

Data Preprocessing:

Data preprocessing is important to ensure the data is suitable for the neural network by making sure that it's cleaned, standardized, and split into its separate groups.

Cleaning

Cleaning the data involves identifying and handling outliers, null values, and incorrect data values. To handle and clean the data I used the Pandas library in python to automate the cleaning of the data.

```
import pandas as pd

# Read the Excel file to get column names
df = pd.read_csv('DataSet.csv')

# Get the list of column names
c = df.columns.tolist()

# Read the Excel file again to get the full DataFrame
df = pd.read_csv('DataSet.csv')

# Iterate over each row and check for negative values in all columns
for x in df.index:
    for col in c:
        # Check if the value is numeric before comparing
        if pd.to_numeric(df.loc[x, col], errors='coerce') < 0:
            df = df.drop(x)
            break # Once a negative value is found in any column, break out of the inner loop

# Convert columns to numeric, coerce errors to NaN
for col in df.columns:
    df[col] = pd.to_numeric(df[col], errors='coerce')

# Remove rows with NaN values
df = df.dropna()

# Calculate correlation matrix
correlation_matrix = df.corr()

# Save the cleaned DataFrame to a new csv file
df.to_csv('CleanedData.csv', index=False)
correlation_matrix.to_csv('CorrelationMatrix.csv', index=False)

print(correlation_matrix)
```

Figure 1

To ensure all values were corrected I looped through each row and if the value was either negative or non-numeric, I went ahead and deleted the row. Initially for taking all the out of place data I thought about getting averages for the data in that column and filling them in.

I would do this by using a program that takes any result: -999 or NaN and ignore it when calculating the mean for the column. Then fill in where the result = -999 or NaN with the mean for the respected column. After doing this i realize i could also just remove that row as there is so much data, deleting one row isn't going to affect the outcome drastically. However, doing this could remove accurate/important data in other rows which would provide better data for the model.

Normalising

By normalising our data, it makes the data less skewed and draws in the outliers, so the ANN has less work to do in accommodating skewed values. There are numerous ways to go about normalising the data, but I went ahead and applied a log transformation to the data values as all of my data points are positive.

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Read the dataset
df = pd.read_csv('cleanedData.csv')

# Extract the 'Example' column
column = df['Example']

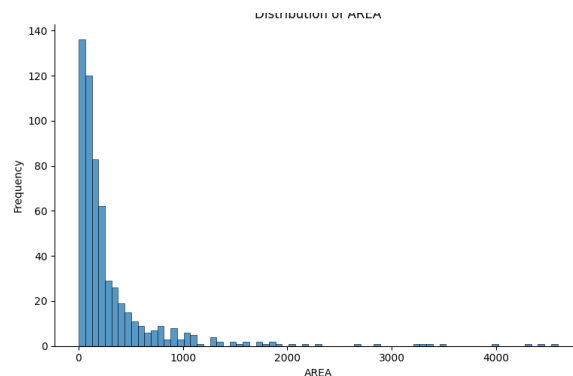
# Calculate skewness for the 'AREA' column
skewness = column.skew()

print(skewness)

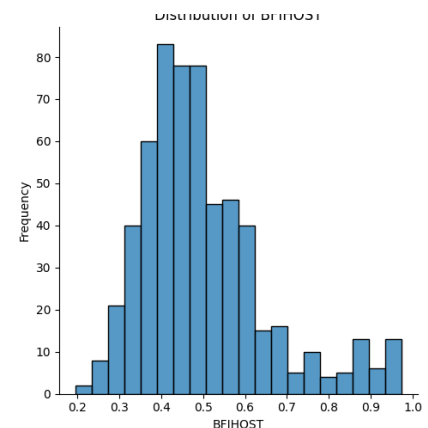
# Plot the distribution of 'AREA' using a distplot
sns.displot(df['Example'])
plt.title('Distribution of Example')
plt.xlabel('Example')
plt.ylabel('Frequency')
plt.show()
```

Figure 2

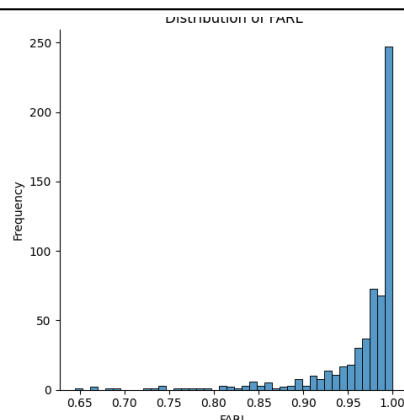
Predictor	Skewness
Area	4.2782094408123



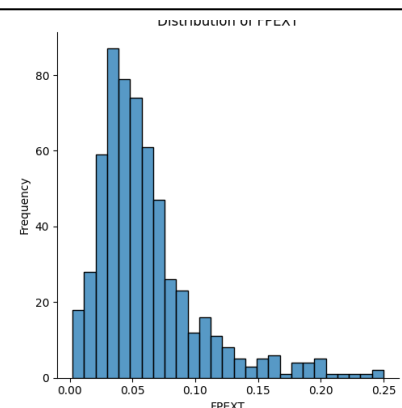
Predictor	Skewness
BFIHOST	1.1732543249838827



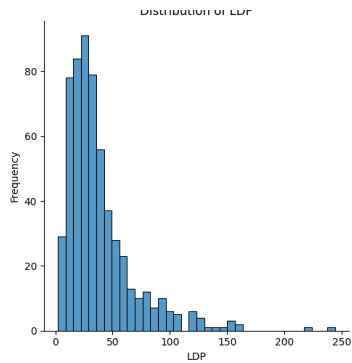
Predictor	Skewness
FARL	-2.976218956726507



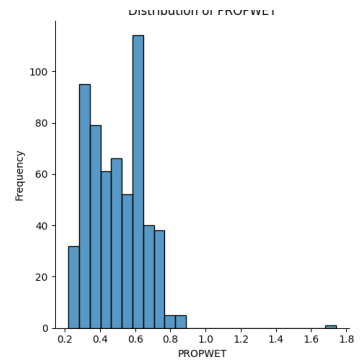
Predictor	Skewness
FPEXT	1.80306349032791



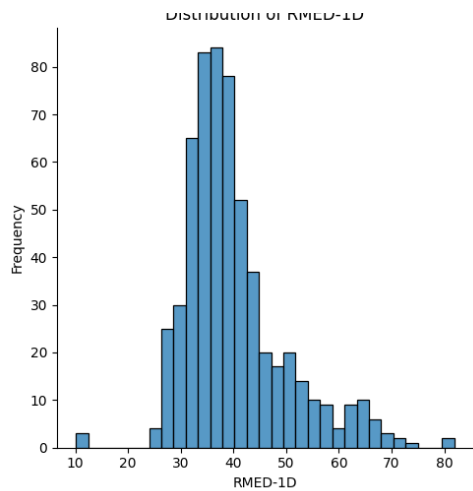
Predictor	Skewness
LDP	2.281258355539437



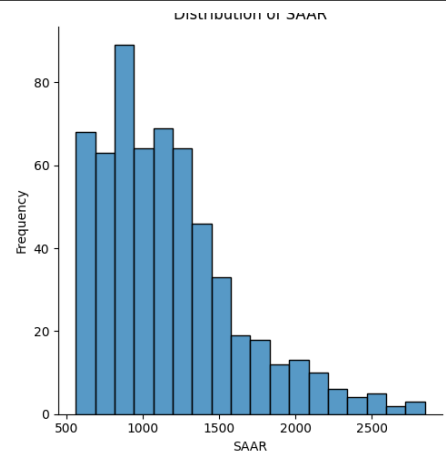
Predictor	Skewness
PROPWET	0.9880093039592643



Predictor	Skewness
RMED-1D	1.1885741825656555



Predictor	Skewness
SAAR	1.0939842665463717



Predictand	Skewness
Index Flood	2.8900186049073064

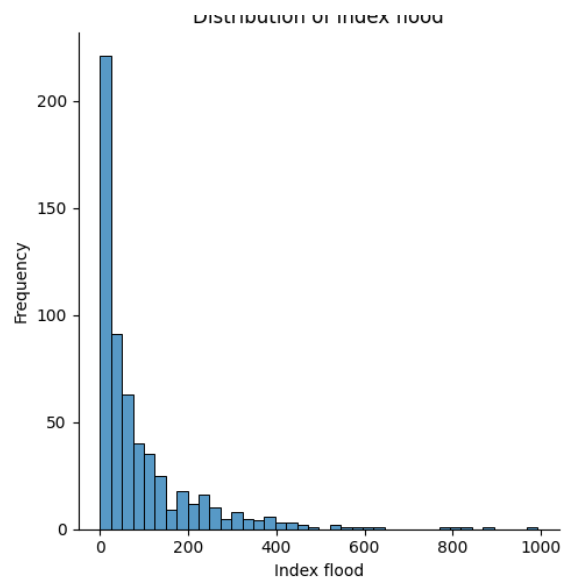


Figure 3

Data skewness is the measure of asymmetry of the probability distribution of a variable relative to its mean. A positive skewness indicates that the tail of the distribution is longer on the right-hand side of the mean and a negative skewness is the opposite. We want to unskew data in such a way that it becomes closer to symmetrical distribution.

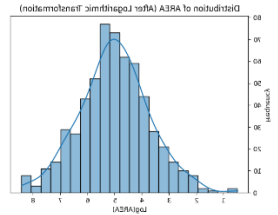
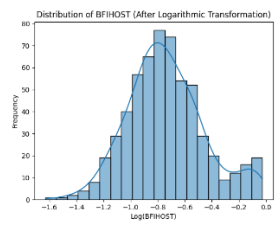
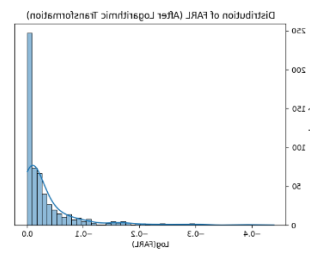
Knowing data skewness of predictors provides insights into the distribution characteristics of the predictors, which can influence the performance and interpretability of the model. Skewed predictors may require special handling or transformation to improve model performance. For example, predictors with high positive skewness might benefit from logarithmic or square root transformations to make their distributions more symmetric.

Ranking the skewness

- **> 1.00** moderate right skewness
- **> 2.00** severe right skewness
- **< -1.00** moderate left skewness
- **< -2.00** severe left skewness

To 'unskew' the Predictors I applied a logarithmic transformation to each variable in question to reduce the skewness. After doing this I ideally want the skewness to be between -0.5 & 0.5!

