AI coursework

Contents

Data pre-processing

Gathering of data has been done prior as sufficient data is already made available. The next stage will involve me cleansing the data and identifying predictors.

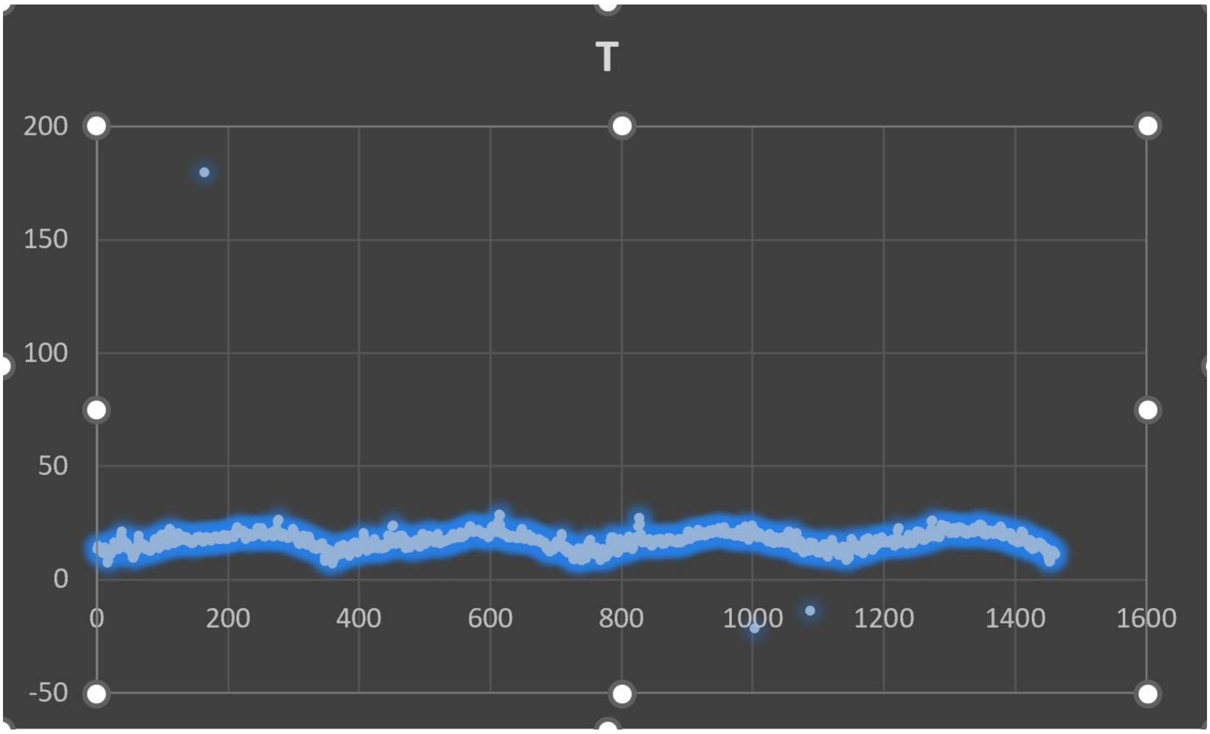
Identifying predictors:

The predictors include; Mean daily temperature in Celsius (T), Wind speed in cm/s (W) , solar radiation in Langley(SR), Air pressure in kpa(DSP) and Humidity in % (DRH).

The predictand is Pan Evaporation in cm/day (PanE)

Cleaning the data : To clean the data, I began with removing values that were not in the data type of the predictors.

T prior



The graph made it easier t spot outliers and as shown there are 3 outliers. I went selected each row and deleted the entire row to remove them.

The value 180 was an outlier because it is large and out of the range.

I also removed the negative numbers as they are negative outliers.

61387	180	483.3	597.3	101.7	74	0.59

122289	-13.6	297.9	281	102.3	57	0.32

92989	-21.5	414.5	528.5	101.4	69	0.6

W prior

11788	12.8	1089.7	101.1	100.2	80	0.36

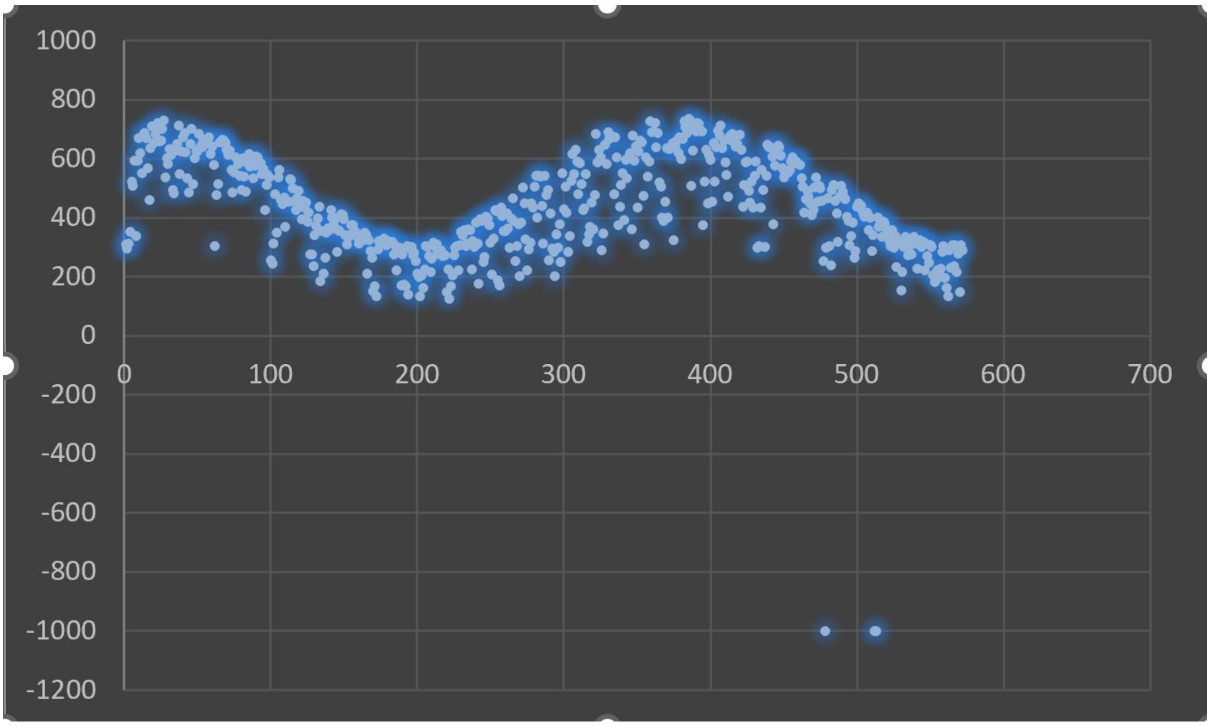
123088	9.5	96.1	306.5	102.3	42	0.24

52988	16.3	838.2	500.1	100.9	58	0.75
122090	11	800.8	162.4	101.2	66	0.43
50287	17	a	627.6	101.4	66	0.62

Removed the letters as the values for W should only be numbers.

96.1, 1089.7, 838.2, 800 were all outliers as they all seemed to be out of the range for W.

SR prior

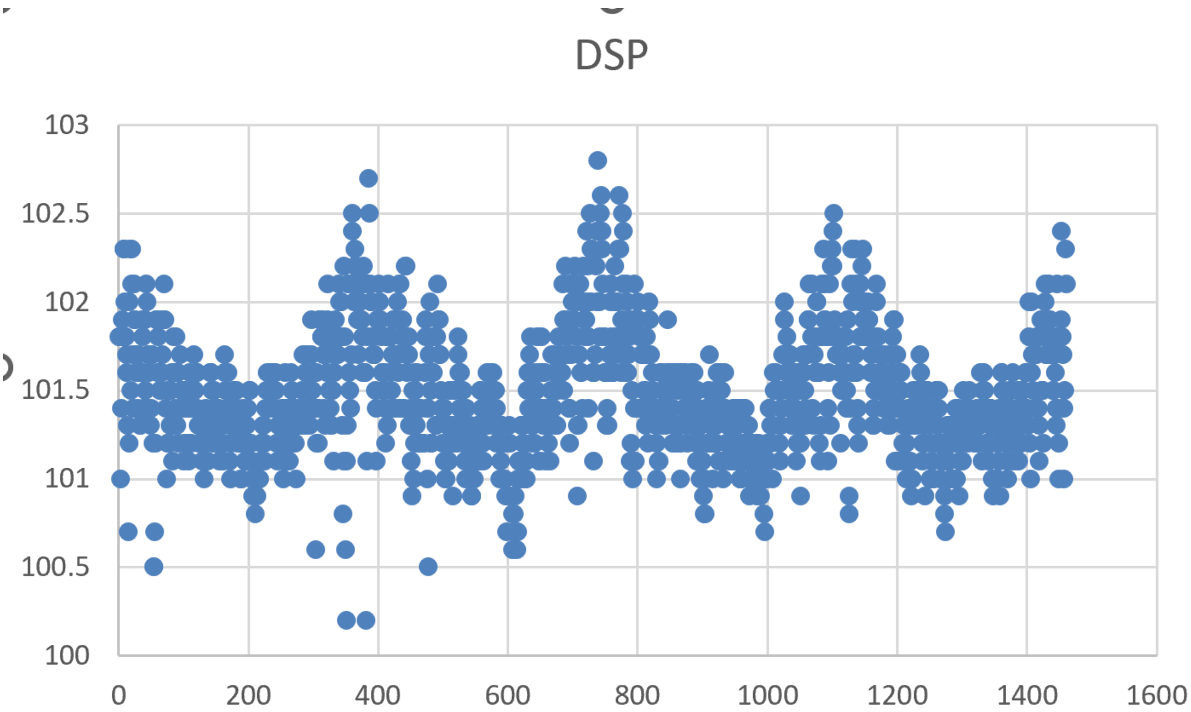


92890	20.3	371.7	-999	101.2	70	0.45

110190	17.3	442.3	-999	101.4	62	0.44
110290	17.1	433.2	-999	101.3	46	0.6

The negative values for SR were removed as they were outliers.

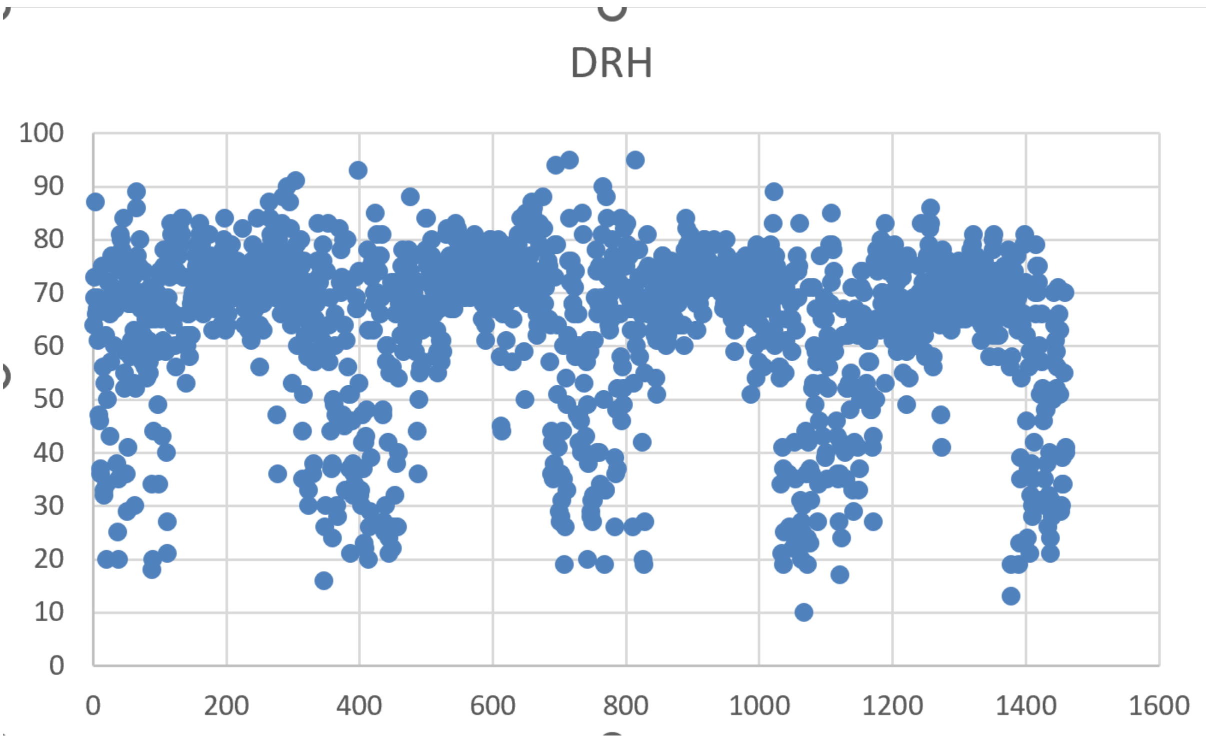
DSP prior



121787	13.3	470.6	125.8	100.2	74	0.25
60789	17.1	404.5	ddd	101.3	78	0.3

Removed the row containing ddd as it didn't match the data type .125.8 was also an outlier in the data.

