AI coursework

Contents

[Data pre-processing 1](#_Toc130334555)

[Implementation 9](#_Toc130334556)

[Extensions 12](#_Toc130334557)

[Training and network Selection 18](#_Toc130334558)

[Evaluation of final model 28](#_Toc130334559)

[Comparing predicted values 30](#_Toc130334560)

## 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

Chart

Description automatically generated

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

Chart

Description automatically generated

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 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

Chart, scatter chart

Description automatically generated

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 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

Chart, scatter chart

Description automatically generated

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 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 |

Table

Description automatically generated

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

Graph for T after standardising

Chart

Description automatically generated

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:

Text, letter

Description automatically generated

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

Rectangle

Description automatically generated with medium confidence

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

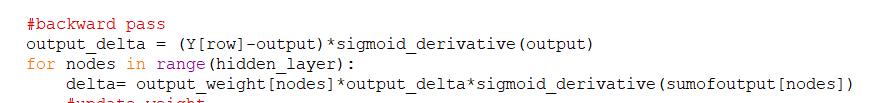
For the forward pass, the code calculates the weighted sum of the inputs to each hidden nodes and after that it apples the sigmoid function to each hidden node’s weighted sum and generates a set of hidden node output values ( sumofoutput). After thatm the code calcutes the weight sum of the hidden node 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.

Graphical user interface

Description automatically generated with medium confidence

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.

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.

A picture containing diagram

Description automatically generated

## Extensions

Momentum

Text

Description automatically generated

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

Text

Description automatically generated with medium confidence

Text

Description automatically generated

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.

Graphical user interface, application

Description automatically generated

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

Diagram

Description automatically generated

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**

Chart, line chart

Description automatically generated

**Test data**

Chart, line chart

Description automatically generated

1500 epochs

**validation**

Chart

Description automatically generated

2500 epochs

**Validation**

Chart, line chart

Description automatically generated

Test data

Chart

Description automatically generated

3000 epoch

validation

Chart

Description automatically generated

Test

Chart, histogram

Description automatically generated

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

Chart, histogram

Description automatically generated

Validation

Chart, histogram

Description automatically generated

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

Chart, bar chart

Description automatically generated

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**

Chart, line chart

Description automatically generated

Orange= predicted panE

Blue= original PanE

**Validation data**

Chart, line chart

Description automatically generated

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

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*()*