```
logged = np.log(column)
```

Predictor	Skewness	Distribution table
Area	-0.095350	
BFIHOST	0.350735	
FARL	-3.334382	

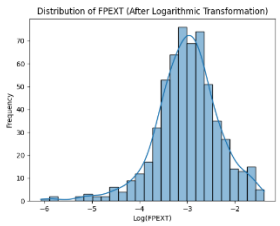
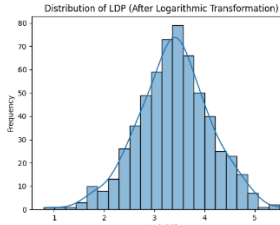
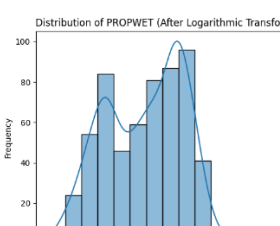
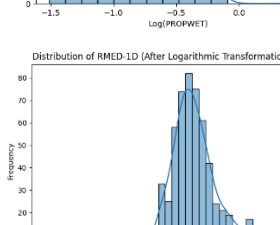
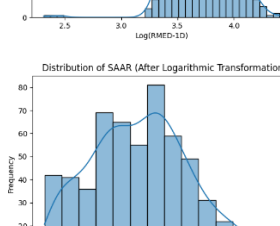
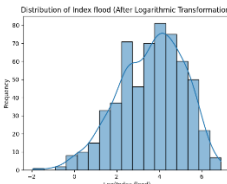
FPEXT	-0.632972	
LDP	-0.142840	
PROPWET	-0.150802	
RMED-1D	-0.142696	
SAAR	0.248650	

Table 1

Predictors	Skewness	Distribution
Index Flood	-0.413561	

Standardising

Once the data was clean, I went on to standardizing the data into a common scale of [0, 1], it's important for neural networks as it helps in faster convergence during training and prevents

certain columns from dominating others during training. I also used Min-Max scaling to standardize the input data alongside the Pandas library:

```
import pandas as pd
from sklearn.preprocessing import MinMaxScaler

# Load the data from cleanedData.csv
data = pd.read_csv('cleanedData.csv')

# Extract features (X) and target (y)
X = data.drop(columns=['Index flood']) # Assuming 'Index flood' is the target column
y = data['Index flood']

