Michael Ly
COB107 Report

1. Data Pre-Processing

The dataset provided is data that represents daily environmental factors at a site in San Diego, USA, from 1/1/1987 – 31/12/1990. In the dataset there are five fields that are recorded.
The data fields are:

- Date in the format of mmddyy
- T – Mean daily temperature in Celsius
- W – Wind speed in cm/s
- SR – Solar radiation in Langleys
- DSP – Air pressure in kPa
- DRH – Humidity in %
- PanE – Pan Evaporation in cm/day

In this dataset, the predictors are: T, W, SR, DSP, DRH. The predictand is PanE.

To begin, I had to decide what language to use to code my MLP. I had decided to use Python as I think that Python offers a large variety of libraries that I think would be useful and helpful when coding the MLP. The libraries in talk are Pandas and NumPy.
Pandas is a Python library that is used for data manipulation and analysis. Pandas provides powerful data structures and tools for working with structured data, including data reading and cleaning. As the dataset was provided in an Excel spreadsheet, Pandas was a perfect library to deal with this.
Another library mentioned is NumPy. NumPy is a Python library used for numerical computing. As I am coding a MLP, a lot of mathematical functions and calculations would be needed. Therefore, NumPy is a suited library to use for this case.
In summary, Python was used so that Pandas and NumPy was available to use. This allowed the process of coding the MLP to be a lot easier.

The first step in preparing the dataset for training was to read in the Excel file dataset using the pandas library. The date column was dropped from the dataframe as it is not a predictor variable in the model and is not relevant for this MLP.

```
# read the dataset into a dataframe
df = pd.read_excel(r'C:\Users\Micha\PycharmProjects\COB107CW\DataSet.xlsx')
# drop the date column
df = df.drop(['Date'], axis=1)
```

Next, string outliers were removed by converting all columns to numerical values and dropping any rows with missing values.

```
# eliminate string outliers
df[df.columns] = df[df.columns].apply(pd.to_numeric, errors='coerce')
df = df.dropna()
```

What this code does is that whenever a value cannot be converted to a numerical value, the value will then be replaced with, ‘NaN’, in which I can drop the rows with any fields with ‘NaN’.

With the data now cleansed from string outliers, numerical outliers must be eliminated. Values 3 standard deviations away from the mean were considered anomalies that could negatively impact the performance of the MLP model. The method used to remove the outliers was using the z-score

method, which involves scaling the data and removing data points that are more than three standard deviations away from the mean.

```
# eliminate outliers 3 s.d. away from the mean
df = df[(np.abs(stats.zscore(df[df.columns])) < 3).all(axis=1)]
```

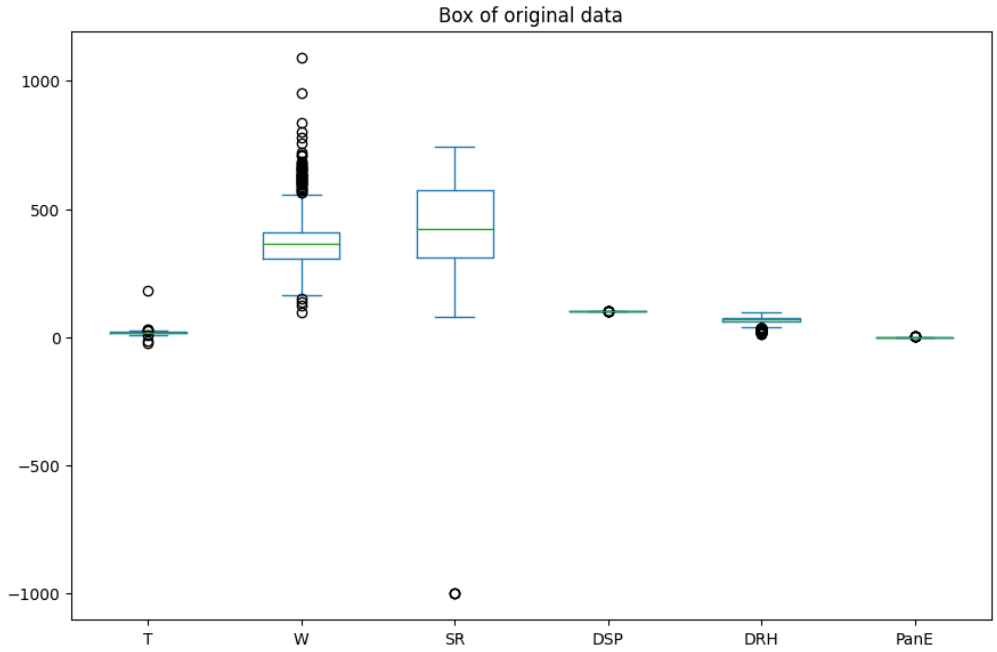

Figure 1. Box plot of data before removing the outliers

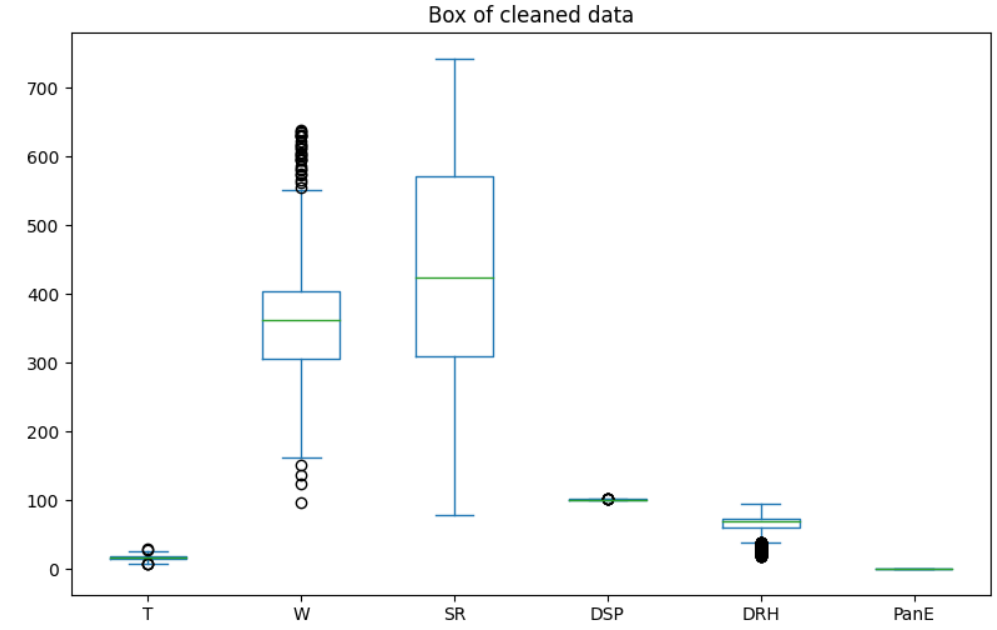

Figure 2. Box Plot of the data after removing outliers

As shown in the graphs above, if we had not removed the outliers from the dataset, there would be a plethora of data points which would skew the performance of the MLP. While after removing the outliers, the dataset has a much more desirable trend.