DRH prior



62290 19.5 465.6 599.7 101.2 0.6

STANDARDISATION

I split the data into 3 parts; Training 60 %, validation 20% and testing 20% and then standardised

TRAINING

Data	Min	Max
SR	78.4	743.2
DSP	100.5	102.8
DRH	16	95
PanE	0.07	1.28
T	7.2	28.9
W		

Validation

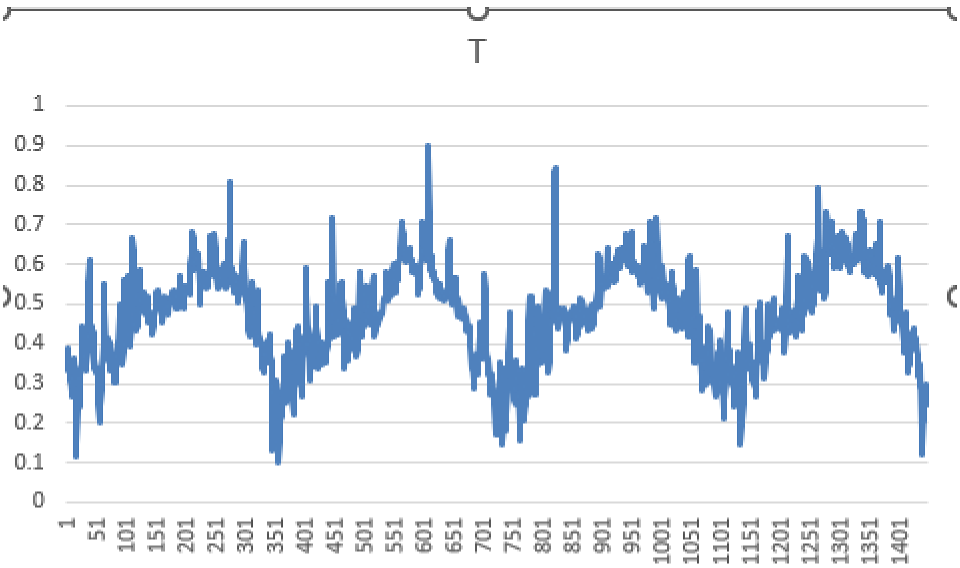
Data	Min	Max

T	8.5	24
W	125.8	757.1
SR	125.7	728.7
DSP	100.7	102.5
DRH	10	89
PanE	0.14	0.94

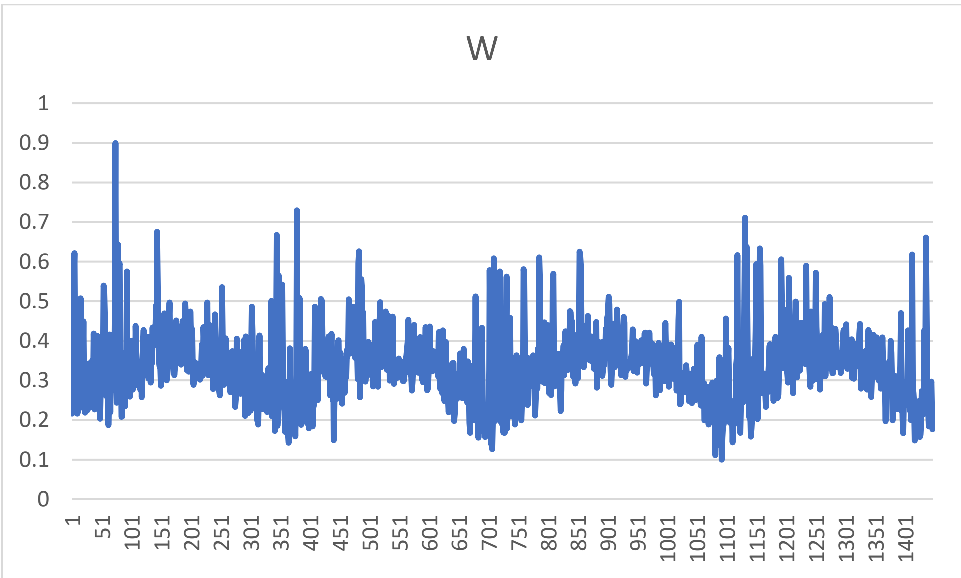
Date	T	W	SR	DSP	DRH	PanE	T2	W2	SR2	DSP2	DRH2	PANE2
10187	13.5	245.5	215.6	101.8	64	0.21	0.332258	0.193312	0.265102	0.552174	0.586076	0.192562
10287	15	350.8	290.4	101.8	69	0.27	0.387558	0.298625	0.355114	0.552174	0.636709	0.232231
10387	14.3	310.7	242.6	101.8	73	0.21	0.361751	0.25852	0.297593	0.552174	0.677215	0.192562
10487	13.5	595.4	97.5	101	87	0.16	0.332258	0.543255	0.122984	0.273913	0.818987	0.159504
10587	13.9	664.2	234.4	101.4	66	0.4	0.347005	0.612064	0.287726	0.413043	0.606329	0.318182
10687	12.8	380.8	216.6	101.9	67	0.24	0.306452	0.328629	0.266306	0.586957	0.616456	0.212397
10787	13.3	404.5	246	101.8	69	0.28	0.324885	0.352332	0.301685	0.552174	0.636709	0.238843
10887	12	302.9	303.6	102.3	61	0.28	0.276959	0.250719	0.370999	0.726087	0.555696	0.238843
10987	12.5	334.8	269.1	102.3	47	0.36	0.295392	0.282623	0.329483	0.726087	0.413924	0.291736
11087	11.7	245.1	321.4	102	46	0.34	0.265899	0.192912	0.392419	0.621739	0.403797	0.278512
11187	13.7	251.5	320.1	101.8	37	0.4	0.339631	0.199312	0.390854	0.552174	0.312658	0.318182
11287	14.3	266.5	319.3	101.7	36	0.46	0.361751	0.214314	0.389892	0.517391	0.302532	0.357851
11387	13.1	258.7	236.2	101.6	68	0.21	0.317512	0.206513	0.289892	0.482609	0.626582	0.192562
11487	13.6	303.8	169.6	101.3	75	0.17	0.335945	0.251619	0.209747	0.378261	0.697468	0.166116
11587	12.1	546.6	225.1	100.7	56	0.42	0.280645	0.494449	0.276534	0.169565	0.505063	0.331405
11687	7.6	429.6	331.1	101.2	32	0.5	0.114747	0.377435	0.404091	0.343478	0.262025	0.384298
11787	8.5	307	325	102	33	0.4	0.147926	0.254819	0.396751	0.621739	0.272152	0.318182
11887	10.8	348	314.4	102.3	53	0.35	0.232719	0.295824	0.383995	0.726087	0.474684	0.285124
11987	11.8	348	297.7	101.5	62	0.3	0.269585	0.295824	0.363899	0.447826	0.565823	0.252066
12087	13	486.1	349.6	102.1	20	0.72	0.313825	0.433942	0.426354	0.656522	0.140506	0.529752
12187	11.1	307.5	324.1	102.3	34	0.44	0.243779	0.255319	0.395668	0.726087	0.282278	0.344628
12287	12.1	346.2	263.6	101.7	50	0.33	0.280645	0.294024	0.322864	0.517391	0.444304	0.271901
12387	13.2	247.4	239.6	101.4	72	0.18	0.321198	0.195212	0.293983	0.413043	0.667089	0.172727
12487	13.1	305.2	328.6	101.9	72	0.23	0.317512	0.253019	0.401083	0.586957	0.667089	0.205785
12587	14.3	251.5	354.1	102.1	66	0.28	0.361751	0.199312	0.431769	0.656522	0.606329	0.238843
12687	16.5	278.8	349.5	101.9	43	0.42	0.442857	0.226616	0.426233	0.586957	0.373418	0.331405
12787	15.7	254.2	177.7	101.6	57	0.28	0.413364	0.202013	0.219495	0.482609	0.51519	0.238843
12887	15.5	362.2	304.4	101.7	77	0.28	0.405991	0.310026	0.371961	0.517391	0.717722	0.238843
12987	13.8	291.5	325.3	101.7	71	0.26	0.343318	0.239317	0.397112	0.517391	0.656962	0.22562
13087	13.6	376.7	257.8	101.4	67	0.28	0.335945	0.324528	0.315884	0.413043	0.616456	0.238843
13187	13.6	292.5	362.3	101.7	70	0.3	0.335945	0.240318	0.441637	0.517391	0.646835	0.252066
20187	14.3	262.8	359.7	101.6	60	0.32	0.361751	0.210614	0.438508	0.482609	0.54557	0.265289
20287	13.5	290.2	329.7	101.3	67	0.27	0.332258	0.238017	0.402407	0.378261	0.616456	0.232231
20387	13.8	306.1	227.9	101.3	77	0.18	0.343318	0.253919	0.279904	0.378261	0.717722	0.172727
20487	14.7	384.5	384.6	101.6	75	0.32	0.376498	0.332329	0.468472	0.482609	0.697468	0.265289

This is the comparison of my standaradised training data compared to the original.