# Combine features and target for scaling
combined_data = pd.concat([X, y], axis=1)

# Min-Max scaling
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(combined_data)

# Create a DataFrame with scaled data
scaled_df = pd.DataFrame(scaled_data, columns=combined_data.columns)

# Write the scaled data to a new file called EqualData.csv
scaled_df.to_csv('EqualData.csv', index=False)
```

Figure 4

Correlation

After standardising all the data between [0, 1] I wanted to find out the trends between each column to see if any were highly related. To do this i calculated the correlation between each column in the data set by creating a correlation matrix using the `df.corr()` function in pandas. To visualize the correlation matrix in a clearer way i utilized seaborn and matplotlib modules to create a heatmap for the correlation:

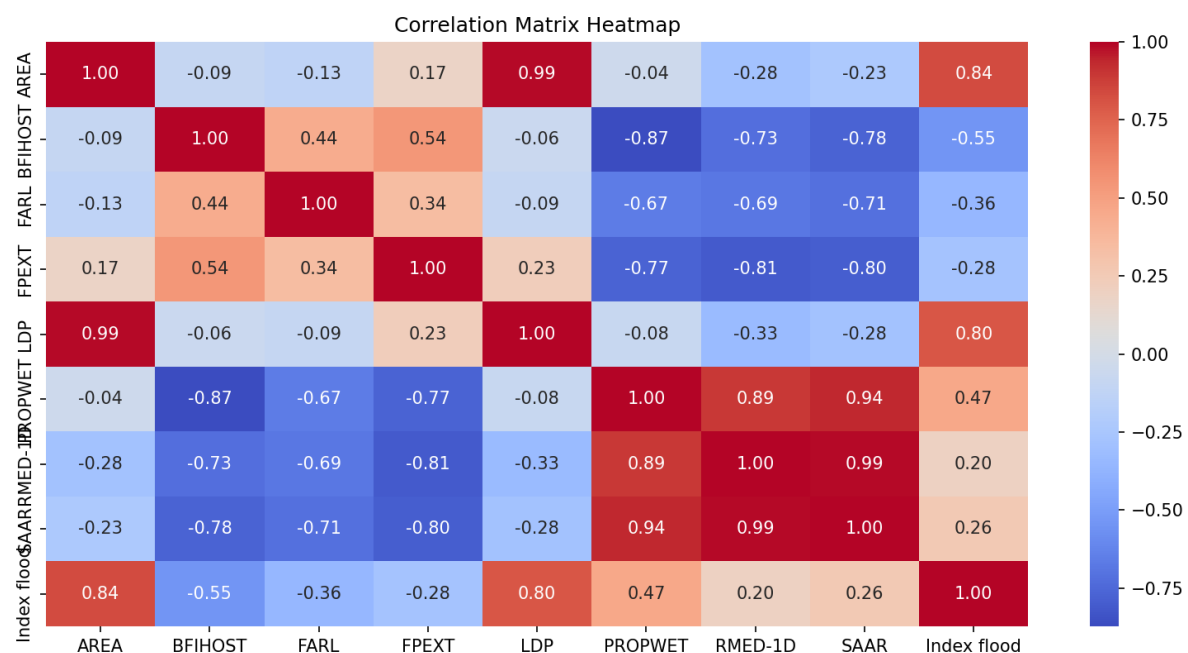
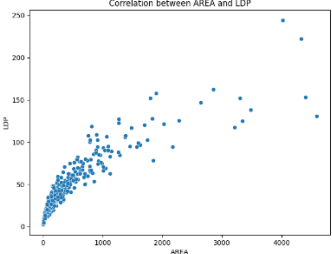
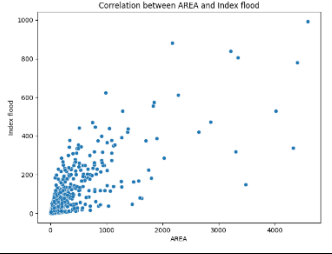
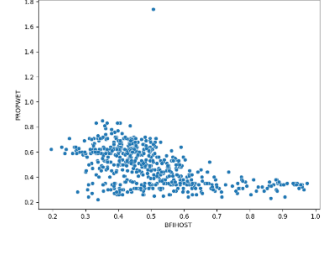
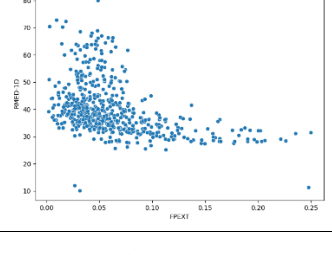
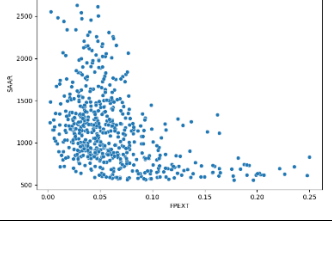


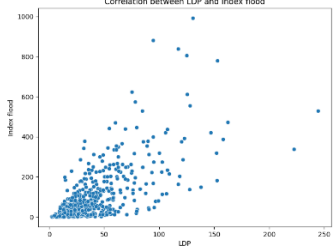
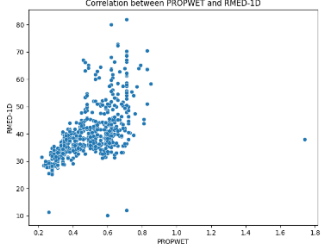
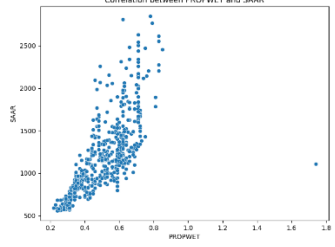
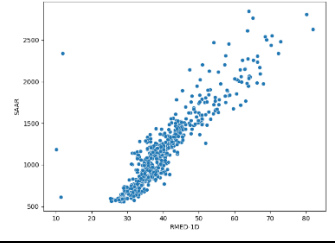
Figure 5

After reviewing the heat map, these are some of the relationships that stood out to me:

- Area & LDP = 0.99
- Area & index flood = 0.84
- BFIHOST & Propwet = -0.87
- FPEXT & RMED-1D = -0.81
- FPEXT & SAAR = -0.80
- LDP & index flood = 0.80
- PROPWET & RMED-1D & 0.89
- PROPWET & SAAR = 0.94
- RMED-1D & SAAR = 0.99

Table 2

Relationship	Correlation	Visualised
Area & LDP	0.99	
Area & Index Flood	0.84	
BFIHOST & PROPWET	-0.87	
FPEXT & RMED-1D	-0.81	
FPEXT & SAAR	-0.80	

LDP & Index Flood	0.80	
PROPWET & RMED-1D	0.89	
PROPWET & SAAR	0.94	
RMED-1D & SAAR	0.99	

Predictor Variables:

- AREA
- LDP
- PROPWET
- RMED-1D
- SAAR

These variables have shown strong to very strong correlations with other variables in the dataset, indicating their potential to predict or explain variation in the outcome variable.

Predictand Variable:

- Index Flooding

The "Index flood" variable is the likely predictand variable because it's the variable that is being predicted or explained by the predictor variables.

What predictors I'm choosing and why:

- Correlations with the predictand:
 - Area & index flood = 0.84
 - LDP & index flood = 0.80
- Correlations among predictors:
 - none because where predictors are highly correlated, it can be challenging for the model to distinguish the individual effects of each predictor on the target variable. This can result in inflated standard errors, which affects the reliability of the coefficient estimates. Additionally, multicollinearity can lead to unstable predictions when the model is applied to new data.
- This ensures that the model only captures meaningful relationships without introducing unnecessary redundancy or noise.

Data Splitting

Once all the data has been pre-processed, I used the NumPy and pandas module to split the data into 3 important sections: Training, Validation and Testing. The Training data represented the bulk of the data set containing 60% of its cleaned size, Validation and Testing both had 20% of the cleaned data's size.

Training data:

The Training data set was used to train the model, which is why it has the most data in, this will improve the model's performance as more training data allows the model to learn more relationships, leading to better performance on unseen data. It also creates better generalisation through a better understanding of patterns and variations within the data. This helps prevent overfitting, where the model remembers data instead of learning patterns.

Testing data:

The testing data set is used to evaluate its performance during development, which is why it only has a minor percentage of the original size. Once the model has become used to patterns on the training data set, it's put to work on the test data to find matching patterns within the data and to produce a reasonable prediction based on the information given. This can allow you to fine tune the model and figure out if anything needs changing. E.g. NO. epochs, NO. hidden nodes and the learning rate.

Validation data:

The Validation data set is used to assess the performance of the model during training in order for us to tune hyper parameters such as the learning rate, activation functions and NO. hidden nodes. By evaluating the model's performance on data that it hasn't seen during training, you can get an unbiased estimate of its generalization performance.

