Neural Network Implementation

AI METHODS COURSEWORK

OWEN WALLER F221887

Table of Contents

1. Data Pre-Processing

1.1 Data Cleansing – Missing and Spurious Data

The first step in preparing the data for use with the neural network was to remove any missing or spurious data from the set. This was done by first searching for any fields which were blank or contained only letters, which resulted in the removal of the following rows:

69	235.21	a		0.979	0.034	34.4	0.32	34.3	736	3.545
115	87.41	0.499		0.979	0.0837	17.88	a	39.1	891	26.158
293	173.32	0.466		0.996	0.0309	bbb	0.61	44.8	1394	119.551
539	164.46	0.419		0.997	0.0414	36.96	0.59	53.1		114.109
588	79.86	0.561		0.975	0.0316	21.35		38.3	949	19.489

The next step was to remove any spurious data, such as negative values or impossible values (e.g. values greater than 1 in the proportion of wet days column):

80	-999	0.483		0.979	0.0339	40.88	0.58	42.8	1145	63.141
182	53.88	0.461		-999	-999	14.1	0.54	43.8	1316	19.135
337	58.16	0.523		0.996	0.0789	13.64	0.25	-999	560	4.628
340	1380.04	0.506		0.976	0.0468	107.51	1.74	38	1108	436.809

(Row 340 contains impossible PROPWET value)

1.2 Data Cleansing – Outliers

```
def remove_outliers(df):
    z_scores = df.apply(lambda x: np.abs((x - x.mean()) / x.std()))
    return df[(z_scores < 3).all(axis=1)]
```

The next stage of data cleansing was to remove any significant outliers, in order to reduce the work the neural network does to compensate. Due to the high variety within the data, I decided to class any values more than 3 standard deviations away from the mean as outliers, and the code above was used to implement their removal.

1.3 Data Cleansing – Skew

In order to reduce the work the neural network needs to do to accommodate for the skew in certain predictors, I applied a log transformation to all the data. This helps make it more evenly distributed, and was achieved with the following code:

```
9   def apply_log10(df):
10      return df.applymap(lambda x: np.log10(x) if x > 0 else None)
11
```

1.4 Selecting Predictors

Of the eight data columns provided (excluding the index flood), the inputs for the neural network must be selected. To simplify the program and reduce the work the ANN needs to do, excluding certain predictors is necessary, and therefore each predictor was evaluated:

Catchment Area (AREA)
Important for the calculations, independent of other predictors
Conclusion: Use as predictor

Base Flow Index (BFIHOST)
Useful data, mostly independent and works with rainfall stats to describe river well
Conclusion: Use as predictor

Flood Attention due to Reservoirs and Lakes (FARL)
Important data, independent of other predictors, accounts for human intervention
Conclusion: Use as a predictor

Flood Plain Extent (FPEXT)
Vital information for calculating the index
Conclusion: Use as a predictor

Longest Drainage Path (LDP)
Important data which is independent of the other predictors
Conclusion: Use as a predictor

Proportion of Wet Days (PROPWET)
Important however qualitative – what classifies as wet
High correlation with annual average rainfall – 0.798

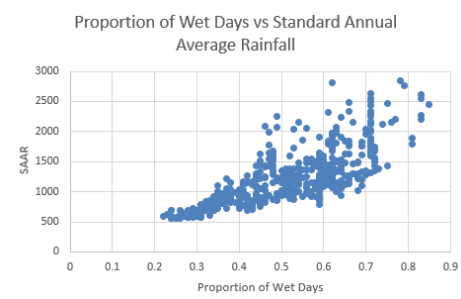

Figure 1

Conclusion: Exclude from predictors

Median Annual Maximum 1-day Rainfall (RMED-1)
Important as maximum rainfall will have high impact on flooding, also mostly independent (influences SAAR but not heavily)
Conclusion: Use as a predictor

Standard Annual Average Rainfall
Influenced by many other factors, however vital information and can't be recreated from other predictors.
Conclusion: Use as a predictor

1.5 Data Standardisation

It is necessary to standardise data in order to prevent certain predictors having little to no impact on the output, as their values are too low in comparison to others. I opted to standardise the data in python using the pandas library, as this was much more efficient than using excel manually. This produced a new excel file containing all the data standardised between 0.1-0.9. The code is shown below:

```python
import pandas as pd

# Function to scale data between 0.1 and 0.9
def standardise_data(df):
    min_val = df.min()
    max_val = df.max()
    return 0.1 + (0.9 - 0.1) * (df - min_val) / (max_val - min_val)

file_name = 'preparedata.xlsx'
sheet_name = 'Sheet1'
df = pd.read_excel(file_name, sheet_name=sheet_name)

# Apply the scaling function to the entire DataFrame
df_scaled = standardise_data(df)

output_file_name = 'standardised_data.xlsx'
df_scaled.to_excel(output_file_name, index=False)
```

1.6 Data Splitting

The final stage is preparing the data for use was to split it into three distinct sets – training data, validation data and test data. I chose to divide this into 60-20-20 splits respectively.

