

# Calculate the difference in MSE between the last two recorded epochs

mse\_diff = mse\_history\_training[-2] - mse\_history\_training[-1] # Use -2 and -1 for the last two items in list

# If the MSE decreased (improvement), consider increasing the learning rate

**if** mse\_diff > 0:

p \*= 1.05

print(f"Increasing learning rate to {p} at epoch {epoch}")

# If the MSE increased (worsening), decrease the learning rate

**elif** mse\_diff < 0:

p \*= 0.7

print(f"Decreasing learning rate to {p} at epoch {epoch}")

**else**:

p = p

total\_error\_training = 0

total\_error\_validation = 0

# Ensure there are at least two previous epochs to compare MSEs

**if** epoch >= 100 **and** epoch % 100 == 0:

df = df\_validation

columns\_of\_interest = columns\_of\_interest\_validation

index\_flood\_column = index\_flood\_column\_validation

epochs\_validation.append(epoch)

**else**:

df = df\_training

columns\_of\_interest = columns\_of\_interest\_training

index\_flood\_column = index\_flood\_column\_training

**for** i **in** range(len(df)):

# Extracting row-wise data

selected\_data\_values = np.array(df.loc[i, columns\_of\_interest])

index\_flood\_actual = df.loc[i, index\_flood\_column]

# Generate matrices for input and perform forward propagation

input\_matrix = generate\_input\_matrix(selected\_data\_values)

hidden\_sigmoid\_matrix = get\_sigmoid\_matrix(get\_weighted\_sum\_matrix(input\_matrix, hidden\_matrix))

output\_weighted\_sum = get\_weighted\_sum\_matrix(hidden\_sigmoid\_matrix, output\_matrix)

index\_flood\_predicted = get\_index\_flood\_predicted(output\_weighted\_sum)

# Calculate error

error\_training = mean\_squared\_error(index\_flood\_predicted, index\_flood\_actual)

total\_error\_training += error\_training

# Update weights and biases based on backpropagation

omega = calculate\_omega(hidden\_matrix,output\_matrix)

upsilon = calculate\_upsilon(p,epoch+1)

output\_delta = get\_output\_delta(index\_flood\_actual,index\_flood\_predicted,omega,upsilon)

hidden\_delta = get\_hidden\_delta(output\_delta,hidden\_sigmoid\_matrix,output\_matrix)

update\_weights\_and\_biases(output\_matrix,hidden\_matrix,output\_delta,hidden\_delta,output\_momentum,hidden\_momentum)

**if** df.equals(df\_training):

avg\_epoch\_error = total\_error\_training / len(df)

mse\_history\_training.append(avg\_epoch\_error)

# Average error for the epoch

**if** df.equals(df\_validation):

# Calculate error

error\_training = mean\_squared\_error(index\_flood\_predicted, index\_flood\_actual)

total\_error\_validation += error\_training

**if** df.equals(df\_validation):

avg\_epoch\_error = total\_error\_validation/ len(df)

mse\_history\_validation.append(avg\_epoch\_error)

print(f"Epoch {epoch+1}: Validation MSE = {avg\_epoch\_error}, {hidden\_nodes} hidden nodes")

# Plot MSE history for validation data

# Assuming validation MSE is recorded every 100 epochs

plt.figure(figsize=(10, 6))

plt.plot(epochs\_validation, mse\_history\_validation, label='MSE Validation', color='red')

plt.xlabel('Epoch')

plt.ylabel('MSE')

plt.title('MSE over Epochs - Validation Data')

plt.legend()

plt.show()

# Convert MSE values to RMSE values

rmse\_array = np.sqrt(mse\_history\_validation)

# Print or use the RMSE values as needed

min\_rmse = np.min(rmse\_array)

index = np.where(rmse\_array == min\_rmse)[0]

optimal\_epoch = epochs\_validation[index[0]]

print("RMSE:",min\_rmse, optimal\_epoch)

# Load test dataset

df\_test = pd.read\_excel("Test\_Set\_MinMax.xlsx")

columns\_of\_interest\_test = df\_test.columns[9:17]

index\_flood\_column\_test = df\_test.columns[17]

# Lists to store predicted and actual values

modelled\_test = []

observed\_test = []

