# Report

**Implementation details**

The program is written using Java. I have chosen this language mostly because of my existing experience with it and because I'm also aware of other ANNs implemented using Java, ensuring that this language will be adequate.

Some data pre-processing has already been done on the data set using Excel, namely removal of erroneous and outlier data. Reading the data from the .xlsx (Excel) file, standardization, and splitting the data set have been done inside the program.

Libraries used:

- Apache POI (Poor Obfuscation Implementation) – gives facilities to read, write and manipulate .xlsx (Excel) files.
- java.io – used to access .xlsx (Excel) data set file
- java.util – common utility library, used specifically for randomness and collections.

**How to use program**

The parameters of the model and used improvements can be customized inside the config.txt file:

config.txt - Notepad

File Edit Format View Help

```
ANN_SHAPE = 10, 8
MOMENTUM = 0.9
MAX_EPOCHS = 100000
START_STEP_SIZE = 0.1
END_STEP_SIZE = 0.01
MAX_STEP_SIZE = 0.5
MIN_STEP_SIZE = 0.001
USE_MOMENTUM = true
USE_BOLD_DRIVER = false
USE_ANNEALING = true
USE_WEIGHT_DECAY = false
USE_BATCH_LEARING = false
```

For example, ANN_SHAPE = 10, 8 means the neural network will have 10 neurons in the fist hidden layer and 8 neurons in the second hidden layer (there is strictly one output neuron).

The data set is represented in the DataSet.xlsx file. The final column of will be treated as the sole predictand, every other column will be treated as an individual predictor. Any rows not containing only numeric data will be ignored. This means the program can be used to train/evaluate a model with any number of inputs (predictors) and one output (predictand).

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| 1 | T | W | SR | DSP | DRH | PanE |
| 2 | 13.5 | 245.5 | 215.6 | 101.8 | 64 | 0.21 |
| 3 | 15 | 350.8 | 290.4 | 101.8 | 69 | 0.27 |
| 4 | 14.3 | 310.7 | 242.6 | 101.8 | 73 | 0.21 |
| 5 | 13.5 | 595.4 | 97.5 | 101 | 87 | 0.16 |
| 6 | 12.8 | 380.8 | 216.6 | 101.9 | 67 | 0.24 |
| 7 | 13.3 | 404.5 | 246 | 101.8 | 69 | 0.28 |
| 8 | 12 | 302.9 | 303.6 | 102.3 | 61 | 0.28 |
| 9 | 12.5 | 334.8 | 269.1 | 102.3 | 47 | 0.36 |
| 10 | 11.7 | 245.1 | 321.4 | 102 | 46 | 0.34 |
| 11 | 13.7 | 251.5 | 320.1 | 101.8 | 37 | 0.4 |
| 12 | 14.3 | 266.5 | 319.3 | 101.7 | 36 | 0.46 |
| 13 | 13.1 | 258.7 | 236.2 | 101.6 | 68 | 0.21 |
| 14 | 13.6 | 303.8 | 169.6 | 101.3 | 75 | 0.17 |
| 15 | 12.1 | 546.6 | 225.1 | 100.7 | 56 | 0.42 |
| 16 | 7.6 | 429.6 | 331.1 | 101.2 | 32 | 0.5 |

Models can be saved to a file after it is trained. This file can then be read to recreate it's neural network for evaluation or execution. This file has the following format where the network shape is first specified followed by the weights and biases for each neuron.

example - Notepad

File Edit Format View Help

```
6,1,
0.3491707944983019,-1.399851727598531,1.7254480949306255,0.940216201128468,-3.51483104632943,2.4940870788100398,
-0.6322276229624743,-3.0715772114065403,-0.6714490592127134,1.8702616081157328,-1.2108311998146946,1.1408122280581419,
0.1940340882931291,-0.13062042806549895,-0.20469462404004035,-0.2085839518902245,0.1515714864946453,-0.2919054729303959,
1.3691229469468085,1.2543901057363418,-1.9340871619133215,-0.6319487073560941,-3.066065576770901,-0.6486826071100033,
-2.1401180737274728,-0.06104517916679003,-3.762755491075802,2.020416321443846,2.8003509138706746,2.366142301414271,
0.17902222703543033,0.32450119451241766,-0.48968281987889156,-0.11018924355117721,0.10141408378359786,-0.04482481850296127,
3.4818786907744386,-2.705754230583162,0.298065502183894,3.217392283052742,-4.764885026002697,0.6840728469352736,1.8685437558845814,
```

The output of the model is saved to the Output.xlsx file inside the ANN Models folder. This Excel file contains two sheets. The "Training Data" sheet displays the RMSE (measured again the validation set) and Step size against the current training Epoch. The "Evaluation Data" sheet displays the modelled output against each sample output. The "Execution Data" sheet displays the standardized modelled output generated from running the model on only inputs.

Use the command line "java -jar ANN_Implementation.jar" to execute the program.

When executing the program, the user is given 4 options. Option 1, 2, and 3 will ask for file names.

1) Train an new ANN on labelled data set?
2) Evaluate an existing ANN on labelled data set?
3) Execute existing ANN on unlabelled data set?
4) Save output and quit?

I have designed this program to be versatile and easy to use. Nothing has been hardcoded expect the number of predictands (strictly one). Parameters are easily customized, and the program can be used to train a model for any labelled data set with only one predictand, simply but changing the data in the Excel file.

I have ensured the program is cleanly coded, well structed, and commented. This report further documents specific parts of the program.

# Data pre-processing

Topics: Erroneous data, outlier data, exploring data set, standardization, splitting the data set, identifying predictors.

**Erroneous data**

There are some obvious erroneous data entries where a non-numerical value or nothing was recorded:

| Date | T | W | SR | DSP | DRH | PanE |
|---|---|---|---|---|---|---|
| 50287 | 17 | a | 627.6 | 101.4 | 66 | 0.62 |
| 60789 | 17.1 | 404.5 | ddd | 101.3 | 78 | 0.3 |
| 62290 | 19.5 | 465.6 | 599.7 | 101.2 | | 0.6 |
| 73190 | 22.1 | 463.3 | | 101.3 | 72 | 0.74 |

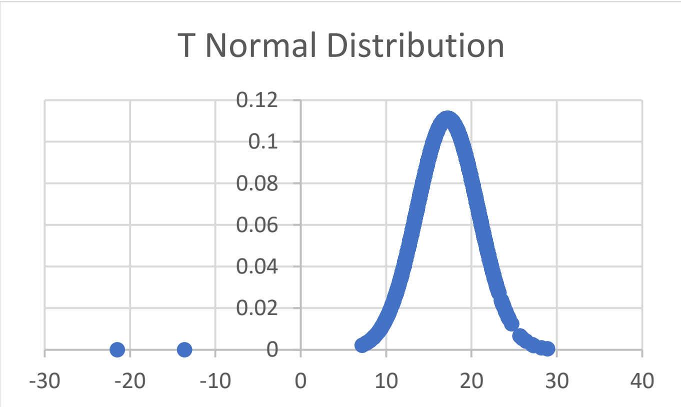
There are further data entries with obviously erroneous data such as the 180 °C value for T, mean daily temperature, or the -999 Langleys measurement for SR, solar radiation, which are both impossible:

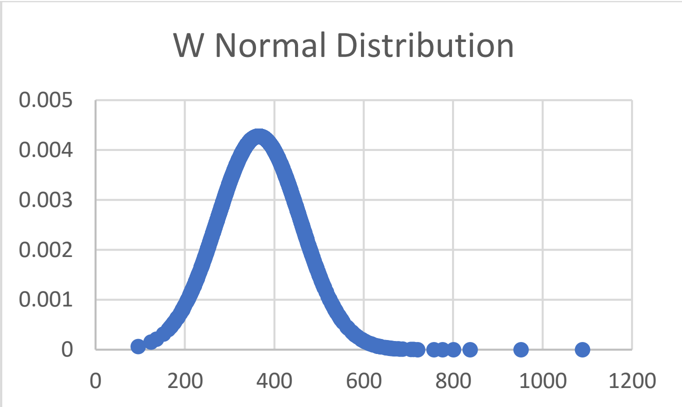
| Date | T | W | SR | DSP | DRH | PanE |
|---|---|---|---|---|---|---|
| 61387 | 180 | 483.3 | 597.3 | 101.7 | 74 | 0.59 |
| 92890 | 20.3 | 371.7 | -999 | 101.2 | 70 | 0.45 |
| 110190 | 17.3 | 442.3 | -999 | 101.4 | 62 | 0.44 |
| 110290 | 17.1 | 433.2 | -999 | 101.3 | 46 | 0.6 |

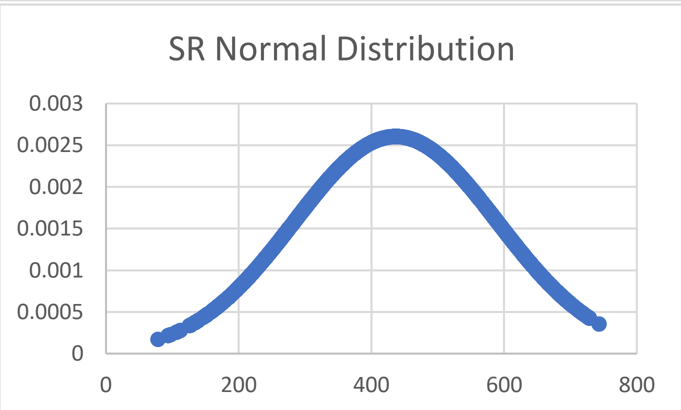
I have removed all these entries from the data set.

