Data Preprocessing

The dataset represents daily environmental factors recorded at a site in San Diego, USA, between 1/Jan/1987 and 31/Dec/1990. There are 9 columns with the following information:

- Date (mm/dd/yy format)
- 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 (the predictand) in cm/day

The predictors for the dataset are the first five columns (excluding the Date): T, W, SR, DSP, and DRH. The goal is to use these predictors to predict the value of PanE.

The first step was to check for invalid values in the dataset. This was done by sorting each column from largest to smallest to identify values that did not make sense. The temperature column had some negative values, but it was deemed plausible. The wind speed column had a letter in one of the entries, and the solar radiation column had a letter as well. The solar radiation column also had negative values, which is not possible. The DRH column had a missing value.

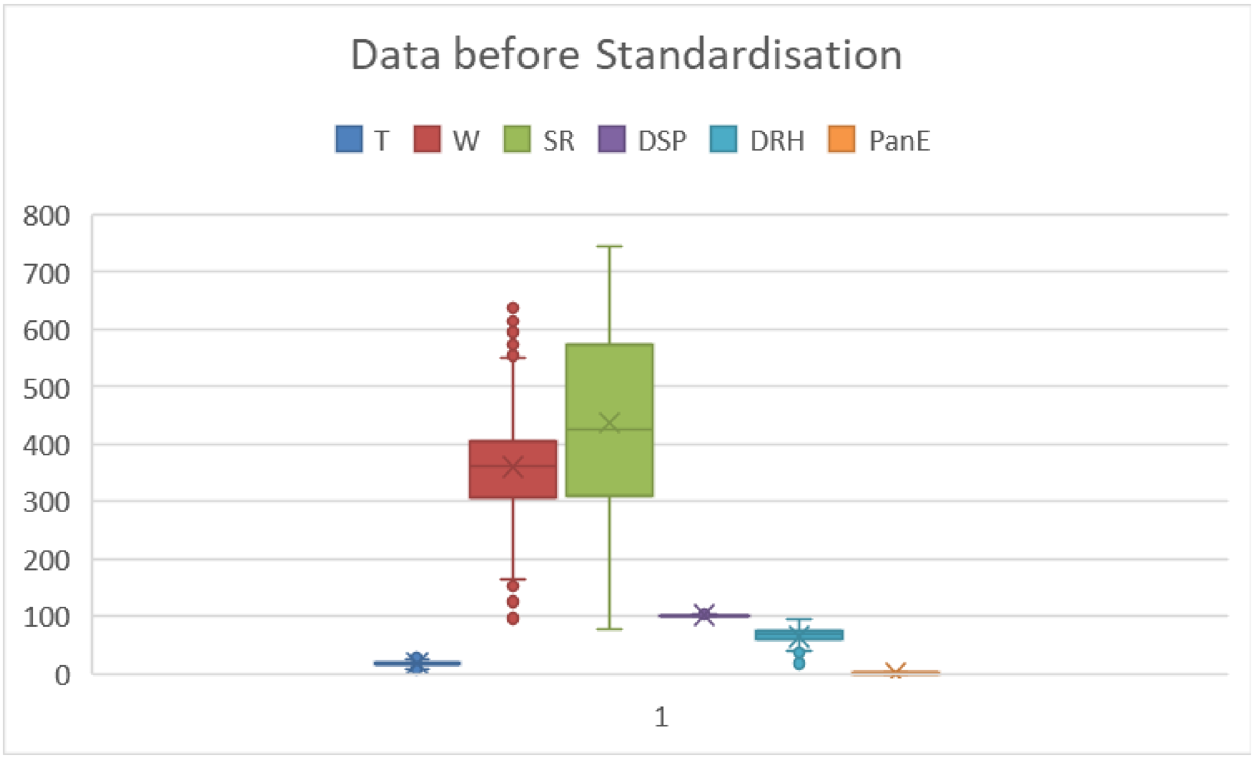
The next step was to check for extreme values. To do this, any data entry greater than three standard deviations from the mean for each field was removed. The average and standard deviation values were calculated using built-in formulas such as AVERAGE and STDEV.S. Cells were named so that they could be used in calculations. S_i values were then calculated for each column, and the distance from the mean in terms of standard deviation was compared. Values greater than three were eliminated.

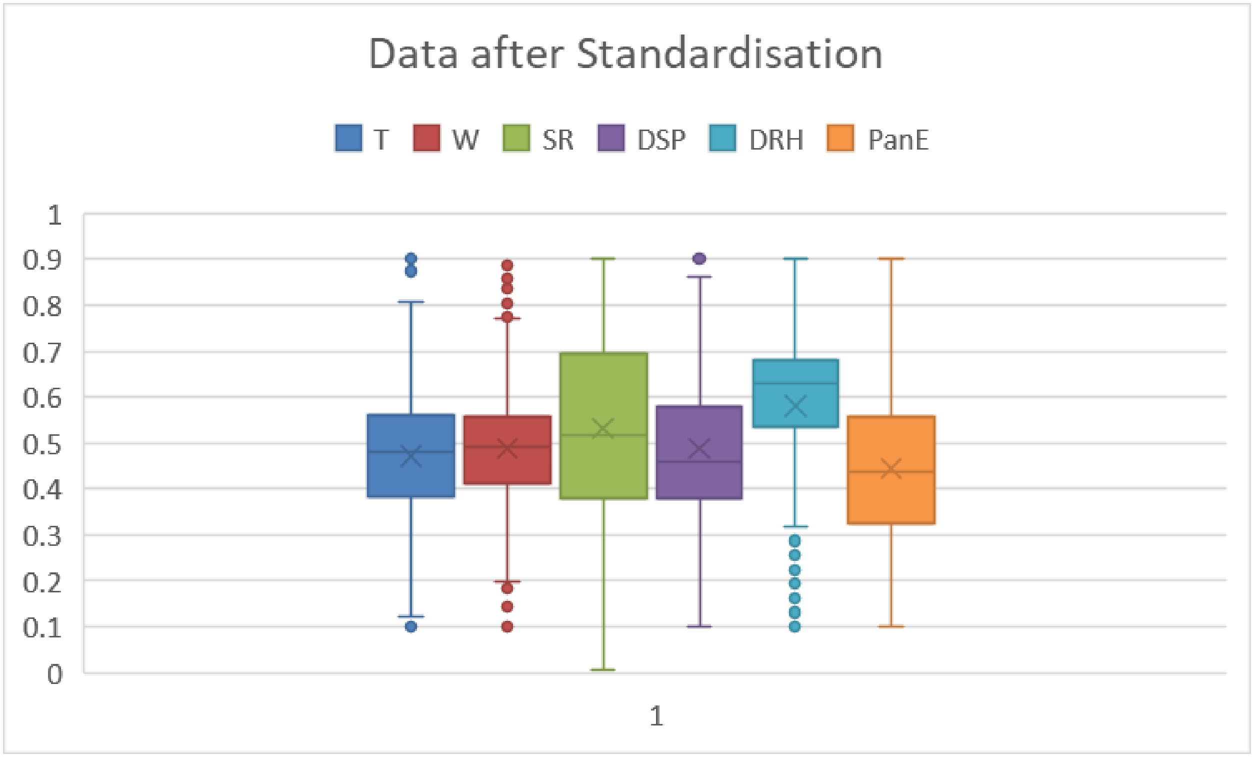
The next step was data standardisation, which involved applying a standardisation equation to each column. For example, to standardise the temperature, a new column was created, and the formula “=0.8*(B2-MINT)/(MAXT-MINT)+0.1” was applied. The graph before and after standardisation was inserted. Modelled after this equation:

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

Next, the predictors were identified. The date column was removed as it does not contribute to predicting the PanE value. Additionally, in the implementation of the algorithm, the PanE data was not used as an input node, but it was used to calculate the error - as it was the predictand.

Effects of standardisation:
It was important for the data to be standardised because numbers that are too large can cause rounding and overflow errors. By standardising the data, we ensure that it is within a manageable range and can be processed efficiently.