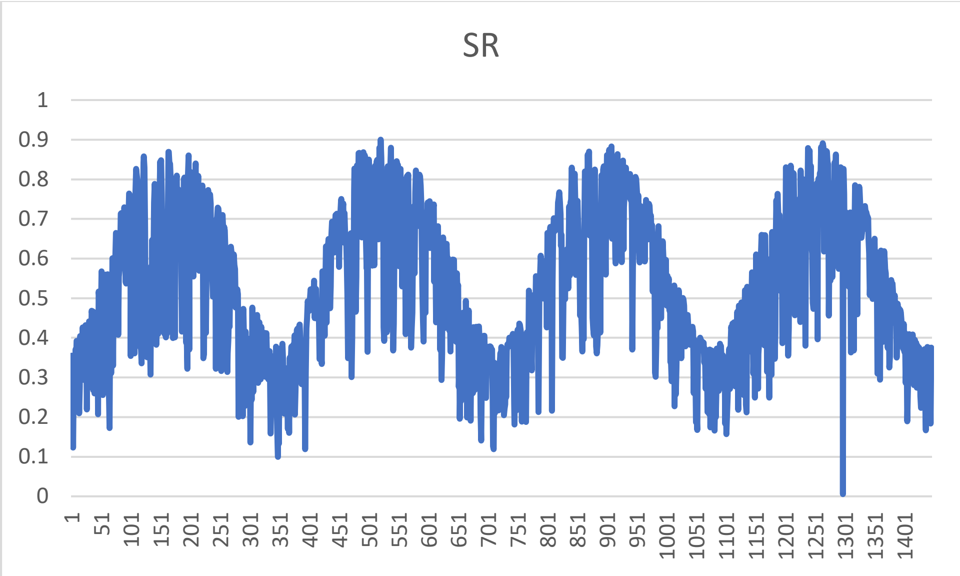
Graph for T after standardising



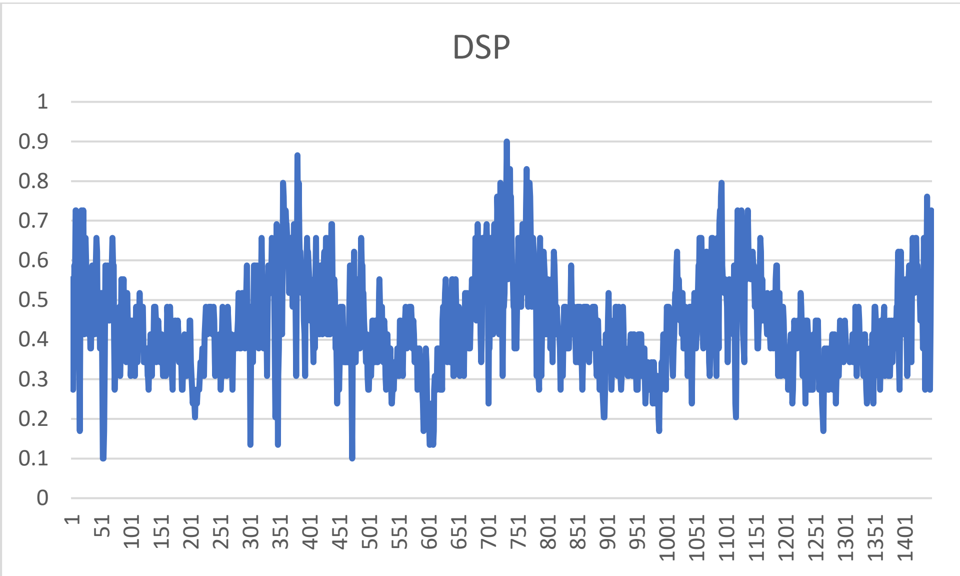
Graph for W after standardising



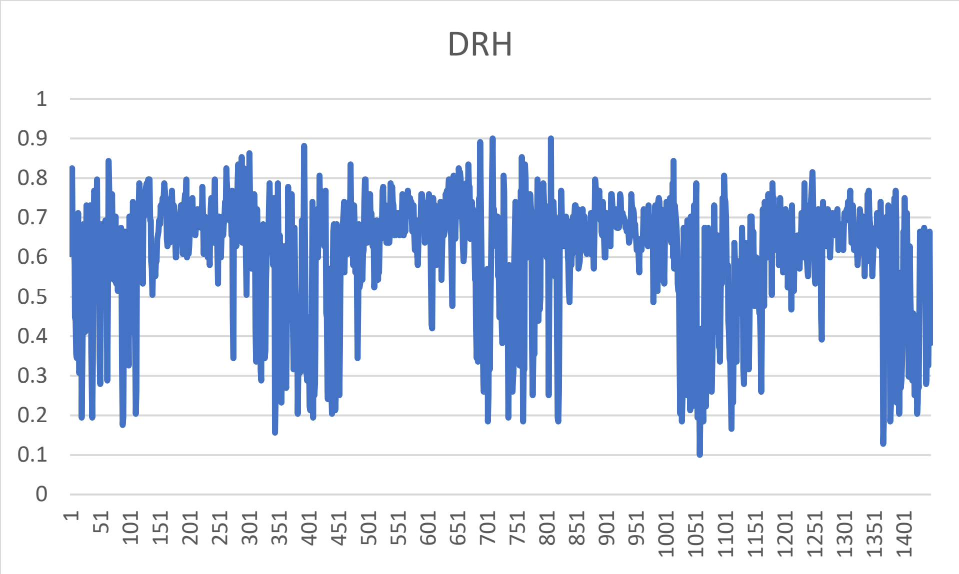
Graph for SR after standardising



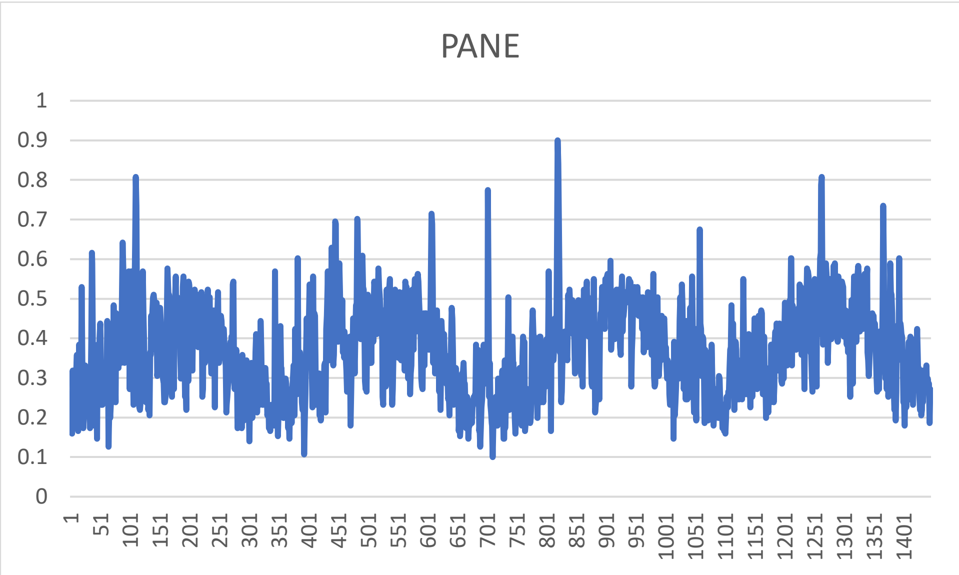
Graph for DSP after standardising



Graph of DRH after standardisation



Graph of PanE after standardisation



I standardised the data using this formula:

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

R= representing each row of the data.

Min= is the minimum for each column

Max is the maximum for each column

Implementation

For this coursework I used python as my programming language because it is the language I am most comfortable with and it is used by a lot of people for data science. The libraries I used consisted of:

Numpy: Numpy is a python library that provides multidimensional array object and an assortment of routines for fast operations on arrays.

Pandas: Pandas is a python library used for working with data sets

Matplotlib: Python library used for plotting and analysing graphs.

I used a procedural programming approach, which involved me structuring my code as a sequence of steps that are executed in particular order to achieve the desired result.

CODE

Functions used

```
# formulas
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def sigmoid_derivative(x):
    return x * (1 - x)
```

I labelled this formula as this contains all the mathematical formulas used. The derivative of sigmoid Is used to calculate gradient.

I read from the excel file using pandas and specified the columns to be read from.

I made use of 4 hidden layers, 5 inputs, 1 output node and the bias was made from within the range [-2/input size,2/ input size]

Learning parameter is 0.01. The general code is implemented in the form: forward pass, backward pass and update weights.

The code iterates for ‘iterations’- (which is a representation of epochs) times.

FORWARD Pass

For the forward pass, the code calculates the weighted sum of the inputs to each hidden nodes and after that it apples the sigmoid function to each hidden node's weighted sum and generates a set of hidden node output values (sumofoutput). After thatm the code calcutes the weight sum of the hidden node to obtain the final output values called output. The output is then passed through the sigmoid function to map it to the range – 0,1.

After calculating the output value, the error is the squared difference between the predicted output and actual output. After which I moved to the backward pass to then update the weights based on the error.

```
for i in range(iterations):
    error = 0
    for row in range(len((X))):
        # forward pass
        sumofoutput  = []
        for node in range(hidden_layer):
            sum_ = hidden_bias[node] + np.dot(X[row], hidden_layer_weights.T[node])
            sumofoutput.append(sigmoid(sum_))
        ouputsum = output_bias + np.dot(sumofoutput, output_weight)
        output = sigmoid(ouputsum)
        sim_annealing(iterations,i)

        error += (Y[row]-output)**2

        #backward pass
```

Backward pass

In order to calculate the delta of the output layer for the backward pass, the code first multiplies the error by the derivative of the sigmoid function that will be applied to the output in a later step. The weight between the hidden node and the output layer, along with the delta for the output layer, are then used to determine the delta for the hidden node. the output of the hidden node after applying the sigmoid function's derivative.

```
#backward pass
output_delta = (Y[row]-output)*sigmoid_derivative(output)
for nodes in range(hidden_layer):
    delta= output_weight[nodes]*output_delta*sigmoid_derivative(sumofoutput[nodes])
    #update weight
```

Update weight

It then updates the weight using the deltas and the learning parameter.

At this stage, I implemented two extensions- momentum and annealing.

Annealing was made using a function while the momentum was just embedded into the code using variables ; change_in_weight and change_in_output_weight to store the change in weight values.

As of now my inputs are hardcoded, as the number of columns being read as to be equal to the input size. I also processed my data outside of my code.

```
        #update weight
        for input_weights in range(input_size):
            x = hidden_layer_weights[input_weights][nodes]
            hidden_layer_weights[input_weights][nodes] += learning_parameter*delta*X[row][input_weights] + alpha*change_in_weight[input_weights][nodes]
            change_in_weight[input_weights][nodes] = hidden_layer_weights[input_weights][nodes] - x
        x = output_weight[nodes]
        output_weight[nodes] += learning_parameter*output_delta*sumofoutput[nodes] + alpha*change_in_output_weight[nodes]
        change_in_output_weight[nodes] = output_weight[nodes] - x
error = (error / len(X))**0.5
errors.append(error)
```

Extensions

Momentum

4. Update the weights: $w^*_{i,j} = w_{i,j} + \rho \delta_j u_i$

- If the previous weight change was :

$$\Delta w_{i,j} = w^*_{i,j} - w_{i,j}$$

$\Delta w_{i,j}$ is the change made to the weights compared with the previous weights at the last step

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

momentum

Let's keep the momentum going in this direction

Typical value for α = 0.9

The slides were a guide for me to do momentum.

I set alpha to 0.9 and implemented the change of weights

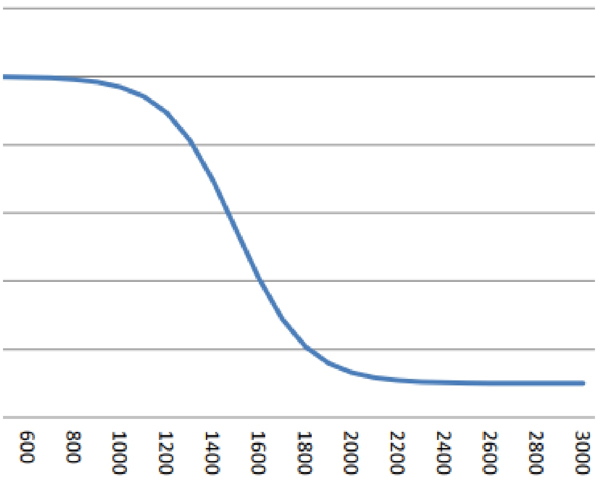
Code:

```
#update weight
for input_weights in range(input_size):
    x = hidden_layer_weights[input_weights][nodes]
    hidden_layer_weights[input_weights][nodes] +=
learning_parameter*delta*X[row][input_weights] +
alpha*change_in_weight[input_weights][nodes]
    change_in_weight[input_weights][nodes] =
hidden_layer_weights[input_weights][nodes] - x
x = output_weight[nodes]
output_weight[nodes] += learning_parameter*output_delta*sumofoutput[nodes]
+ alpha*change_in_output_weight[nodes]
change_in_output_weight[nodes] = output_weight[nodes] - x
```

Annealing

Annealing made using this function

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



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

```
def sim_annealing(max_epoch,currentepoch):
    p=0.01
    q=0.1
    expo= 10-((20 * currentepoch)/max_epoch)
    return p + (q - p)*(1 - (1 /(1+np.exp(expo))))
```

Which I then called in the forward pass, setting the learning parameter to call the function on the epochs and the current epoch.

