**AI Coursework Report**

# Introduction

I used Java to create the MLP, along with the Apache POI API for reading from and writing to the excel dataset. I have used an object oriented approach, with arraylists to store the weights and biases of each node, including the output node, as well as arraylists to store the errors given. It is possible to change the amount of nodes in a single hidden layer by changing the amount of generated arrays in the arraylist.

# Data Pre-processing

## Cleaning the Data

First I noticed that the date column was in a format with no clear splits between day, month and year. I used Excel’s “Text to Column” feature to convert every value in the date column to an actual date, making it possible for me to analyse the data with graphs.

I subsequently made graphs comparing each of the 5 predictors with PanE, as well as a graph of PanE against date, to see if there were any visible anomalies. Below are the graphs before cleaning:

Chart, scatter chart

Description automatically generatedChart, scatter chart

Description automatically generatedChart, scatter chart

Description automatically generatedChart, scatter chart

Description automatically generatedChart, scatter chart

Description automatically generatedChart

Description automatically generated

There were several examples of anomalous data, such as in the graph of pan evaporation against temperature, where the temperature was apparently around 180 degrees Celsius on a specific day. Also, when looking through the dataset, there were several clear anomalies, such as values that were letters instead of numbers. Some anomalies were clearly caused by mistypes, such as the 180 degress celsius example most likely actually being 18.0 degrees celsius. Others, such as the datapoints where letters were given, had to be removed.

## Standardising the Data

After cleaning the data, I needed to standardise each datapoint before using it in my MLP algorithm. To do this, I used the Apache POI library in Java, which allows a user to read and write to excel files. I used it to calculate the maximums and minimums of each column excluding the date, and subsequently standardise each value in each row and add it to a Data object. This Data object is as follows:

public class Data {

float T;

float W;

float SR;

float DSP;

float DRH;

float PanE;

//constructor, getters and setters not included

}

I made an arraylist called “dataset” that would store all the data objects, which would eventually be cycled through each epoch. I also initialised the Excel workbook and sheet instance that I would be taking the data from:

ArrayList<Data> dataset = new ArrayList<Data>();

File file = new File("C:\\Users\\laptop-a\\OneDrive - Loughborough University\\dataset.xlsx"); //creating a new file instance

FileInputStream f = new FileInputStream(file); //gets data from excel

//create instance of workbook

XSSFWorkbook wb = new XSSFWorkbook(f);

XSSFSheet sheet = wb.getSheetAt(0); // identify specific sheet that data is taken from

I then created variables that would hold the maximum and minimum values of each column. These values were calculated in Excel and copied over to these variables:

float Tmax = 28.9f;

float Tmin = 7.2f;

float Wmax = 1089.7f;

float Wmin = 96.1f;

float SRmax = 743.2f;

float SRmin = 78.4f;

float DSPmax = 102.8f;

float DSPmin = 100.2f;

float DRHmax = 95f;

float DRHmin = 10f;

float PanEmax = 1.28f;

float PanEmin = 0.07f;

after this, each row in the Excel sheet was examined and the cell contents taken. Each value was standardised and put into the object, which was then put into the arraylist using the following code:

for (Row row: sheet) {

try {

Data dataRow = new Data(0f,0f,0f, 0f,0f, 0f);

//gets contents of each cell in the row

Cell Tcell = row.getCell(0);

Cell Wcell = row.getCell(1);

Cell SRcell = row.getCell(2);

Cell DSPcell = row.getCell(3);

Cell DRHcell = row.getCell(4);

Cell PanEcell = row.getCell(5);

//converts contents of each cell into a string

String TcellValue = dataFormatter.formatCellValue(Tcell);

String WcellValue = dataFormatter.formatCellValue(Wcell);

String SRcellValue = dataFormatter.formatCellValue(SRcell);

String DSPcellValue = dataFormatter.formatCellValue(DSPcell);

String DRHcellValue = dataFormatter.formatCellValue(DRHcell);

String PanEcellValue = dataFormatter.formatCellValue(PanEcell);

//converts contents of each string variable into a float

float T = Float.*parseFloat*(TcellValue);

float W = Float.*parseFloat*(WcellValue);

float SR = Float.*parseFloat*(SRcellValue);

float DSP = Float.*parseFloat*(DSPcellValue);

float DRH = Float.*parseFloat*(DRHcellValue);

float PanE = Float.*parseFloat*(PanEcellValue);

//System.out.println(DRHcell+" : "+DRHcellValue+" : "+DRH);