Implementation

Standard MLP

I chose to implement the Multi-Layer Perceptron (MLP) neural network using Python. The program reads input data from two Excel files: 'DataSet - Training.xlsx' for training and 'DataSet - Validation.xlsx' for validation. We train the MLP using a gradient descent algorithm with backpropagation to minimise the error between the predicted output and the actual output.

The program can be broken down into multiple sections:
Importing libraries:

1. The code imports necessary libraries: **`math`** , **`numpy`**, **`matplotlib.pyplot`**, and **`pandas`**.
2. Defining some helper functions, to make the code look cleaner:

- **`sigmoid(x)`**: A function which calculates the sigmoid activation for a given input 'x'.

```
def sigmoid(x):
    return 1/(1 + np.exp(-x))
```

- **`plot(training_errors, validation_errors)`**: A function which plots training and validation errors against the number of epochs.

```
def plot(training_errors, validation_errors):
    plt.plot(training_errors, label='Training Error')

    # Plot the validation errors every 100 epochs
    validation_x_values = list(range(0, len(training_errors), 100))
    plt.plot(validation_x_values, validation_errors, label='Validation Error',
             marker='o', linestyle='dashed', markersize=5)

    plt.xlabel('Epoch')
    plt.ylabel('Error')
    plt.title('Training and Validation Errors vs. Epoch')
    plt.legend()
    plt.show()
```

Defining the MLP class:

3. The class MLP consists of the following methods:

- **`__init__(self, no_hidden_nodes, learning_parameter)`:** The constructor initialises the weights, biases, and learning rate for the MLP.

```
class MLP:

    def __init__(self, no_hidden_nodes, learning_parameter):
        self.hidden_node_weights = np.random.uniform(-2/5, 2/5, size=(no_hidden_nodes, 5))
        self.hidden_node_biases = np.random.uniform(-2/5, 2/5, size=(no_hidden_nodes, 1))
        self.output_node_weights = np.random.uniform(-2/5, 2/5, size=no_hidden_nodes)
        self.output_node_bias = np.random.uniform(-2/5, 2/5)
        self.no_hidden_nodes = no_hidden_nodes
        self.learning_parameter = learning_parameter
```

- **`forward_pass(self, input_nodes)`**: This method calculates the weighted sums and activations for the hidden and output layers using the input nodes.

  We apply the sigmoid function on the hidden node weighted sums to get the hidden node outputs. We also calculate the output node weighted sum by getting the dot product of the output node weights(weights connecting from hidden nodes to output nodes) and the hidden node outputs, then adding the output node bias.

  We then calculate the output node activation by applying the sigmoid function to the output node weighted sum.

```
# compute weighted sums for every node
def forward_pass(self, input_nodes):
    # get weighted sums by calculating dot product of input nodes and their weights
    #  then adding the hidden node biases
    hidden_node_weighted_sums = (
        np.dot(self.hidden_node_weights, input_nodes) + self.hidden_node_biases
    )
    #apply sigmoid function
    hidden_node_outputs = sigmoid(hidden_node_weighted_sums)

    # dot product of output weights and activations
    output_node_weighted_sum = np.dot(self.output_node_weights,hidden_node_outputs) + self.output_node_bias
    #return output activation
    return hidden_node_outputs, sigmoid(output_node_weighted_sum)
```

- **`backwards_pass(self, inputs_and_target, hidden_node_outputs, predicted_PanE_value)`:** This method updates the weights and biases using the input nodes, hidden node outputs, and the predicted output.

  It calculates the deltas for the output and hidden nodes, and then updates the weights and biases using the learning parameter, node deltas, and node outputs.

```
# pass in input nodes aswell as pane target value
def backwards_pass(self, inputs_and_target, hidden_node_outputs, predicted_PanE_value):
    # calculate delta of the output node by getting the difference between the correct and predicted output
    output_node_sigmoid_derivative = predicted_PanE_value * (1 - predicted_PanE_value)
    output_node_delta = (inputs_and_target[-1][0] - predicted_PanE_value)*output_node_sigmoid_derivative
    # calculate delta of each hidden node by multiplying...
    # ...[the weight of its connection to the output node] with [the delta of the output node]...
    # ...and the derivative equation for the hidden layer
    hidden_node_deltas =[]
    for i in range(self.no_hidden_nodes):
        hidden_node_sigmoid_derivative = hidden_node_outputs[i] * (1 - hidden_node_outputs[i])
        hidden_node_deltas.append(self.output_node_weights[i] * output_node_delta * hidden_node_sigmoid_derivative)

    # ammend all weights by adding the product of (learning parameter, delta of node, output of the node) to the weight
    for i in range(self.no_hidden_nodes):
        # update hidden node weights and biases
        for j in range(5):
            self.hidden_node_weights[i][j] += self.learning_parameter * hidden_node_deltas[i] * inputs_and_target[j][0]

        self.hidden_node_biases[i][0] += self.learning_parameter * hidden_node_deltas[i]

        # next ammend output weights and output biases
        self.output_node_weights[i] += self.learning_parameter * output_node_delta * hidden_node_outputs[i]

    self.output_node_bias += self.learning_parameter * output_node_delta
```

- **`train(self, no_epochs, input_nodes)`**: This method trains the MLP for a specified number of epochs using input nodes. It performs the forward and backwards pass at each epoch and calculates the training and validation errors then plots them using the **`plot()`** function. In the current implementation it calculates the validation errors every 100 epochs.

```
def train(self, no_epochs, input_nodes):
    training_errors = []
    validation_errors = []

    for x in range(no_epochs):
        sum_of_squared_differences = 0
        for entry in input_nodes:
            # pass in a row of entry data from excel file excluding pane data
            hidden_node_outputs, output_node_activation = self.forward_pass(entry[: -1])
            # get sum of squared differences to use to calculate the rmse (the error value at the current epoch)
            sum_of_squared_differences += (output_node_activation - entry[-1])**2
            # call backwards pass and update weights
            self.backwards_pass(entry, hidden_node_outputs, output_node_activation)
        # calculate the error value at the current epoch
        error = math.sqrt(sum_of_squared_differences / len(input_nodes))
        training_errors.append(error)

        # at regular intervals run forward_pass to validate the model
        if not x % 100:
            sum_of_squared_differences = 0
            for entry in validation_input_nodes:
                # pass in a row of entry data from excel file excluding pane data
                hidden_node_outputs, output_node_activation = self.forward_pass(entry[: -1])
                # get sum of squared differences to use to calculate the rmse (the error value at the current epoch)
                sum_of_squared_differences += (output_node_activation - entry[-1])**2
            # calculate the error value at the current epoch
            error = math.sqrt(sum_of_squared_differences / len(validation_input_nodes))
            validation_errors.append(error)

    plot(training_errors, validation_errors)
```

Main execution:

- In the main execution, an instance of the MLP class is created with **`x`** hidden nodes (4 in the example below) and a learning parameter of 0.05. The training and validation datasets are read from Excel files:

```python
if __name__ == "__main__":
    mlp = MLP(4, 0.05)
    data = pd.read_excel('DataSet - Training.xlsx')

    validation_data = pd.read_excel('DataSet - Validation.xlsx')
    validation_input_nodes = []
    for index, row in validation_data.iterrows():
        validation_input_nodes.append([[row['T']], [row['W']], [row['SR']], [row['DSP']], [row['DRH']], [row['PanE']]])
        input_nodes = []
        for index, row in data.iterrows():
            input_nodes.append([[row['T']], [row['W']], [row['SR']], [row['DSP']], [row['DRH']], [row['PanE']]])
        mlp.train(200, input_nodes)
```

Annealing

The second MLP model modifies the initial MLP, by incorporating simulated annealing. The aim is to adjust the learning rate over time, allowing the model to explore the solution space effectively and avoid getting stuck in local minima:

Summary of implementation changes:

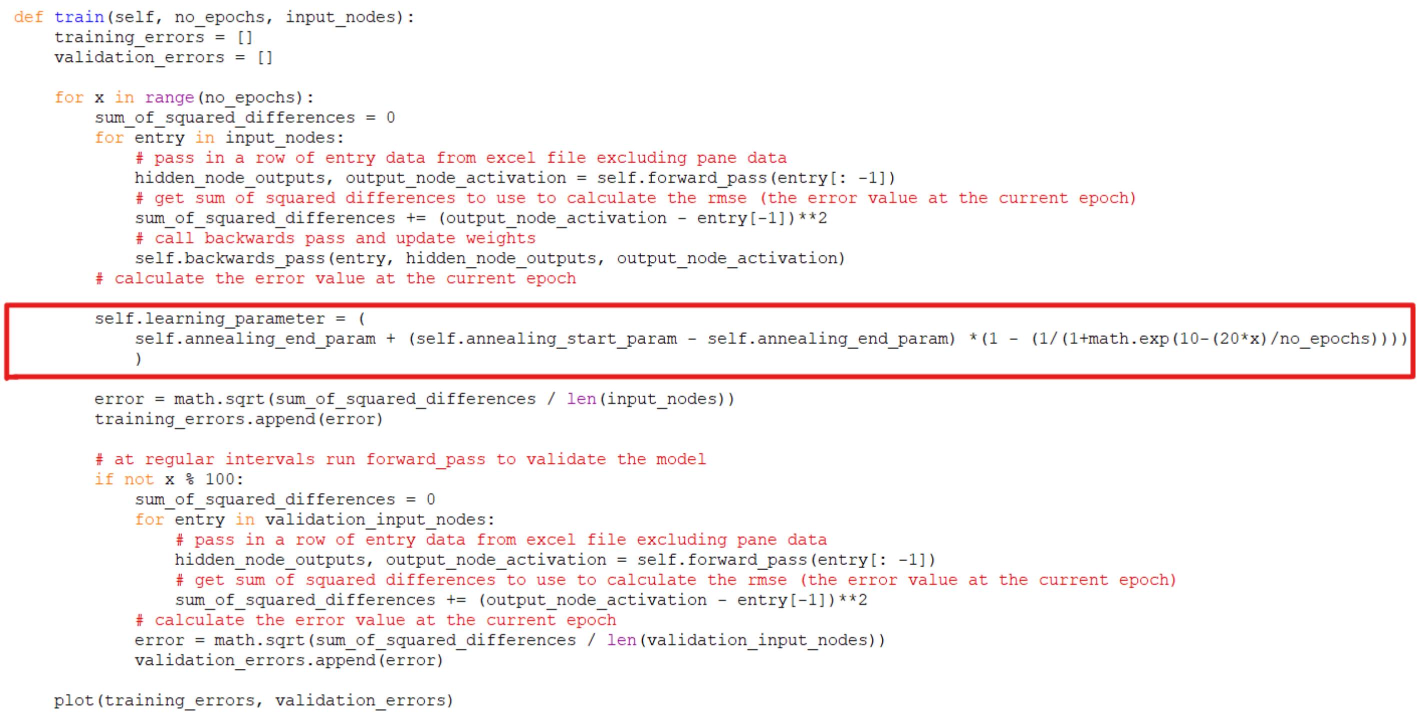
1. Modification of the MLP class constructor: We add **`annealing_start_param`** and **`annealing_end_param`** to the constructor, which are used to define the starting and ending learning rates.
2. Modification of the **`train()`** method: Learning rate is updated during the training using the annealing formula. The formula calculates the learning rate each epoch as a function of the starting learning rate, ending learning rate, and the current epoch number.

Annealing formula:

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

Example:
p = end parm: 0.01
q = start parm: 0.1
r = max epochs = 3000
x = epochs so far

This is placed within the **`train`** function as such:



```python
def train(self, no_epochs, input_nodes):
    training_errors = []
    validation_errors = []

    for x in range(no_epochs):
        sum_of_squared_differences = 0
        for entry in input_nodes:
            # pass in a row of entry data from excel file excluding pane data
            hidden_node_outputs, output_node_activation = self.forward_pass(entry[: -1])
            # get sum of squared differences to use to calculate the rmse (the error value at the current epoch)
            sum_of_squared_differences += (output_node_activation - entry[-1])**2
            # call backwards pass and update weights
            self.backwards_pass(entry, hidden_node_outputs, output_node_activation)
        # calculate the error value at the current epoch

        self.learning_parameter = (
            self.annealing_end_param + (self.annealing_start_param - self.annealing_end_param) *(1 - (1/(1+math.exp(10-(20*x)/no_epochs))))
            )

        error = math.sqrt(sum_of_squared_differences / len(input_nodes))
        training_errors.append(error)

        # at regular intervals run forward_pass to validate the model
        if not x % 100:
            sum_of_squared_differences = 0
            for entry in validation_input_nodes:
                # pass in a row of entry data from excel file excluding pane data
                hidden_node_outputs, output_node_activation = self.forward_pass(entry[: -1])
                # get sum of squared differences to use to calculate the rmse (the error value at the current epoch)
                sum_of_squared_differences += (output_node_activation - entry[-1])**2
            # calculate the error value at the current epoch
            error = math.sqrt(sum_of_squared_differences / len(validation_input_nodes))
            validation_errors.append(error)

    plot(training_errors, validation_errors)
```

The rest of the code remains the same as in the previous implementation.

Weight Decay

In this implementation we add a weight decay mechanism to avoid overfitting. We introduce a penalty term to the error function and control its strength with a regularisation parameter that decays over time.

The changes from the previous implementation are as follows:

1. Add a regularisation parameter to the `**backwards_pass**` method. It's a function of the epoch and the learning rate and controls the strength of the weight decay.

   $$\upsilon = \frac{1}{\rho e}$$

   ```
   # ! get regularisation parameter
   regularisation_parameter = 1 / (epoch * self.learning_parameter)
   ```

2. Calculate omega, in the `**backwards_pass**` method. Omega is the sum of the squares of all the weights and biases in the network, normalised by a constant. *Penalty term is highlighted in red.*

   $$\Omega = \frac{1}{2n}\sum_{i=1}^{n} w_i^2$$

   ```
   # ! get omega
   omega = (
       sum(w**2 for node in self.hidden_node_weights for w in node)
       + sum(w**2 for w in self.output_node_weights)
       + sum(w**2 for w in self.hidden_node_biases)
       + self.output_node_bias**2
   ) / (2 * (6*self.no_hidden_nodes + self.no_hidden_nodes + 1))
   ```

3. Add the penalty term to the error function in the `**backwards_pass**` method. Remember the penalty term is equal to the regularisation parameter * omega.

   $$\widetilde{E} = E + \upsilon\Omega$$

   ```
   # ! get error
   error = (inputs_and_target[-1][0] - predicted_PanE_value) + regularisation_parameter * omega
   ```

Updated `**backwards_pass**` method:

```
def backwards_pass(self, epoch, inputs_and_target, hidden_node_outputs, predicted_PanE_value):
    # calculate delta of the output node by getting the difference between the correct and predicted output
    output_node_sigmoid_derivative = predicted_PanE_value * (1 - predicted_PanE_value)

    # ! get regularisation parameter
    regularisation_parameter = 1 / (epoch * self.learning_parameter)

    if regularisation_parameter < 0.001:
        regularisation_parameter = 0.001
    elif 0.1 < regularisation_parameter:
        regularisation_parameter = 0.1

    # ! get omega
    omega = (
        sum(w**2 for node in self.hidden_node_weights for w in node)
        + sum(w**2 for w in self.output_node_weights)
        + sum(w**2 for w in self.hidden_node_biases)
        + self.output_node_bias**2
    ) / (2 * (6*self.no_hidden_nodes + self.no_hidden_nodes + 1))

    # ! get error
    error = (inputs_and_target[-1][0] - predicted_PanE_value) + regularisation_parameter * omega

    output_node_delta = (inputs_and_target[-1][0] - predicted_PanE_value)*output_node_sigmoid_derivative
    # calculate delta of each hidden node by multiplying...
    # ...[the weight of its connection to the output node] with [the delta of the output node]...
    # ...and the derivative equation for the hidden layer
    hidden_node_deltas =[]
    for i in range(self.no_hidden_nodes):
        hidden_node_sigmoid_derivative = hidden_node_outputs[i] * (1 - hidden_node_outputs[i])
        hidden_node_deltas.append(self.output_node_weights[i] * output_node_delta * hidden_node_sigmoid_derivative)

    # ammend all weights by adding the product of (learning parameter, delta of node, output of the node) to the weight
    for i in range(self.no_hidden_nodes):
        # update hidden node weights and biases
        for j in range(5):
            self.hidden_node_weights[i][j] += self.learning_parameter * hidden_node_deltas[i] * inputs_and_target[j][0]

        self.hidden_node_biases[i][0] += self.learning_parameter * hidden_node_deltas[i]

        # next ammend output weights and output biases
        self.output_node_weights[i] += self.learning_parameter * output_node_delta * hidden_node_outputs[i]

    self.output_node_bias += self.learning_parameter * output_node_delta
```

Overall, the weight decay MLP implementation adds a regularisation term to the error function and controls its strength with a parameter that decays over time.