This was achieved by assigning a column in excel a random value, and then sorting by that value, resulting in the order of the rows being randomised. Given there are 527 rows after cleansing, the first rows are for training data, and the next 106 for validation and finally the last 105 for testing.

```
file_path = 'standardised_data.xlsx'
X = pd.read_excel(file_path, nrows=316)
inputs = X.iloc[:, 0:7].values
outputs = X.iloc[:, 7].values

validationFile = pd.read_excel(file_path, nrows = 106, skiprows = 316)
validationSet = validationFile.iloc[:, 0:7].values
```

The splitting of data is achieved as shown above when importing the spreadsheet into the neural network file.

2. Implementation

2.1 Paradigm Selection

Object oriented programming will be used to develop the network rather than a procedural approach, as the class approach lends itself to this task. OOP is known for its modularity and reusability, and by defining the network as an object with consistent methods it will be much easier to test different input data or quantity of neurons.

2.2 Language and Libraries Selection

In order to implement the multi-layer perceptron, I opted to use Python as the language, for numerous reasons. First of all, familiarity was an advantage as I have had more experience using Python than other languages, and therefore this would allow for a smoother development process. Python also allows for an object-oriented approach as suggested above, through the use of built-in class and method functionalities, meaning it provides suitable complexity and means for solving the problem.

Python also offers a vast array of libraries, all of which are heavily documented, allowing for faster development by providing functions for frequently needed utility without having to redefine it in each separate project. The libraries I will be using for this implementation include Pandas and Numpy.

Pandas provides easy access to excel files, letting the user manipulate the data within using code and also save their results as a separate excel file. This is very useful for stages like normalisation and removing outliers.

Numpy provides a great number of mathematical utility not provided by python alone, such as dot products and exponentials. This is vital to implementing the mathematics needed for the backpropagation algorithm and the activation functions of the neurons.

2.3 Data Storage, Structure and Initialisation

Data is intricately structured and stored to facilitate the neural network's operations. Inputs and outputs are initially handled as pandas DataFrames, sourced from an Excel file, before being converted into numpy arrays. This conversion is crucial for the efficient execution of matrix operations during the forward and backward propagation phases of the network's training process.

Weights and biases, integral to the network's learning capability, are also stored as numpy arrays. These are initialized following the Xavier/Glorot initialization (first proposed by Xavier Glorot in his 2010 research paper “Understanding the difficulty of

training deep feedforward neural networks") method for weights and set to zeros for biases, ensuring a balanced starting point for training. This type of random initialisation was selected as when partnered with the sigmoid function it allows the activations and gradients to flow effectively during both forward and backward passes.

As the network undergoes training through epochs, the loss calculated at each step—a measure of the difference between the actual and predicted outputs—is recorded in a list named loss_over_time. This list serves not only as a log of the network's performance over time but also as a dataset for visualizing the learning progress through a plot, providing insights into the effectiveness and dynamics of the training process. This structured approach to data storage within the code underscores the layered and iterative nature of neural network training, where data manipulation and matrix operations form the backbone of the learning mechanism.

2.4 Classes and Methods

The code uses an object-oriented approach as discussed earlier, and this is done through a key class, **NeuralNetwork**, which encapsulates the entirety of the network's functionality,
including initialization, forward and backward propagation, and the training process. This allows for different neural networks to be initialised during training, with flexible numbers of nodes or inputs.

The **constructor** for this class initialises the weights and biases for the network (see 2.3 for initialisation), as well as setting up the input, output, and hidden layers' neurons. The array for tracking the networks performance is also initialised here.

```
12  class NeuralNetwork:
13      def __init__(self, input_nodes, hidden_nodes, output_nodes, seed=10):
14          self.input_nodes = input_nodes
15          self.hidden_nodes = hidden_nodes
16          self.output_nodes = output_nodes
17          self.loss_over_time = []
18
19          np.random.seed(seed) ## added to ensure consistency between networks with different num of neurons during testing
20          # Initialize weights and biases
21          # Xavier/Glorot initialization
22          self.weights_input_hidden = np.random.randn(self.input_nodes, self.hidden_nodes) * np.sqrt(2 / (self.input_nodes + self.hidden_nodes))
23          self.weights_hidden_output = np.random.randn(self.hidden_nodes, self.output_nodes) * np.sqrt(2 / (self.hidden_nodes + self.output_nodes))
24
25          # Biases can be initialized to zeros
26          self.bias = np.zeros(self.hidden_nodes)
27          self.bias_output = np.zeros(self.output_nodes)
28
```