```

import pandas as pd
import numpy as np

# Load your dataset
df = pd.read_csv('EqualData.csv')

# Shuffle the dataset
df = df.sample(frac=1, random_state=42).reset_index(drop=True)

# Calculate the sizes for each set
total_size = len(df)
train_size = int(0.6 * total_size)
validation_size = int(0.5 * (total_size - train_size))

# Split the dataset
train_data = df.iloc[:train_size]
validation_data = df.iloc[train_size:train_size + validation_size]
test_data = df.iloc[train_size + validation_size:]

# Save the datasets to CSV files
train_data.to_csv('TrainData.csv', index=False)
validation_data.to_csv('ValidationData.csv', index=False)
test_data.to_csv('TestData.csv', index=False)

```

Figure 6

Implementation of the MLP algorithm

Architecture

The MLP is one big class which consists of an input layer, one or more hidden layers, which can be changed in order to alter the MLP's performance, and an output layer. Each layer contains multiple neurons, and connections between neurons have associated weights and biases. These parameters are all defined at the beginning of the class. Before any functions are written I also initialised the weights and biases with random values using NumPy. Inside the MLP class I created functions for the forward propagation algorithm, backwards propagation algorithm, the Activation function and derivative and the training function.

Forward pass

During the forward pass, input data propagates through the network and the weighted sums of inputs are passed with a bias through activation functions to produce output activations.

To begin with I computed the activation function for the nodes in the hidden layer by using np.dot function in NumPy which computes the dot product of two arrays. In this case its between the input weights for the hidden layer and input data 'X' where each row represents a sample, and each column represents a feature. A bias is then added to each row of the hidden_activation array. The hidden_output applies the activation function to each row in the hidden activation array to get an output for each hidden node.

The output_activation performs the same mathematics as the hidden_activation variable but uses the output from the hidden node as input data., then we finally apply the activation function to the output_activation variable of the hidden node to get the final output for the network.

Activation functions

Then I defined the activation_function that I would be using during the backward propagation, and in this case I've decided to use the sigmoid function. I used the sigmoid function because it's smooth and continuous which ensures smooth gradients during backwards propagation, it also introduces non-linearity to the network so the neural network can learn complex patterns in data.

However, there was an issue which was present in my model which is the vanishing gradient problem, where the gradients became so small for extreme values that it made the convergence during training extremely slow.

I also defined a function for the derivative of the activation function because it's used to calculate the gradients of the loss function with respect to the network's parameters, by knowing these gradients, the backpropagation allows us to update parameters in a direction that minimizes the loss function

Back propagation

To calculate the output error I computed the error between the predicted output (self.output) and the actual output (y). This error represents how far off the network's prediction is from the true target values.

To calculate the output delta, I computed the delta (gradient) for the output layer by multiplying the output error by the derivative of the activation function applied to the output layer's activations. This delta indicates the direction and magnitude of adjustment needed for the output layer's weights and biases to reduce the error.

Then to calculate the hidden nodes error I backpropagated the output delta through the weights connecting the hidden and output layers. And to calculate the hidden nodes delta I multiplied the hidden layer's error by the derivative of the activation function applied to the hidden layer's activations. This delta indicates the adjustment needed for the hidden layer's weights and biases to reduce the error in the output layer.

Finally, I ended the propagation by updating the weights and biases of both the hidden-to-output and input-to-hidden connections based on their respective deltas and the learning rate. These updates are performed to minimize the error in the output layer by adjusting the parameters of the network through gradient descent.

Training

To begin with my training function, I initialised the learning rate and began the training loop, for each epoch I performed a forward pass and backward pass to obtain predicted output for input data 'X' then I computed the **Mean Squared Error** between predicted output and actual output 'Y'.

Every 100 epochs I print the corresponding loss at that epoch.

Improvements

After running my code and not being happy with the performance of it, I decided to make some improvements that were mentioned in the lecture. After some trial and error with the modifications I stuck with a bold driver and weight decay.

Bold Driver

To implement the **bold driver**, I had to define the threshold for error decay at the beginning, alongside the learning rate. I then went on to check the change in loss after every 1000 epochs to determine if the learning rate needs adjustment. If the error increases by more than what was set for the predefined threshold (4% increase/decrease), it indicates that the learning rate might be too large. In this case you decrease the learning rate by a factor of 0.7 (30% decrease).