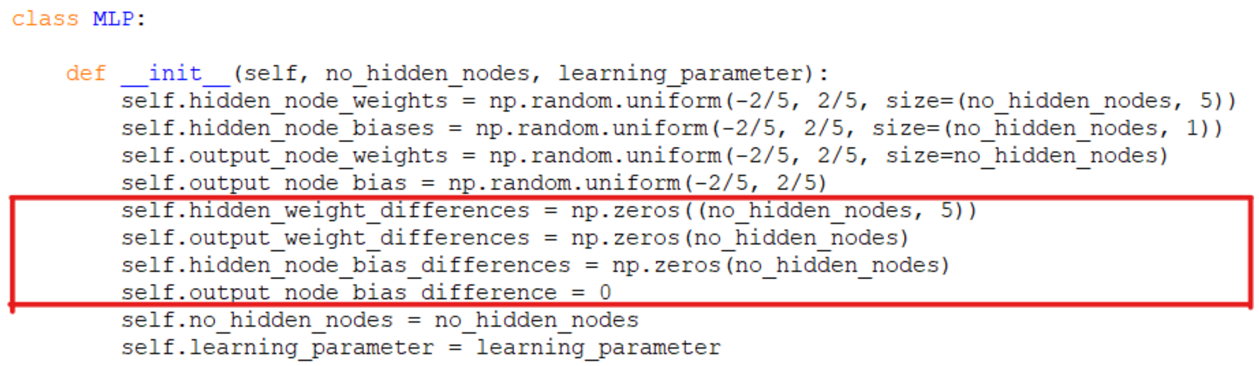
Momentum

This model modifies the initial MLP, by incorporating momentum into the weight updating section.

We add new attributes to the **`MLP`** class to store the weight and bias differences from the previous iteration. These are:

- **`hidden_weight_differences`**
- **`output_weight_differences`**
- **`hidden_node_bias_differences`**
- **`output_node_bias_difference`**

We also update the constructor of the class to initialise these new attributes with arrays of zeros, with the same size as the corresponding weights or biases.

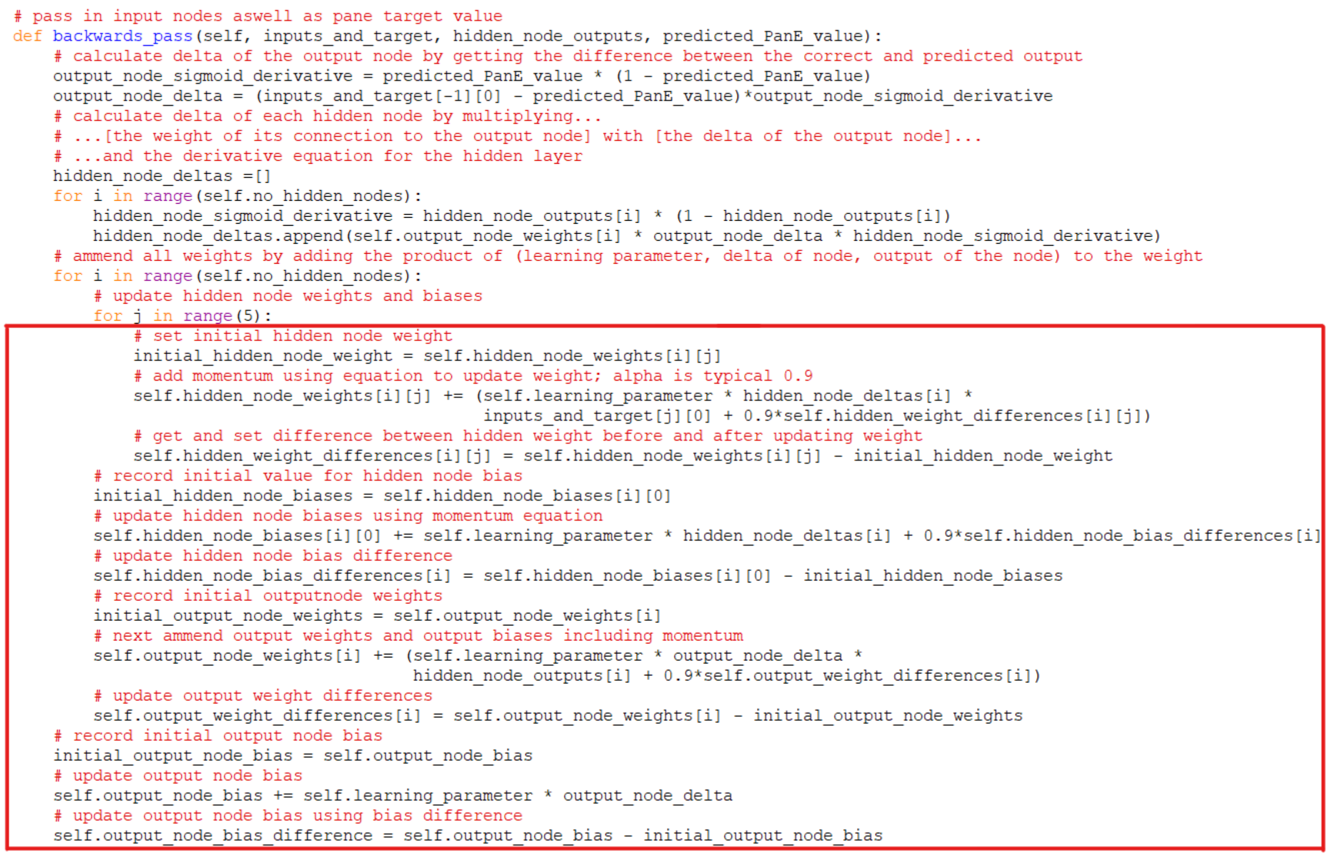


```
class MLP:

    def __init__(self, no_hidden_nodes, learning_parameter):
        self.hidden_node_weights = np.random.uniform(-2/5, 2/5, size=(no_hidden_nodes, 5))
        self.hidden_node_biases = np.random.uniform(-2/5, 2/5, size=(no_hidden_nodes, 1))
        self.output_node_weights = np.random.uniform(-2/5, 2/5, size=no_hidden_nodes)
        self.output_node_bias = np.random.uniform(-2/5, 2/5)
        self.hidden_weight_differences = np.zeros((no_hidden_nodes, 5))
        self.output_weight_differences = np.zeros(no_hidden_nodes)
        self.hidden_node_bias_differences = np.zeros(no_hidden_nodes)
        self.output_node_bias_difference = 0
        self.no_hidden_nodes = no_hidden_nodes
        self.learning_parameter = learning_parameter
```