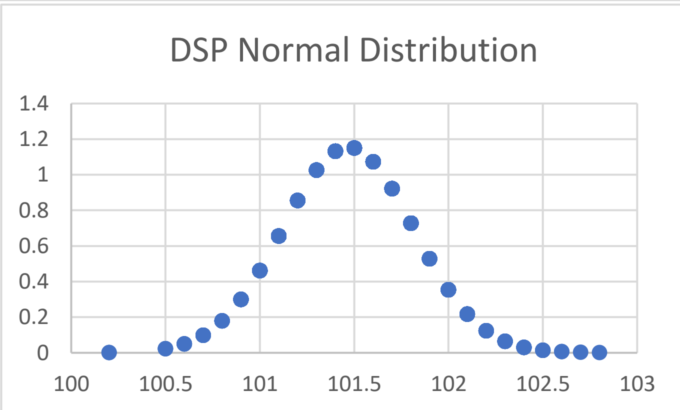
**Outlier data**

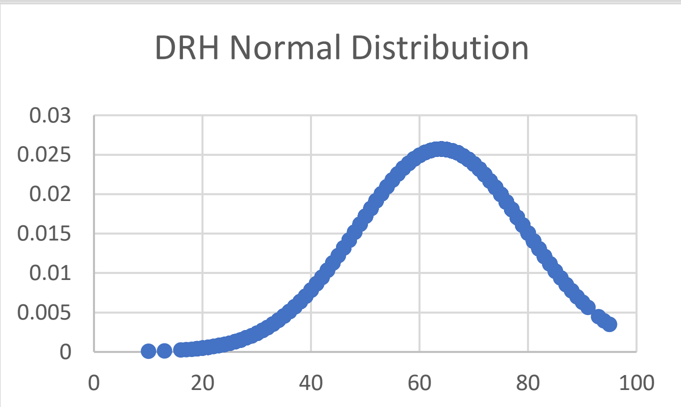
T (mean daily temperature), W (wind speed), SR (solar radiation), DSP (air pressure), DRH (humidity) and PanE (Pan Evaporation) are normally distributed. You can clearly see this from the normal distribution charts below:

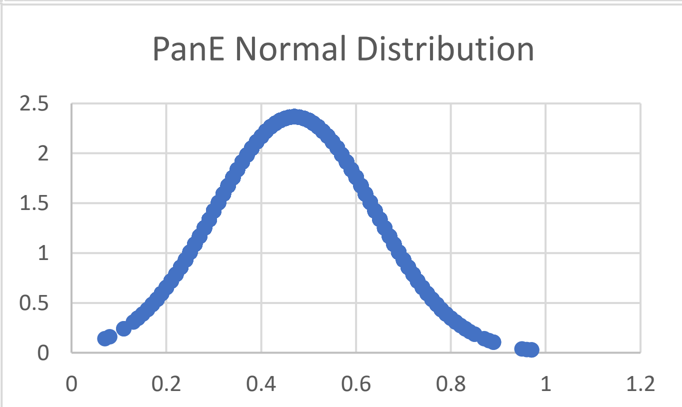


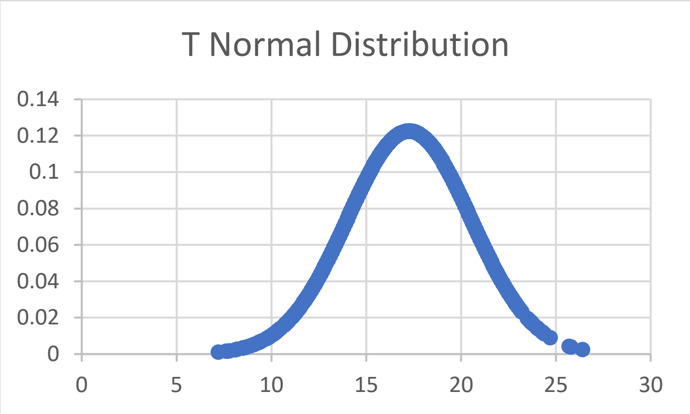



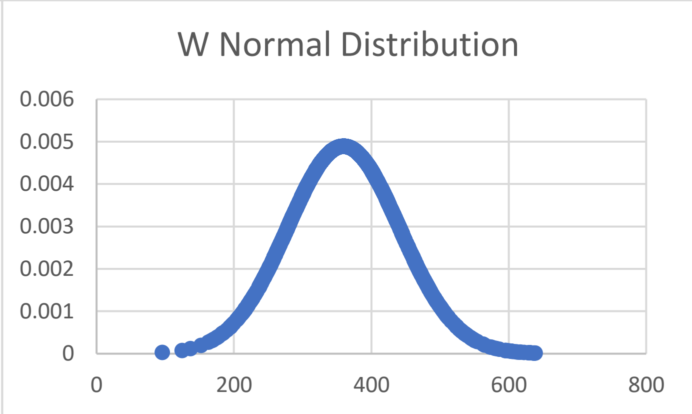
Some data points are clearly outliers (such as the negative values for T, mean daily temperature)

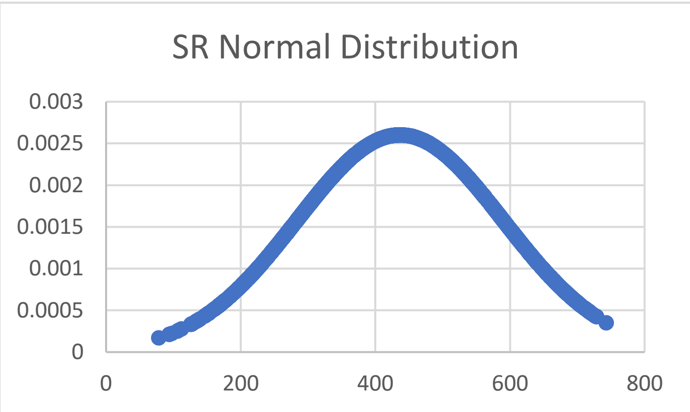
Outliers will be calculated as values that fall outside the range:

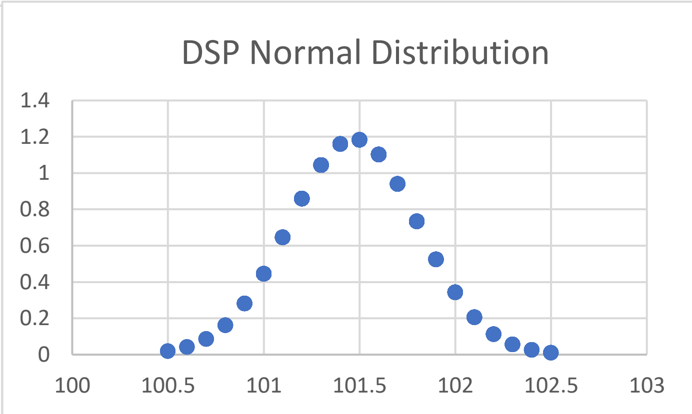
[mean + 3 * standard deviation, mean - 3 * standard deviation].