The **forward_pass** method is used for executing the forward propagation, by applying the sigmoid function and the weights and biases to the input data in order to predict the output. This method therefore allows for testing of the network without training it when ran in isolation on a certain set of inputs.

```
30      def forward_pass(self, inputs):
31          self.inputs = inputs
32          self.hidden_layer_activation = np.dot(inputs, self.weights_input_hidden) + self.bias
33          self.hidden_layer_output = sigmoid(self.hidden_layer_activation)
34          self.output_layer_activation = np.dot(self.hidden_layer_output, self.weights_hidden_output) + self.bias_output
35          self.predicted_output = sigmoid(self.output_layer_activation)
```

The **backward_pass** method includes the main part of the backpropagation algorithm, as it works back through the network and updates the weights and biases. It does this based on the mean loss function which is calculated by comparing the predicted output to the expected output. The adjustment is then calculated using the derivative of the sigmoid function and the dot product of matrices. Finally, the loss is stored in the array in order to track performance.

```
36
37      def backward_pass(self, actual_output):
38
39          loss = np.mean(np.square(actual_output - self.predicted_output)) #Mean loss function used
40          self.loss_over_time.append(loss)  # Store the calculated loss
41
42          d_loss_predicted_output = -2 * (actual_output - self.predicted_output)
43          d_predicted_output = d_loss_predicted_output * sigmoid_derivative(self.predicted_output)
44
45          # Calculate the error at the hidden layer
46          error_hidden_layer = d_predicted_output.dot(self.weights_hidden_output.T)
47          d_hidden_layer = error_hidden_layer * sigmoid_derivative(self.hidden_layer_output)
48
49          # Update the weights and biases
50          learning_rate = 0.01
51          self.weights_hidden_output -= learning_rate * self.hidden_layer_output.T.dot(d_predicted_output)
52          self.weights_input_hidden -= learning_rate * self.inputs.T.dot(d_hidden_layer)
53          self.bias_output -= learning_rate * np.sum(d_predicted_output, axis=0)
54          self.bias -= learning_rate * np.sum(d_hidden_layer, axis=0)
55
```

The **train** method orchestrates the training of the network, calling the forward and backward pass methods on the provided inputs and expected outputs. This method can be called within a loop to train the network over a number of epochs. The method uses batch training to improve efficiency.

```
def train(self, inputs, actual_output):
    self.forward_pass(inputs)
    self.backward_pass(actual_output)
```

2.5 Alternative Transfer Functions

The neural network I have implemented initially uses the sigmoid function, as it is reliable and very commonly used. However, other transfer functions were also tested for the program.

For the examples used below, a fixed number of 7 nodes was used to ensure consistency between functions. The number of nodes will be optimised later during the training process.

Sigmoid Function

The sigmoid function was the first function tested for this network. It maps the output values been 0 and 1, which makes it useful for predicting probabilities – similar to the flood index.

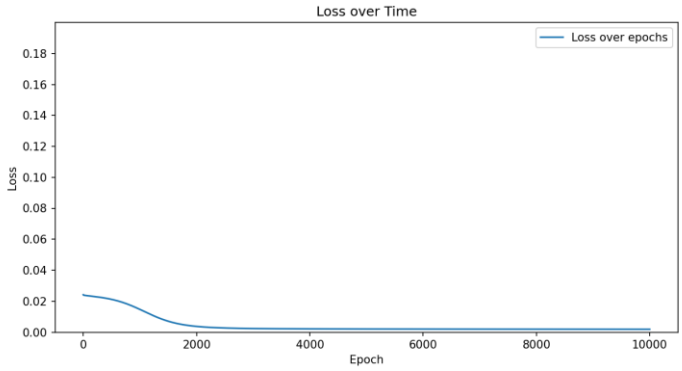

Figure 2

As shown in figure 2, the loss for the sigmoid function follows the anticipated pattern and is quickly extremely small.

Hyperbolic Tangent Function

This transfer function maps squashes values between -1 and 1. It starts with a much higher initial loss, however falls quicker than the sigmoid function.

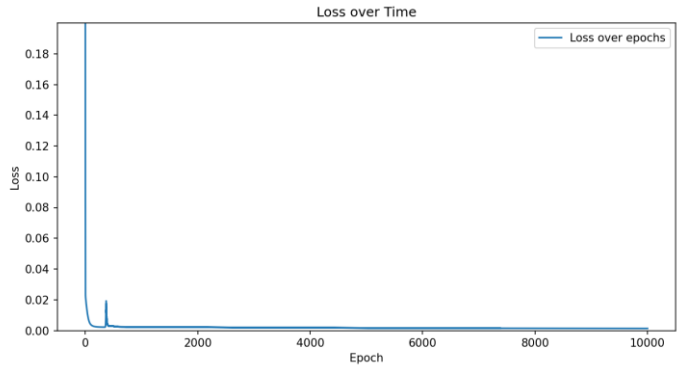

Figure 3

It mostly follows the anticipated pattern, although has a fluctuation early in the training process.