The dataset was then mixed randomly using a random seed to ensure that the MLP would not be generalised to a certain order of data.

```
# mix the dataset randomly
df_mixed = df.sample(frac=1, random_state=random.randint(1, 100))
```

Next, to split the data into subsets for training, validation, and testing, the dataset was divided into two subsets, 80% for training and validation, and 20% for testing. The subset containing the training and validation data were standardised by scaling the data to a range of 0.1 to 0.9, using the minimum and maximum of each column. The testing data will be standardised on its own, using its own minimum and maximum. This is so the testing data is not entirely the same as the training and validation data.

```
# calculate the number of rows for each subset
num_rows = len(df_mixed)
n_data_60 = int(0.6 * num_rows)
n_data_20 = int(0.2 * num_rows)

# split dataframe into two subsets, 80:20
df_train_val = df_mixed[:n_data_60 + n_data_20]
df_test_20 = df_mixed[n_data_60 + n_data_20:]

# standardise the dataset carrying training and validation data
df_train_val_std = 0.8 * ((df_train_val - df_train_val.min()) / (df_train_val.max() - df_train_val.min())) + 0.1

# split 80 into 60:20
df_train_60_std = df_train_val_std[:n_data_60]
df_val_20_std = df_train_val_std[n_data_60:]

# standardised test subset with own min, max
df_test_20_std = 0.8 * ((df_test_20 - df_test_20.min()) / (df_test_20.max() - df_test_20.min())) + 0.1
```

After this, the subsets are now formed and are saved to separate sheets in an Excel file named "CleanedData.xlsx". The training subset consists of 60% of the data, the validation subset consists of 20% of the data, and the testing subset consists of the remaining 20% of the data.

```
# write the subsets to separate sheets in an Excel file
with pd.ExcelWriter('CleanedData.xlsx') as writer:
    df_train_60_std.to_excel(writer, sheet_name='Training Subset', index=False)
    df_val_20_std.to_excel(writer, sheet_name='Validation Subset', index=False)
    df_test_20_std.to_excel(writer, sheet_name='Testing Subset', index=False)
```

Subset	Number of rows	Percentage of data
Training Subset	850	60%
Validation Subset	284	20%
Testing Subset	284	20%

Table 1. Summary of the subsets

Limitations

The limitations presented are the assumptions during data pre-processing. Firstly, I am dropping the 'Date' column. This is only viable to drop as the column is not relevant to the MLP and is not a predictor. However, if this MLP was presented with a dataset which required the date to be a predictor then this would remove that.

2. Implementation of the MLP algorithm

There are three Python files for the whole system:

File name	Use
dataClean.py	This file pre-processes the data, and writes the appropriate subsets to an excel spreadsheet.
mlp.py	This file holds the MLP itself. The training, validation and evaluation is done here.
main.py	This file gathers inputs from the user and runs the functions of dataClean and mlp.

Table 2. Summary of the files

In dataClean.py, there is one function clean_data. This function does all the data pre-processing as discussed in the first section. The function is called by the main file every time it is ran, as we want to randomise what data is being used.

```
from dataClean import clean_data

# clean the dataset
clean_data()
```

I have implemented the MLP algorithm using OOP in Python. Despite Python not being a strictly natural OOP language, Python still offers a way to implement OOP. I figured that the MLP should be able to store its weights and biases so that they can be easily accessible to be evaluated or used whenever needed in the code. The MLP has a single hidden layer and gathers inputs from the user to determine what the number of hidden nodes are in the layer, and how many epochs the MLP should train for.

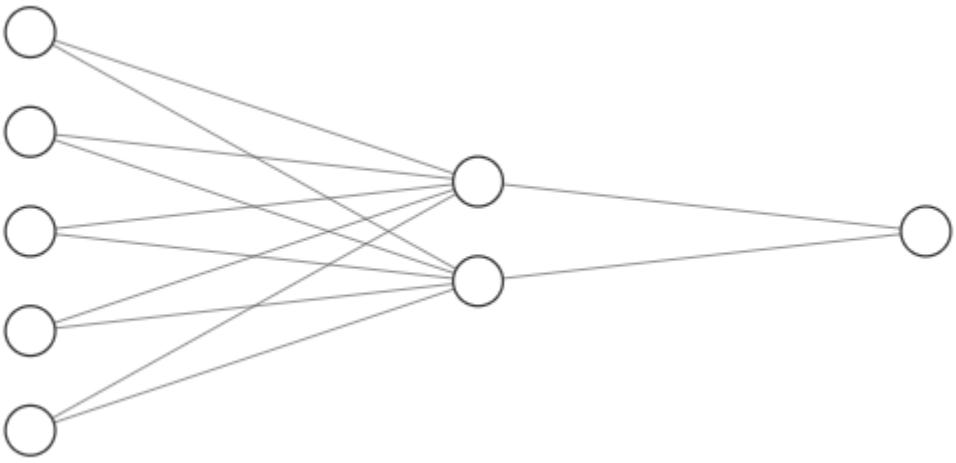

Figure 3. Visualisation of the MLP.
Note that the number of nodes in the hidden layer is determined by the user. This is just an example.

The user input resides in main.py, and passes the necessary data to the MLP class to make the MLP object.

```
# get the training, validation and testing subset
df = pd.read_excel(r'C:\Users\Micha\PycharmProjects\COB107CW\CleanedData.xlsx', sheet_name=None)
df_train = df.get('Training Subset')
df_valid = df.get('Validation Subset')
df_test = df.get('Testing Subset')

# get the number of predictors, assuming the last column of the dataset is the predictand
n_predictors = len(df_train.iloc[[0]].values[0]) - 1
# allow the user to enter the number of hidden nodes in the MLP
# allow the user to enter the number of epochs to train
n_nodes = int(input("Enter the number of hidden nodes: "))
epochs = int(input("Enter the number of epochs: "))

# create MLP object and give it the training and validation subsets, along with the number of hidden
# nodes and predictors
mlp1 = MLP(df_train, df_valid, n_nodes, n_predictors)
# train the MLP with user given epochs and get the training, validation and testing errors
trn_vld_errors = mlp1.train(epochs)
test_error = mlp1.test(df_test)
```

In the MLP class, there are five methods:

Name	Use
__init__	Initialise the MLP with the dataset and user given values.
forward_pass	Calculate the output of the MLP. i.e. forward pass
backpass	Update the weights and biases of the MLP. i.e. backward pass
train	Train the MLP.
test	Test the MLP against the test subset.

Table 3. Summary of functions in the MLP class.