//standardises each variable

T = (float) (0.8\*((T-Tmin)/(Tmax-Tmin))+0.1);

W = (float) (0.8\*((W-Wmin)/(Wmax-Wmin))+0.1);

SR = (float) (0.8\*((SR-SRmin)/(SRmax-SRmin))+0.1);

DSP = (float) (0.8\*((DSP-DSPmin)/(DSPmax-DSPmin))+0.1);

DRH = (float) (0.8\*((DRH-DRHmin)/(DRHmax-DRHmin))+0.1);

PanE = (float) (0.8\*((PanE-PanEmin)/(PanEmax-PanEmin))+0.1);

//put standardised variables inside array

dataRow.setT(T);

dataRow.setW(W);

dataRow.setSR(SR);

dataRow.setDSP(DSP);

dataRow.setDRH(DRH);

dataRow.setPanE(PanE);

//adds to the dataset arraylist

dataset.add(dataRow);

}

The “dataset” arraylist is then used by the MLP to access the training data. We will see how the MLP does this in the following section.

# Algorithm Implementation

Initially I created a basic multi-layer perceptron algorithm that would take in the 5 predictors for each datapoint in the dataset and output the predicted value of PanE. Afterwards, I added improvements and modifications such as momentum, bold driving, annealing and weight decay.

## Initial Algorithm Construction

I started the learning parameter at a value of 0.1 and I initialised a 2D array called “biases”, which would contain randomised weights and biases for each node.

//p = learning parameter

float p = 0.1f;

//creates a 2d array containing weights and biases of each node float[][]biases=newfloat[][]{NeuralNetwork.*generateW*(2),NeuralNetwork.*generateW*(5),NeuralNetwork.*generateW*(5)};

I created a function called “generateW” that would generate an array containing a certain number of random weights and a bias given a certain number of inputs. These weights and biases would be in the interval [-2/n,2/n] where n is the number of inputs given. Using this function, the 2D array would have a format like {{0.7,-0.3,0.2},{-0.2,0.57,-0.31,0.1,0.05},{0.9,-0.3,0.2,-0.07,0.29}}. The first array indicates the output node, with a bias of 0.7 and input weights of -0.3 and 0.2. the next 2 arrays indicate the respective biases and weights of the 2 nodes in the hidden layer. This is visualised on the diagram:

Diagram

Description automatically generated

The code for the function that generates the random weights and biases is as follows:

public static float[] generateW(int inputs){

float min = -1\*(2/(float)inputs);

float max = 2/(float)inputs;

//use below code to get random numbers within interval

float[] randomWeights = new float[inputs+1];

for (int i = 0; i<=inputs; i++) {

Random r = new Random();

float random = min + r.nextFloat() \* (max - min);

randomWeights[i] = random;

}

return randomWeights;

}

The following code initialises an array to store the values of the activation function for each node, an array to store the error for each node and a counter for the number of epochs so far. There is also an arraylist that stores the average error of each epoch:

float[] activations = {0,0,0};

float[] errors = {0,0,0};

int epochCount = 0;

ArrayList<Float> epochError = new ArrayList<Float>();

I also created a function that would take an arraylist of floats as input, and output the root mean square value. I used this function to calculate the average error in each epoch:

public static float rootMeanSquare(ArrayList<Float> array){

int n = array.size();

float square = 0;

float mean = 0;

float sqrt = 0;

//calculates sum of squares of each item

for(Float item : array){

square += Math.*pow*(item, 2);

}

mean = (square / (float) n);

sqrt = (float)Math.*sqrt*(mean);

return sqrt;

}

for the main MLP, I used a while loop that conducts 10000 epochs. In each epoch, the foreach loop goes through each datapoint, does a forwards pass and then a backwards pass, updating the weights and biases arrays:

while (epochCount<10000) {

ArrayList<Float> PanEerror = new ArrayList<Float>();

for (Data d : dataset) {

float T = d.getT();

float W = d.getW();

float SR = d.getSR();

float DSP = d.getDSP();

float DRH = d.getDRH();

float PanE = d.getPanE();

float[] inputs = {T,W,SR,DSP,DRH};

//do forward pass

//cycles through nodes and calculates activation function value

for (int i = 1; i<biases.length; i++) {

float S = biases[i][0]+(biases[i][1]\*T)+(biases[i][2]\*W)+(biases[i][3]\*SR)+(biases[i][4]\*DSP)+(biases[i][5]\*DRH);

float U = (float) (1/(1+Math.*exp*(-1\*S)));

activations[i] = U;