Rectified Linear Unit (ReLU)

The ReLU function takes the maximum of 0 and the value, and such helps with vanishing gradient removal. However, it can lead to dead neurons, and in this case is unsuitable:

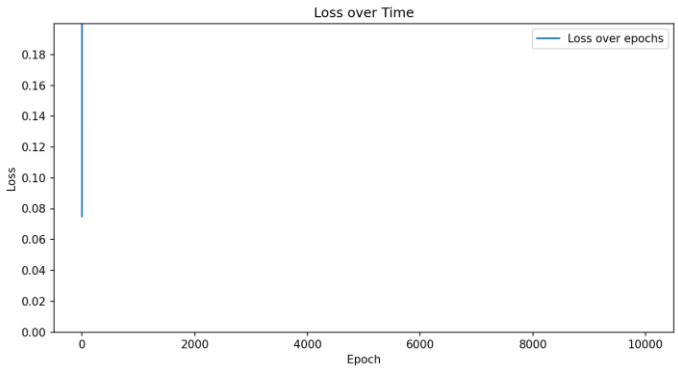

Figure 4

Softmax Activation Function

The softmax activation is commonly used in the output layer of a network in classification problems. Given the nature of this data and network, the function is not suitable and therefore was not tested, as the output should be quantitative rather than qualitative.

2.6 Backpropagation Improvements - Momentum

Momentum involves building upon the original backpropagation algorithm to improve the speed at which the network converges. This is achieved by taking the “velocity” of the learning process into account as well as the current gradient when updating the weights and biases.

To implement this, first the class constructor was updated to add the velocity properties:

```
        # Momentum
        self.velocity_weights_input_hidden = np.zeros_like(self.weights_input_hidden)
        self.velocity_weights_hidden_output = np.zeros_like(self.weights_hidden_output)
        self.velocity_bias_hidden = np.zeros_like(self.bias_hidden)
        self.velocity_bias_output = np.zeros_like(self.bias_output)
        self.momentum = 0.9
```

The backward pass method was then updated to utilise these properties when updating the weights and biases:

```
        # Calculate updates with momentum
        self.velocity_weights_hidden_output = (self.momentum * self.velocity_weights_hidden_output -
                                               learning_rate * self.hidden_layer_output.T.dot(d_predicted_output))
        self.velocity_weights_input_hidden = (self.momentum * self.velocity_weights_input_hidden -
                                              learning_rate * self.inputs.T.dot(d_hidden_layer))
        self.velocity_bias_output = (self.momentum * self.velocity_bias_output -
                                     learning_rate * np.sum(d_predicted_output, axis=0))
        self.velocity_bias_hidden = (self.momentum * self.velocity_bias_hidden -
                                     learning_rate * np.sum(d_hidden_layer, axis=0))

        # Update weights and biases
        self.weights_hidden_output += self.velocity_weights_hidden_output
        self.weights_input_hidden += self.velocity_weights_input_hidden
        self.bias_output += self.velocity_bias_output
        self.bias_hidden += self.velocity_bias_hidden
```