If the error either decreases or remains the same, you accept the weight changes and adjust the learning rate accordingly and, in this case, I will increase the learning rate by a factor of 1.05 (5% increase) and bound the learning rate within the range of 0.01 to 0.5 to avoid extreme changes.

Weight Decay

To implement the **weight decay** modification, I had to initialise the weight decay value at the beginning of the function to a typical value such as 0.001. Then, after computing the gradient of the weights with respect to the loss (e.g., `np.dot(self.hidden_output.T, output_delta)`), I went ahead and subtracted the weight decay term (`weight_decay * self.weights_hidden_output`) from the gradient.

This term penalizes the larger weights and encourages smaller weights during updates, preventing overfitting and promoting a more generalized model.

Overall nothing in my code was hardcoded in so it can be suitable for many different sets of data.

Training and Network Selection

Comparisons between modification methods

Weight decay

Weight decay is a regularization technique used to prevent overfitting and improve the generalization performance of a neural network model. By incorporating weight decay, the model penalizes large weights to prevent the function from becoming too "rough," where minor changes in input can lead to large changes in output. This is particularly important as large weights on output nodes can cause the model to produce outputs beyond the range of the training data, resulting in poor generalization. In essence, weight decay adds a penalty term to the error function, encouraging the model to learn simpler patterns and reducing the risk of overfitting.

Bold driver

The bold driver is a technique used to dynamically adjust the learning rate during training based on the gradient of the loss function. With a fixed learning rate parameter, gradient descent down a surface of low curvature can lead to successively smaller steps, potentially causing slow convergence or getting stuck in local minima. The bold driver addresses this issue

by modifying the learning rate in response to the gradient, allowing for more adaptive learning. However, there is a risk of the learning rate increasing excessively, which is why it's important to impose boundaries on its values to prevent this snowball effect. This adaptive approach helps strike a balance between convergence speed and stability during training.

Momentum

Adding a momentum term to gradient descent leads to more rapid progress along the valley compared with unmodified gradient descent. This momentum contributes to faster convergence, more stable training dynamics, and improved exploration of the parameter space. By incorporating momentum, the algorithm gains momentum during descent, enabling it to navigate through local minima more effectively and reach the optimal solution with greater efficiency.

Annealing

It helps balance between exploring different solutions widely and exploiting promising ones effectively. By allowing occasional uphill moves, it helps the algorithm escape local optima and explore diverse regions of the search space. Annealing schedules aid convergence by gradually narrowing down the search space towards promising solutions.

Comparison

In order to accurately compare the different modifications, I individually included them within my data and ran them each 3 times to get an average value of Loss after 5000 epochs, with these parameters remaining constant:

- Input size = 2
- NO. hidden nodes = 4
- Learning rate = 0.1

Table 3

Modification	AVG loss after 5000 epochs
Weight Decay	0.015039261724994954
Bold Driver	0.007017010854093803
Momentum	0.015486436563988866
Annealing	0.009792918475328903

This gave me the information I needed to make my final decision on what improvements to choose.

What I avoided:

I chose to avoid momentum because it can sometimes cause overfitting when the data set isn't very big, this is where I would rather preserve my computational resources and opt for something a lot simpler such as weight decay.

However, when it came to annealing, I would've had to remove the bold driver because you cannot have both. Because I saw the impact that the bold driver had I couldn't remove it as annealing only narrows down the search space however the bold driver allows the mlp to effectively manoeuvre around the search space and reach a location of reduced errors faster.

Evaluation of Final Model

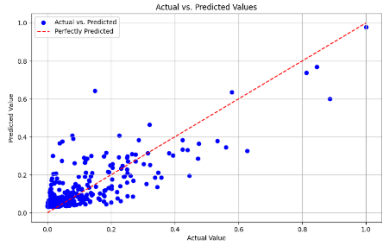
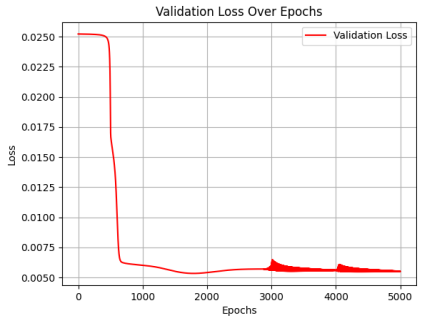
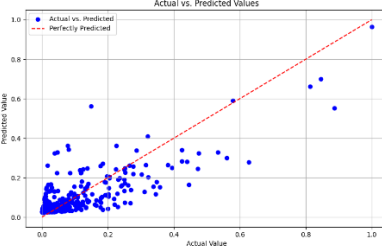
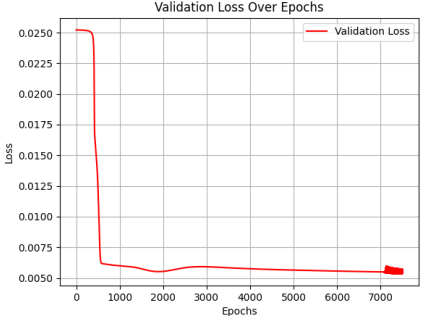
Modifying the Bold driver and Weight Decay parameters:

These tests are completed for 15,000 epochs with these parameters staying constant:

- Input size = 2
- NO. hidden nodes = 4
- Learning rate = 0.1

Table 4

Weight Decay	Bold Driver	Graph of predictions	Validation Loss
0.01	Error change threshold: 1.04 Decreasing learning rate: 0.7 Increasing learning rate: 1.05 Boundaries: [0.01, 0.5]		
0.01	Error change threshold: 1.08 Decreasing learning rate: 0.5 Increasing learning rate: 1.05 Boundaries: [0.01, 0.05]		
0.01	Error change threshold: 1.10 Decreasing learning rate: 0.7 Increasing learning rate: 1.10 Boundaries [0.01, 0.75]		

0.01	<p>Error change threshold: 1.04</p> <p>Decreasing learning rate: 0.7</p> <p>Increasing learning rate: 1.05</p> <p>Boundaries [0.01, 0.25]</p>		