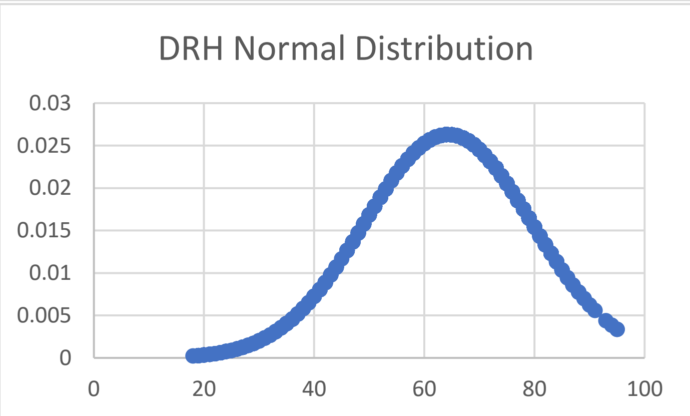
I will remove entries from the data set that contain at least one value from T, W, SR, DSP, DRH, and PanE where that value is outside said range. See graphs below showing normal distribution after outliers have been removed:

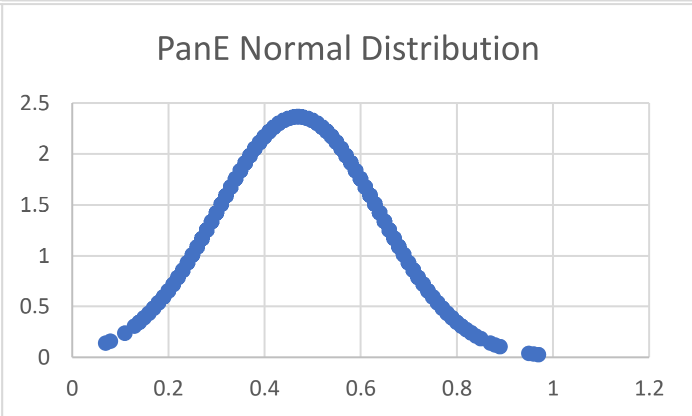



Clearly, the values T, W, SR, DSP, DRH, and PanE can now be better predicted by a normal distribution and have become significantly less skewed. Most importantly, outlier data has clearly been removed.

Removing erroneous and outlier data (data cleansing) was done using Excel and has removed 3.29% of the entries in the data set. Even if the outlier data was accurately recorded, since so few entries have been removed, the model will still be accurate.

### Exploring the data set

The data set is read once from the Excel file once using the code that follows. The number of inputs is based on the column count. Interacting with the Excel file is done using the Apache POI library.

```
final ArrayList<Sample> dataSet = new ArrayList<>();

Random rand = new Random();

//Read data set from DataSet.xlsx
FileInputStream inputFile = new FileInputStream(new File("DataSet.xlsx"));
XSSFWorkbook workbook = new XSSFWorkbook(inputFile);
XSSFSheet sheet = workbook.getSheetAt(0);
FormulaEvaluator evaluator = workbook.getCreationHelper().createFormulaEvaluator();

int sampleCount = sheet.getPhysicalNumberOfRows();
int inputSize = sheet.getRow(0).getPhysicalNumberOfCells();

//Extra information for each colum of data (used for data pre-processing)
final double[] minimums = new double[inputSize];
final double[] maximums = new double[inputSize];

//Read every row in sheet
for (int row = 0; row < sampleCount; row++) {
    //Set to false if a non-numeric value is detected in a sample
    boolean validSample = true;
    //Read row from data set
    double[] sampleData = new double[inputSize];
    for (int col = 0; col < inputSize; col++) {
        if (sheet.getRow(row).getCell(col) != null && evaluator.evaluateInCell(
                sheet.getRow(row).getCell(col)).getCellTypeEnum() == NUMERIC) {
            sampleData[col] = sheet.getRow(row).getCell(col).getNumericCellValue();
            //Calculate min and max values for input (column), used for standardization later
            minimums[col] = Math.min(minimums[col], sampleData[col]);
            maximums[col] = Math.max(maximums[col], sampleData[col]);
        } else {
            validSample = false;
            break;
        }
    }
    //Invalid samples are ignored
    if (!validSample) {
        continue;
    }
    //Convert each row of data into a Sample data object and add it to dataSet
    dataSet.add(new Sample(sampleData));
}
```

### Standardization

For the sake of avoiding floating point errors and avoiding certain inputs having more influence than others, every input value is standardized to the range [0.1,0.9] by the following methods:

```
private static double standardise(double value, double max, double min) {
    return 0.8 * ((value - min) / (max - min)) + 0.1;
}

private static double deStandardise(double value, double max, double min) {
    return ((value - 0.1) / 0.8) * (max - min) + min;
}
```

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

The values are standardized to the range [0.1,0.9] instead of [0,1] so that the model can predict values outside the range of the original data set.

The minimum and maximum value of each input (column) is calculated when the data set is first read from the Excel file. Every sample is then standardized as shown in the following code:

```
//Loop through data set
for (Sample sample : dataSet) {
    double[] sampleData = sample.getInputs();
    //Standardize each value between range [0.1, 0.9]
    for (int i = 0; i < sampleData.length; i++) {
        sampleData[i] = standardize(sampleData[i], maximums[i], minimums[i]);
    }
```

### Splitting the data set

About 20% of the data set is placed in the test set, about 20% placed in the validation set, and about 60% placed in the training set. The data is split randomly between these subsets because the data is already sorted by date. The time of year (date) likely affects the predicatand and therefore splitting the data non-randomly may lead to bias, for example the validation set may only contain entries from winter and the training set may only contain entries from the rest of the year leading to poor performance correctly predicting the validation set's predictands. Randomly splitting the data avoids this bias.

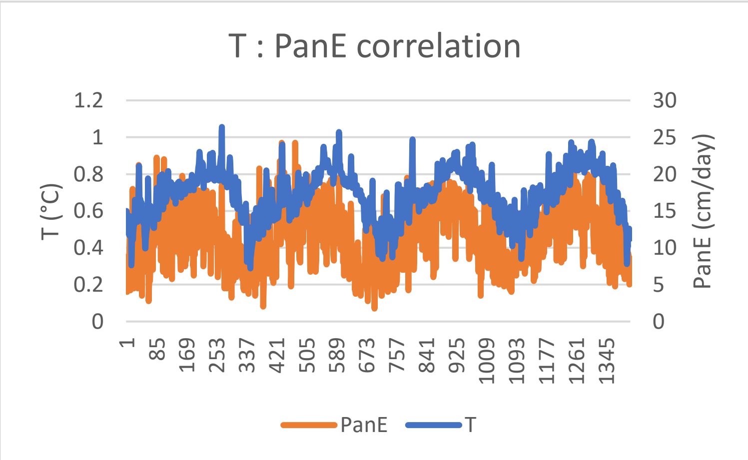
```
//Data subsets
private static final ArrayList<Sample> testSet = new ArrayList<>(); //About 20% of data
private static final ArrayList<Sample> validationSet = new ArrayList<>(); //About 20% of data
private static final ArrayList<Sample> trainingSet = new ArrayList<>(); //About 60% of data
```

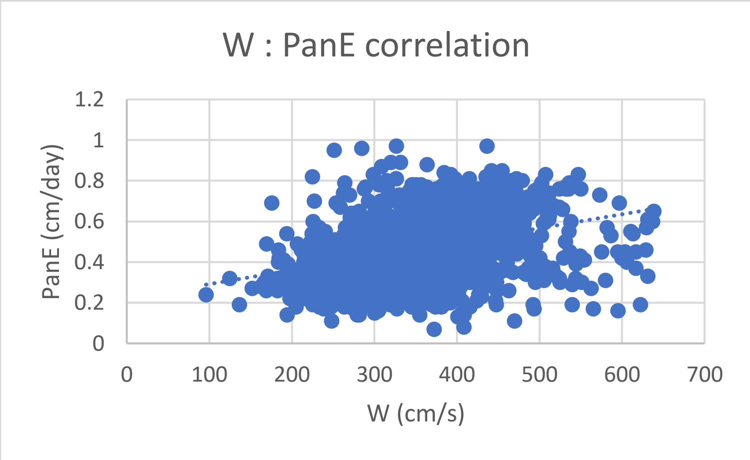
The data set is split randomly between these data subsets as can be seen in the code below. While, because of the random nature of subset assignment, there is potential for the sizes of the data subsets not to be distributed as intended, the large size of the given data set ensures the data is split very close to 20:20:60 between the 3 subsets.

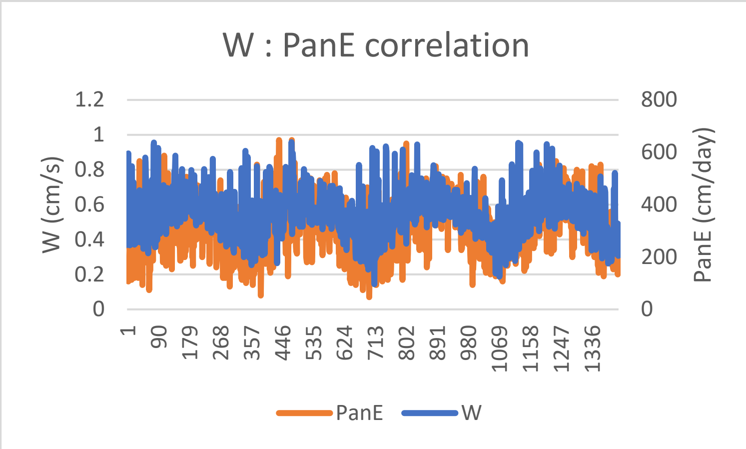
```
//Loop through data set
for (Sample sample : dataSet) {
    double[] sampleData = sample.getData();
    //Standardize each value between range [0.1, 0.9]
    for (int i = 0; i < sampleData.length; i++) {
        sampleData[i] = standardize(sampleData[i], maximums[i], minimums[i]);
    }
    //Display sample data
    System.out.println(sample);
    //Randomly place sample between 3 data sets
    switch (rand.nextInt(5)) {
        case (0): //20% chance
            testDataSet.add(sample);
            break;
        case (1): //20% chance
            validationDataSet.add(sample);
            break;
        default: //60% chance
            trainingDataSet.add(sample);
            break;
    }
    entireDataSet.add(sample);
}
//Display data subset sizes
System.out.printf("\nTotal number of samples: %d\n", entireDataSet.size());
System.out.printf("Test set size: %.2f%%\n", (double) testDataSet.size() / dataSet.size() * 100);
System.out.printf("Validation set size: %.2f%%\n", (double) validationDataSet.size() / dataSet.size() * 100);
System.out.printf("Training set size: %.2f%%\n", (double) trainingDataSet.size() / dataSet.size() * 100);
```