# Iterate over the test data

**for** i **in** range(len(df\_test)):

# Extracting row-wise data

selected\_data\_values = np.array(df\_test.loc[i, columns\_of\_interest\_test])

index\_flood\_actual = df\_test.loc[i, index\_flood\_column\_test]

# Generate matrices for input and perform forward propagation

input\_matrix = generate\_input\_matrix(selected\_data\_values)

hidden\_sigmoid\_matrix = get\_sigmoid\_matrix(get\_weighted\_sum\_matrix(input\_matrix, hidden\_matrix))

output\_weighted\_sum = get\_weighted\_sum\_matrix(hidden\_sigmoid\_matrix, output\_matrix)

index\_flood\_predicted = get\_index\_flood\_predicted(output\_weighted\_sum)

# Store predicted and actual values

modelled\_test.append(index\_flood\_predicted)

observed\_test.append(index\_flood\_actual)

# Plot predicted and actual values

plt.figure(figsize=(10, 6))

plt.plot(range(len(df\_test)), modelled\_test, label='Predicted', color='blue')

plt.plot(range(len(df\_test)), observed\_test, label='Actual', color='red')

plt.xlabel('Sample')

plt.ylabel('Index Flood Value')

plt.title('Predicted vs Actual - Test Data Bold Driver with weight decay')

plt.legend()

plt.show()

# Create scatter plot

plt.scatter(observed\_test, modelled\_test)

plt.title('MLP Scatter Plot')

plt.xlabel('Observed')

plt.ylabel('Modelled')

plt.grid(**True**)

plt.show()

**def** correlation\_coefficient(x, y):

"""

Calculate the correlation coefficient between two arrays of data.

Parameters:

x (array): First array of data.

y (array): Second array of data.

Returns:

float: Correlation coefficient between the two arrays.

"""

# Convert input arrays to numpy arrays if they are not already

x = np.array(x)

y = np.array(y)

# Calculate means of x and y

mean\_x = np.mean(x)

mean\_y = np.mean(y)

# Calculate covariance and standard deviations

covariance = np.mean((x - mean\_x) \* (y - mean\_y))

std\_dev\_x = np.std(x)

std\_dev\_y = np.std(y)

# Calculate correlation coefficient

correlation = covariance / (std\_dev\_x \* std\_dev\_y)

**return** correlation

print("Correlation coefficient:", correlation\_coefficient(observed\_test, modelled\_test))

## Appendix C

### Regression.py

**import** pandas **as** pd

**import** numpy **as** np

# Coefficients obtained from the LINEST output

coefficients = [-0.104674002,0.17494234,0.22843003,0.26175687,-0.0478894,0.1239252,-0.0535111,0.58261092]

# Load validation dataset

df\_validation = pd.read\_excel("Validation\_Set\_MinMax.xlsx")

columns\_of\_interest\_validation = df\_validation.iloc[:, 9:17]

index\_flood\_column\_validation = df\_validation.iloc[:, 17]

# Multiply each predictor variable by its corresponding coefficient and sum up the products

y\_pred\_validation = np.dot(columns\_of\_interest\_validation, coefficients)

# Load actual target values for validation dataset

y\_val\_actual = index\_flood\_column\_validation.to\_numpy()

# Calculate squared differences

squared\_diff = (y\_val\_actual - y\_pred\_validation) \*\* 2

# Calculate mean squared difference

mean\_squared\_diff = np.mean(squared\_diff)

# Calculate RMSE for validation dataset

rmse\_validation = np.sqrt(mean\_squared\_diff)

print("RMSE for validation dataset:", rmse\_validation)

# Load test dataset

df\_test = pd.read\_excel("Test\_Set\_MinMax.xlsx")

columns\_of\_interest\_test = df\_test.iloc[:, 9:17]

index\_flood\_column\_test = df\_test.iloc[:, 17]

# Apply the model (coefficients) to the test dataset to obtain predicted values

y\_pred\_test = np.dot(columns\_of\_interest\_test, coefficients)

# Load actual target values for test dataset

y\_test\_actual = index\_flood\_column\_test.to\_numpy()

# Calculate squared differences

squared\_diff\_test = (y\_test\_actual - y\_pred\_test) \*\* 2

# Calculate mean squared difference

mean\_squared\_diff\_test = np.mean(squared\_diff\_test)

# Calculate RMSE for test dataset

rmse\_test = np.sqrt(mean\_squared\_diff\_test)

print("RMSE for test dataset:", rmse\_test)