I set the end parameter to 0.01 and the start parameter to 0.1

Code:

```
learning_parameter = sim_annealing(iterations, i)
```

Weight Decay

Backpropagation employs weight decay as a regularisation approach to lessen the complexity of a neural network model and avoid overfitting. Backpropagation aims to minimise the loss function, which evaluates the discrepancy between the anticipated output and the actual output, by optimising the weights and biases of the neural network.

- Weight decay adds a 'penalty' term to the error function:

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

(*upsilon, omega*)

$$\delta_O = \underbrace{(C - u_O)}_{E} f'(S_O)$$

Becomes:

$$\delta_O = (C - u_O + \upsilon\Omega) f'(S_O)$$

101 © Christian Dawson Loughborough University

Weight decay

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

- We can penalize large weights by choosing:

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

- For a network with n weights/biases, w_i (i=*1 to n*).

This was my hardest to implement logically but I finally got it to work.

It works by adding a penalty term to the loss/error function. This penalty term is then multiplied by a regularization parameter which controls the strength of the regularization

```
v = 1/(learning_parameter * (1 + i))
#this contains weight decay
o = (sum(x**2 for row in hidden_layer_weights for x in row) + sum(x**2 for
x in hidden_bias) + sum(x**2 for x in
        output_weight) + output_bias**2) / (1 + hidden_layer * (2 +
input_size))
output_delta = (Y[row]-output + v*o)*sigmoid_derivative(output)
for nodes in range(hidden_layer):
```

I

My whole algorithm was done iteratively at first , but I then decided to fit my code into functions for the following reasons:

- Functions are reusable
- Modularity
- Efficiency

These are some of the functions I used

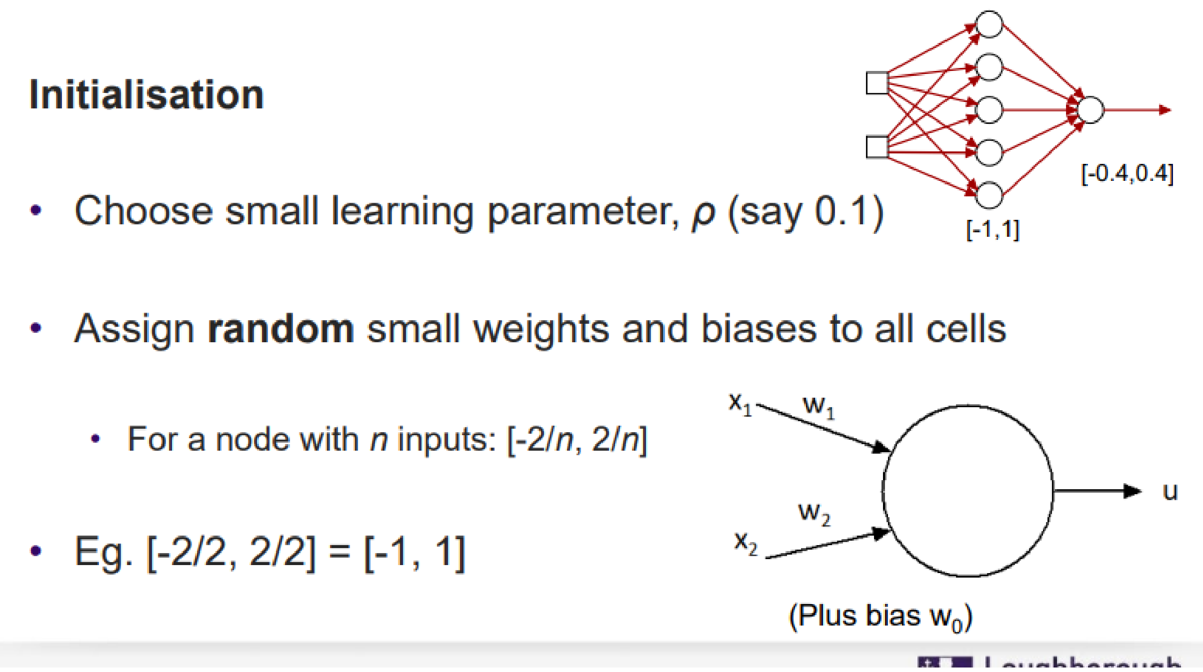
```
def initialize_weights(input_size, hidden_layer):
    hidden_bias = np.random.uniform(-2/input_size, 2/input_size,
size=hidden_layer)
    output_bias = random.uniform(-2/input_size, 2/input_size)
    hidden_layer_weights = np.random.uniform(-2/input_size, 2/input_size,
size=(input_size, hidden_layer))
    output_weight = np.random.uniform(-2/hidden_layer, 2/hidden_layer,
size=hidden_layer)
    return hidden_bias, output_bias, hidden_layer_weights, output_weight
```

This function initialises the weights as the name suggests.

Input size : is the number of inputs in the dataset

Hidden_layer= the number of nodes in the hidden layer

Hidden bias= contains the biases of the hidden layers. This was gotten from this formula



With n as the inputs.

The rest of the variables are self explanatory with the names

Another function I created was

```
def train_model(X, Y, hidden_layer_weights, output_weight, hidden_bias,
output_bias, input_size, hidden_layer,
                learning_parameter, iterations, alpha):
```

the train model function takes multiple inputs, with X and Y representing input and output respectively. The function works by first initialising variables to keep track of the errors and changes in weights during training. It then loops over a number of iterations (Epochs) and for each iteration, it loops over each training example in the dataset. This is when the **forward pass** comes in and after that it perform the backward pass and updates the weights and biases. Finally, the function computes the error for the entire dataset after each iteration and appends it to a list of errors. At the end of all iterations, the function returns the list of errors, as well as the updated weights and biases of the neural network model.

```
def predict(X, hidden_layer_weights, output_weight, hidden_bias,
output_bias, hidden_layer):
    predictions = []
    for row in range(len((X))):
        # Forward pass
        sumofoutput = []
        for node in range(hidden_layer):
            sum_ = hidden_bias[node] + np.dot(X[row],
hidden_layer_weights.T[node])
            sumofoutput.append(sigmoid(sum_))
        sumofoutput = np.array(sumofoutput)
        ouputsum = output_bias + np.dot(sumofoutput.reshape(1, -1),
output_weight.reshape(-1, 1))
        output = sigmoid(ouputsum[0])
        predictions.append(output)
    return predictions
```

The function **predict** takes a set of input data **X** and the learned weights and biases from a trained neural network model as inputs. It uses the learned weights and biases to make predictions for each input in **X** using a forward pass through the neural network.

During the forward pass, the function first applies the learned hidden layer weights and biases to the input data to produce a set of hidden layer activations. It then applies the learned output weight and bias to the hidden layer activations to produce the final output prediction.The function returns a list of predictions, one for each input in **X**.

Last one is main function

```
def main():
# Load data
    filepath = r'C:\Users\idowu\Desktop\AI DATA\Validation.xlsx'
    X, Y = load_data(filepath)
# Initialize weights and biases
    input_size = 5
    hidden_layer = 4
    hidden_bias, output_bias, hidden_layer_weights, output_weight =
initialize_weights(input_size, hidden_layer)

# Train the model
```

```
    learning_parameter = 0.01
    iterations = 1000
    alpha = 0.9
    errors, hidden_layer_weights, output_weight, hidden_bias, output_bias =
train_model(X, Y, hidden_layer_weights,
        output_weight, hidden_bias, output_bias, input_size, hidden_layer,
learning_parameter, iterations, alpha)

# Make predictions
    predictions = predict(X, hidden_layer_weights, output_weight,
hidden_bias, output_bias, hidden_layer)

# Plot errors
    plot_errors(errors)

# Print minimum error
    min_error_index = np.argmin(errors)
    min_error = errors[min_error_index]
    print("Minimum error:", min_error, "at iteration", min_error_index)
if __name__ == "__main__":
    main()
```

this is the main entry point of the program. It reads from the excel file using the load_data function and initialises the weights and biases of the neural network model. In this case I used 5.

It then calls the train model function to train the network with a learning rate of 0.01 and 1000 iterations.

Predictions call the predict function and stores the predictions.

I also made a plot_error function which just plots the error graphs.

Training and network Selection

I will be training my ANN based on 3 different models, this includes the base model, Annealing , momentum and Weight decay.

Standard/Base algorithm

Model	Hidden nodes	Learning rate	Epoch	Validation Error	Test error
Base	4	0.01	1000	0.04194	0.0671
Base	4	0.01	1500	0.03689	0.03243
Base	4	0.01	2500	0.02246	0.03053
Base	4	0.01	3000	0.01967	0.03048

Annealing

Model	Hidden nodes	Learning rate	Epoch	Validation Error	Test error
Annealing	4	0.01	1000	0.01855	0.02794
Annealing	4	0.01	1500	0.01771	0.02889
Annealing	4	0.01	3000	0.01676	0.02585
Annealing	4	0.1	3000	0.01683	0.02747
Annealing	8	0.01	3000	0.01725	0.026688

Momentum

Momentum	4	0.01	1000	0.018220	0.02694
Momentum	4	0.01	2000	0.01810	0.02646
Momentum	4	0.01	2500	0.01706	0.02561
Momentum	4	0.1	3000	0.01682	0.02601

Weight Decay

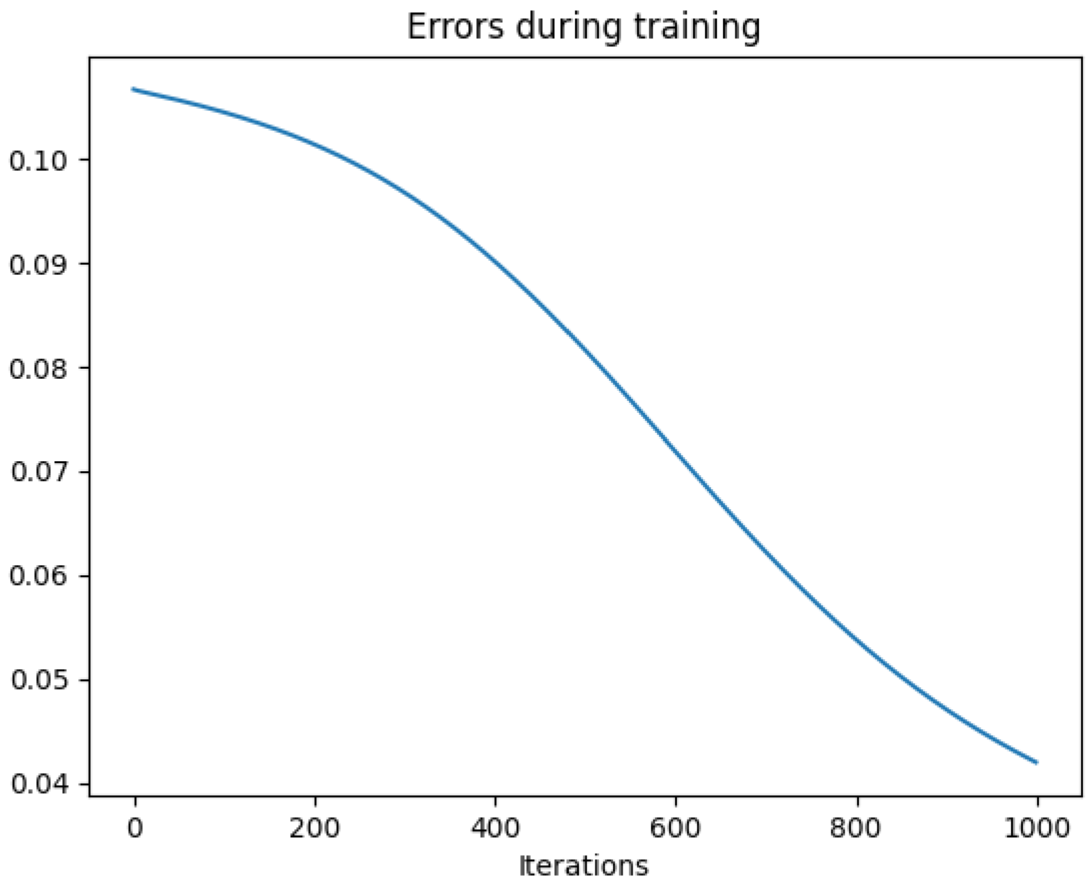
Model	Hidden nodes	Learning rate	Epoch	Validation Error	Test error
Weight Decay	4	0.1	1000	0.02065	0.03172
Weight decay	4	0.1	1500	0.01923	0.02847
Weight Decay	8	0.1	1500	0.01649	0.02178

Base: For all learning rates and hidden nodes, the validation and test errors dropped as the number of epochs rose.