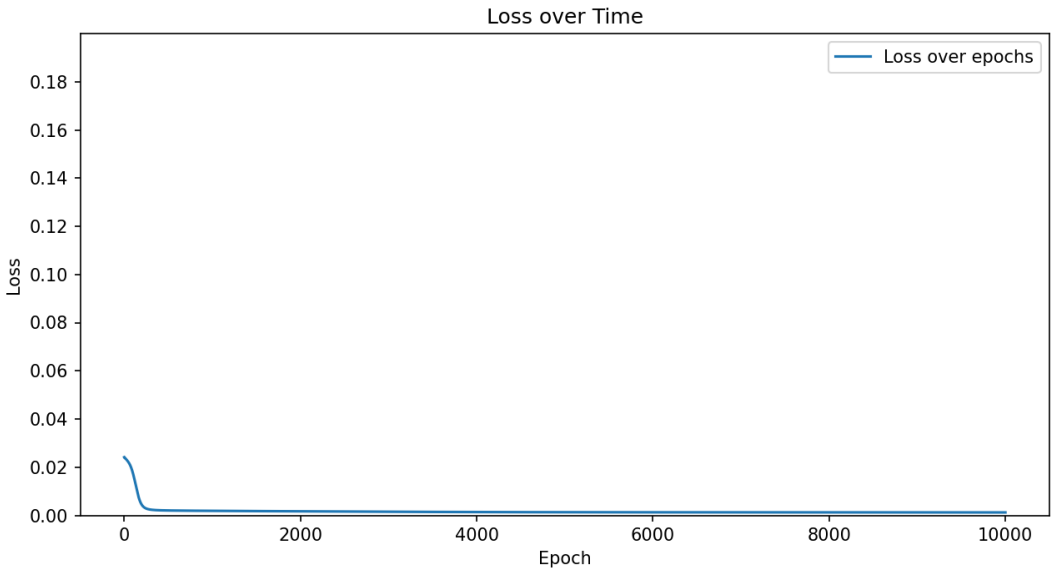

Figure 5

As shown in figure 5, the use of momentum allows the loss to be reduced quicker and therefore results in faster convergence (Figure 5 can be compared to figure 2 in section 2.5, and both are shown using 7 nodes in the hidden layer).

2.7 Backpropagation Improvements – Learning Rate

Bold Driver

The bold driver algorithm adjusts the learning rate based on the changes in loss between epochs. This works to prevent overshooting and increases the speed at which the network converges. It is implemented by changing the constructor method to include learning rate as a property, as well as its increase or decrease.

```
32          # Bold Driver
33          self.learning_rate = 0.0015
34          self.lr_increase = 1.05
35          self.lr_decrease = 0.7
36          self.prev_loss = float('inf')
37
```

The backward pass method is then adjusted to use this learning rate in calculations, as well as adjusting it as necessary.

```
# Adjust learning rate with Bold Driver
if loss < self.prev_loss:
    self.learning_rate *= self.lr_increase
else:
    self.learning_rate *= self.lr_decrease
self.prev_loss = loss
```

As shown in figure 6 below, this combined with momentum further increases the speed at which the network converges:

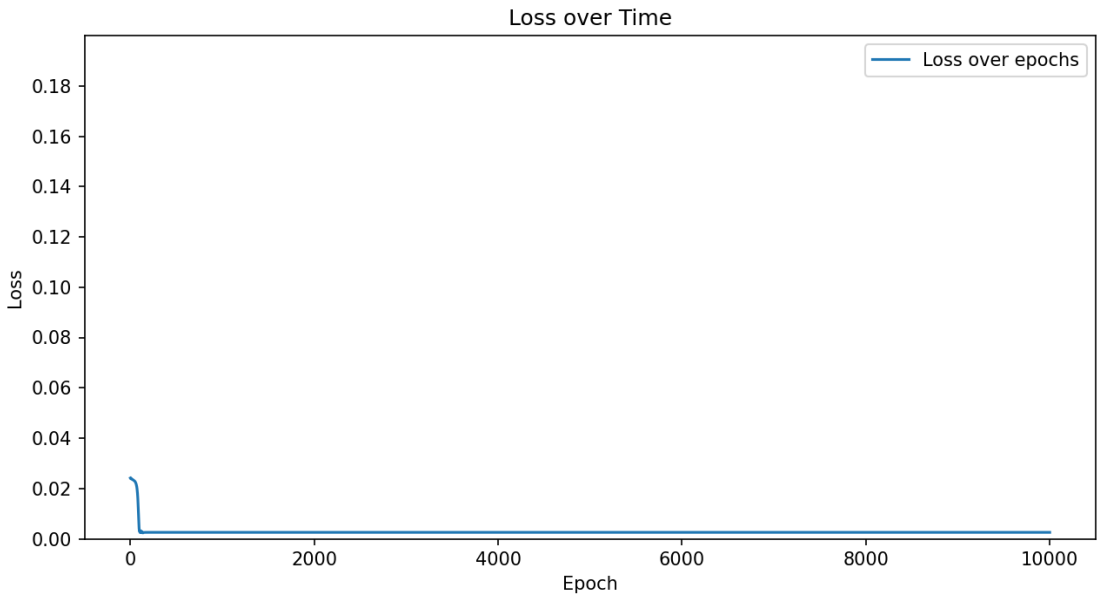

Figure 6

Annealing

The learning rate can also be used to optimise the algorithm through annealing. Annealing involves gradually decreasing the learning rate, preventing overtraining at the end of the process and speeding up convergence at the start. It is implemented by introducing the learning rate properties without the constructor method, and then adjusting these later within the backward pass method.

```
32        # Annealing
33        self.initial_learning_rate = 0.03
34        self.learning_rate = self.initial_learning_rate
35        self.annealing_rate = 0.995
36
```

```
70          # Anneal learning rate
71          self.learning_rate *= self.annealing_rate
72
```

As shown below in figure 7, the annealing process definitely streamlines the networks convergence, with the loss falling within the first few hundred epochs