__init__

```
class MLP:
    p = 0.1

    # initialise the MLP
    def __init__(self, df_train, df_valid, hidden_nodes, n_predictors):
        # store the datasets and number of hidden nodes and predictors
        self.df_train = df_train
        self.df_valid = df_valid
        self.hidden_nodes = hidden_nodes
        self.n_predictors = n_predictors

        # randomise the weights
        self.in_weights = np.random.uniform(-2/n_predictors, 2/n_predictors, (n_predictors, hidden_nodes))
        self.in_weights = np.round(self.in_weights, 2)
        self.out_weights = np.random.uniform(-2/n_predictors, 2/n_predictors, hidden_nodes)
        self.out_weights = np.round(self.out_weights, 2)

        # randomise the biases
        self.biases = np.random.uniform(-2/n_predictors, 2/n_predictors, hidden_nodes + 1)
        self.biases = np.round(self.biases, 2)
        # set the bias of the output node
        self.biases[-1] = round(random.uniform(-2/hidden_nodes, 2/hidden_nodes), 2)
```

This is the initialisation method of the MLP class. We store the datasets and the number of hidden nodes and predictors as variables. To store the weights, I had decided to store them as arrays and decided to split up the weights in two separate parts, one part for the weights incoming to the hidden layer, and the second part for the weights outgoing to the output node. I stored them this way because it was the most visual and logical to me. The index of the 2D array determines what incoming connection we are looking at, i.e. if the index was [2,3] then that means that we are looking at the connection between predictor 3 and hidden node 4. As I only implemented one output node, a normal array would be necessary as the indexing would represent what hidden node the connection is coming from. This is a similar reasoning to the biases being stored in an array. The indexing of the biases array determines what node we are looking at, i.e. index 1 would mean we are looking at hidden node 2 or the output node , depending on how many hidden nodes there are.
I initialised the weights and biases to be random values between -2/predictors and 2/predictors. However, the bias for the output node is a random value between -2/hidden_nodes and 2/hidden_nodes.

forward_pass

```
def forward_pass(self, values):
    # values - predictor values
    s_value = 0
    u_values = []
    u_output = 0
    # for each hidden node calculate it's S value and U value
    # calculate the output at the end
    for i in range(self.hidden_nodes):
        # for each predictor
        for j in range(self.n_predictors):
            # calculate S value, predictor * weight
            s_value += values[j] * self.in_weights[j][i]
        s_value += self.biases[i]
        # calculate U value for each hidden node using the sigmoid activation function
        u_value = 1 / (1 + np.exp(-s_value))
        u_values.append(u_value)
        u_output += u_value * self.out_weights[i]

    output = 1 / (1 + np.exp(-u_output))
    # return output and U values as they are needed for updating the weights and biases
    return output, u_values
```

The forward_pass function is essentially the forward pass of the MLP. It calculates the output from the output node and the activations for every hidden node, which uses the sigmoid function to do this. This function is dynamic because of the user input that determines the number of hidden nodes, therefore this is not hard coded. The value returned from this function is the output and the activation values, as they will be needed for the backward pass.

backpass

```
def backpass(self, values, output_values):
    momentum = 0.9
    # calculate delta values for nodes
    output = output_values[0]
    output_delta = (values[5] - output) * (output * (1 - output))

    # for each hidden node, calculate the delta values and update the weights and biases
    for i in range(self.hidden_nodes):
        delta = (self.out_weights[i] * output_delta) * (output_values[1][i] * (1 - output_values[1][i]))
        # for each predictor, update each weight it is connected to
        for j in range(self.n_predictors):
            temp = self.in_weights[j][i]
            self.in_weights[j][i] += MLP.p * delta * values[j]
            weight_change = self.in_weights[j][i] - temp
            self.in_weights[j][i] += momentum * weight_change

        # update the out-going weights of the hidden node
        temp = self.out_weights[i]
        self.out_weights[i] += MLP.p * output_delta * output_values[1][i]
        weight_change = self.out_weights[i] - temp
        self.out_weights[i] += momentum * weight_change
        # update biases
        self.biases[i] += MLP.p * delta

    # update the output node bias accordingly
    self.biases[-1] += MLP.p * output_delta
```

This function is essentially where the backpropagtion algorithm is implemented. In the function the weights and biases are updated. Along with this, the improvement of momentum is included. We first initialise the momentum and output delta, so that they can be used when updating. output_values[1][i] is an array which stores the activations of each node. Next, we go through each hidden node and its predictors that are attached to it and update the weights accordingly, using the delta value that was calculated for the node and the learning parameter. As mentioned before I have included momentum. For each weight I store the old weight in a temporary variable, then calculate the new weight, and add the appropriate calculations to the weight. This is done to the biases as well.

train

```
def train(self, epochs):
    train_errors = []
    valid_errors = []
    train_error = 0
    valid_error = 0
    train_len = len(self.df_train)
    valid_len = len(self.df_valid)
    # for each epoch
    for i in range(epochs):
        # for each row in the training subset
        for j in range(train_len):
            # get predictors and calculate values for the hidden nodes
            t_values = self.df_train.iloc[[j]].values[0]
            train_values = self.calc_output(t_values)
            output = train_values[0]
            # update weights and biases
            self.update_wb(t_values, train_values)
            train_error += (t_values[5] - output)**2
        # calculate training error using RMSE
        train_error = np.sqrt(train_error / train_len)
        train_errors.append(train_error)

        # for each row in the validation subset
        for k in range(valid_len):
            # get predictors and calculate values for the hidden nodes
            v_values = self.df_valid.iloc[[k]].values[0]
            valid_values = self.calc_output(v_values)
            v_output = valid_values[0]
            valid_error += (v_values[5] - v_output)**2
        # calculate validation error using RMSE
        valid_error = np.sqrt(valid_error / valid_len)
        valid_errors.append(valid_error)

        # check if the validation error has been improved compared to the previous 2 epochs
        if (len(valid_errors) > 1) and (valid_error >= max(valid_errors[-2:])):
            # check if model has improved on the validation subset for more than 10 epochs
            if i - np.argmax(valid_errors) > 10:
                # end training loop
                break
```

This function is the main training algorithm and is where both the functions, '*forward_pass'* and '*backpass*' is called. First we initialise some variables and lists. Then we loop through the number of epochs and loop through the training subset. For each row in the subset, we pass the values to the *'forward_pass'* function where we perform the forward pass. On return, we get the output and calculate the error, which added to a running total. At the same time we update the weights and biases. After, the overall error is calculated using the RMSE(Root mean square error) function. I used this because it is the most standard way to measure the error of a model. It is then added to the list of training errors, later to be used to be represented on a graph. These steps are the same for the validation error, except the updating of the weights and biases.