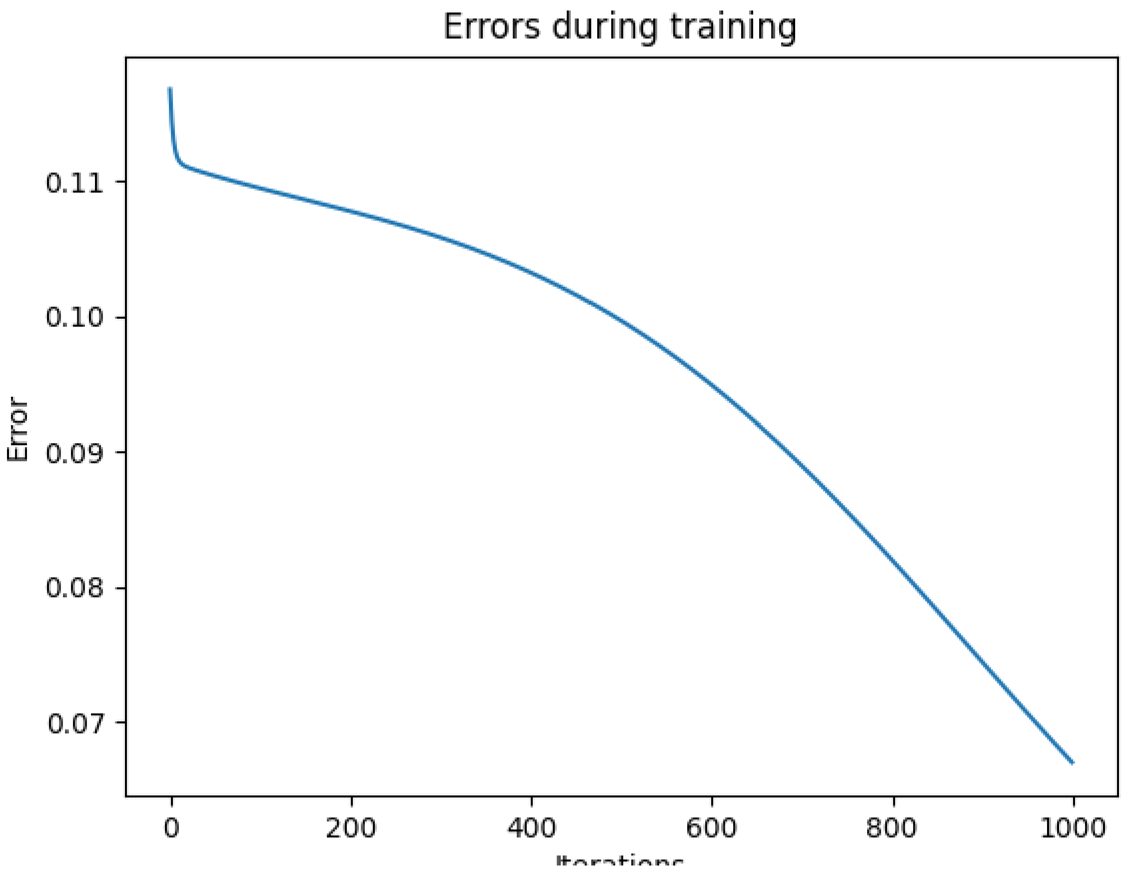
• With 4 hidden nodes, a learning rate of 0.01, and 3000 epochs, the best outcome was attained.

At 1000, epoch and learning rate of 0.01.