0.01	<p>Error change threshold: 1.07</p> <p>Decreasing learning rate: 0.8</p> <p>Increasing learning rate: 1.025</p> <p>Boundaries [0.01, 0.75]</p>		

My Findings

In my first variation of training the model with different parameters for only the improvements, bold driver and weight decay, I had what I believed to be the standard model with an Error change threshold: 1.04, Decreasing learning rate: 0.7, Increasing learning rate: 1.05, Boundaries: [0.01, 0.5] and a weight decay of 0.01, this gave me a base prediction graph which what I will compare the other variations to.

In my second variation of training the model I increased the error changing threshold to 1.08 (8%) and increased the decreasing learning rate to 0.5, I saw a slight difference in the accuracy of the predicted values as they were a greater distance away from the perfectly predicted line.

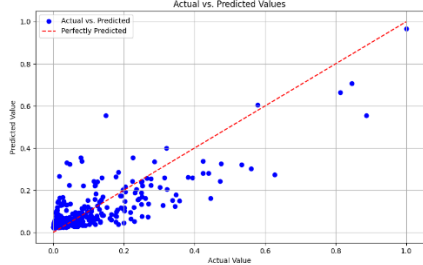
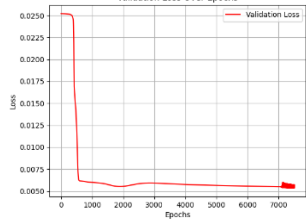
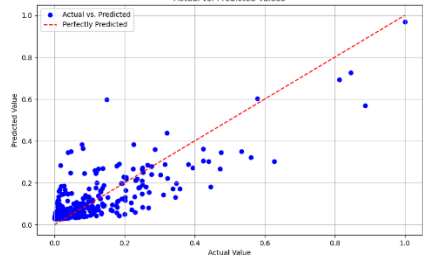
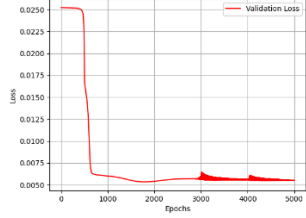
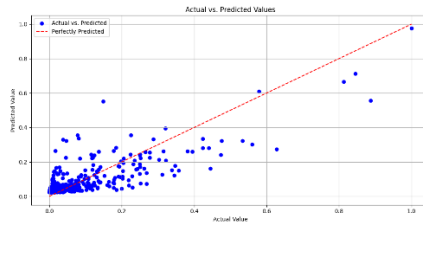
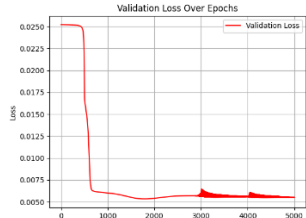
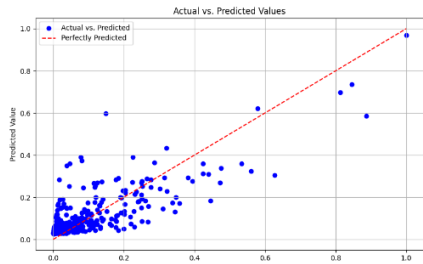
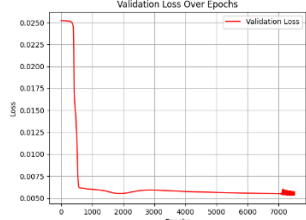
In my third variation of training the model I increase the error change threshold to 1.10, the increasing learning rate to 1.10 and increased the maximum boundary to 0.75. doing this I was expecting it to search the weight space more effectively and after reviewing the outputted graph I came to the realisation that it predicted values slightly better but not perfectly when comparing the graph to the graph of version 1.

In my fourth version I decreased the maximum boundary to 0.25 when comparing it to the model in version 1. Comparing the graphs it shows that the predictions weren't as good, so these options are ones to avoid.

In my final comparison, version 5, I increased the error change threshold, the decreasing learning rate to 0.8 and the boundary max to 0.75, however I decreased the increasing learning rate to 1.025. after making these changes it was clear to me that these changes had a negative effect on the prediction capabilities of the model.

Changing the weight decay and keeping bold drivers' parameters the same:

Table 5

Weight Decay	Bold Driver	Predictions	Validation Loss
0.02	Error change threshold: 1.04 Decreasing learning rate: 0.7 Increasing learning rate: 1.05 Boundaries: [0.01, 0.5]		
0.015	Error change threshold: 1.04 Decreasing learning rate: 0.7 Increasing learning rate: 1.05 Boundaries: [0.01, 0.5]		
0.005	Error change threshold: 1.04 Decreasing learning rate: 0.7 Increasing learning rate: 1.05 Boundaries: [0.01, 0.5]		
0.0025	Error change threshold: 1.04 Decreasing learning rate: 0.7 Increasing learning rate: 1.05 Boundaries: [0.01, 0.5]		

After all the changes I've made the following assessment of the parameters of the bold driver and weight decay:

- Don't reduce/ change the weight decay as 0.01 seems to be the optimal value for me.

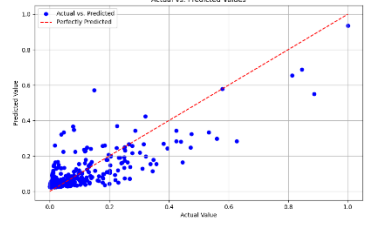
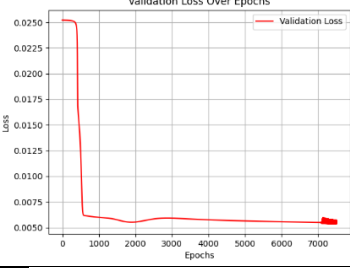
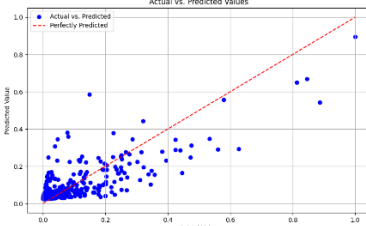
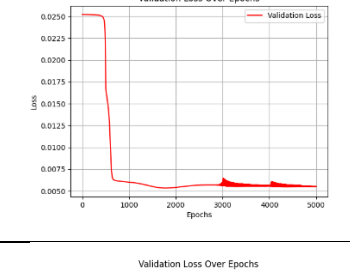
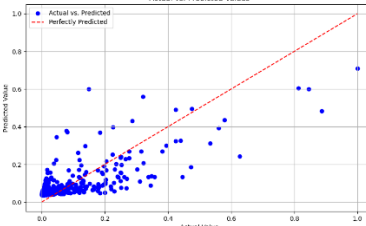
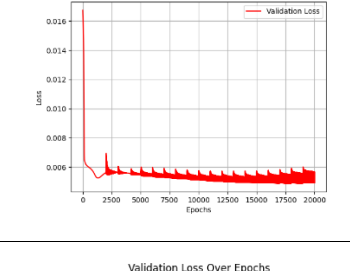
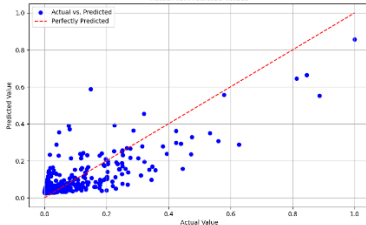
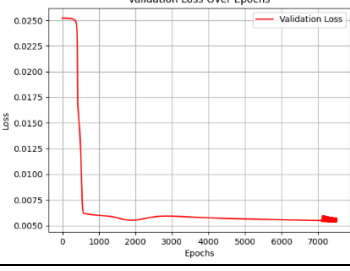
- Increasing the maximum boundary is key to search the weight space with more efficiency.
- Keep the error change threshold to a minimum so it can actively change the learning rate.
- Keep the decreasing learning value constant as 0.7 is optimal.
- Increase the increasing learning rate so it can search the weight space quicker.

Parameter changes- graphs and showing changes.

All tests done over 15,000 epochs with a weight decay of 0.01, Error change threshold: 1.04,

Decreasing learning rate: 0.7, Increasing learning rate: 1.05 and Boundaries: [0.01, 0.5]:

Table 6

Learning Rate	Hidden Nodes	Predictions	Validation Loss
0.01	4		
0.05	4		
0.001	4		
0.02	4		

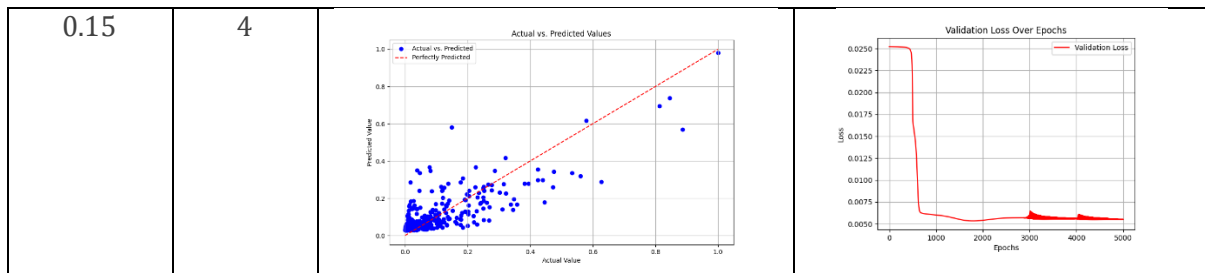


Table showing how predictions are affected by learning rates with constant hidden nodes. Comparing this with the validation data gives me an indication of when the parameters need to be changed and don't work over so many epochs

My NO. Hidden nodes are optimal between $n/2$ and $2n$, n being the input size. Because my input size was 2 the NO. hidden nodes that would be best for my model fall between 1 and 4.

Table 7

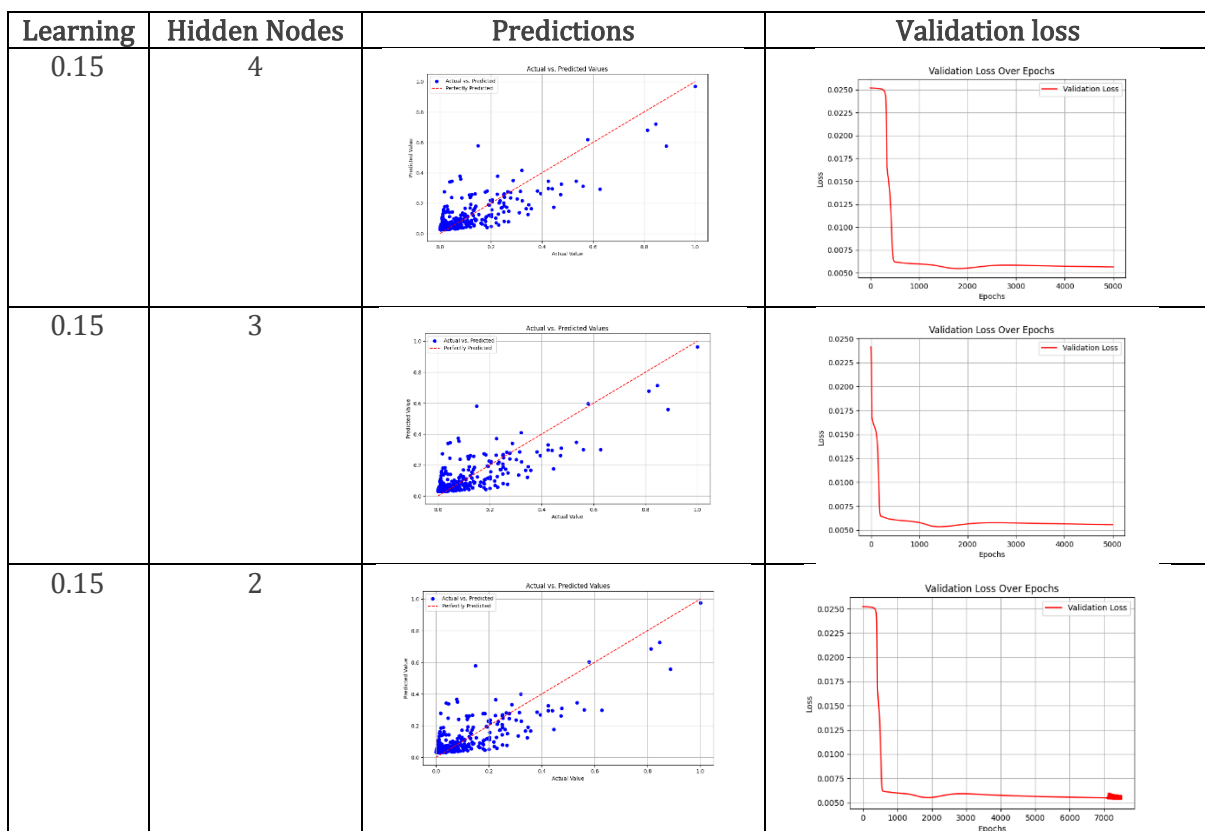


Table showing how predictions are affected by Hidden nodes with constant learning rates.

After computing this controlled test on both the hidden nodes and learning rate I reached these conclusions:

- Having more hidden nodes is best for most of the predictions but not so important/relevant when it comes to predicting them on spot.

- Increasing the learning rate was much more beneficial for the neural network as it grouped the predicted values closer to the line of perfect prediction.
- I've settled on:
 - Learning rate = 0.15
 - NO. hidden nodes = 4

Using a bold driver

I used a bold driver because the everchanging learning rates seemed to manoeuvre around the weight space with high levels of efficiency. I believe it is the main improvement which reduced the loss significantly.

Performing weight decay

I used weight decay, so it prevents over fitting and to improve the generalisation of the model and encourages the model to learn simple patterns. The better generalization allows the model to perform better on unseen data and allows it to be more robust and adaptable. This ensures that the model can be used on many different data sets and is reusable.