The code also implements early stopping based on validation error improvement over a number of epochs. The nested if statement first checks whether the length of the list of validation errors is greater than 1, and then checks if the most recent validation error is greater than the previous two validation errors. i.e. has the validation error improved over the last two epochs. If the validation error hasn't improved, then we do another check. The second if statement checks if the number of

epochs since the last improvement is greater than 10. If this condition is true then that means the model has not improved on the validation subset for more than 10 epoch. Therefore, if this is true then we shall break out of the training loop.

These if statements helps to prevent the model from overfitting on the training subset and ensures that the model generalises well on new data. If the validation error does not improve for more than 10 epochs, then it is likely that the model has reached a local minima and is not learning anything new. In this case it is better to stop the training loop and use the current set of weights and biases as the final model parameters.

test

```
def test(self, df_test):
    # initialise variables
    test_len = len(df_test)
    test_error = 0
    test_errors = []
    predicted_outputs = []
    real_values = []

    # for each row in the test subset calculate the errors
    for i in range(test_len):
        t_values = df_test.iloc[[i]].values[0]
        # calculate the predicted predictand
        output = self.calc_output(t_values)[0]
        predicted_outputs.append(output)
        # calculate test error
        test_error += (t_values[5] - output)**2
        real_values.append(t_values[5])

    # calculate RMSE
    test_error = np.sqrt(test_error / test_len)
    test_errors.append(test_error)
```

This function is responsible for evaluating the performance of the MLP. Given a testing subset, loop for each row, get the predicted predictand, and calculate the total test error while adding the predictand to the expected value list. After calculate the overall test error of the testing subset and add that to the test errors list. From this, we can use the test errors and real values to show them on a graph which will be discussed in section 4.

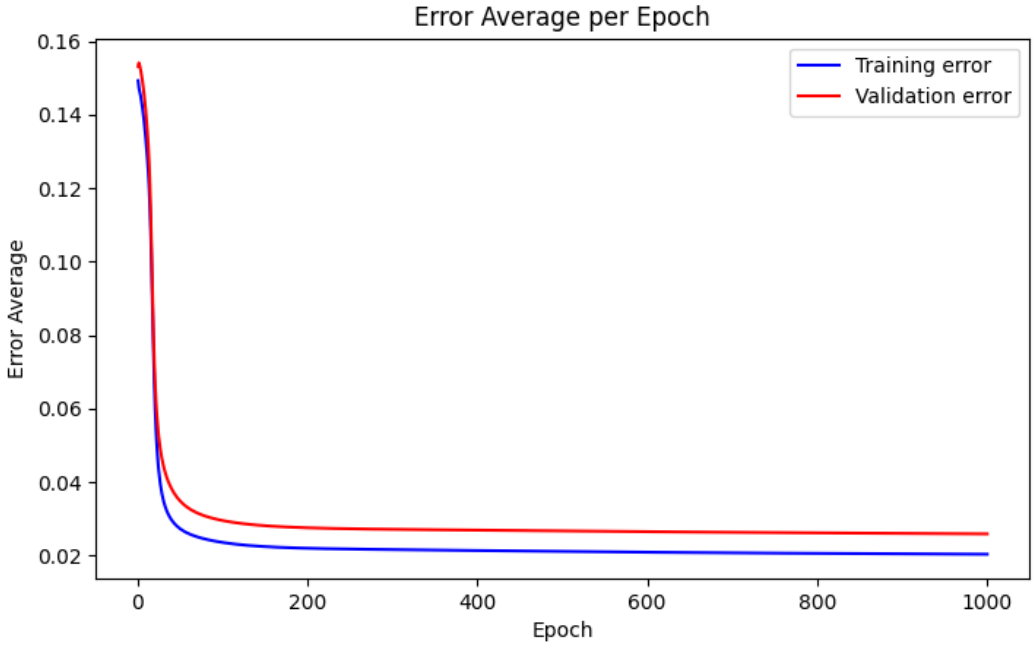

Figure 4. MLP learning curve with 3 hidden nodes over 1000 epochs.

Limitations

The limitations presented in the MLP algorithm are the assumptions in main.py. Firstly, there is the assumption that the last column in the given dataset is the predictand. However, if given a dataset which did not have the last column as the predictand then this would not work.

3. Training and Network Selection

The MLP was trained on the given dataset explained in the first section, and is given the necessary data when the MLP is first initialised.

```
# get the training, validation and testing subset
df = pd.read_excel(r'CleanedData.xlsx', sheet_name=None)
df_train = df.get('Training Subset')
df_valid = df.get('Validation Subset')
df_test = df.get('Testing Subset')

# get the number of predictors, assuming the last column of the dataset is the predictand
n_predictors = len(df_train.iloc[[0]].values[0]) - 1
# allow the user to enter the number of hidden nodes in the MLP
# allow the user to enter the number of epochs to train
n_nodes = int(input("Enter the number of hidden nodes: "))
epochs = int(input("Enter the number of epochs: "))

# create MLP object and give it the training and validation subsets, along with the number of hidden
# nodes and predictors
mlp1 = MLP(df_train, df_valid, n_nodes, n_predictors)
# train the MLP with user given epochs and get the training, validation and testing errors
trn_vld_errors = mlp1.train(epochs)
test_error = mlp1.test(df_test)
```

We see that we get the required subsets from the spreadsheet and we pass them as arguments to their corresponding functions. The training and validation subset is passed into the MLP function to create an instance of a MLP object. Along with the subsets, the program will ask the user how many hidden nodes and epochs they wish to use. This is because with the way I had implemented the MLP, it allows for the flexibility of differing epochs and hidden nodes. As mentioned before, the only limiting factor here is that I assume that the last column of the dataset is the predictand. This code should be easily modifiable for other datasets, differing number of inputs and number of hidden nodes.

The training is done when the '*train*' function is called given the number of epochs. In the '*train*' function, I had implemented a simple line graph to show the training curve of the MLP.

```
# plot the errors of the training and validation subsets
plt.plot(range(1, len(train_errors) + 1), train_errors, 'b-')
plt.plot(range(1, len(valid_errors) + 1), valid_errors, 'r-')
plt.xlabel("Epoch")
plt.ylabel("Error Average")
plt.title("Error Average per Epoch")
plt.legend(["Training error", "Validation error"])
plt.show()
return train_error, valid_error
```

With this I can see how the MLP performs, and can compare the performance between different combinations of hidden nodes and epochs. As well as this I return the final train error and valid error recorded.

Network Architectures

As the dataset used to train the MLP has 5 predictors I tried architectures with 2 to 10 hidden nodes to see which one would perform the best. I will show the performance of the MLP with the epochs as 1000 and 2000. Keep in mind that the MLP has one hidden layer, RMSE is the error function and the sigmoid activation function is used. Furthermore, the weights and biases are being randomised as well. The dataset should be kept the same when finding out which MLP performs the best. This is to ensure that the performance of the models is comparable.