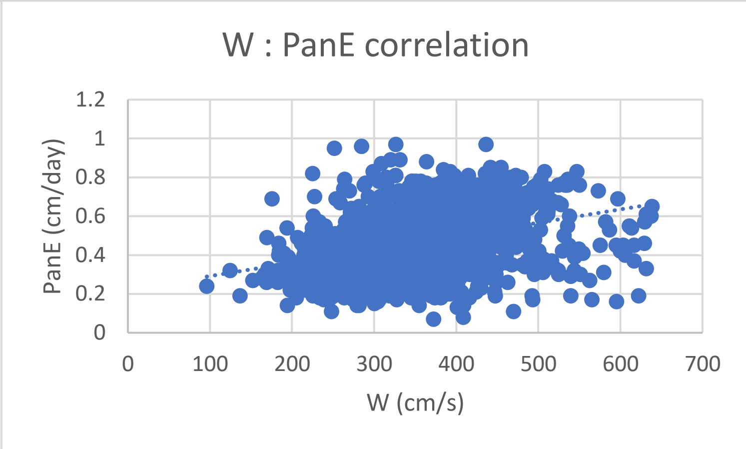
### Identifying suitable predictors

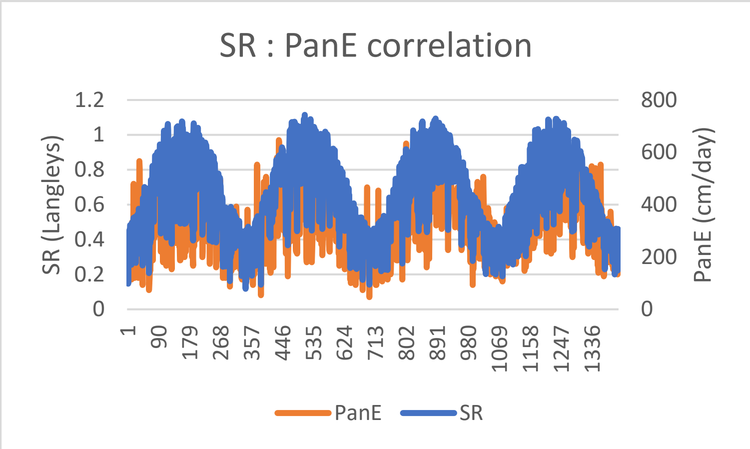
Below, for every predictor, you can see both a line and scatter comparing it to the predictand, showing a clear relationship between each predictor and the predictand. In every case, the peaks and troughs algin, or do the inverse, meaning the phase of both the predictor predictand are similar in every case. There is also a weak but notable correlation in every case. For this reason, and based on given domain knowledge, I have decided to use every given predictor in the model as they are all clearly suitable. In addition, after testing the model with subsets of the given inputs, the model is clearly more accurate when all inputs are used.

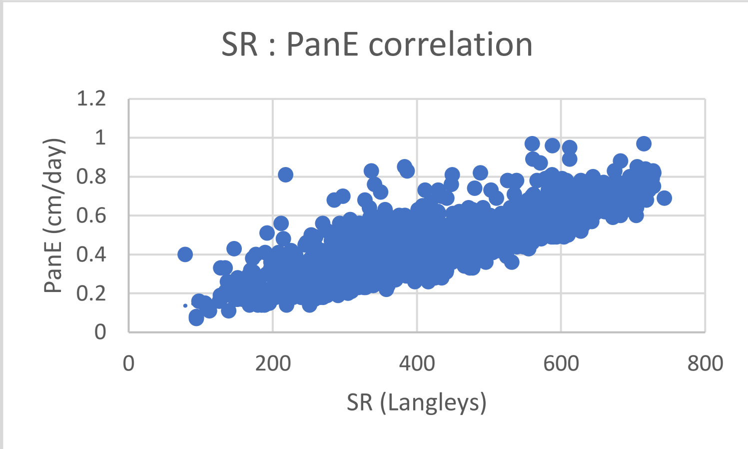


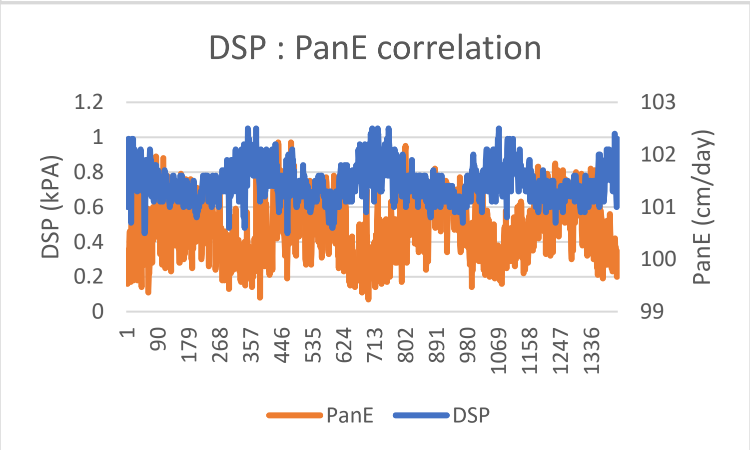



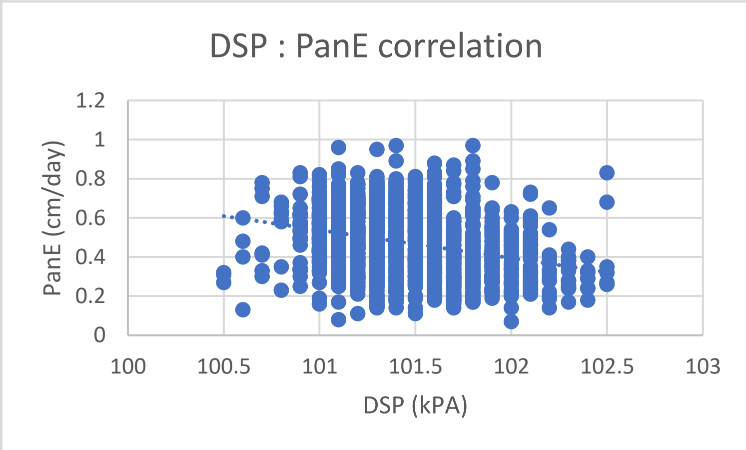


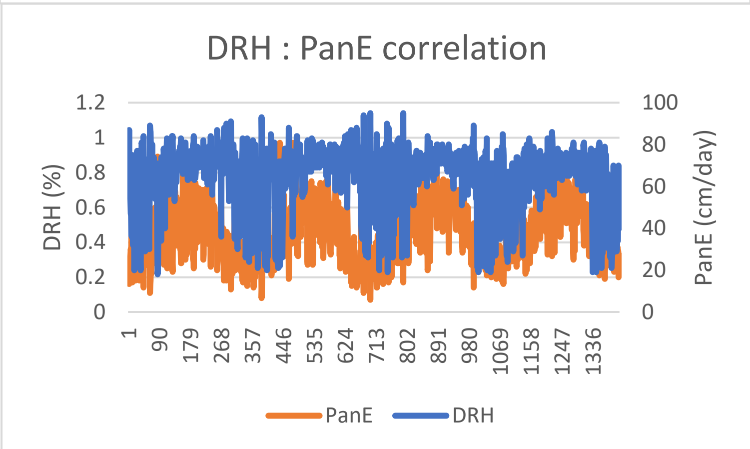



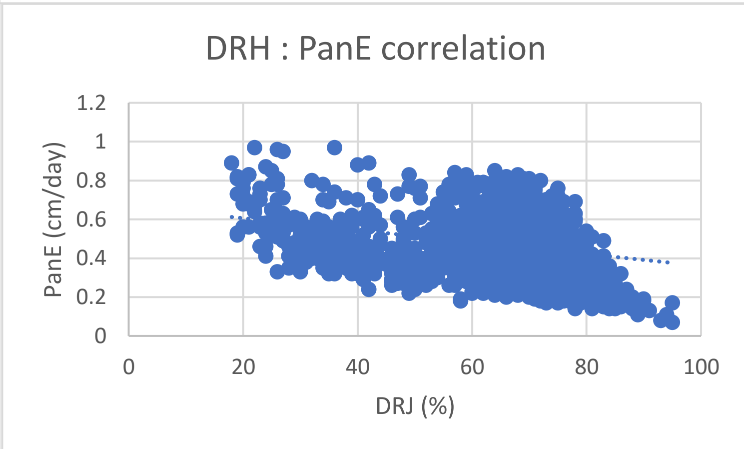




**Overall**

Erroneous and outlier data has been removed from the data set, 96.71% of the original data set remain.