Graphs of final models' performance

Learning rate = 0.15

NO hidden nodes = 4

Weight decay = 0.01

Error change threshold: 1.04

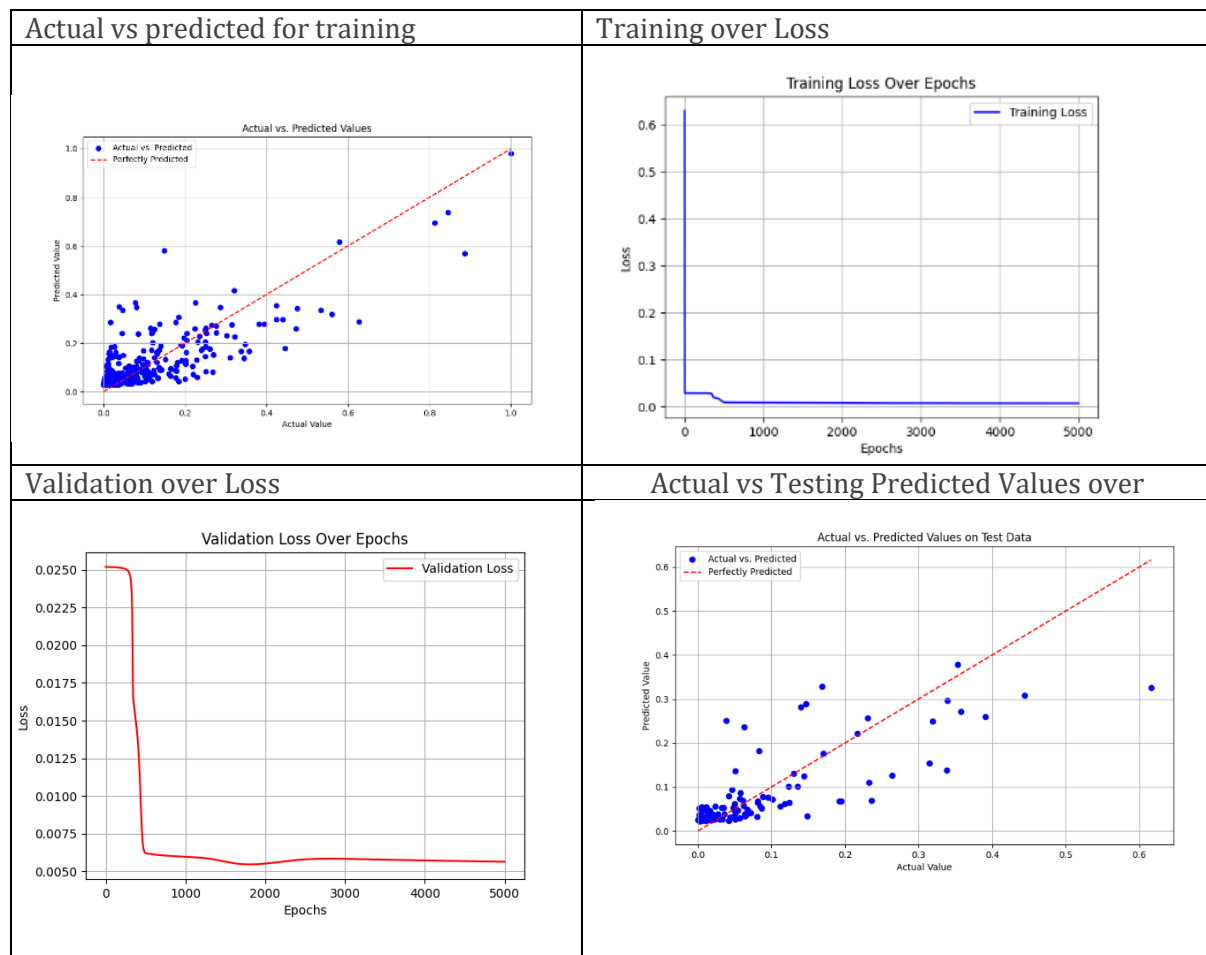
Decreasing learning rate: 0.7

Increasing learning rate: 1.10

Boundaries: [0.01, 0.75]

Epochs = 5,000

Table 8



After a final evaluation model and testing it against the test data by performing 1 forward pass, I achieved a MSE for the test data to be “0.004442615025710937” which is an improvement on the MSE for the training data “0.006918618885870329”. showing me that the model was learning and constantly improving. It's crucial to note that the model's ability to perform well on unseen test data indicates strong generalization capabilities. the model's ability to generalize suggests that it has learned meaningful features from the training data and can make accurate predictions on similar but previously unseen examples.

Moreover, the fact that the model's performance on the test data was not significantly worse than on the training data indicates that the model did not overfit to the training data. Overfitting occurs when a model learns noise or irrelevant patterns from the training data, leading to poor performance on unseen data. However, in this scenario, the model's performance on the test data demonstrates that it captured the underlying patterns in the data without memorizing specific instances. This suggests that the model struck an appropriate balance between complexity and generalization, resulting in robust performance on unseen data.

Regarding the future, more avenues should be explored to improve the efficiency and reusability of the model. One possible area of focus is examining neural networks, such as deep networks or networks at other levels. Furthermore, adding additional features or data sources to the model can improve its predictive capability. Finally, exploring real-world applications and applying the model to practical situations can provide valuable insights and help solve related problems in various sectors.

Comparison with another data driven model or baseline.

<https://www.statology.org/excel-linest-multiple-regression/>

Compare to LINEST on excel.

Comparing my model against LINEST in Excel can provide valuable insights into the capabilities of my MLP.

In Excel LINEST is often used as a simple baseline for regression analysis. It calculates the least squares fit for a given set of data points, providing a basic linear regression model. By comparing my model against LINEST, i can evaluate whether my model performs better or worse than a basic linear regression approach.

Also, LINEST produces coefficients that are easily interpretable. These coefficients represent the slope and intercept of the linear regression line. Comparing the coefficients from LINEST with the weights learned by my model can help me understand how my model's features are contributing to predictions.

Finally, LINEST provides a simple linear regression model, while my model may incorporate more complex relationships between variables. Comparing the performance of both models can help me determine whether the added complexity of my model is justified by improvements in predictive accuracy.

To use LINEST I removed all the predictors which I don't use and moved all my AREA, LDP and Index flood data to columns A, B, C. From here I picked a random box on the spreadsheet and plugged in this line of code:

=LINEST (C2:C65, A2:B65,TRUE,TRUE)

And excel outputted these values:

Table 9

-1.80195	0.244053	66.20979
1.113491	0.059496	26.55212
0.531838	98.99685	#N/A
34.64843	61	#N/A
679135.4	597823	#N/A

What do these values show:

0.059496 is the standard error for the AREA column

1.113491 is the standard error for the LDP column

The R^2 for the model is 0.531838 shows that 53% of the variance can be explained so there can be extreme variability in the dataset.

$$Y = 66.20979 + 0.2440053(\text{AREA}) - 1.80195(\text{LDP})$$

This is the prediction model from LINEST, and it provides a Mean Squared Error (MSE) of 0.028895876543456789. My model produces a Mean Squared Error (MSE) of 0.006918618885870329, which is somewhat like my implementation of the MLP. Therefore, I shall conclude that excel is as good as calculations as my model.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

class MLP:
    def __init__(self, input_size, hidden_size, output_size):
        """
        Initialize Multilayer Perceptron (MLP) model.

        Parameters:
        - input_size (int): Number of input features.
        - hidden_size (int): Number of neurons in the hidden layer.
        - output_size (int): Number of output neurons.

        Initializes weights and biases with random values.
        """
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        # Initialize weights with random values
        self.weights_input_hidden = np.random.rand(self.input_size, self.hidden_size)
        self.weights_hidden_output = np.random.rand(self.hidden_size, self.output_size)

        # Initialize biases with random values
        self.bias_hidden = np.random.rand(1, self.hidden_size)
        self.bias_output = np.random.rand(1, self.output_size)

    def forward(self, X):
        """
        Perform forward pass through the network and Returns the output of the network after forward pass.

        The forward pass computes the output of the neural network given the input data X.
        It involves the following steps:
        1. Compute the activation of the hidden layer neurons.