To choose the most optimal MLP network for the given dataset, we would choose the MLP with the lowest validation error. This is because this shows how well the MLP will generalise to completely new data. The model with the lowest validation error is the one that has performed the best on average across the epochs.

Epochs as 1000

No. hidden nodes	Training Error .9f	Validation Error .9f
2	0.020676397	0.020616631
3	0.021092626	0.021045956
4	0.021936501	0.021985741
5	0.020569847	0.020437237
6	0.020541051	0.020479838
7	0.021012622	0.021437734
8	0.021557169	0.021921950
9	0.021039359	0.021053163
10	0.020568698	0.020552012

Table 4. MLP training and validation errors with epochs as 1000

Epochs as 2000

No. hidden nodes	Training Error .9f	Validation Error .9f
2	0.020612131	0.020609655
3	0.019198507	0.019131675
4	0.020407180	0.020421726
5	0.019194984	0.019470214
6	0.019699105	0.020146186
7	0.019773307	0.019648303
8	0.019119136	0.019645204
9	0.020604095	0.021051371
10	0.020782796	0.021243213

Table 5. MLP training and validation errors with epochs as 2000, with different subset from epochs as 1000

From these results we can observe that from Table 4, the best validation error is from when the number of hidden nodes is 5, with 6, 10, and 2 close behind. When looking closely at these results, we can see that the validation error is actually less than the training error for these results mentioned. When the number of hidden nodes is 5, we can say that it is the most optimal MLP for this given dataset at 1000 epochs. From Table 5, we can observe that the lowest validation error is when the number of nodes is 3, with 5, 7, and 8 close behind. Comparing these MLPs mentioned, the most optimal MLP is when the number of hidden nodes is 3. It is by far the most optimal as the difference between it and the next most optimal has a difference of ~0.0003.

Due to the epochs being twice as large in Table 5 as Table 4, we will take the MLP from Table 5, as it by far performs better than the rest.

To conclude, when the number of nodes is 3 the validation error is at its lowest. This means that the MLP has generalised well to new data that it is not trained on. Therefore, the MLP with the number of nodes as 3 performs the best.

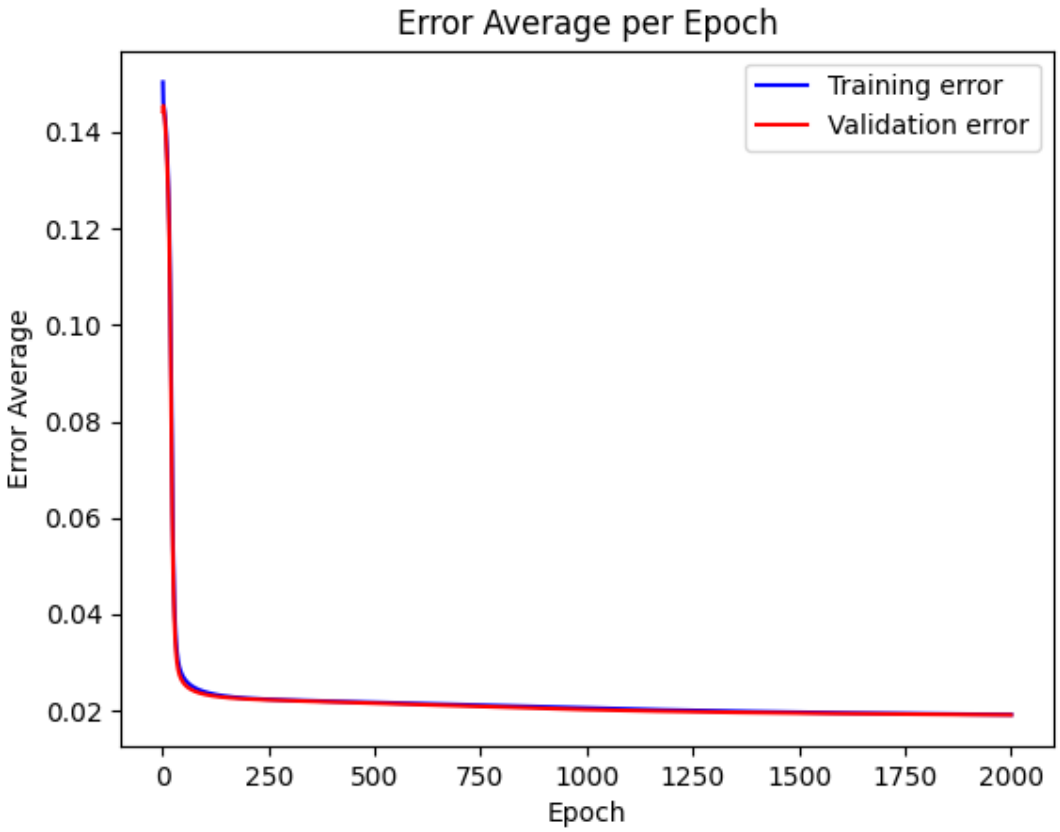

Figure 5. MLP learning curve when the number of nodes is 3.

4. Evaluation of Final Model

To evaluate the final model, I had used to '*test*' function in the MLP class. The '*test*' function evaluates the model against the testing subset. It does this by checking the predicted predictand values(the PanE value) against the real 'PanE' values, and shows this on a graph.

```
def test(self, df_test):
    # initialise variables
    test_len = len(df_test)
    test_error = 0
    test_errors = []
    predicted_outputs = []
    real_values = []

    # for each row in the test subset calculate the errors
    for i in range(test_len):
        t_values = df_test.iloc[[i]].values[0]
        # calculate the predicted predictand
        output = self.calc_output(t_values)[0]
        predicted_outputs.append(output)
        # calculate test error
        test_error += (t_values[5] - output)**2
        real_values.append(t_values[5])

    # calculate RMSE
    test_error = np.sqrt(test_error / test_len)
    test_errors.append(test_error)

    # plot the errors of the testing subset
    plt.plot(range(1, len(real_values) + 1), real_values, 'b-')
    plt.plot(range(1, len(predicted_outputs) + 1), predicted_outputs, 'y-')
    plt.xlabel("Test Number")
    plt.ylabel("PanE Value")
    plt.title("Error Average per Test")
    plt.legend(["Expected output", "Predicted output"])
    plt.show()
    return test_error
```

In this function, we apply the forward pass algorithm to each row of the dataset and record its output. With the list of the predicted output, we can show this on a graph which compares the precited values with the real values of the predictand. As we have chose the most optimal MLP as when the number of hidden nodes is 3, we can apply this '*test*' function to the MLP to evaluate the MLP

When running this '*test*' function on the MLP.