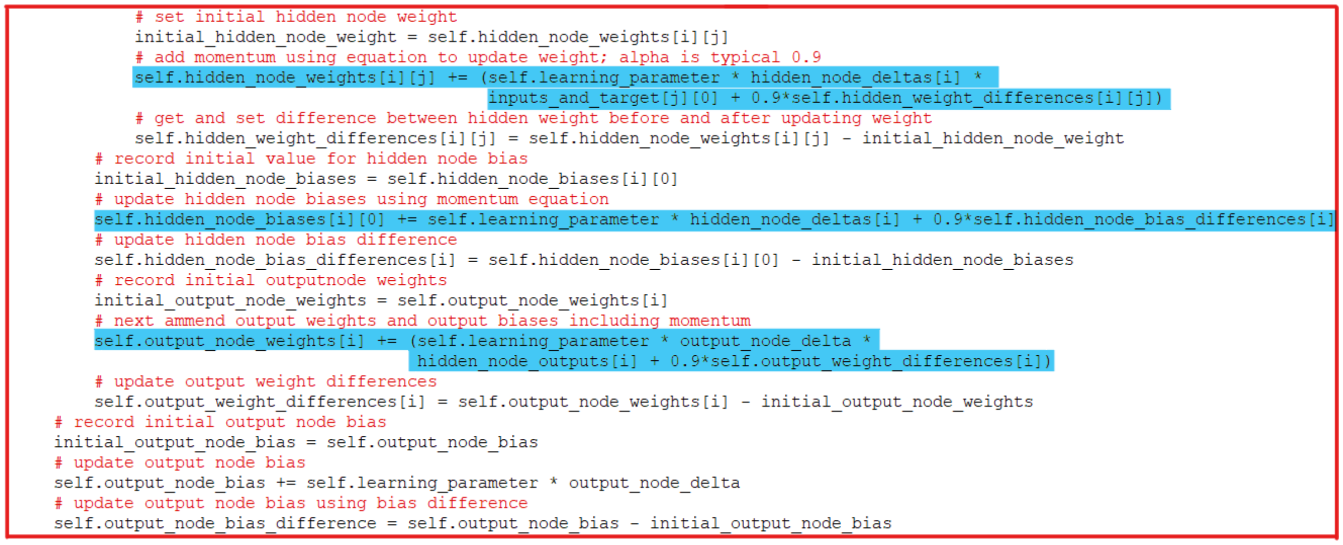
In the **`backwards_pass`** method, the updates made to the weights and biases also include the previous iteration's weight and bias differences; this is done by incorporating momentum into the weight and bias updates. The momentum term is given by a scalar, 0.9, multiplied by the previous iteration's weight or bias difference.

$$w^*_{i,j} = w_{i,j} + \rho\delta_j u_i + \alpha\Delta w_{i,j}$$



```
# pass in input nodes aswell as pane target value
def backwards_pass(self, inputs_and_target, hidden_node_outputs, predicted_PanE_value):
    # calculate delta of the output node by getting the difference between the correct and predicted output
    output_node_sigmoid_derivative = predicted_PanE_value * (1 - predicted_PanE_value)
    output_node_delta = (inputs_and_target[-1][0] - predicted_PanE_value)*output_node_sigmoid_derivative
    # calculate delta of each hidden node by multiplying...
    # ...[the weight of its connection to the output node] with [the delta of the output node]...
    # ...and the derivative equation for the hidden layer
    hidden_node_deltas =[]
    for i in range(self.no_hidden_nodes):
        hidden_node_sigmoid_derivative = hidden_node_outputs[i] * (1 - hidden_node_outputs[i])
        hidden_node_deltas.append(self.output_node_weights[i] * output_node_delta * hidden_node_sigmoid_derivative)
    # ammend all weights by adding the product of (learning parameter, delta of node, output of the node) to the weight
    for i in range(self.no_hidden_nodes):
        # update hidden node weights and biases
        for j in range(5):
            # set initial hidden node weight
            initial_hidden_node_weight = self.hidden_node_weights[i][j]
            # add momentum using equation to update weight; alpha is typical 0.9
            self.hidden_node_weights[i][j] += (self.learning_parameter * hidden_node_deltas[i] *
                                               inputs_and_target[j][0] + 0.9*self.hidden_weight_differences[i][j])
            # get and set difference between hidden weight before and after updating weight
            self.hidden_weight_differences[i][j] = self.hidden_node_weights[i][j] - initial_hidden_node_weight
        # record initial value for hidden node bias
        initial_hidden_node_biases = self.hidden_node_biases[i][0]
        # update hidden node biases using momentum equation
        self.hidden_node_biases[i][0] += self.learning_parameter * hidden_node_deltas[i] + 0.9*self.hidden_node_bias_differences[i]
        # update hidden node bias difference
        self.hidden_node_bias_differences[i] = self.hidden_node_biases[i][0] - initial_hidden_node_biases
        # record initial outputnode weights
        initial_output_node_weights = self.output_node_weights[i]
        # next ammend output weights and output biases including momentum
        self.output_node_weights[i] += (self.learning_parameter * output_node_delta *
                                        hidden_node_outputs[i] + 0.9*self.output_weight_differences[i])
        # update output weight differences
        self.output_weight_differences[i] = self.output_node_weights[i] - initial_output_node_weights
    # record initial output node bias
    initial_output_node_bias = self.output_node_bias
    # update output node bias
    self.output_node_bias += self.learning_parameter * output_node_delta
    # update output node bias using bias difference
    self.output_node_bias_difference = self.output_node_bias - initial_output_node_bias
```

Highlighted momentum equation:



```
            # set initial hidden node weight
            initial_hidden_node_weight = self.hidden_node_weights[i][j]
            # add momentum using equation to update weight; alpha is typical 0.9
            self.hidden_node_weights[i][j] += (self.learning_parameter * hidden_node_deltas[i] *
                                               inputs_and_target[j][0] + 0.9*self.hidden_weight_differences[i][j])
            # get and set difference between hidden weight before and after updating weight
            self.hidden_weight_differences[i][j] = self.hidden_node_weights[i][j] - initial_hidden_node_weight
        # record initial value for hidden node bias
        initial_hidden_node_biases = self.hidden_node_biases[i][0]
        # update hidden node biases using momentum equation
        self.hidden_node_biases[i][0] += self.learning_parameter * hidden_node_deltas[i] + 0.9*self.hidden_node_bias_differences[i]
        # update hidden node bias difference
        self.hidden_node_bias_differences[i] = self.hidden_node_biases[i][0] - initial_hidden_node_biases
        # record initial outputnode weights
        initial_output_node_weights = self.output_node_weights[i]
        # next ammend output weights and output biases including momentum
        self.output_node_weights[i] += (self.learning_parameter * output_node_delta *
                                        hidden_node_outputs[i] + 0.9*self.output_weight_differences[i])
        # update output weight differences
        self.output_weight_differences[i] = self.output_node_weights[i] - initial_output_node_weights
    # record initial output node bias
    initial_output_node_bias = self.output_node_bias
    # update output node bias
    self.output_node_bias += self.learning_parameter * output_node_delta
    # update output node bias using bias difference
    self.output_node_bias_difference = self.output_node_bias - initial_output_node_bias
```

The momentum value is added to the weight update equation, which helps the model remember the previous weight update and move in the same direction (Reference: **self.hidden_node_weights[i][j] += self.learning_parameter * hidden_node_deltas[i] * inputs_and_target[j][0] + 0.9*self.hidden_weight_differences[i][j])**

The rest of the code remains the same as in the initial implementation.

Training and Network Selection

In this section, we compare the different MLP models, and identify how their performance changes, after manipulating different hyperparameters. We identify key hyperparameters as the number of hidden nodes, and the learning parameter. The goal is to obtain the lowest possible error on the test dataset. However in the case of the annealing model we modify the start and end learning parameters, instead of just the single variable.

We keep the number of epochs as 1000 as it provides enough iterations for the ANN to converge to a stable yet accurate solution. If the number of epochs is too low, the network may not have enough time to learn and adjust its weights properly. However, if the number of epochs is too high, the network may overfit the training data and perform poorly on new, unseen data, which could increase validation errors.