All data has been standardized to be between the range [0.9, 0.1]

The entire data set has been randomly split 20:20:60 between the 3 data subsets: test, validation, and training set.

Every predictor is deemed suitable and used in the model because of their strong correlations with the predictand.

# MLP Algorithm Implementation

Topics: Representation of Neuron and ANN, random weights, training, backpropagation algorithm, OOP.

**Representation of a Neuron**

A neuron is represented as an initialization of the Neuron class (Neuron object).

Each neuron has the following attributes:

```
private double[] inputs;
private final double[] weights;


private double weightSum = 0;
private double activation = 0;
private double deltaValue = 0;
```

As well as the following methods, used to compute the weight sum, activation, and delta value for the neuron.

```
//Initialises neuron with initial wieghts
public Neuron(double[] weights) {
    this.weights = weights;
    this.previousWeights = weights;
    weightSum = 0;
}

public void setInputs(double[] inputs) {
    this.inputs = inputs;
}

public void computeWeightSum() {
    //weightSum = Σ(weights[i] * inputs[i])
    weightSum = 0;
    for (int i = 0; i < inputs.length; i++) {
        weightSum += weights[i] * inputs[i];
    }
}

public double computeActivation() {
    //activation, "output" = f(weightSum)
    activation = sigmoidFunction(weightSum);
    return activation;
}

public void computeDeltaValue(double sampleOutput) {
    //For output neurons:
    //deltaValue = (sampleOutput - activation, "modelled output") * f'(weightSum)
    deltaValue = (sampleOutput - activation) * (activation * (1 - activation));
}

public void computeDeltaValue(double[] nextWeights, double[] nextDeltaValues) {
    //For non-output neurons:
    //deltaValue = (sum of following weights * following delta values ) * f'(weightSum)
    double sum = 0;
    for (int i = 0; i < nextWeights.length; i++) {
        sum += (nextWeights[i] * nextDeltaValues[i]);
    }
    deltaValue = sum * (activation * (1 - activation));
}
public void updateWeightsAndBias(double stepSize) {
    //Update each weight:
    //new weight = old weight + stepSize * deltaValue * respective input
    for (int i = 0; i < weights.length; i++) {
        previousWeights[i] = weights[i];
        weights[i] += stepSize * deltaValue * inputs[i];
    }
}

private double sigmoidFunction(double value) {
    return (1 / (1 + Math.pow(Math.E, -value)));
}

public double[] getWeights() {
    return weights;
}

public double getDeltaValue() {
    return deltaValue;
}
```

**Representation of the ANN**

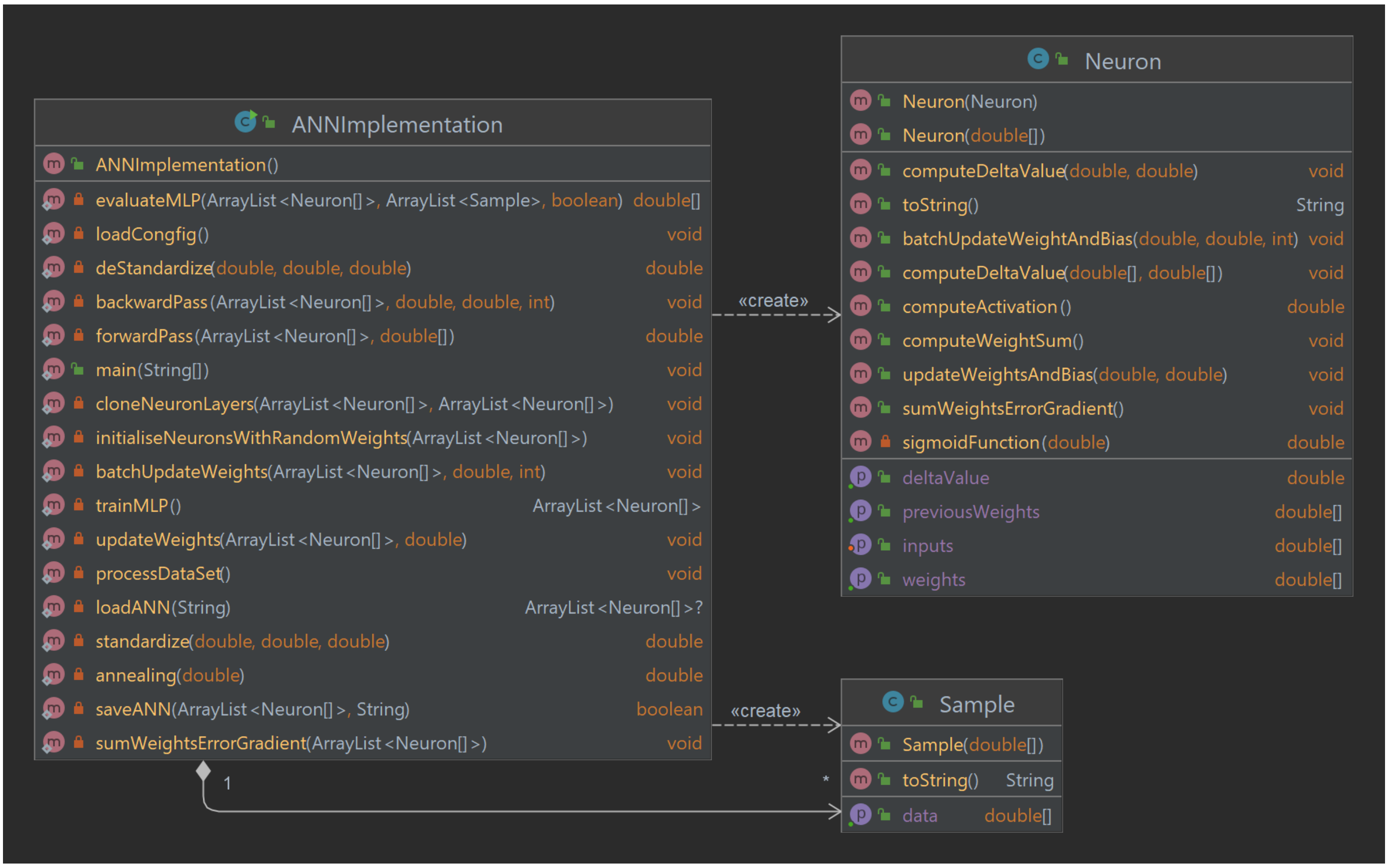
The neuron network is simply represented as an array list of any number of neuron layers. Each neuron layer itself intuitively represents an array of neurons. This implementation allows the number of neuron layers and number of neurons in each layer to easily be altered/customized.

```
//Array list of neuron layers represents ANN
final ArrayList<Neuron[]> neuronLayers = new ArrayList<>();
```

**OOP and class diagram**

This project had some clear use cases for object-oriented programming. Namely, representing each neuron then representing a neuron layer as an array of neurons and further representing then ANN as an array of neuron layers.

All calculations for the an individual neuron (weight sum, activation, and delta value computations) have been encapsulated inside the Neuron class while the main static ANNImplementation class interacts with the neurons such as training and evaluation. It also handles the processing of the dataset and uses the Sample data class to represent a single entry from the data set. See the class diagram below:




**Initializing the neurons with random weights and bias**

For every neuron in the ANN, each weight (and bias) is assigned a random value between [-2/n, 2/n] where n is the input size of the neuron. This ensures that random small weights are assigned, larger weights have a greater influence in the neuron network so it makes sense to have lower impact smaller weights to begin with, resulting in a more stable model less likely to overfit the training data set.