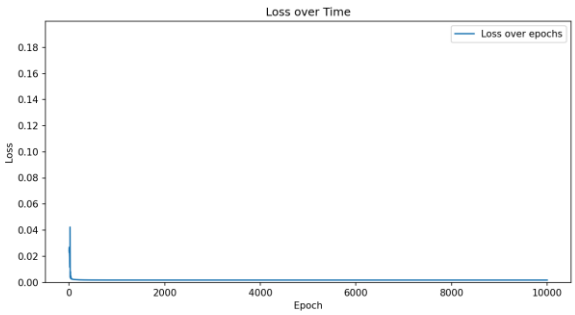

Figure 7

Selection

Both options for using the learning rate to improve the network were successful and very efficient, however cannot work in unison. I have opted to use bold driver for the implementation rather than annealing, as the dynamic learning rate adjustments give it a slight edge.

3. Training and Network Selection

3.1 Training Ideology

In order to train the network, the split data from section 1.6 was passed into the program as an array, with the first 7 columns (the predictors) being split from the expected output.

```
83  def train_network_with_user_input():
84      file_path = 'log_standardised_data.xlsx'
85      X = pd.read_excel(file_path, nrows=316)
86      inputs = X.iloc[:, 0:7].values
87      outputs = X.iloc[:, 7].values
88
89      validationFile = pd.read_excel(file_path, nrows = 106, skiprows = 316)
90      validationSet = validationFile.iloc[:, 0:7].values
91      validationOutput = validationFile.iloc[:, 7].values
92
```

The code above shows the process of splitting and importing the data within the program.

With regard to the ideology of the training process, I chose to adopt large batch training, with the whole of the training set being sent through the network with each epoch. This helps to prevent overshooting if certain subsections of the data are particular skewed or contain outliers. I also chose to use a lower learning rate (before the improvements such as bold driver were added) and higher epochs, which increases the time the network spends learning, however the smaller adjustments result in less chance of overtraining or overshooting.

3.2 Hyperparameters

Hyperparameters are configuration settings set at the start of the learning process and not inferred from the data. They are important for the network to learn smoothly and guide its training.

Learning rate – the initial learning rate has a big impact on the first epoch and consequently the rest of the training process, as too high or low can prevent proper convergence.

```
33          self.learning_rate = 0.0015
```

Momentum – the initial momentum determines the speed the gradient descent is increased by, and if set too high or low could prevent the network from learning properly, by either slowing the gradient down to 0, or boosting it too much and resulting in instant overtraining.

```
30          self.momentum = 0.9
```

Epochs – epochs is the total number of times the data will be passed forwards and backwards through the network. Too few epochs can result in the network not detecting the underlying patterns properly, whilst too many can lead to overfitting on the training data.

```
75      epochs = 2000
```

Weight Initialisation – the weights at the beginning of the learning process are also hyperparameters, as they must prevent vanishing or exploding gradients. See section 2.3 for explanation on the choice of initialisation used.

```
18      np.random.seed(seed)  # Ensure consistency
19      # Initialize weights and biases with Xavier/Glorot initialization
20      self.weights_input_hidden = np.random.randn(self.input_nodes, self.hidden_nodes) * np.sqrt(2. / (self.input_nodes + self.hidden_nodes))
21      self.weights_hidden_output = np.random.randn(self.hidden_nodes, self.output_nodes) * np.sqrt(2. / (self.hidden_nodes + self.output_nodes))
22      self.bias_hidden = np.zeros(self.hidden_nodes)
23      self.bias_output = np.zeros(self.output_nodes)
```

3.3 Training Interface and Monitoring

The interface provided was important in allowing monitoring of training in different circumstances, and the one implemented allowed for the number of hidden nodes to be re-entered every time to allow for a variety in testing.

```
98  while True:
99      try:
100         hidden_nodes = int(input("Enter the number of hidden nodes (or type 'exit' to quit): "))
101     except ValueError:
102         print("Exiting program.")
103         break
104
105     print(f"\nTraining with {hidden_nodes} hidden nodes.")
106     floodIndexNetwork = NeuralNetwork(input_nodes, hidden_nodes, output_nodes)
107
108     for epoch in range(epochs):
109         floodIndexNetwork.train(inputs, outputs.reshape(-1, 1))
110         if epoch == 1999:
111             predicted_outputs = []
112             actual_outputs = []
113             i = 0
114             squared_errors = []
115             while i < 100:
116                 floodIndexNetwork.forward_pass(validationSet[i])
117                 predicted_outputs.append(floodIndexNetwork.predicted_output)
118                 actual_outputs.append(validationOutput[i])
119                 squared_error = (floodIndexNetwork.predicted_output - validationOutput[i])**2
120                 squared_errors.append(squared_error)
121                 i = i + 1
122
```