We sort this table by the lowest testing error. Which refers to the value of the testing error at the final epoch. Then in the next section we evaluate the best performing models.

Figure 1: Comparing effects of varying hyperparameters on different versions of MLP (Base, Weight Decay, Momentum)

Model	No. Hidden nodes	Learning Parameter	Training error	Testing Error
Standard	3	0.01	0.0280	0.0266
Standard	3	0.05	0.0244	0.0263
Standard	3	0.1	0.0242	0.0285
Standard	6	0.01	0.0281	0.0286
Standard	6	0.05	0.0249	0.0274
Standard	6	0.1	0.0242	0.0277
Standard	9	0.01	0.0278	0.0277
Standard	9	0.05	0.0249	0.0273
Standard	9	0.1	0.0245	0.0277
Weight Decay	3	0.01	0.0266	0.0253
Weight Decay	3	0.05	0.0244	0.0267
Weight Decay	3	0.1	0.0240	0.0270
Weight Decay	6	0.01	0.0273	0.0265

Weight Decay	6	0.05	0.0246	0.0273
Weight Decay	6	0.1	0.0242	0.0280
Weight Decay	9	0.01	0.0286	0.0300
Weight Decay	9	0.05	0.0276	0.0304
Weight Decay	9	0.1	0.0246	0.0286
Momentum	3	0.01	0.0260	0.0284
Momentum	3	0.05	0.0221	0.0344
Momentum	3	0.1	0.0214	0.0302
Momentum	6	0.01	0.0256	0.0282
Momentum	6	0.05	0.0246	0.0273
Momentum	6	0.1	0.0212	0.0317
Momentum	9	0.01	0.0244	0.0280
Momentum	9	0.05	0.0211	0.0347
Momentum	9	0.1	0.0197	0.0346

Annealing, we maintain the hidden nodes hyperparameter but alter the way the learning parameter is utilised:

Figure 2:Figure 1: Comparing effects of varying hyperparameters on Annealing model

Hidden Nodes	**Start Learning Parameter**	**End Learning Parameter**	**Training Error**	**Testing Error**
3	0.1	0.01	0.0249	0.0249
3	0.1	0.05	0.0243	0.0247
3	0.05	0.01	0.0254	0.0254
6	0.1	0.01	0.0247	0.0248
6	0.1	0.05	0.0243	0.0247
6	0.05	0.01	0.0254	0.0254
9	0.1	0.01	0.0254	0.0254

9	0.1	0.05	0.0244	0.0249
9	0.05	0.01	0.0262	0.0263
3	0.1	0.01	0.0247	0.0247
3	0.1	0.05	0.0252	0.0258
3	0.05	0.01	0.0251	0.0252
6	0.1	0.01	0.0247	0.0247
6	0.1	0.05	0.0245	0.0251
6	0.05	0.01	0.0254	0.0255
9	0.1	0.01	0.0250	0.0250
9	0.1	0.05	0.0263	0.0270
9	0.05	0.01	0.0254	0.0255

The tables show the performance of different Multi-Layer Perceptron (MLP) models with variations in the number of hidden nodes, learning parameters, and other configurations such as weight decay, momentum, and annealing. The aim is to find the lowest testing error, which would indicate the best model performance on unseen data.

Number of Hidden Nodes

There doesn't seem to be a linear relationship between the number of hidden nodes and the testing error. In some cases, increasing the number of hidden nodes led to a slight increase in the testing error, while in other cases, it had a minimal effect or even improved the performance - suggesting that the optimal number of hidden nodes is not consistent across different configurations and learning parameters.

Learning Parameter

The learning parameter controls how quickly the model adapts to the data during training. Smaller learning rates result in the model learning slower, taking smaller steps in updating the weights, while larger learning rates lead to faster learning with more significant weight updates. The impact of changing the learning parameter varied across different configurations:

Standard:
Learning Parameter 0.01: Average testing error = (0.0266 + 0.0286 + 0.0277) / 3 = 0.0276
Learning Parameter 0.05: Average testing error = (0.0263 + 0.0274 + 0.0273) / 3 = 0.0270
Learning Parameter 0.1: Average testing error = (0.0285 + 0.0277 + 0.0277) / 3 = 0.0279

For the standard configuration, a learning parameter of 0.05 generally provided the best performance in terms of testing error. Both lower learning rates (0.01) and higher learning rates tend to result in slightly higher testing errors; this suggests that a moderate learning rate may be optimal for the standard configuration.

Momentum:
Learning Parameter 0.01: Average testing error = (0.0284 + 0.0282 + 0.0280) / 3 = 0.0282
Learning Parameter 0.05: Average testing error = (0.0344 + 0.0273 + 0.0347) / 3 = 0.0321
Learning Parameter 0.1: Average testing error = (0.0302 + 0.0317 + 0.0346) / 3 = 0.0322

For the momentum model, the optimal learning rate seems to be lower (0.01), than that of the standard configuration (0.05). This further highlights how the optimal learning rate varies across different configurations and may be influenced by the specific combination of hidden nodes and other hyperparameters.

Weight Decay:
Learning Parameter 0.01: Average testing error = (0.0253 + 0.0265 + 0.0300) / 2 = 0.0273
Learning Parameter 0.05: Average testing error = (0.0267 + 0.0273 + 0.0304) / 3 = 0.0281
Learning Parameter 0.1: Average testing error = (0.0270 + 0.0280 + 0.0286) / 3 = 0.0279

For the weight decay configuration, similarly to the Momentum model, a learning parameter of 0.01 generally yields the best performance in terms of testing error. Higher learning rates (0.05 and 0.1) tend to result in slightly increased testing errors. Suggesting that a lower learning rate may be more suitable for the weight decay configuration, allowing the model to learn more slowly and avoid overfitting.
Annealing:

Annealing:
Start Learning Parameter of 0.1 and End Learning Parameter of 0.01:
This configuration provided a larger learning rate decay over the training process. Starting with a higher learning rate (0.1) helps the model explore the solution space quicker and escape local minima. As the training progresses, the learning rate decays to a lower value (0.01), allowing the model to fine-tune its weights and reach a more accurate solution. Having a smaller learning rate towards the end of training helps the model to converge more precisely and avoid overshooting the optimal weights.

Start Learning Parameter of 0.1 and End Learning Parameter of 0.05:
In this case, the learning rate decay is smaller in comparison to the previous configuration. Although the model still starts with a relatively high learning rate (0.1), it only decreases to 0.05 by the end of training. This means that the model still explores the solution space quickly at the beginning, but the fine-tuning phase might not be as precise as in the first case as the final learning rate is greater. The model might swing more around the optimal weights due to this..

Start Learning Parameter of 0.05 and End Learning Parameter of 0.01:
Here, the model starts with a moderate learning rate (0.05) and decays to a lower value (0.01) as training progresses. In this case, the exploration of the solution space is slower compared to starting with a learning rate of 0.1, which can result in slower convergence. However, the model still has the fine-tuning benefits offered by the lower learning rate (0.01) towards the end of training.

Based on the results in *Figure 2*, the Annealing model with a start learning parameter of 0.1 and an end learning parameter of 0.01 performed the best, with the lowest testing error. We can infer from this that the larger learning rate decay, allowing for both quick exploration and precise fine-tuning, was more effective.

Evaluation:
In conclusion, the impact of changing the learning parameter is not the same across different configurations. It’s important to find the optimal learning rate for each configuration to balance the speed of learning and generalisation to unseen data.

Evaluation

Based on the testing errors, the best performing configurations of each model are:

Standard: 3 hidden nodes and a learning parameter of 0.05, with a testing error of 0.0263.
Weight Decay: 3 hidden nodes and a learning parameter of 0.01, with a testing error of 0.0253.
Momentum: 6 hidden nodes and a learning parameter of 0.05, with a testing error of 0.0273.
Annealing: 3 hidden nodes with a start parameter of 0.1 and an end parameter of 0.01, with a testing error of 0.0249.