The code used to do this is shown below, the neurons are initialized with random before weights before training:

```
private static void initialiseNeuronsWithRandomWeights(ArrayList<Neuron[]> neuronLayers) {
    //Initialise all neurons with random weights and bias
    for (int i = 0; i < neuronLayers.size(); i++) {
        int weightsCount;
        //If first layer of hidden neurons then #weights = #sampleInputs + 1
        if (i == 0) {
            weightsCount = dataPointsCount;
        } //Otherwise, #weights = #previousLayerNeurons + 1
        else {
            weightsCount = neuronLayers.get(i - 1).length + 1;
        }
        for (int j = 0; j < neuronLayers.get(i).length; j++) {
            //Number of weights (#inputs + 1 for bias) for neuron
            double[] weightsAndBias = new double[weightsCount];
            //Randomise weights and bias values between [-2/n, 2/n] where n is the input size of the neuron
            int range = weightsAndBias.length - 1;
            for (int k = 0; k < weightsAndBias.length; k++) {
                //Random value between [-2/n, 2/n] where n is the input size
                weightsAndBias[k] = -2d / range + (2d / range - -2d / range) * rand.nextDouble();
            }
            neuronLayers.get(i)[j] = new Neuron(weightsAndBias);
        }
    }
}
```

### Training the ANN

The following code shows the implementation of the backpropagation algorithm without any additional improvements. It first creates an ANN with structure specified by the int array with the addition of a single output node before randomising all weights. Then for the specified number of epochs, it loops through every sample in the training set and does a forward pass (computing each neurons activation), a backward pass (computing a delta value for each neuron), and then finally updating the weights of each neuron. The training will also terminate early if the change in error (RMSE) is negligible 3 times in a row.

```java
private static ArrayList<Neuron[]> trainMLP() {

    double stepSize = START_STEP_SIZE;

    //Array list of neuron layers represents ANN
    final ArrayList<Neuron[]> neuronLayers = new ArrayList<>();
    //Create uninitialised hidden layers of neurons
    for (int hiddenLayerSize : ANN_SHAPE) {
        if (hiddenLayerSize > 0) {
            neuronLayers.add(new Neuron[hiddenLayerSize]);
        }
    }
    //Create uninitialised output neuron
    neuronLayers.add(new Neuron[1]);
    //Initialise neurons with random weights
    initialiseNeuronsWithRandomWeights(neuronLayers);

    //Previously measured RMSE
    double error = Double.MAX_VALUE;
    int tinyErrorCount = 0;

    //Train for specified number of epochs
    for (int i = 0; i < MAX_EPOCHS; i++) {
        //Loop through every sample in training set
        for (Sample sample : trainingDataSet) {
            //Inputs and bias passed to first layer of hidden neurons
            double[] inputsAndBias = new double[sample.getData().length];
            System.arraycopy(sample.getData(), 0, inputsAndBias, 0, inputsAndBias.length - 1);
            inputsAndBias[inputsAndBias.length - 1] = 1;
            //Correct output specified by sample
            double sampleOutput = sample.getData()[sample.getData().length - 1];
            //Forward pass through ANN
            forwardPass(neuronLayers, inputsAndBias);
            //Backward pass through ANN
            backwardPass(neuronLayers, sampleOutput, stepSize, i + 1);
            //Update weights for every neuron in ANN
            updateWeights(neuronLayers, stepSize);
        }
        //100 times while training (every 1% of training complete)
        if (i % (Math.max(MAX_EPOCHS / 100, 1)) == 0) {
            //Get Root Mean Squared Error from evaluating model on validation set
            double RMSE = evaluateMLP(neuronLayers, validationDataSet, false)[0];
            //Display epoch data
            //Terminate training if error change in negligable 3 times in a row
            if (Math.abs(RMSE - error) < 0.00001d) {
                tinyErrorCount++;
                if (tinyErrorCount >= 3) {
                    System.out.println(" - Change in Error minimal, stopping training");
                    break;
                }
            } else {
                tinyErrorCount = 0;
            }
            error = RMSE;
        }
    }
    return neuronLayers;
}
```

### Activation function

I have used the following function for the activation fucntion

```java
private double sigmoidFunction(double value) {
    return (1 / (1 + Math.pow(Math.E, -value)));
}
```

$$f(x) = \frac{1}{1 + e^{-x}}$$

### Forward pass

Forward passes through ANN, passing outputs (activations) from the previous layer of neurons as inputs to every neuron in the next layer, then calculating the weight sum and activation of that neuron. It passes the inputs from the sample to the first layer of neurons.

```
private static double forwardPass(ArrayList<Neuron[]> neuronLayers, double[] sampleInputsAndBias) {
    //Outputs from every neuron in the layer, used as inputs to the next layer
    double[] layerOutputs = new double[0];

    //Forward pass through every layer in ANN
    for (int i = 0; i < neuronLayers.size(); i++) {
        //Inputs and bias for every neuron in the layer
        double[] inputsAndBias;
        //If first layer then inputs equal to inputs from the sample
        if (i == 0) {
            inputsAndBias = sampleInputsAndBias;
        } //Otherwise, inputs equal to outputs of previous layer
        else {
            inputsAndBias = layerOutputs;
        }
        //Clears layer outputs and sets bias input to 1
        layerOutputs = new double[neuronLayers.get(i).length + 1];
        layerOutputs[layerOutputs.length - 1] = 1;
        //Set inputs, calculate weight sum, compute activation for every neuron in layer
        for (int j = 0; j < neuronLayers.get(i).length; j++) {
            neuronLayers.get(i)[j].setInputs(inputsAndBias);
            neuronLayers.get(i)[j].computeWeightSum();
            layerOutputs[j] = neuronLayers.get(i)[j].computeActivation();
        }
    }
    //Returns the output of the final neuron (output neuron), the output of the entire ANN
    return layerOutputs[layerOutputs.length - 2];
}
```

$$S_j = \sum_i w_{i,j} u_i$$

$$u_j = f(S_j) = \frac{1}{1+e^{-S_j}}$$

### Backward pass

Backward passes through ANN, computing the delta value for each neuron. The delta value is later used to update the neuron’s eights.

```
private static void backwardPass(ArrayList<Neuron[]> neuronLayers, double sampleOutput) {
    //Backward pass through every layer in ANN
    for (int i = neuronLayers.size() - 1; i >= 0; i--) {
        //If last layer (output layers)
        if (i == neuronLayers.size() - 1) {
            //Compute delta value for output neuron
            for (Neuron outputNeuron : neuronLayers.get(i)) {
                outputNeuron.computeDeltaValue(sampleOutput);
            }
        } //Otherwise, if hidden layer
        else {
            //Loop through neurons in hidden layer
            for (int j = 0; j < neuronLayers.get(i).length; j++) {
                double[] forwardWeights = new double[neuronLayers.get(i + 1).length];
                double[] forwardDeltaValues = new double[neuronLayers.get(i + 1).length];
                //Loop through forward neurons
                for (int k = 0; k < neuronLayers.get(i + 1).length; k++) {
                    Neuron forwardNeuron = neuronLayers.get(i + 1)[k];
                    //Store weights between current neuron and all forward neurons
                    forwardWeights[k] = forwardNeuron.getWeights()[j];
                    //Store delta values of every forward neuron
                    forwardDeltaValues[k] = forwardNeuron.getDeltaValue();
                }
                //Compute delta value for hidden neuron
                neuronLayers.get(i)[j].computeDeltaValue(forwardWeights, forwardDeltaValues);
            }
        }
    }
}
```

$$f'(S_j) = u_j(1-u_j)$$

$$\delta_j = \begin{cases} (C_j - u_j)\, f'(S_j) & \text{if } j \text{ is an output node (do 1}^{\text{st}}\text{)} \\ \left( \sum_{m:m>j} w_{j,m} \delta_m \right) f'(S_j) & \text{for other nodes (working back through layers)} \end{cases}$$

### Updating the weights

Finally, updates the weights for every neuron in the ANN.

```
private static void updateWeights(ArrayList<Neuron[]> neuronLayers, double stepSize) {
    //Calculates new wieghts for every neuron in ANN
    for (Neuron[] neuronLayer : neuronLayers) {
        for (Neuron neuron : neuronLayer) {
            neuron.updateWeightsAndBias(stepSize);
        }
    }
}
```

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

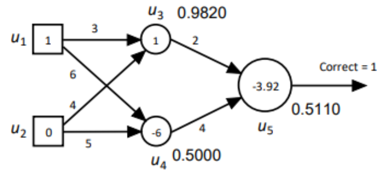
**Verifying the unmodified backpropagation algorithm works as intended**

On the left is what was produced by my implementation of the ANN and on the right is a given correct implementation of the backpropagation algorithm. As you can see, for the given small data set, both have the exact same results.

| Input 1 | Input 2 | Output |
|---|---|---|
| 1 | 0 | 1 |

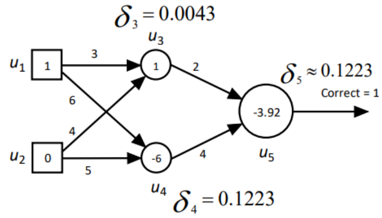
After the first forward pass:

```
1st Neuron{inputs=[1.0, 0.0, 1.0], weights=[3.0, 4.0, 1.0],
        weightSum=4.0, activation=0.9820137900379085, deltaValue=0.0}
2nd Neuron{inputs=[1.0, 0.0, 1.0], weights=[6.0, 5.0, -6.0],
        weightSum=0.0, activation=0.5, deltaValue=0.0}
3rd Neuron{inputs=[0.9820137900379085, 0.5, 1.0], weights=[2.0, 4.0, -3.92],
        weightSum=0.044027580075816974, activation=0.5110051173575921, deltaValue=0.0}
```