Validation graph

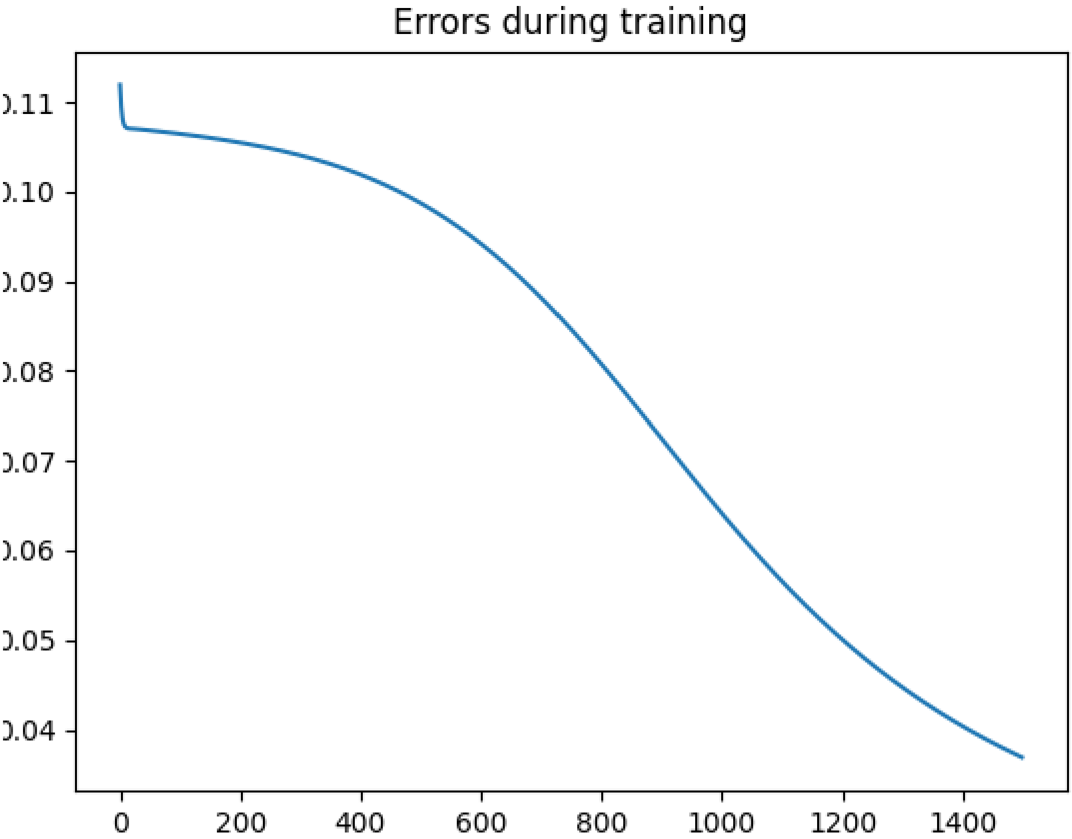


Test data



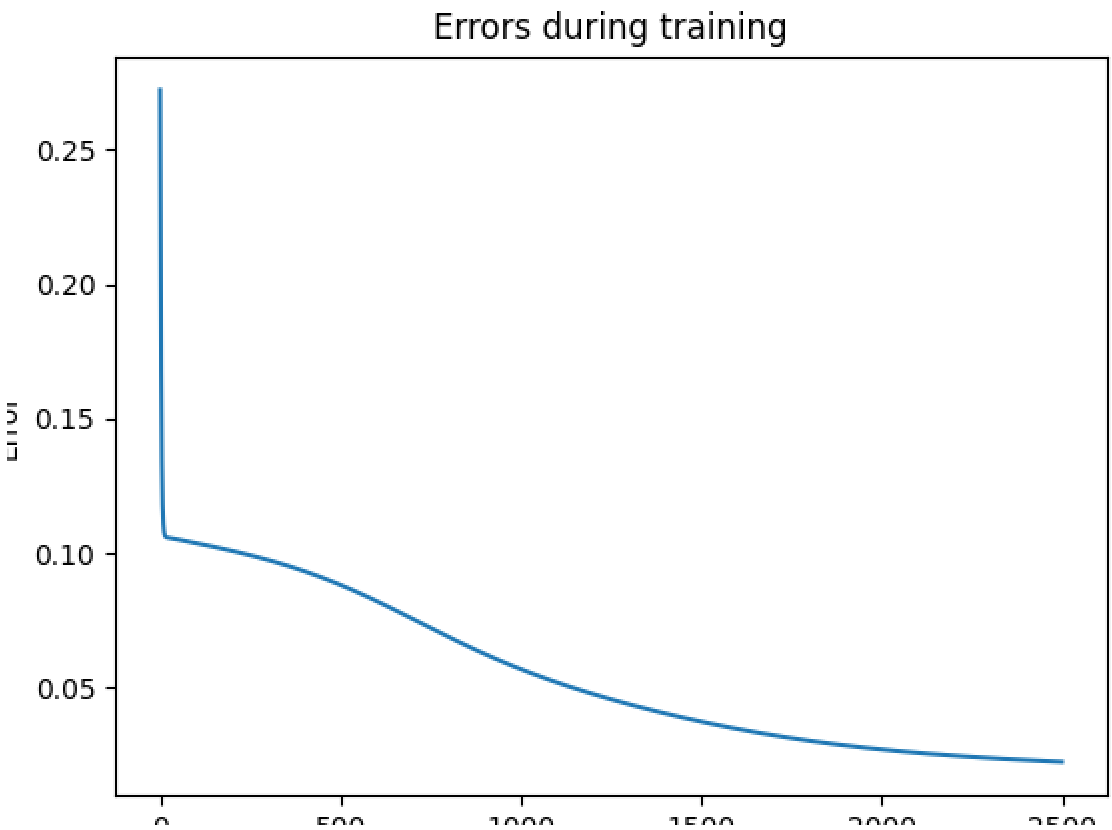
1500 epochs

validation

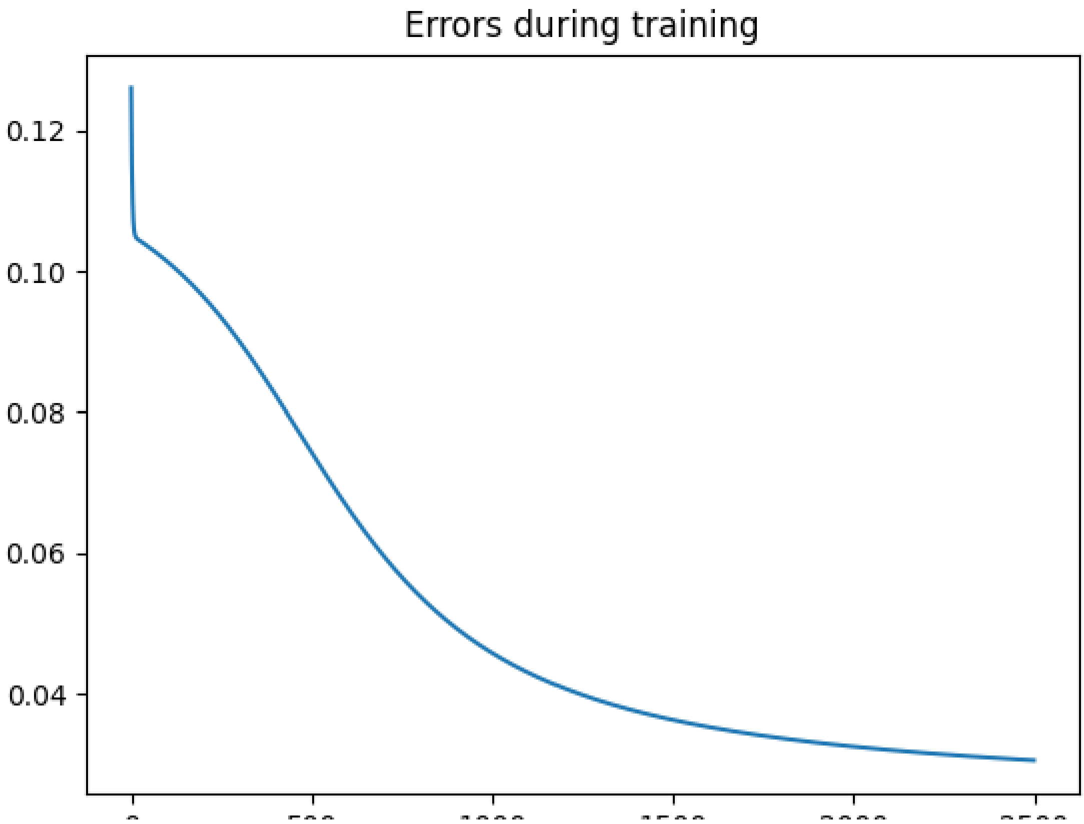


2500 epochs

Validation

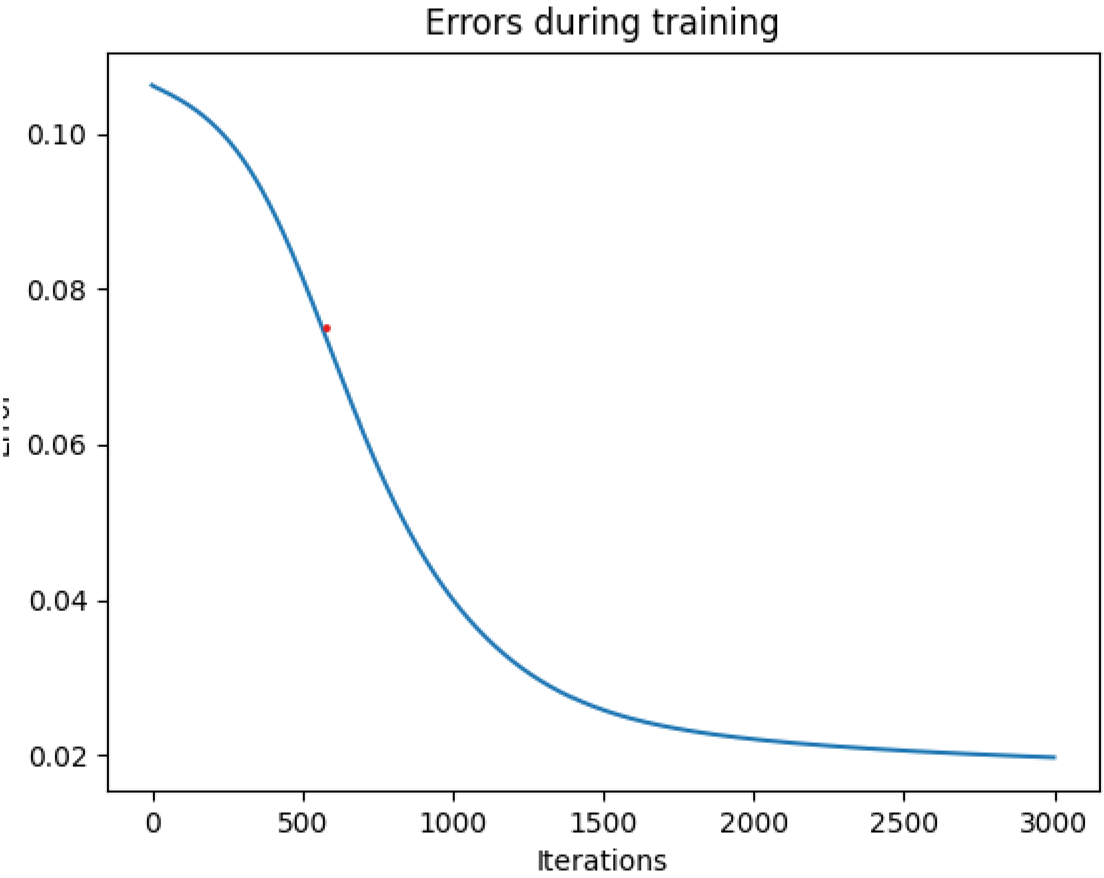

Errors during training
0.25
0.20
0.15
0.10
0.05

Test data

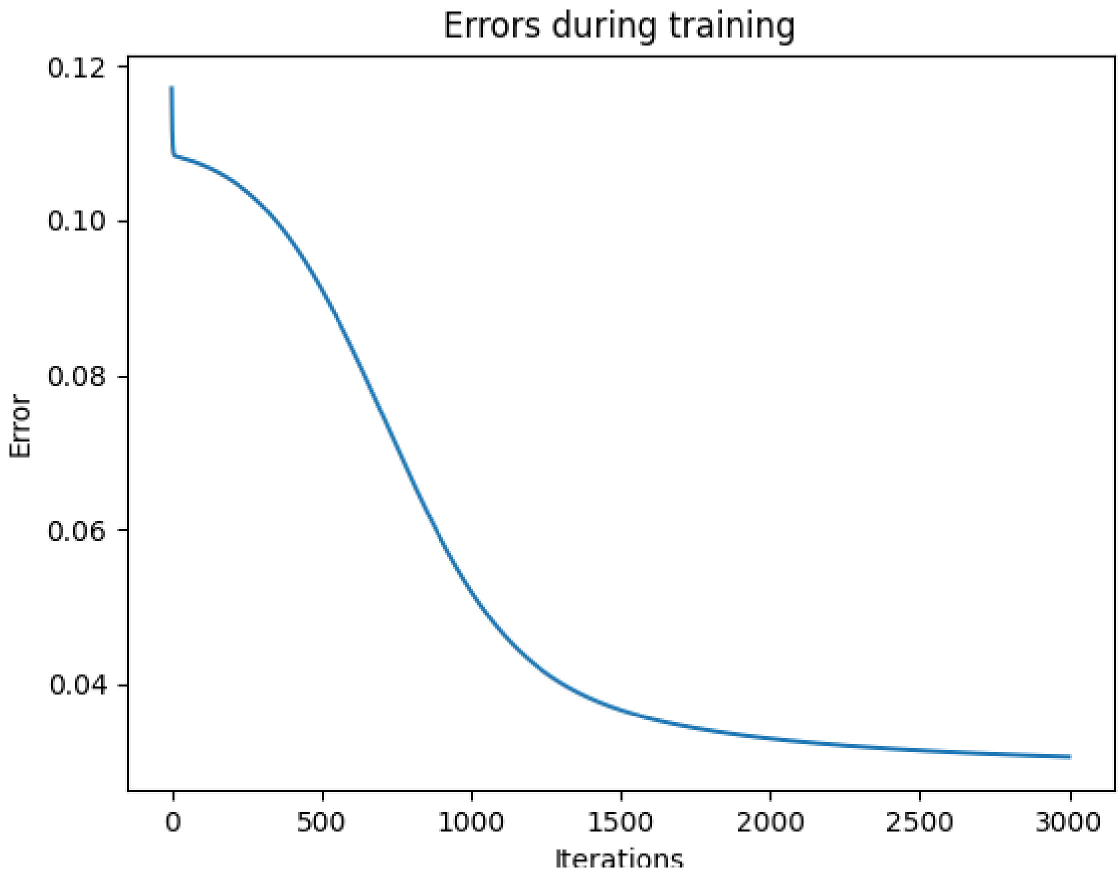

Errors during training
0.12
0.10
0.08
0.06
0.04

3000 epoch

validation



Test

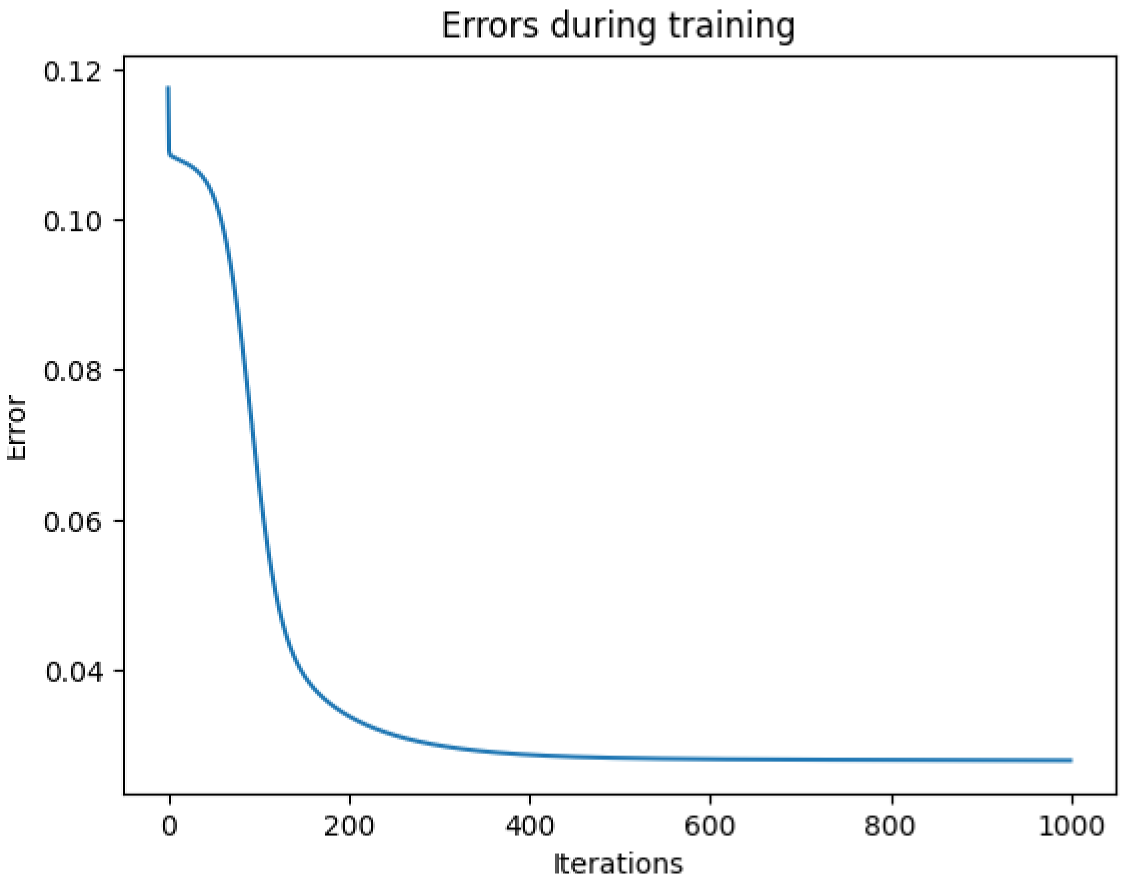


Annealing: For all learning rates and hidden nodes, the validation and test errors dropped as the number of epochs rose.