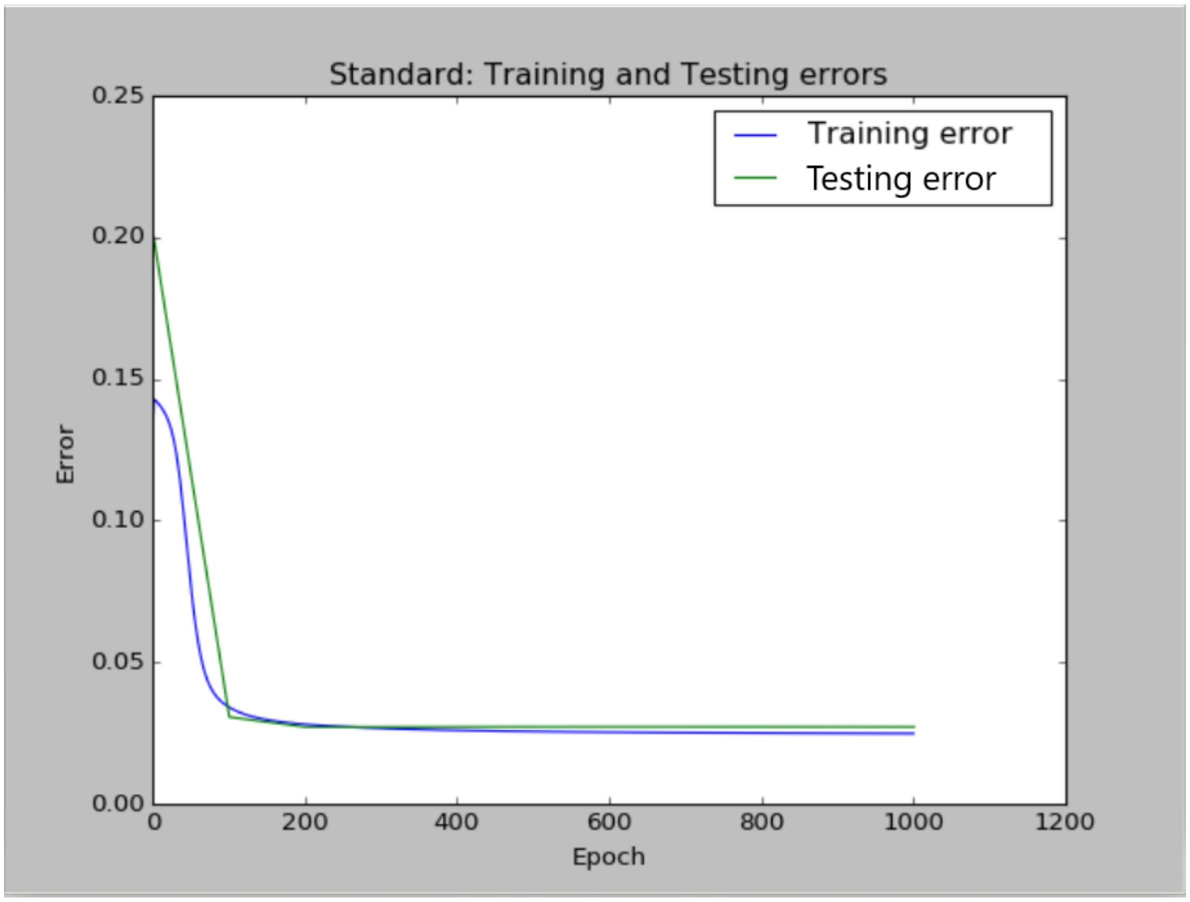
It is important to note however, running each configuration multiple times and taking the mean of the testing errors would provide a more accurate assessment of the model performance, as it would account for randomness and variability in the training process. As it stands each configuration was only tested once and the results were collected from that run.

Evaluation of final model

Overall, the Annealing model with 3 hidden nodes, a start learning parameter of 0.1, and an end learning parameter of 0.01 returns the lowest testing error of 0.0249, making it the best configuration among all the models. Below we plot the values of the errors of the best configurations for the different models.

Figure 3: Training and Testing Errors

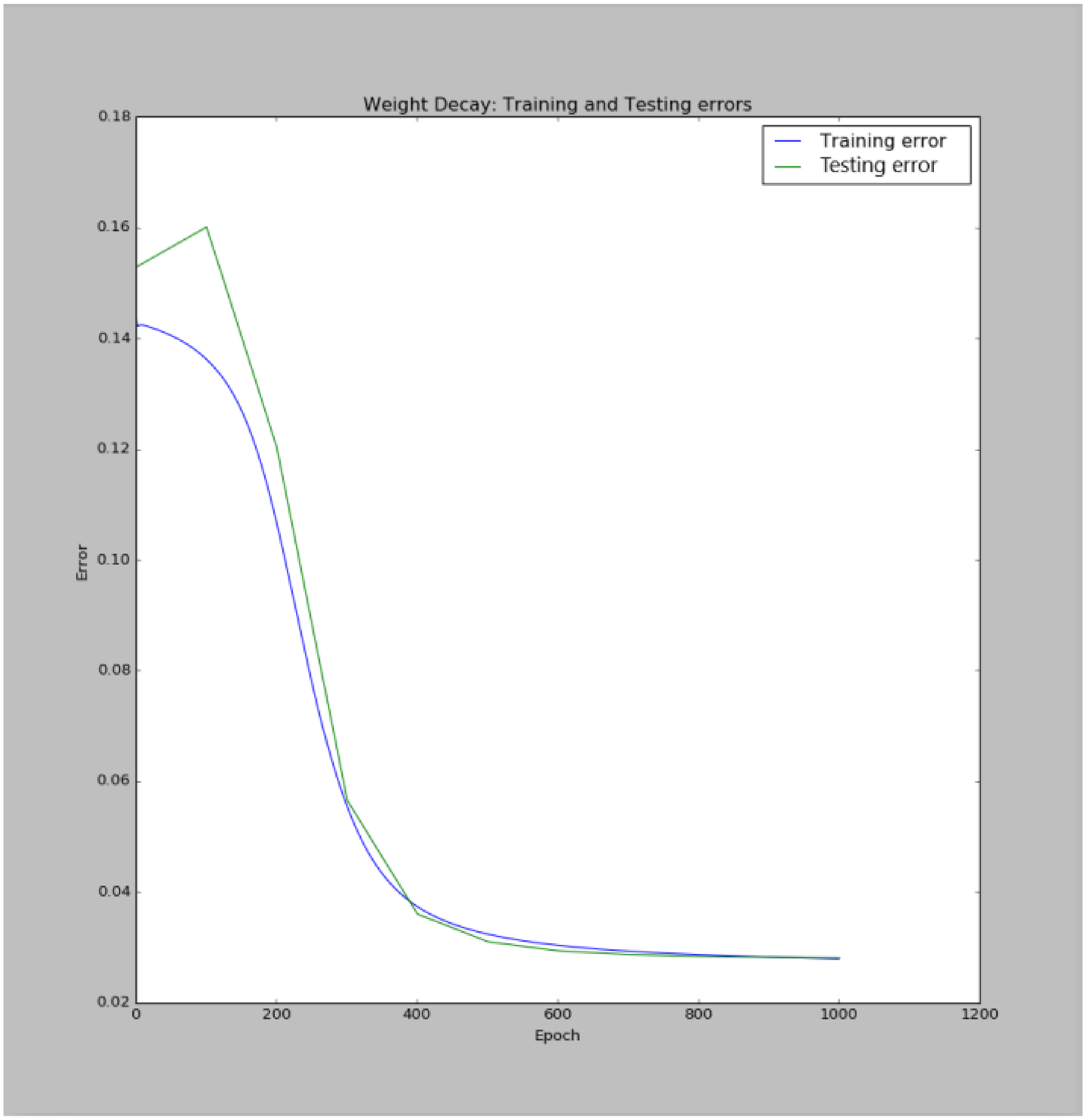
Standard: 3 hidden nodes and a learning parameter of 0.05, with a testing error of 0.0263.