}

//find output node activation function value

float S = biases[0][0]+(biases[0][1]\*activations[1])+(biases[0][2]\*activations[2]);

float U = (float) (1/(1+Math.*exp*(-1\*S)));

activations[0] = U;

//calculate percentage error and add to arraylist

float errorValue = (float)Math.*abs*(PanE-activations[0])/(PanE);

PanEerror.add(errorValue);

//do backwards pass

//error for output node

errors[0] = (float)(PanE-activations[0])\*activations[0]\*(1-activations[0]);

//error for node 1

errors[1] = (float)(biases[0][1]\*errors[0])\*activations[1]\*(1-activations[1]);

//error for node 2

errors[2] = (float)(biases[0][2]\*errors[0])\*activations[2]\*(1-activations[2]);

//update weights and biases of output

biases[0][0] = (float)biases[0][0]+(p\*errors[0]\*1);

biases[0][1] = (float)biases[0][1]+(p\*errors[0]\*activations[1]);

biases[0][2] = (float)biases[0][2]+(p\*errors[0]\*activations[2]);

//update weights and biases of hidden layer

for (int i = 1; i<biases.length; i++) {

biases[i][0] = biases[i][0]+(p\*errors[i]\*1);

for (int j = 1; j<biases[i].length; j++) {

//j-1 since inputs[0] corresponds to biases[i][1] and so forth

biases[i][j] = biases[i][j]+(p\*errors[i]\*inputs[j-1]);

}

}

}

System.***out***.println(epochCount);

//calculate the root mean square error of the epoch given the arraylist containing the percentage errors of each datapoint

epochError.add(NeuralNetwork.*rootMeanSquare*(PanEerror));

epochCount++;

}

//print out root mean square errors

for (Float error : epochError) {

System.***out***.println(String.*format*("%.4g%n", error);

}

At the end, the program outputs the contents of epochError into a new Excel sheet, making it easy to view the data and create graphs using the error:

XSSFSheet outputSheet = wb.createSheet("output");

int rowCount = 0;

for (Float error : epochError) {

//System.out.print(String.format("%.4g%n", error));

Row row = outputSheet.createRow(++rowCount);

Cell cell = row.createCell(0);

cell.setCellValue((float) error);

}

try (FileOutputStream outputStream = new FileOutputStream(file)) {

wb.write(outputStream);

}

## Modifications to the Algorithm

### Changing the Number of Hidden nodes

Changing the number of hidden nodes in the layer is done by changing the array of weights and biases. The output node is indicated to have 3 inputs, and an extra array of random weights and biases is created using the generateW() function:

float[][] biases = new float[][]{NeuralNetwork.*generateW*(3),NeuralNetwork.*generateW*(5),NeuralNetwork.*generateW*(5),NeuralNetwork.*generateW*(5)};

### Bold Driver

One of the improvements I made to the program was adding a bold driver, so that the learning rate could change depending on how the errors were trending. Every 1000 epochs, the MLP would check if the last epoch’s error was larger than the error before it. If it was larger, then the learning rate would decrease, else it would increase.

if (epochCount % 1000 == 0) {

if (epochError.get(epochError.size()-1)>epochError.get(epochError.size()-2)) {

p = p\*0.7f;

}else {

p = p\*1.05f;

}

}

### Annealing

I applied the following formula once per epoch to the learning rate:

p = (float) (p+(0.01-p)\*(1-(1/(1+Math.*exp*(10-(20\*epochCount/10000))))));

### Weight Decay

Once per epoch, the upsilon value would be set:

float upsilon = (float)(1/(p\*epochCount));

Inside the main algorithm, before the new error values were calculated, the omega value would be calculated using the following:

float omega = 0;

int weightCount = 0;

for (float[] array : biases) {

for (float item : array) {

weightCount += 1;

omega += item \* item;

}

}

omega = (float)(omega/(2\*weightCount));

The product of upsilon and omega was then applied to every error calculation:

errors[0] = (float)(PanE-activations[0]+(upsilon\*omega))\*activations[0]\*(1-activations[0]);

//error for node 1

errors[1] = (float)(biases[0][1]\*errors[0]+(upsilon\*omega))\*activations[1]\*(1-activations[1]);

//error for node 2

errors[2] = (float)(biases[0][2]\*errors[0]+(upsilon\*omega))\*activations[2]\*(1-activations[2]);

# Training the Algorithm

## Training the basic MLP without modifications

Training the basic MLP 100 times for 10000 epochs resulted in an average rms error of 0.04874. over 10000 epochs, the graph looked like this:

Table

Description automatically generated

Evidently, the error decreases rapidly to around 0.05 and does not decrease very much beyond that point. After seeing this, I decided to use a smaller amount of epochs:

A picture containing application

Description automatically generated

## Training the MLP with modifications

### 3 nodes in the Hidden Layer

Training the 3 node hidden layer MLP 100 times for 10000 epochs resulted in an average rms error of 0.04310.