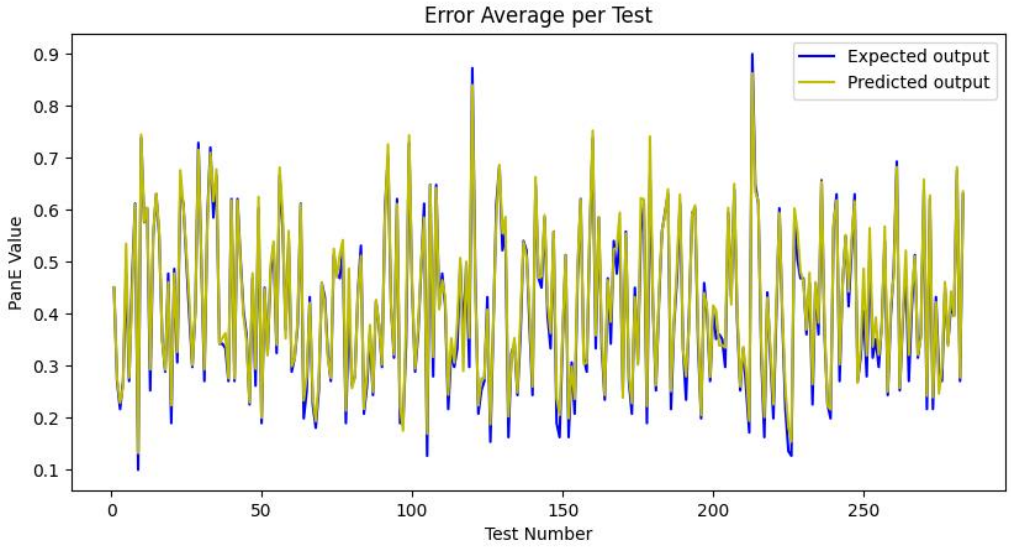

Figure 6. Graph showing the predicted output against the expected output

For this graph, the test error was 0.025324247 to 9 decimal places and the validation error was 0.022693084 to 9 decimal places. We can see that the MLP tends to the general shape of the expected outputs, however, the MLP struggles when the PanE values are on the ends of the PanE spectrum. We can especially see this when the PanE values are at ~0.9.

We will now evaluate the performance without the improvement of momentum on the same subsets. We will do this by looking at the test errors of them all. We will train the MLP with the epoch set as 1000.

No. hidden nodes	With Momentum	Without Momentum
2	0.025194576	0.024817258
3	0.025117288	0.025755831
4	0.025389245	0.025696632
5	0.024697517	0.026342539
6	0.025356447	0.026537656
7	0.024623715	0.024798501
8	0.025118568	0.026851658
9	0.024749868	0.026437454
10	0.024774642	0.026140402

Table 6. MLP test error with and without the improvement of momentum

The lower the test error, the better the MLP performance. As we can see from the results, on most occasions, the momentum plays a large role in reducing the test error. There are only 2 instances where the test error is greater without momentum than with momentum, this is when the number of nodes is 2 and 7. However, we can observe that the differences when the nodes are 2 and 7 are quite minute. We can also acknowledge that when the number of hidden nodes increase the difference between the test errors increase. To conclude momentum is a good way to improve its performance.

Here are some graphs that show the difference.

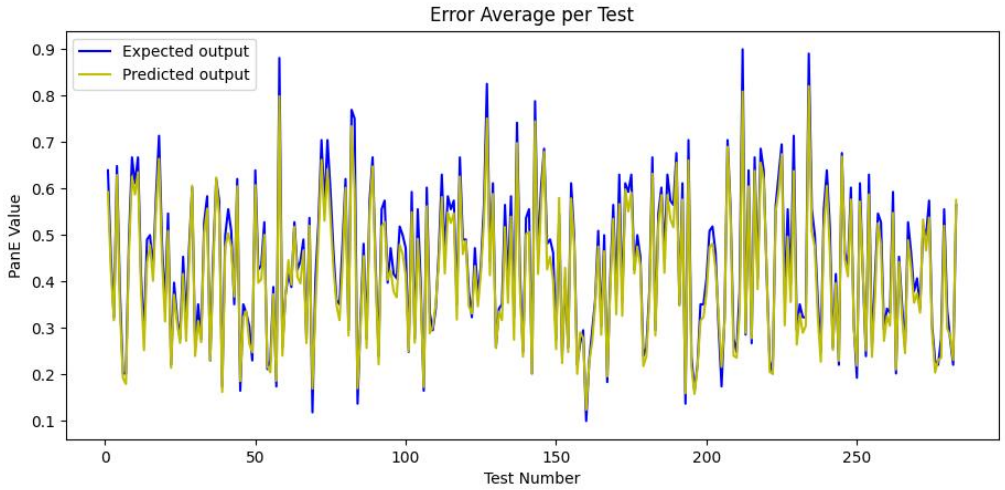

Figure 7. MLP test error performance with momentum.

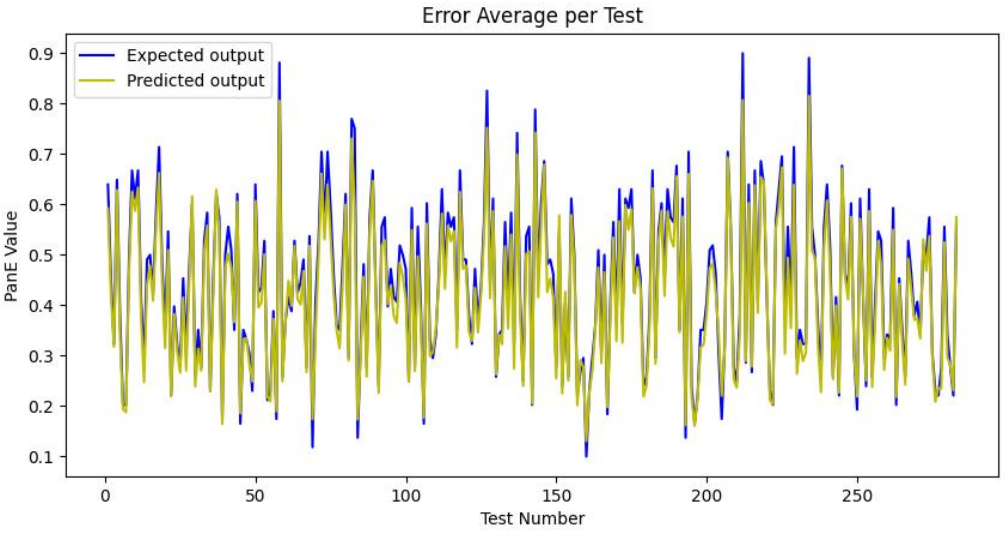

Figure 8. MLP test error performance without momentum.