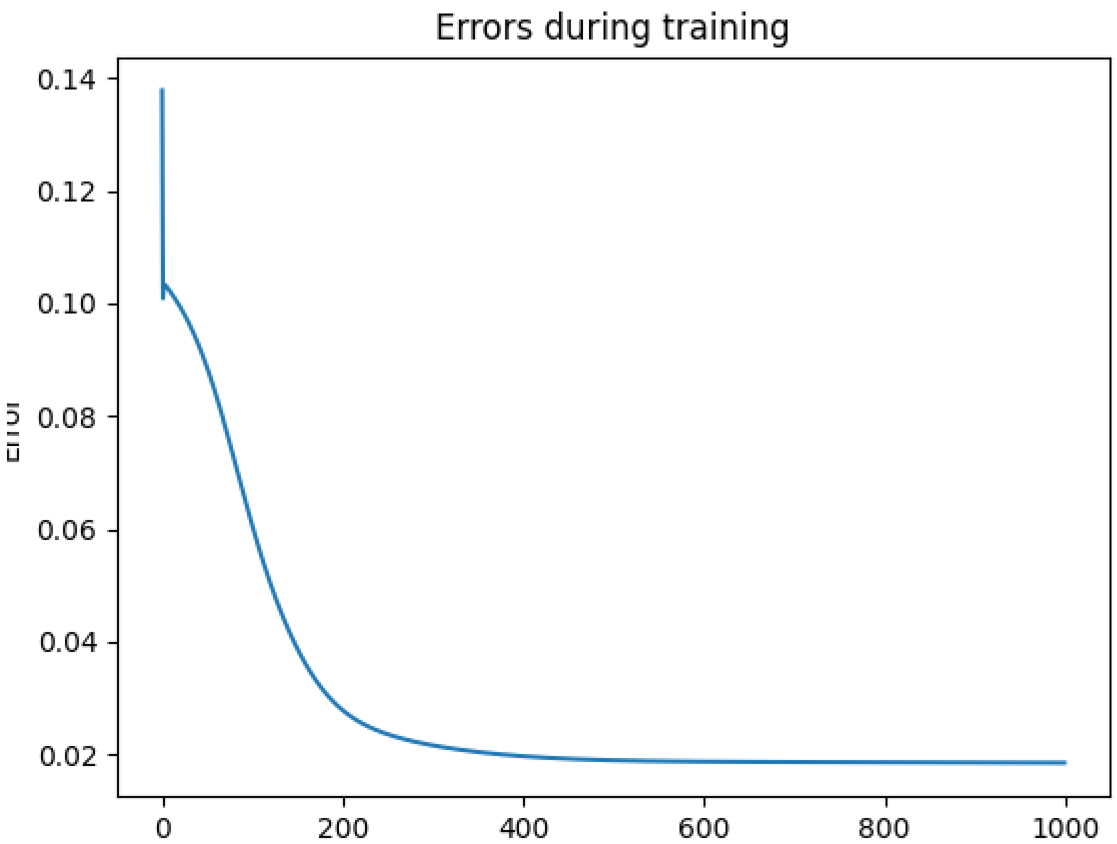
• The best result was obtained with 4 hidden nodes, a learning rate of 0.01, and 3000 epochs.

• A learning rate of 0.01 consistently performed better than 0.1 for all hidden nodes.

Test


Errors during training
0.12
0.10
0.08
0.06
0.04
Error
0
200
400
600
800
1000
Iterations

Validation


Errors during training
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0
200
400
600
800
1000

Weight Decay: For all learning rates and hidden nodes, the test and validation errors dropped as the number of epochs rose.

• Using 8 hidden nodes, a learning rate of 0.1, and 1500 epochs, the best option was obtained.

Momentum: For all learning rates and hidden nodes, the validation and test errors decreased as the number of epochs rose.

• A learning rate of 0.1, 4 hidden nodes, and 3000 epochs produced the best results.

Overall: For the majority of approaches, a learning rate of 0.01 generally outperformed 0.1.

• For all strategies, the validation and test errors generally decreased as the number of epochs rose.

The majority of the experiments employed 4 hidden nodes.

Evaluation of the Data

The table above contains a full analysis of the outcomes from the various models and setups. The following are the main findings and conclusions drawn from the table:

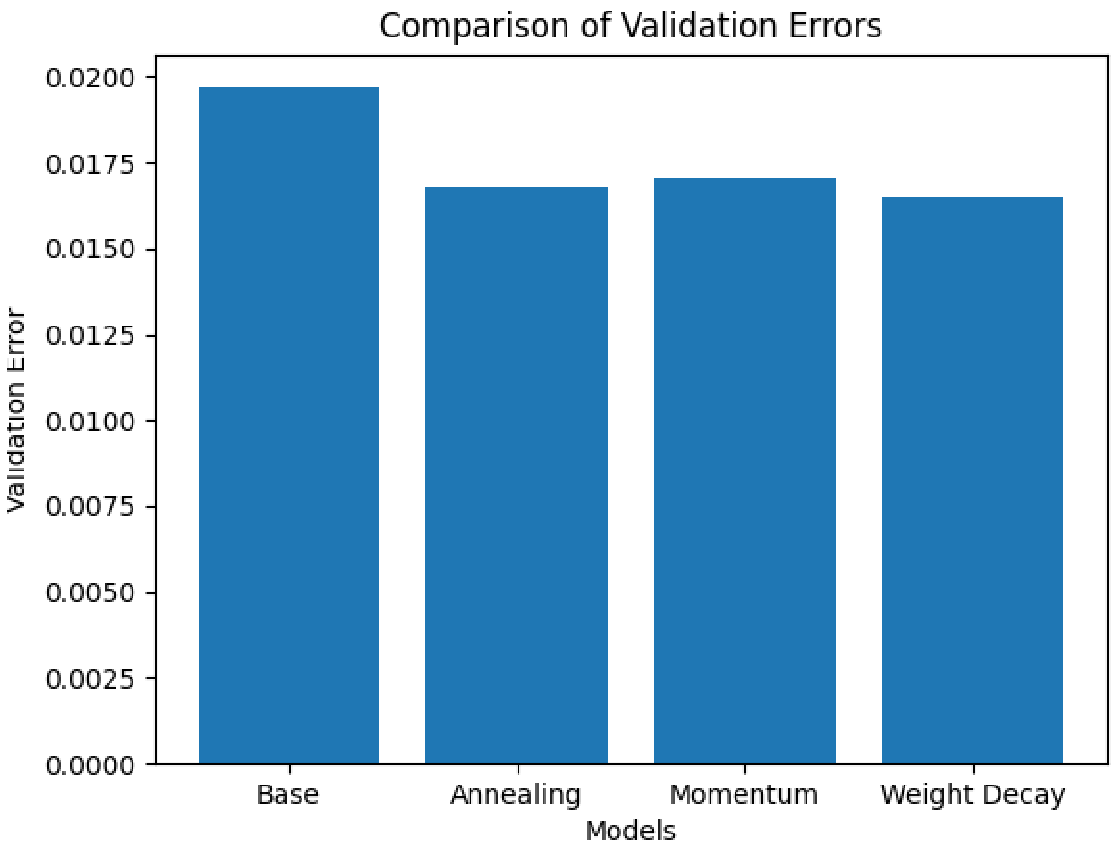
The base model, which used 4 hidden nodes, a learning rate of 0.01, and 3000 epochs to attain its lowest test error of 0.03048 without modifying the base backpropagation procedure. Even while this performance is acceptable, the altered models showed much better outcomes.

Over a range of settings, the Simulated Annealing model consistently offered lower validation and test errors than the original model. This result indicates that the performance of the model can be greatly enhanced by including simulated annealing in the training method. The Simulated Annealing model performed best when 4 hidden nodes, a learning rate of 0.01, and 3000 epochs were used, yielding a test error of 0.02585.

When compared to the base model, the Momentum model performed significantly better, attaining reduced errors in a variety of settings. The performance of the model seems to benefit from the training algorithm's addition of momentum. The Momentum model performed best with 4 hidden nodes, 0.1 learning rate, and 3000 epochs, yielding a test error of 0.02601.

When paired with more hidden nodes, the Weight Decay model showed a variety of performance enhancements. This study implies that weight decay can be a useful regularisation strategy, especially when combined with the right hidden node arrangement. The best performance for the Weight Decay model was achieved with a configuration of 8 hidden nodes, a learning rate of 0.1, and 1500 epochs, resulting in a test error of 0.02178. This is the lowest test error among all tested configurations.

Evaluation of final model



Comparison of Validation Errors: Looking at the graph that displays the validation mistakes for the Basic, Annealing, Momentum, and Weight Decay models, the following is what I can see:

The Base model, Annealing model, Momentum model, and Weight Decay model, in that order, have the highest validation errors. According to the validation dataset, it appears that the Weight Decay model outperforms the others in terms of generalisation.

The Weight Decay model generalises to unseen data the best since it has the lowest test error. The Annealing and Momentum models' test errors are extremely close, indicating that they performed similarly in terms of generalisation on the test dataset. The Basic model does the least well when it comes to generalising to new data because it has the biggest test error.

Model Selection: The Weight Decay model appears to be the best-performing model based on the examination of both validation and test errors, as it has the lowest errors on both datasets. It would be wise to decide to employ or implement the Weight Decay model in the future.

Model Selection: The Weight Decay model seems to be the most appropriate one based on the examination of both validation and test mistakes.

Overall

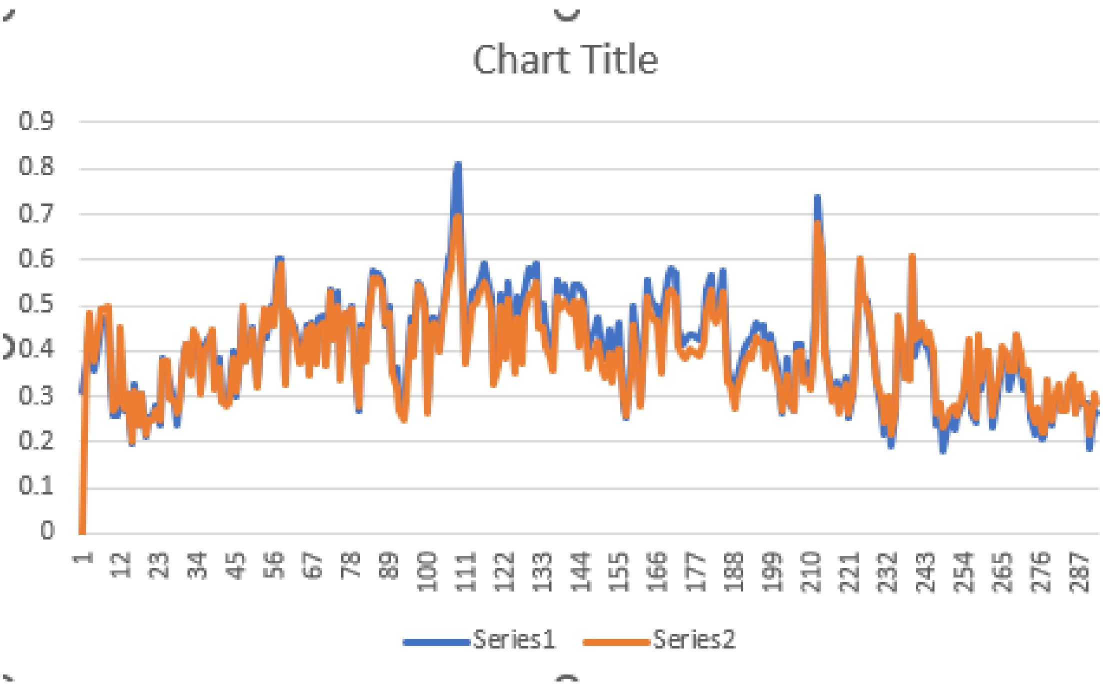
Measures for Overall Performance

Validation error and test error were my two main performance criteria for evaluating the effectiveness of the various models. These mistakes show the mismatch between the predicted values of the model and the actual target values. Because they show a lower discrepancy between anticipated and actual values, lesser errors are a sign of greater model performance. These metrics are crucial for evaluating the performance of various models and setups on a given dataset by comparing them.

In conclusion, based on the performance metrics and analysis of the results, the Weight Decay model with the specified configuration is the best choice among the tested models.