A picture containing application

Description automatically generated

### 4 nodes in the Hidden Layer

Training the 4 node MLP 100 times for 10000 epochs resulted in an average rms error of 0.04230.

Application

Description automatically generated with medium confidence

### 5 nodes in the Hidden Layer

Training the 5 node MLP 100 times for 10000 epochs resulted in an average rms error of 0.04160.

A picture containing text

Description automatically generated

### 6 nodes in the Hidden Layer

Training the 6 node MLP 100 times for 10000 epochs resulted in an average rms error of 0.04445.

A picture containing table

Description automatically generated

### 7 nodes in the Hidden Layer

Training the 7 node MLP 100 times for 10000 epochs resulted in an average rms error of 0.04501.

A picture containing application

Description automatically generated

### 8 nodes in the Hidden Layer

Training the 8 node MLP 100 times for 10000 epochs resulted in an average rms error of 0.04541.

Application

Description automatically generated with low confidence

### Bold Driver

According to my previous testing, having 5 nodes in the hidden layer was most effective, so I kept the MLP at 5 nodes.

After 100 runs of 10000 epochs each, the bold driver did not appear to improve the MLP’s error by much, giving an average error of 0.3821. However, after changing the increase to the learning rate to 30% instead of 5%, the MLP managed to get an average error of 0.03036.

A picture containing table

Description automatically generated

### Annealing

I also used 5 nodes in the hidden layer for this test. Annealing did not improve the performance of the MLP, with the average error rate after 100 runs being 0.04371.

Chart, line chart

Description automatically generated

### Weight Decay

Again I used 5 nodes. Using the weight decay code that I created, after 100 runs of 10000 epochs each, the average error was 0.03300, a considerable improvement.

### Combination

Using all 3 improvements, after 100 runs I managed to achieve an average error of 0.2423 with 5 nodes in the hidden layer, making it the most successful training so far.

Chart, line chart

Description automatically generated

# Evaluation

Changing the amount of hidden nodes was effective in improving the algorithm’s error.

Table

Description automatically generated with low confidenceChart, line chart

Description automatically generated

The data above clearly shows that having 5 hidden nodes produced the least average error. However, implementing further modifications to the algorithm also significantly improved the algorithm’s error. Testing at 5 hidden nodes, the bold driver managed to improve the error to 0.3821 and the weight decay improved the error to 0.03300. However, annealing did not improve the error at all, in fact making it worse, to 0.04371. Using all 3 in combination resulted in a low error of 0.2423. Compared to the basic 2 hidden node algorithm, this was an improvement of over 50%.

Overall, this MLP is quite accurate and fast. However, I would say that a lot of it has been hard coded, making it more difficult to change things like adding more hidden layers, changing the amount of inputs and outputs or changing the maximum and minimum values in the data set. I also could have made some more improvements, like experimenting more with a different number of hidden nodes along with implementing the bold driver, annealing and weight decay, rather than keeping a constant 5 nodes in the hidden layer. Furthermore, it could have been interesting to experiment with a different activation function instead of using the Sigmoid function.

# Comparison to other Models

I used Excel’s LINEST multiple linear regression model as a comparison. I standardised the data set in excel using the same formula that I used in the MLP:

Text

Description automatically generated

I applied this rule to every column in the data set, resulting in a new, standardised data set. I then applied the LINEST function, with T, W, SR, DSP and DSH as x values and PanE as the y value.

Text

Description automatically generated

Table

Description automatically generated

As shown, the standard error for Y estimate is at 0.0251. at 10000 epochs, my MLP could not get an error below 0.04 for any number of hidden nodes without any other modifications. However, with the bold driver, annealing and weight decay modifications I was able to get an average error of 0.2423, which is slightly more accurate.

The error of the LINEST regression was 0.0251, compared to the MLP’s error of 0.02423, making the MLP better by about 3.5%.