5. Comparison with Another Data Driven Model or Baseline

To compare with another data driven model, I will be using the baseline as a linear regression model. Firstly, we can compare my MLP with the baseline by comparing RMSE(root mean square error) of the two models. The RMSE of the baseline was 0.029162458 to 9 decimal places, however, the RMSE of my MLP was 0.025088065, keep in mind that the MLP used has 3 hidden nodes. From these values you can see that the RMSE of the MLP is smaller than the linear regression model, which shows that the MLP performs far greater than the liner regression model.

To display the performance of the MLP we will show the predicted outputs against the real outputs, with a regression line.

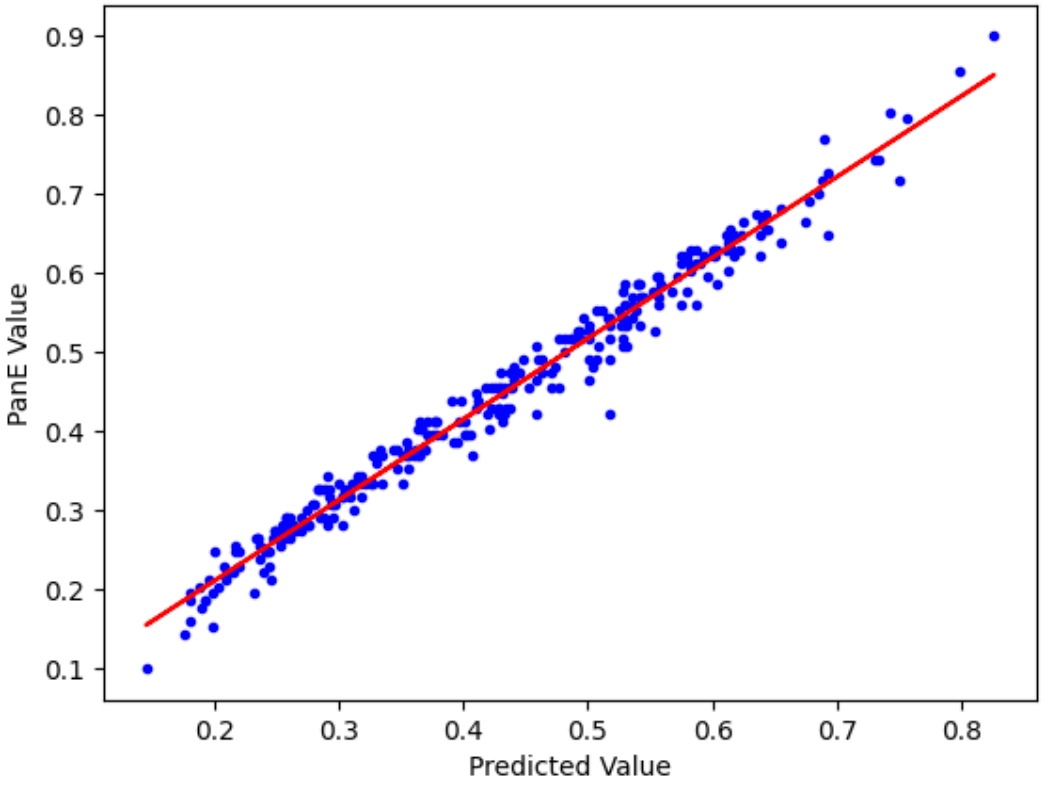

Figure 9. Regression line of predicted value against PanE value

What this graph shows is the relationship between the predicted values and the real values. We can observe that the predicted values are close to the regression line, meaning that the error is small. The regression line is overtly positive, therefore, this means that as the predicted values increase so do the actual values and so we can predict the PanE value with this line.

To conclude, the MLP performs well against the regression model as its RMSE is far less than the regression model. As well as this, we further expand on this by looking at the MLPs performance against the real values of the dataset. The predicted values follow a linear shape and therefore can approximately predict PanE values.

Program Listing

dataClean.py

```
import numpy as np
import pandas as pd
import random
from scipy import stats


def clean_data():
    # read the dataset into a dataframe
    df = pd.read_excel(r'DataSet.xlsx')
    # drop the date column
    df = df.drop(['Date'], axis=1)

    # eliminate string outliers
    df[df.columns] = df[df.columns].apply(pd.to_numeric, errors='coerce')
    df = df.dropna()

    # eliminate outliers 3 s.d. away from the mean
    df = df[(np.abs(stats.zscore(df[df.columns])) < 3).all(axis=1)]
    # mix the dataset randomly
    df_mixed = df.sample(frac=1, random_state=random.randint(1, 100))

    # calculate the number of rows for each subset
    num_rows = len(df_mixed)
    n_data_60 = int(0.6 * num_rows)
    n_data_20 = int(0.2 * num_rows)

    # split dataframe into two subsets, 80:20
    df_train_val = df_mixed[:n_data_60 + n_data_20]
    df_test_20 = df_mixed[n_data_60 + n_data_20:]

    # standardise the dataset carrying training and validation data
    df_train_val_std = 0.8 * ((df_train_val - df_train_val.min()) /
(df_train_val.max() - df_train_val.min())) + 0.1

    # split 80 into 60:20
    df_train_60_std = df_train_val_std[:n_data_60]
    df_val_20_std = df_train_val_std[n_data_60:]

    # standardised test subset with own min, max
    df_test_20_std = 0.8 * ((df_test_20 - df_test_20.min()) /
(df_test_20.max() - df_test_20.min())) + 0.1

    # write the subsets to separate sheets in an Excel file
    with pd.ExcelWriter('CleanedData.xlsx') as writer:
        df_train_60_std.to_excel(writer, sheet_name='Training Subset',
index=False)
        df_val_20_std.to_excel(writer, sheet_name='Validation Subset',
index=False)
        df_test_20_std.to_excel(writer, sheet_name='Testing Subset',
index=False)
```

main.py

```
import pandas as pd
from MLP import MLP
from dataClean import clean_data

# clean the dataset
clean_data()

# get the training, validation and testing subset
df = pd.read_excel(r'CleanedData.xlsx', sheet_name=None)
df_train = df.get('Training Subset')
df_valid = df.get('Validation Subset')
df_test = df.get('Testing Subset')

# get the number of predictors, assuming the last column of the dataset is
the predictand
n_predictors = len(df_train.iloc[[0]].values[0]) - 1

# allow the user to enter the number of hidden nodes in the MLP
# allow the user to enter the number of epochs to train
n_nodes = int(input("Enter the number of hidden nodes: "))
epochs = int(input("Enter the number of epochs: "))

# create MLP object and give it the training and validation subsets, along
with the number of hidden
# nodes and predictors
mlp1 = MLP(df_train, df_valid, n_nodes, n_predictors)
# train the MLP with user given epochs and get the training, validation and
testing errors
trn_vld_errors = mlp1.train(epochs)
test_error = mlp1.test(df_test)

# write the MLP performance to a text file
with open("MLPperformance.txt", "a") as f:
    f.write(f"Hidden nodes: {n_nodes},Epochs: {epochs}, Training error:
{trn_vld_errors[0]}, Validation error: {trn_vld_errors[1]},"
            f"Test error: {test_error}\n")
```