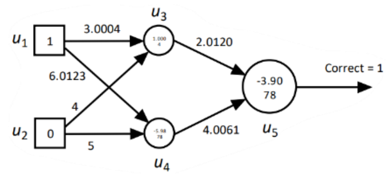

After the first backward pass:

```
1st Neuron{inputs=[1.0, 0.0, 1.0], weights=[3.0, 4.0, 1.0],
        weightSum=4.0, activation=0.9820137900379085, deltaValue=0.004316394383180035}
2nd Neuron{inputs=[1.0, 0.0, 1.0], weights=[6.0, 5.0, -6.0],
        weightSum=0.0, activation=0.5, deltaValue=0.1221894972l503991}
3rd Neuron{inputs=[0.9820137900379085, 0.5, 1.0], weights=[2.0, 4.0, -3.92],
        weightSum=0.044027580075816974, activation=0.5110051173575921, deltaValue=0.12218949721503991}
```



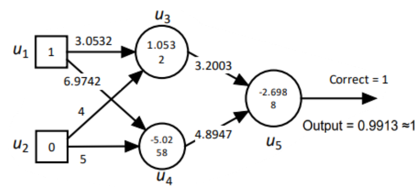

After first updating the weights:

```
1st Neuron{inputs=[1.0, 0.0, 1.0], weights=[3.000431639438332, 4.0, 1.000431639438332],
        weightSum=4.0, activation=0.9820137900379085, deltaValue=0.004316394383180035}
2nd Neuron{inputs=[1.0, 0.0, 1.0], weights=[6.012218949721504, 5.0, -5.987781050278496],
        weightSum=0.0, activation=0.5, deltaValue=0.12218949721503991}
3rd Neuron{inputs=[0.9820137900379085, 0.5, 1.0], weights=[2.0119991771262966, 4.006109474860752, -3.907781050278496],
        weightSum=0.044027580075816974, activation=0.5110051173575921, deltaValue=0.12218949721503991}
```




After 20,000 epochs:

```
1st Neuron{inputs=[1.0, 0.0, 1.0], weights=[3.0532272698644793, 4.0, 1.0532272698644898],
        weightSum=4.106453771442899, activation=0.9838006755684762, deltaValue=3.841430352192595E-6}
2nd Neuron{inputs=[1.0, 0.0, 1.0], weights=[6.97416872207521, 5.0, -5.02583127792479],
        weightSum=1.948329394190834, activation=0.8752643649713825, deltaValue=4.024979793041726E-5}
3rd Neuron{inputs=[0.9838006755684762, 0.8752643649713825, 1.0], weights=[3.2002547590177395, 4.894734079415909, -2.6988228540745114],
        weightSum=4.73375563672591, activation=0.991283265638109, deltaValue=7.531914775051635E-5}
```




**Performance of model (no improvements)**

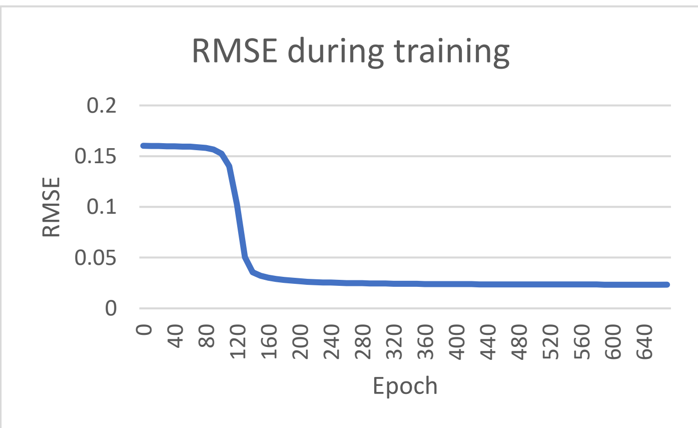
In this section, all evaluation is done on the same data subsets with these exact parameters:

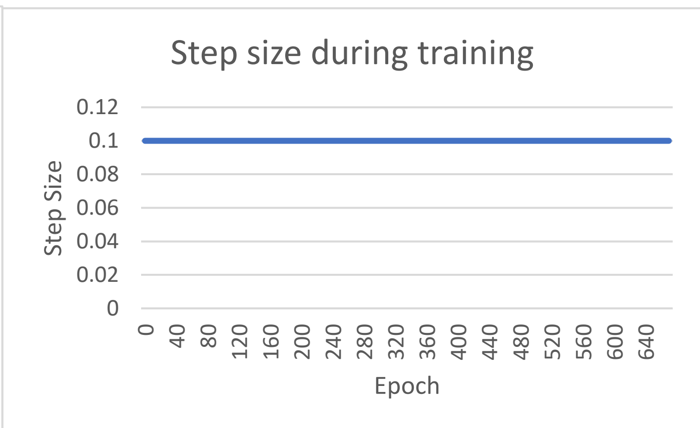
ANN_SHAPE = 10, 8 (10 nodes in first layer, 8 nodes in second layer, 1 output node). MOMENTUM = 0.9. MAX_EPOCHS = 1000. START_STEP_SIZE = 0.1. END_STEP_SIZE = 0.01. MAX_STEP_SIZE = 0.5. MIN_STEP_SIZE = 0.001

Evaluation metrics (measured on test set):

| | |
|---|---|
| RMSE – Root Mean Squared Error | 0.0273 (closer to 0 is better) |
| SRE - Mean Squared Relative Error | 0.0046 (closer to 0 is better) |
| CE - Coefficient of Efficiency | 0.9731 (closer to 1 is better) |
| RSQR - R-Squared (Determination Coefficient): | 0.9737 (closer to 1 is better) |

Training metrics (error calculated using validation set):




Clearly, the model is already performing quite well. It enters some local minimum after about 160 epochs. Training also terminated in this case because the change in error became negligible.

# Improvements

Topics: Momentum, Bold Driver, Annealing, Weight Decay, Batch Learning.

### Momentum

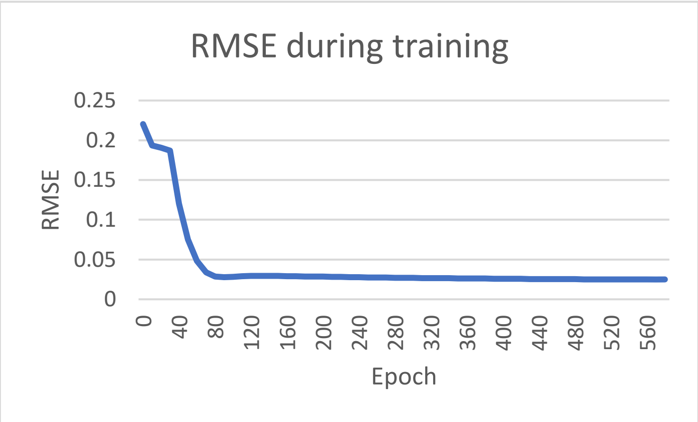
Leads to more rapid progress along the valley compared with unmodified gradient descent.

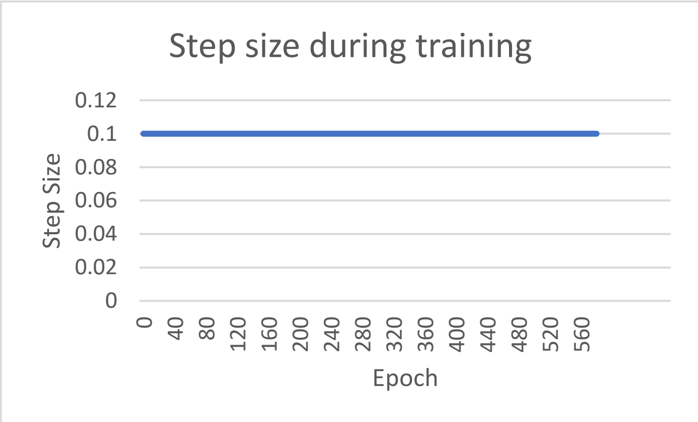
```
public void updateWeightsAndBias(double stepSize, double momentum) {
    //Update each weight:
    //new weight = old weight + stepSize * deltaValue * respective input
    for (int i = 0; i < weights.length; i++) {
        double tempWeight = weights[i];
        weights[i] += (stepSize * deltaValue * inputs[i])
                //Momentum
                + (momentum * (weights[i] - previousWeights[i]));
        previousWeights[i] = tempWeight;
    }
}
```

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

Evaluation metrics (measured on test set):

| | |
|---|---|
| RMSE – Root Mean Squared Error | 0.0276 (closer to 0 is better) |
| SRE - Mean Squared Relative Error | 0.0051 (closer to 0 is better) |
| CE - Coefficient of Efficiency | 0.9725 (closer to 1 is better) |
| RSQR - R-Squared (Determination Coefficient): | 0.9754 (closer to 1 is better) |




Training metrics (error calculated using validation set):

The final result of the model is near identical but in this case training is significatnly faster, reaching a local minima in about half as may epochs. Training again terminated early in this case because of error change became negligible.