        2. Apply the activation function to the hidden layer activation.
        3. Compute the activation of the output layer neurons.
        4. Apply the activation function to the output layer activation.
        5. Return the output of the network.
        """
        # Forward pass through the network
        self.hidden_activation = np.dot(X, self.weights_input_hidden) + self.bias_hidden # Compute the activation of the hidden layer
        self.hidden_output = self.activation_function(self.hidden_activation) # Apply activation function to the hidden layer activation

        self.output_activation = np.dot(self.hidden_output, self.weights_hidden_output) + self.bias_output # Compute the activation of the output layer
        self.output = self.activation_function(self.output_activation) # Apply activation function to the output layer activation

        return self.output

    def activation_function(self, x):
        # Activation function (sigmoid in this case)
        return 1 / (1 + np.exp(-x))

    def activation_derivative(self, x):
        # Derivative of the activation function
        return x * (1 - x)

    def backward(self, X, y, learning_rate):
        # Compute the error between the predicted output and the actual output
        output_error = y - self.output
        # Compute the delta (gradient) for the output layer using the error and the derivative of the activation function
        output_delta = output_error * self.activation_derivative(self.output).reshape(-1, 1)

        # Compute the error for the hidden layer by backpropagating the error from the output layer
        hidden_error = output_delta.dot(self.weights_hidden_output.T)
        # Compute the delta (gradient) for the hidden layer using the error and the derivative of the activation function
        hidden_delta = hidden_error * self.activation_derivative(self.hidden_output)

        # Update the weights between the hidden and output layers based on the hidden layer's output and the output delta
        self.weights_hidden_output += np.dot(self.hidden_output.T, output_delta) * learning_rate
        # Update the weights between the input and hidden layers based on the input data and the hidden delta
        self.weights_input_hidden += np.dot(X.T, hidden_delta) * learning_rate

        # Update the biases for the output layer based on the output delta and hidden delta
        self.bias_output += np.sum(output_delta, axis=0) * learning_rate
        self.bias_hidden += np.sum(hidden_delta, axis=0) * learning_rate

        return output_delta
```

```

def train(self, X_train, y_train, X_val, y_val, epochs, initial_learning_rate, final_learning_rate):
    # Define weight decay
    weight_decay = 0.01 # You can adjust this value as needed
    losses_train = [] # List to store training loss values over epochs
    losses_val = [] # List to store validation loss values over epochs

    learning_rate = initial_learning_rate # Initialize learning rate

    for epoch in range(epochs):
        output_train = self.forward(X_train)
        output_delta = self.backward(X_train, y_train, learning_rate) # Get output_delta from backward pass
        loss_train = np.mean(np.square(y_train - output_train))
        losses_train.append(loss_train)

        # Compute validation loss
        output_val = self.forward(X_val)
        loss_val = np.mean(np.square(y_val - output_val))
        losses_val.append(loss_val)

        # Bold Driver
        error_change_threshold = 1.04 # Predefined error change threshold
        if epoch % 1000 == 0: # Update every thousand epochs
            prev_loss = losses_train[-2] if len(losses_train) >= 2 else float('inf') # Previous loss
            current_loss = losses_train[-1] # Current loss

            if current_loss > prev_loss * error_change_threshold: # Error increased
                # Undo weight changes
                self.weights_hidden_output += (np.dot(self.hidden_output.T, output_delta) - weight_decay * self.weights_hidden_output) * learning_rate
                self.weights_input_hidden += (np.dot(X_train.T, hidden_delta) - weight_decay * self.weights_input_hidden) * learning_rate

                # Decrease learning rate
                learning_rate *= 0.7 # Decrease learning rate by 30%
            else: # Error decreased or remained same
                # Adjust learning rate
                learning_rate *= 1.05 # Increase learning rate by 5%
                learning_rate = min(max(learning_rate, 0.01), 0.75) # Bound learning rate

        # Print loss (optional)
        if epoch % 100 == 0:
            print(f'Epoch {epoch}, Training Loss: {loss_train}, Validation Loss: {loss_val}, Learning Rate: {learning_rate}')

    # Plot training loss
    plt.plot(range(len(losses_train)), losses_train, label='Training Loss', color='blue')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Training Loss Over Epochs')
    plt.legend()
    plt.grid(True)
    plt.show()

    # Plot validation loss
    plt.plot(range(len(losses_val)), losses_val, label='Testing Loss', color='red')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Testing Loss Over Epochs')
    plt.legend()
    plt.grid(True)
    plt.show()

    # Calculate predicted output for the training data
    predicted_output = mlp.forward(X_train)

    # Compute squared differences between actual and predicted outputs
    squared_errors = np.square(y_train - predicted_output)

    # Calculate Mean Squared Error (MSE)
    mse = np.mean(squared_errors)

    print("Mean Squared Error (MSE):", mse)

```