As shown above, the loss at each epoch was stored, as well as the predicted output and error for each of the values in the validation set once training was completed. This data was then presented as the following:

- A graph showing the loss over time of the network
- A graph showing the actual output vs the predicted output of the network
- The mean squared error of the final network returned as an integer

The code to achieve this is shown below.

```
        # loss graph
        plt.figure(figsize=(10, 5))
        plt.plot(floodIndexNetwork.loss_over_time, label='Loss over epochs')
        plt.title('Loss over Time')
        plt.xlabel('Epoch')
        plt.ylabel('Loss')
        plt.legend()
        plt.ylim(0, 0.2)
        plt.yticks(np.arange(0, 0.2, step=0.02))
        plt.show()

        # predicted vs actual graph
        plt.figure(figsize=(10, 5))
        plt.scatter(actual_outputs, predicted_outputs, color='blue')
        plt.title('Validation Set Testing')
        plt.xlabel('Actual Output')
        plt.ylabel('Predicted Output')
        plt.ylim(0, 1)
        plt.xlim(0, 1)
        # line y = x
        min_val = min(min(predicted_outputs), min(actual_outputs))
        max_val = max(max(predicted_outputs), max(actual_outputs))
        plt.plot([min_val, max_val], [min_val, max_val], 'k--', lw=2, label='Perfect Prediction')
        plt.legend()
        plt.grid(True)
        plt.show()
        mse = np.mean(squared_errors)
        print(f"Mean Squared Error (MSE) on Validation Set: {mse}")
```

3.4 Network Selection

In order to determine the number of nodes needed for the most accurate prediction, it was necessary to test a large number of different amounts and compare their results. To ensure fairness, a random seed was set so that each network would have the same initial random weights and biases.

```
class NeuralNetwork:
    def __init__(self, input_nodes, hidden_nodes, output_nodes, seed=10):
        self.input_nodes = input_nodes
        self.hidden_nodes = hidden_nodes
        self.output_nodes = output_nodes
        self.loss_over_time = []

        np.random.seed(seed)  # Ensure consistency
```

As the network has 7 predictors, I decided to test between 4 and 14 nodes (n/2 and 2n) and the mean square error for each was recorded.

Number of hidden neurons	Mean squared error (MSE) (4.d.p)
4	0.0035
5	0.0033
6	0.0058
7	0.0034
8	0.0026
9	0.0031
10	0.0027
11	0.0025
12	0.0026
13	0.0025
14	0.0025

Figure 8

This testing was done prior to the addition of the bold driver and momentum additions, as these are independent of the number of hidden neurons. It was also conducted over 2000 epochs as opposed to 10000 when testing transfer functions and improvements earlier, as over a longer period the difference would be much harder to distinguish.

4. Evaluation of Final Model

4.1 Performance Metrics – Unrefined Backpropagation

Now that the model has been trained successfully, there are several metrics that can be used to assess its performance. These were recorded for testing on the testing set (rather than the validation set which was used at the end of training), with the following code:

```
160         predicted_outputs = np.array(predicted_outputs).flatten()
161         actual_outputs = np.array(actual_outputs).flatten()
162
163         mse = np.mean(squared_errors)
164         print(f"Mean Squared Error (MSE) on Testing Set: {mse}")
165         rmse = np.sqrt(mse)
166         print(f"Root Mean Squared Error (RMSE) on Testing Set: {rmse}")
167         mae = np.mean(np.abs(actual_outputs - predicted_outputs))
168         print(f"Mean Absolute Error (MAE) on Testing Set: {mae}")
169         # coefficient of determination
170         ss_res = np.sum((actual_outputs - predicted_outputs) ** 2)
171         ss_tot = np.sum((actual_outputs - np.mean(actual_outputs)) ** 2)
172         r_squared = 1 - (ss_res / ss_tot)
173         print(f"R-squared (R^2): {r_squared}")
174
```

All testing was done with 7 neurons in the hidden layer, as determined from section 3.4 network selection, and without momentum and bold driver included.

Mean Squared Error (MSE)

MSE is the average of the squares of the errors (difference between network prediction and actual value). It gives a higher weight to larger errors, so that if a network is significantly off target for a small number of values it will still impact the MSE.

Recorded MSE for the network: **0.0034**

Root Mean Squared Error (RMSE)

RMSE is the root of the mean squared error, meaning it gives a relatively high weight to large errors but not to the extent of MSE.

Recorded RMSE for the network: **0.059**

Mean Absolute Error (MAE)

MAE is the mean of the absolute errors of the predictions. It is much less sensitive to extreme outliers than MSE and RMSE.

Recorded MAE for the network: **0.045**

Coefficient of Determination (R^2)

The coefficient of determination describes how well observed outcomes are predicted by the model, based on the proportion of total variation of outcomes explained by the model. A score of 1 would represent perfect predictions for the model, whilst 0 would be completely inaccurate.