Comparing predicted values

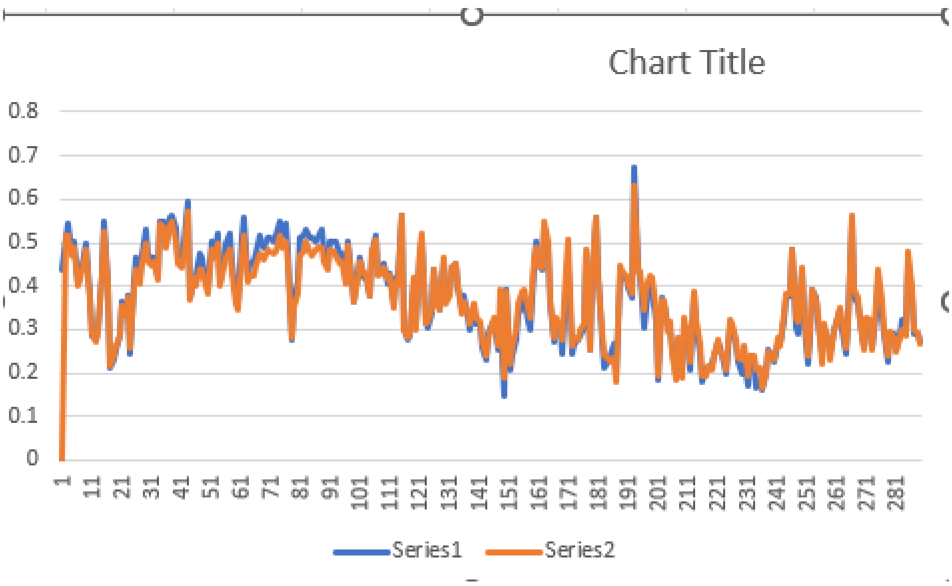
Test data



Orange= predicted panE

Blue= original PanE

Validation data



Orange = predicted Pan E

Blue = original PanE

Overall this looks like a good model, the predicted panE is quite close to the original.

Code

I used this mostly

```
import random
import numpy as np
import pandas as pd


import matplotlib.pyplot as plt


# data prep
df = pd.read_excel(r'C:\Users\codoii\Downloads\Book1.xlsx', usecols=[0, 1,
2, 3, 4, 5])




X = df.iloc[:, :-1].values #this contains the inputs , I read from specific
columns
Y = df.iloc[:, -1:].values
Y = [y[0] for y in Y]

# formulas
# Define the sigmoid function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Define the derivative of the sigmoid function
def sigmoid_derivative(x):
    return x * (1 - x)



'''step 1: initialisation'''
# create variables needed

# number of hidden nodes
hidden_layer = 4

# number of input nodes
input_size = 5

# number of output nodes
output_node = 1
#bias

hidden_bias=np.random.uniform(-2/input_size,2/input_size,
```

```
size=hidden_layer)

#output bias
output_bias=random.uniform(-2/input_size,2/input_size)


learning_parameter = 0.1
iterations = 1000 #epochs
alpha = 0.9# this is the alpha for the momentum

'''step 1b: add weights to layers'''

# weight for hidden layer
hidden_layer_weights = np.random.uniform(-2/input_size,2/input_size,
size=(input_size, hidden_layer))

# weight for output layer
output_weight = np.random.uniform(-2/hidden_layer,2/hidden_layer,
size=hidden_layer)

'''step 2: forward passes'''

errors = []
''' This creates a nested list (matrix) with input_size + 1 rows and
hidden_layer columns. Each element is initialized
to 0. This matrix will be used to store the changes in weights between the
input layer and the hidden layer during the
training process. '''
change_in_weight = [[0 for _ in range(hidden_layer)] for _ in
range(input_size + 1)]  #this is change in weight for momentum
change_in_output_weight = [0 for _ in range(hidden_layer)] # change in
output weight

# Define the simulated annealing function
def sim_annealing(max_epoch,currentepoch):
    p=0.01
    q=0.1
    expo= 10-((20 * currentepoch)/max_epoch)
    return p + (q - p)*(1 - (1 /(1+np.exp(expo))))

for i in range(iterations): #each iteration of epochs
    error = 0
    for row in range(len((X))):
        # forward pass
        sumofoutput  = []
        for node in range(hidden_layer): #going through size of hidden
layers
            sum_ = hidden_bias[node] + np.dot(X[row],
hidden_layer_weights.T[node])
            sumofoutput.append(sigmoid(sum_))
        ouputsum = output_bias + np.dot(sumofoutput, output_weight)
        output = sigmoid(ouputsum)
        learning_parameter = sim_annealing(iterations,i) # calling the
annealing function

        error += (Y[row]-output)**2

        #backward pass
        v = 1/(learning_parameter * (1 + i)) # this is for the weight decay

        o = (sum(x**2 for row in hidden_layer_weights for x in row) +
```

```
sum(x**2 for x in hidden_bias) + sum(x**2 for x
        in output_weight) + output_bias**2) / (1 + hidden_layer * (2 +
input_size)) #weight decay
        output_delta = (Y[row]-output + v*o)*sigmoid_derivative(output)
#adds weight decay to the output delta
        for nodes in range(hidden_layer):
            delta=
output_weight[nodes]*output_delta*sigmoid_derivative(sumofoutput[nodes])
            #update weight
            for input_weights in range(input_size):
                x = hidden_layer_weights[input_weights][nodes]
                #update hiddenlayer weights
                hidden_layer_weights[input_weights][nodes] +=
learning_parameter*delta*X[row][input_weights] +
alpha*change_in_weight[input_weights][nodes]
                #change in weight is used for momentum
                change_in_weight[input_weights][nodes] =
hidden_layer_weights[input_weights][nodes] - x
            x = output_weight[nodes]
            output_weight[nodes] +=
learning_parameter*output_delta*sumofoutput[nodes] +
alpha*change_in_output_weight[nodes] #momentum
            change_in_output_weight[nodes] = output_weight[nodes] - x
#momentum
    error = (error / len(X))**0.5 # mean squared error
    errors.append(error)
    if not i %100:  # i used this as a timer while it is loading
        print(1 + i // 100)

# ------------------------------------

predictions = []
'''I didnt use functions much so i ended up reusing code but this was to
allow me store the predictions
i then wrote the predictions into an excel file called results'''
for row in range(len((X))):
        # forward pass
        sumofoutput  = []
        for node in range(hidden_layer):
            sum_ = hidden_bias[node] + np.dot(X[row],
hidden_layer_weights.T[node])
            sumofoutput.append(sigmoid(sum_))
        ouputsum = output_bias + np.dot(sumofoutput, output_weight)
        output = sigmoid(ouputsum)
        predictions.append(output)

# ----------------------------------

pd.DataFrame(predictions).to_excel("results.xlsx")
plt.plot(errors) # ploting the mean squared error against the iterations
plt.title('Errors during training')
plt.xlabel('Iterations')
plt.ylabel('Error')
plt.show()
```

code in a more functional structure

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


# Load and prepare data
def load_data(filepath):
    df = pd.read_excel(filepath, usecols=[0, 1, 2, 3, 4, 5])
    X = df.iloc[:, :-1].values
    Y = df.iloc[:, -1:].values
    Y = [y[0] for y in Y]
    return X, Y


# Sigmoid function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))


# Sigmoid derivative
def sigmoid_derivative(x):
    return x * (1 - x)





# Initialize weights and biases
def initialize_weights(input_size, hidden_layer):
    hidden_bias = np.random.uniform(-2/input_size, 2/input_size,
size=hidden_layer)# this is in the range[-2/n,2/n]
    output_bias = random.uniform(-2/input_size, 2/input_size)
    hidden_layer_weights = np.random.uniform(-2/input_size, 2/input_size,
size=(input_size, hidden_layer))
    output_weight = np.random.uniform(-2/hidden_layer, 2/hidden_layer,
size=hidden_layer)
    return hidden_bias, output_bias, hidden_layer_weights, output_weight


# Training the model
def train_model(X, Y, hidden_layer_weights, output_weight, hidden_bias,
output_bias, input_size, hidden_layer,
                learning_parameter, iterations, alpha):
    errors = []
    change_in_weight = [[0 for _ in range(hidden_layer)] for _ in
range(input_size + 1)]
    change_in_output_weight = [0 for _ in range(hidden_layer)]

    for i in range(iterations):
        error = 0
        for row in range(len((X))):
            # Forward pass
            sumofoutput = []
            for node in range(hidden_layer):
                sum_ = hidden_bias[node] + np.dot(X[row],
hidden_layer_weights.T[node])
                sumofoutput.append(sigmoid(sum_))
            ouputsum = output_bias + np.dot(sumofoutput, output_weight)
            output = sigmoid(ouputsum)
```

```
            error += (Y[row]-output)**2

            # Backward pass
            v = 1/(learning_parameter * (1 + i))
            o = (sum(x**2 for row in hidden_layer_weights for x in row) +
sum(x**2 for x in hidden_bias)
             + sum(x**2 for x in output_weight) + output_bias**2) / (1 +
hidden_layer * (2 + input_size))

            output_delta = (Y[row] - output + v * o) *
sigmoid_derivative(output)
            for nodes in range(hidden_layer):
                delta=
output_weight[nodes]*output_delta*sigmoid_derivative(sumofoutput[nodes])
                # Update weights
                for input_weights in range(input_size):
                    x = hidden_layer_weights[input_weights][nodes]
                    hidden_layer_weights[input_weights][nodes] +=
learning_parameter*delta*X[row][input_weights] +
alpha*change_in_weight[input_weights][nodes]
                    change_in_weight[input_weights][nodes] =
hidden_layer_weights[input_weights][nodes] - x
                x = output_weight[nodes]
                output_weight[nodes] +=
learning_parameter*output_delta*sumofoutput[nodes] +
alpha*change_in_output_weight[nodes]
                change_in_output_weight[nodes] = output_weight[nodes] - x
        error = (error / len(X)) ** 0.5
        errors.append(error)
    return errors, hidden_layer_weights, output_weight, hidden_bias,
output_bias

def predict(X, hidden_layer_weights, output_weight, hidden_bias,
output_bias, hidden_layer):
    predictions = []
    for row in range(len((X))):
        # Forward pass
        sumofoutput = []
        for node in range(hidden_layer):
            sum_ = hidden_bias[node] + np.dot(X[row],
hidden_layer_weights.T[node])
            sumofoutput.append(sigmoid(sum_))
        sumofoutput = np.array(sumofoutput)
        ouputsum = output_bias + np.dot(sumofoutput.reshape(1, -1),
output_weight.reshape(-1, 1))
        output = sigmoid(ouputsum[0])
        predictions.append(output)
    return predictions

def plot_errors(errors):
    plt.plot(errors)
    plt.title('Errors during training')
    plt.xlabel('Iterations')
    plt.ylabel('Error')
    plt.show()

def main():
# Load data
    filepath = r'C:\Users\idowu\Desktop\AI DATA\Validation.xlsx'
    X, Y = load_data(filepath)
# Initialize weights and biases
```

```
    input_size = 5
    hidden_layer = 4
    hidden_bias, output_bias, hidden_layer_weights, output_weight =
initialize_weights(input_size, hidden_layer)

# Train the model
    learning_parameter = 0.01
    iterations = 1000
    alpha = 0.9
    errors, hidden_layer_weights, output_weight, hidden_bias, output_bias =
train_model(X, Y, hidden_layer_weights,
        output_weight, hidden_bias, output_bias, input_size, hidden_layer,
learning_parameter, iterations, alpha)

# Make predictions
    predictions = predict(X, hidden_layer_weights, output_weight,
hidden_bias, output_bias, hidden_layer)

# Plot errors
    plot_errors(errors)

# Print minimum error
    min_error_index = np.argmin(errors)
    min_error = errors[min_error_index]
    print("Minimum error:", min_error, "at iteration", min_error_index)
if __name__ == "__main__":
    main()
```