- Quick convergence: Model learns patterns rapidly
- Slight rise in testing error: Possible overfitting after 100 epochs

Figure 4: Weight Decay: Training and Testing Errors

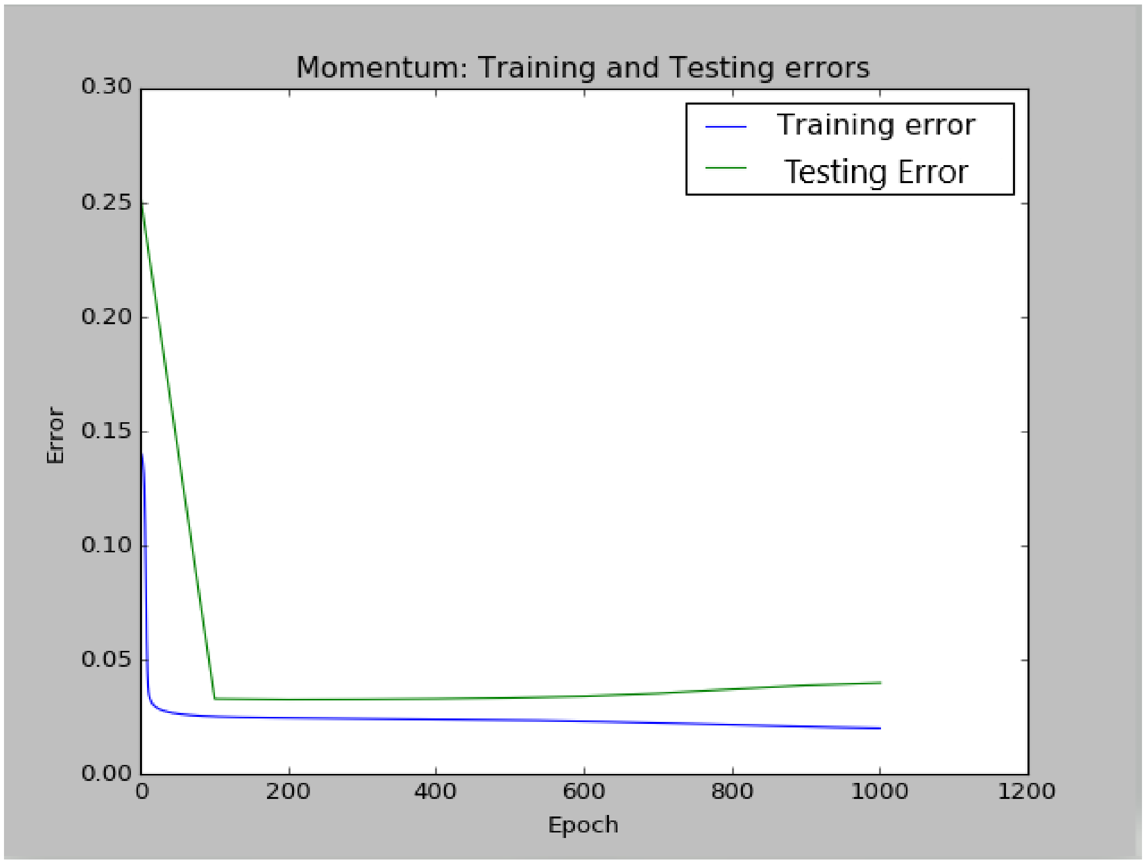
Weight Decay: 3 hidden nodes and a learning parameter of 0.01, with a testing error of 0.0253.



- Convergence around 300 epochs: Model achieves balance between learning and generalisation (success with new unseen data)
- Stability from 700 epochs: Consistent performance on unseen data

Figure 5: Momentum: Training and Testing Errors

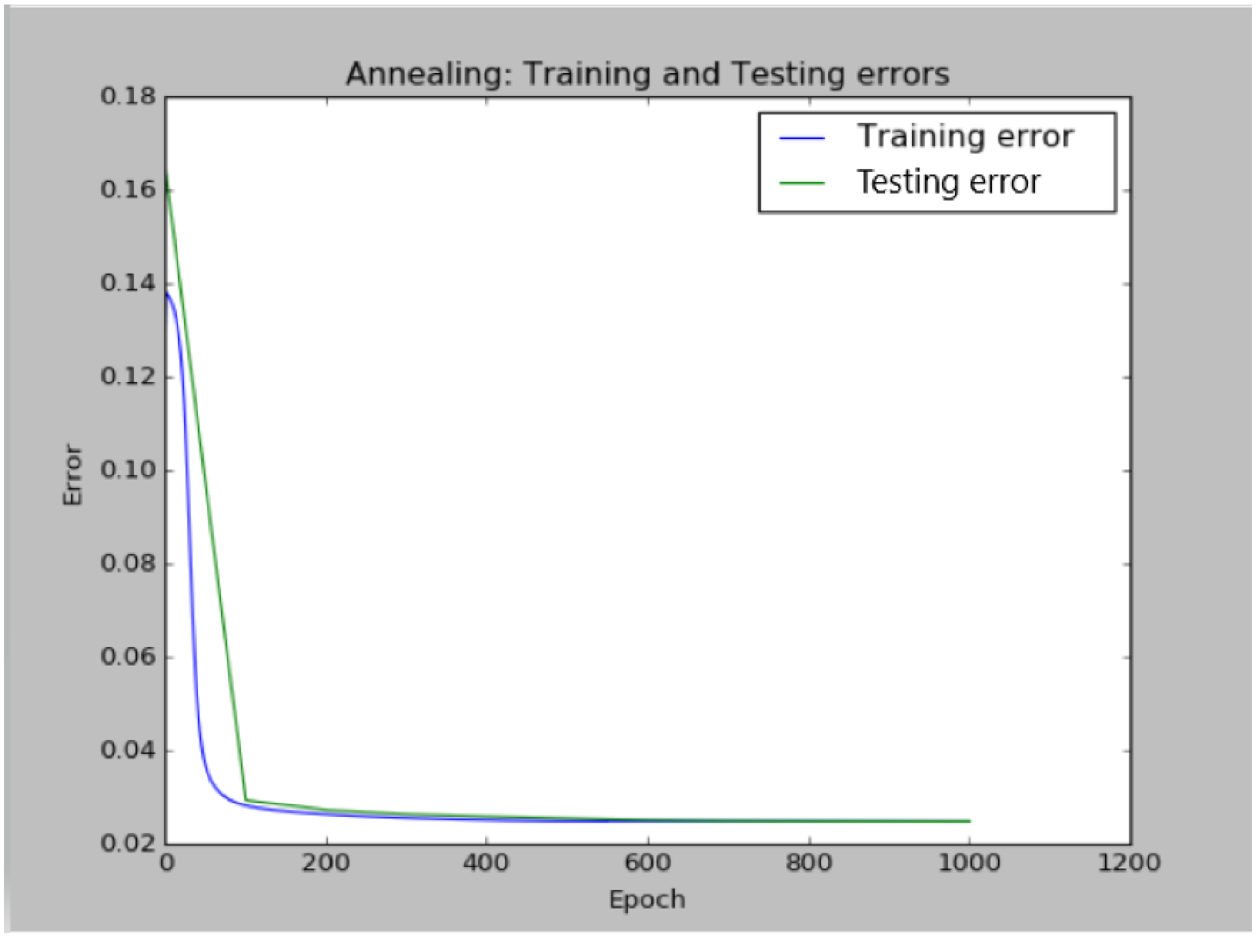
Momentum: 6 hidden nodes and a learning parameter of 0.05,



- Quick convergence: Model learns patterns rapidly
- Testing error increases later: Overfitting as training progresses

Figure 6: Annealing: Training and Testing Errors

Annealing: 3 hidden nodes with a start parameter of 0.1 and an end parameter of 0.01, with a testing error of 0.0249.



- Fast and stable convergence: Model learns efficiently and generalises well to unseen data
- Stable performance: Reduced risk of overfitting
- This has the best parts of every model: fast and stable convergence, without any of the other observed weaknesses: high risks of overfitting