Recorded R^2 for the network: **0.86**

4.2 Performance Metrics – Improvements

Once the improvements to the algorithm (momentum and bold driver) were added, the metrics for the system improved as expected:

Before Improvements	Momentum and Bold Driver Added
0.0034	0.0025
0.059	0.050
0.045	0.039
0.86	0.91

Figure 9

Figure 9 clearly shows that the implementation of the learning rate adjustment algorithms was successful in improving the networks performance (see section 2.7 for details on implementation.)

4.3 Graphical Representation

Graphical representations can provide good understanding of the networks performance through visualisation. Figure 10 shows the networks training performance, with the loss decreasing as it learns, whilst figure 11 shows the comparison between the predicted outputs of the network and the target output.

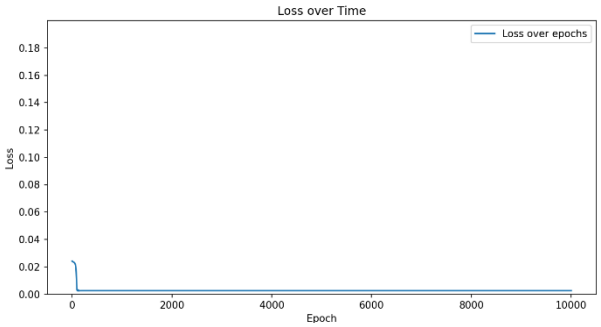

Figure 10

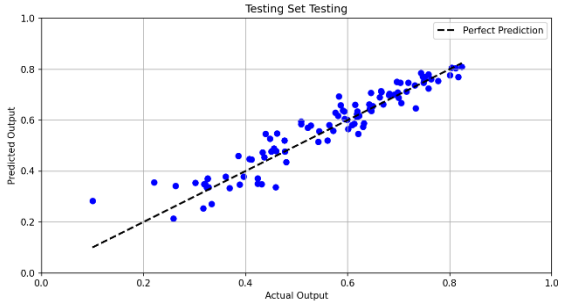

Figure 11

4.4 Versatility and Limitations

The neural network implementation created allows for great versatility, as there are very few hard coded elements. The object's contructor takes in parameters for number of input nodes, number of output nodes and the number of nodes in the hidden layer. This means the program can be ran for any number of inputs or outputs, and a reasonable number of hidden nodes can be inputted to accompany this.

However, the implementation is limited in that it only provides 1 hidden layer within the network. Whilst this is more than adequate for the current problem, much larger scale predictions may require several layers rather than solely an increase in the number of hidden nodes.

In future, it may be beneficial to develop this program to create new classes depending on the number of hidden layers requested.

5. Comparison to Data-Driven Model

5.1 Model Selection

In order to evaluate the network against a linear, data-driven model, it is important to select one which takes multiple independent variables and explores the impact of them on the dependent one.

To satisfy this criteria, I have selected LINEST from Microsoft excel as the regression model. This is because it allows for quick results by not requiring a graphical interface, as well as a plethora of useful statistics for the comparison.

5.2 LINEST Implementation

LINEST can be used directly within the excel spreadsheet containing the data. In order to maintain fairness between the comparison, the model was given the same pre-processed data as the neural network, and was also limited to the testing set:

```
=LINEST(H423:H529,A423:G529,FALSE,TRUE)
```

The above formula was applied to an 8x5 table to produce the output array. From this array certain metrics can be interpreted for comparison to the neural network model.

It is also possible to implement the excel regression function using the analysis toolkit add on, however this wasn't necessary in this situation.

5.3 Statistical Comparison

Due to the linear nature of the excel function, stats such as MSE and RMSE used in section 4 are difficult to accurately obtain. However, the coefficient of determination is provided by excel when the LINEST function is used.

R^2 for Neural Network: **0.91**
R^2 for excel LINEST: **0.70**

As shown above, the excel function had a much worse coefficient of determination, likely due to the non-linear nature of the predictors relationship with the predictand.

5.4 Use Case Comparison

Whilst the neural network was superior in this situation, it may not always be the case when using different predictors and relationships. Figure 12 shows the advantages of each model compared to the other.

Neural Network	LINEST
Can handle more complex relationships between predictors and predictand which linear models cannot	Provides simplicity for linear relationships and is often more accurate when this is the case
Adapts better to different styles of problems, e.g. interpreting non-numerical data where a linear model could not	No risk of overtraining and discovering patterns which shouldn't be applied to predictions
Much more scalable, as for more inputs more nodes and layers can be added – compared to LINEST, where larger data sets will make finding accurate coefficients extremely difficult	Provides more transparency – when working on simpler problems, this allows for better testing than the "black box" of the neural network.