mlp.py

```
import numpy as np
import matplotlib.pyplot as plt
import random
from scipy import stats


class MLP:
    p = 0.1

    # initialise the MLP
    def __init__(self, df_train, df_valid, hidden_nodes, n_predictors):
        # store the datasets and number of hidden nodes and predictors
        self.df_train = df_train
        self.df_valid = df_valid
        self.hidden_nodes = hidden_nodes
        self.n_predictors = n_predictors

        # randomise the weights
        self.in_weights = np.random.uniform(-2/n_predictors,
2/n_predictors, (n_predictors, hidden_nodes))
        self.in_weights = np.round(self.in_weights, 2)
        self.out_weights = np.random.uniform(-2/n_predictors,
2/n_predictors, hidden_nodes)
        self.out_weights = np.round(self.out_weights, 2)

        # randomise the biases
        self.biases = np.random.uniform(-2/n_predictors, 2/n_predictors,
hidden_nodes + 1)
        self.biases = np.round(self.biases, 2)
        # set the bias of the output node
        self.biases[-1] = round(random.uniform(-2/hidden_nodes,
2/hidden_nodes), 2)

    def forward_pass(self, values):
        # values - predictor values
        s_value = 0
        u_values = []
        u_output = 0
        # for each hidden node calculate it's S value and U value
        # calculate the output at the end
        for i in range(self.hidden_nodes):
            # for each predictor
            for j in range(self.n_predictors):
                # calculate S value, predictor * weight
                s_value += values[j] * self.in_weights[j][i]
            s_value += self.biases[i]
            # calculate U value for each hidden node using the sigmoid
activation function
            u_value = 1 / (1 + np.exp(-s_value))
            u_values.append(u_value)
            u_output += u_value * self.out_weights[i]

        output = 1 / (1 + np.exp(-u_output))
        # return output and U values as they are needed for updating the
weights and biases
        return output, u_values

    def backpass(self, values, output_values):
        momentum = 0.9
        # calculate delta values for nodes
```

```
        output = output_values[0]
        output_delta = (values[5] - output) * (output * (1 - output))

        # for each hidden node, calculate the delta values and update the
weights and biases
        for i in range(self.hidden_nodes):
            delta = (self.out_weights[i] * output_delta) *
(output_values[1][i] * (1 - output_values[1][i]))
            # for each predictor, update each weight it is connected to
            for j in range(self.n_predictors):
                temp = self.in_weights[j][i]
                self.in_weights[j][i] += MLP.p * delta * values[j]
                weight_change = self.in_weights[j][i] - temp
                self.in_weights[j][i] += momentum * weight_change

            # update the out-going weights of the hidden node
            temp = self.out_weights[i]
            self.out_weights[i] += MLP.p * output_delta *
output_values[1][i]
            weight_change = self.out_weights[i] - temp
            self.out_weights[i] += momentum * weight_change
            # update biases
            self.biases[i] += MLP.p * delta

        # update the output node bias accordingly
        self.biases[-1] += MLP.p * output_delta

    def train(self, epochs):
        train_errors = []
        valid_errors = []
        train_error = 0
        valid_error = 0
        train_len = len(self.df_train)
        valid_len = len(self.df_valid)
        # for each epoch
        for i in range(epochs):
            # for each row in the training subset
            for j in range(train_len):
                # get predictors and calculate values for the hidden nodes
                t_values = self.df_train.iloc[[j]].values[0]
                train_values = self.forward_pass(t_values)
                output = train_values[0]
                # update weights and biases
                self.backpass(t_values, train_values)
                train_error += (t_values[5] - output)**2
            # calculate training error using RMSE
            train_error = np.sqrt(train_error / train_len)
            train_errors.append(train_error)

            # for each row in the validation subset
            for k in range(valid_len):
                # get predictors and calculate values for the hidden nodes
                v_values = self.df_valid.iloc[[k]].values[0]
                valid_values = self.forward_pass(v_values)
                v_output = valid_values[0]
                valid_error += (v_values[5] - v_output)**2
            # calculate validation error using RMSE
            valid_error = np.sqrt(valid_error / valid_len)
            valid_errors.append(valid_error)

            # check if the validation error has been improved compared to
```

```
the previous 2 epochs
            if (len(valid_errors) > 1) and (valid_error >=
max(valid_errors[-2:])):
                # check if model has improved on the validation subset for
more than 10 epochs
                if i - np.argmax(valid_errors) > 10:
                    # end training loop
                    break

        print("Training RMSE: ", train_error)
        print("Validation RMSE: ", valid_error)
        # plot the errors of the training and validation subsets
        plt.plot(range(1, len(train_errors) + 1), train_errors, 'b-')
        plt.plot(range(1, len(valid_errors) + 1), valid_errors, 'r-')
        plt.xlabel("Epoch")
        plt.ylabel("Error Average")
        plt.title("Error Average per Epoch")
        plt.legend(["Training error", "Validation error"])
        plt.show()
        return train_error, valid_error

    def test(self, df_test):
        # initialise variables
        test_len = len(df_test)
        test_error = 0
        test_errors = []
        predicted_outputs = []
        real_values = []

        # for each row in the test subset calculate the errors
        for i in range(test_len):
            t_values = df_test.iloc[[i]].values[0]
            # calculate the predicted predictand
            output = self.forward_pass(t_values)[0]
            predicted_outputs.append(output)
            # calculate test error
            test_error += (t_values[5] - output)**2
            real_values.append(t_values[5])

        # calculate RMSE
        test_error = np.sqrt(test_error / test_len)
        test_errors.append(test_error)

        # get slope and intercept of the predicted outputs and real values
        slope, intercept = stats.linregress(predicted_outputs, real_values)

        # function to calculate the slope and intercept of a point
        def slope_intercept(x):
            return slope * x + intercept

        # apply function to all predicted outputs
        mymodel = list(map(slope_intercept, predicted_outputs))

        # plot regression line and predicted outputs on graph
        plt.plot(predicted_outputs, real_values, 'b.')
        plt.plot(predicted_outputs, mymodel, color='red')
        plt.ylabel("PanE Value")
        plt.xlabel("Predicted Value")
        plt.show()

        print('Testing RMSE: ', test_error)
```

```
        # plot the errors of the testing subset
        plt.plot(range(1, len(real_values) + 1), real_values, 'b-')
        plt.plot(range(1, len(predicted_outputs) + 1), predicted_outputs,
'y-')
        plt.xlabel("Test Number")
        plt.ylabel("PanE Value")
        plt.title("Error Average per Test")
        plt.legend(["Expected output", "Predicted output"])
        plt.show()

        return test_error
```