### Bold driver

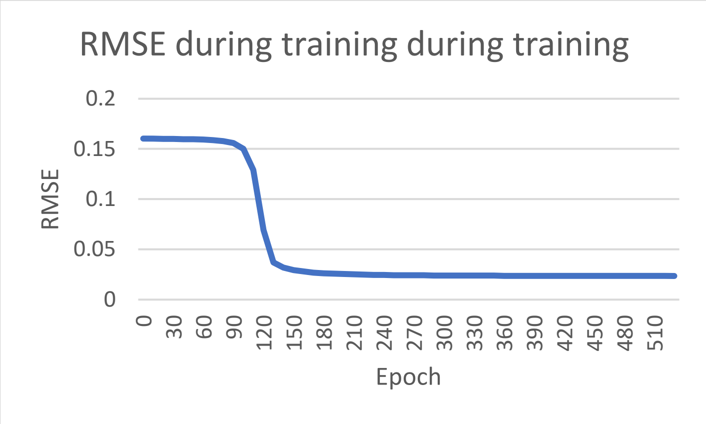
Alters the step size based on changes in the error function.

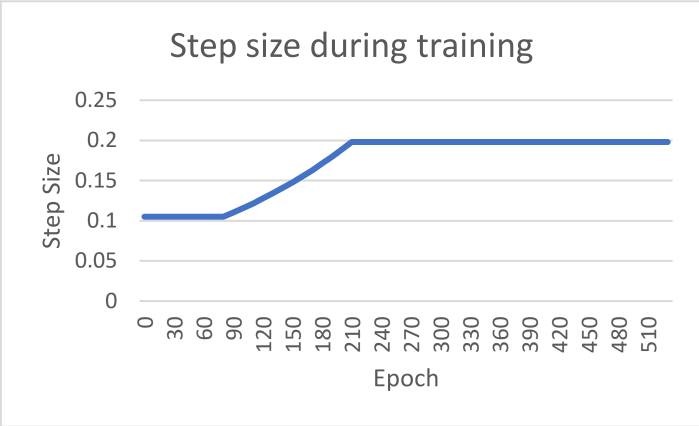
```
if (USE_BOLD_DRIVER) {
    double errorDiff = ((RMSE - error) / error);
    //If the error increases by over 1 % then half the step size and revert the weights back to the prev weights
    if (errorDiff > 0.01d) {
        stepSize *= 0.5;
        stepSize = Math.max(MIN_STEP_SIZE, stepSize);
        //revert model to last bold driver
        cloneNeuronLayers(prevNeuronLayers, neuronLayers);
    } else {
        //If the error decreases by over 1 % then slightly increase step size
        if (errorDiff < -0.01d) {
            stepSize *= 1.05;
            stepSize = Math.min(MAX_STEP_SIZE, stepSize);
        }
        cloneNeuronLayers(neuronLayers, prevNeuronLayers);
    }
}
```

Evaluation metrics (measured on test set):

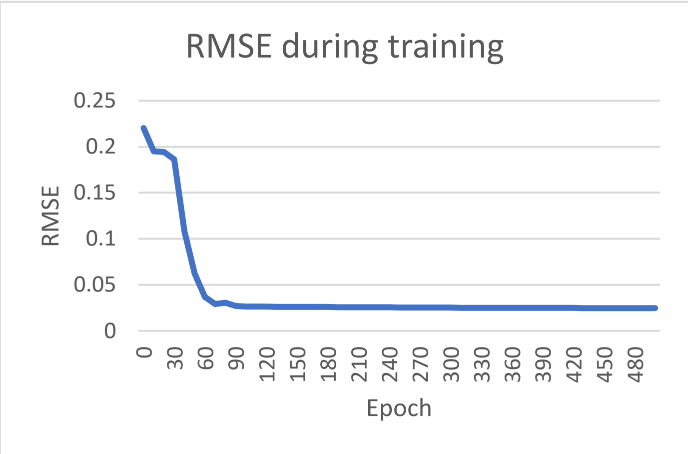
| | |
|---|---|
| RMSE – Root Mean Squared Error | 0.0273 (closer to 0 is better) |
| SRE - Mean Squared Relative Error | 0.0046 (closer to 0 is better) |
| CE - Coefficient of Efficiency | 0.9731 (closer to 1 is better) |
| RSQR - R-Squared (Determination Coefficient): | 0.9737 (closer to 1 is better) |

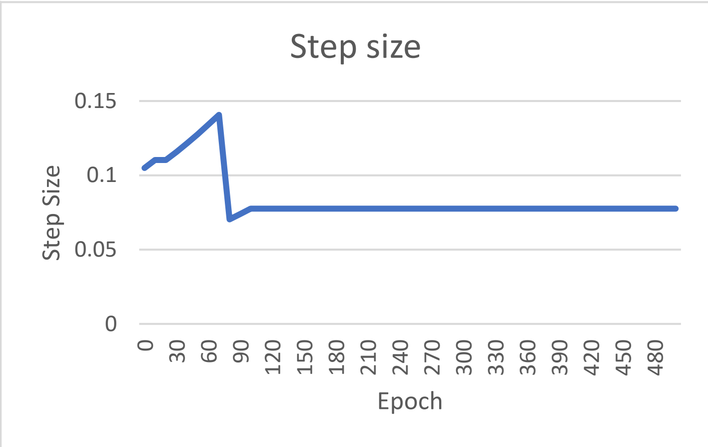
Training metrics (error calculated using validation set):



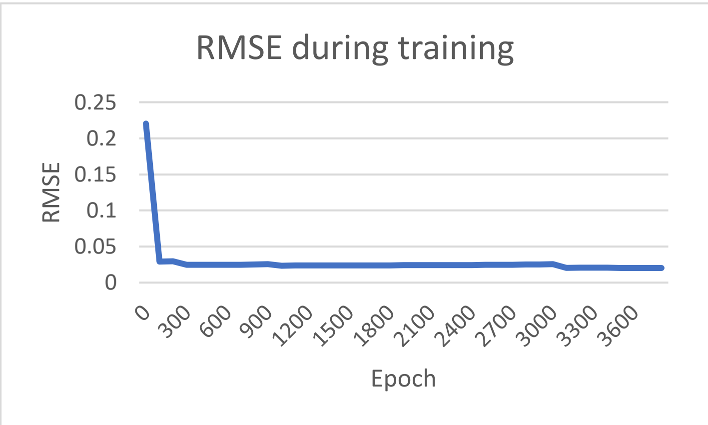



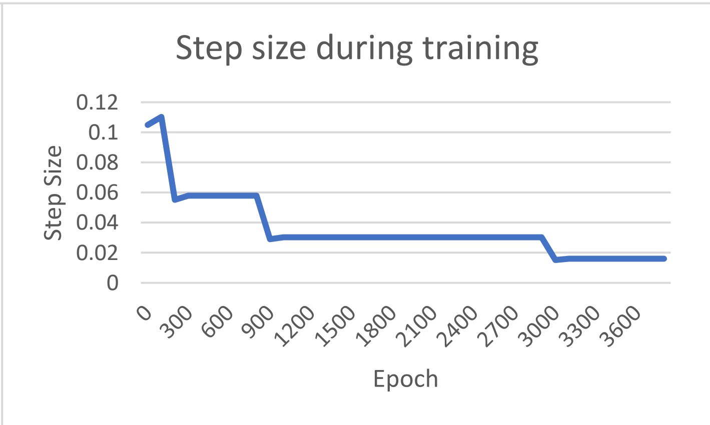

Training has again quickened compared to the basic algorithm, as shown the step size in this case steadily increased because the error never increased by over 1%. The below graphs show momentum combined with bold driver, clearly displaying the adjustment of step size.




To further emphases the change in step size, see training with momentum + bold driver for a few thousand epochs:



### Simulated Annealing

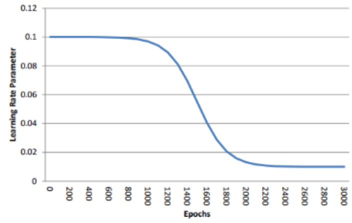
At the end of each epoch, the step size is recalculated using simulated annealing.

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

```
//Simulated annealing
stepSize = annealing(i + 1, epochs);
```

This causes the step size to decay over time, meaning smaller adjustments are made to the model later in training:

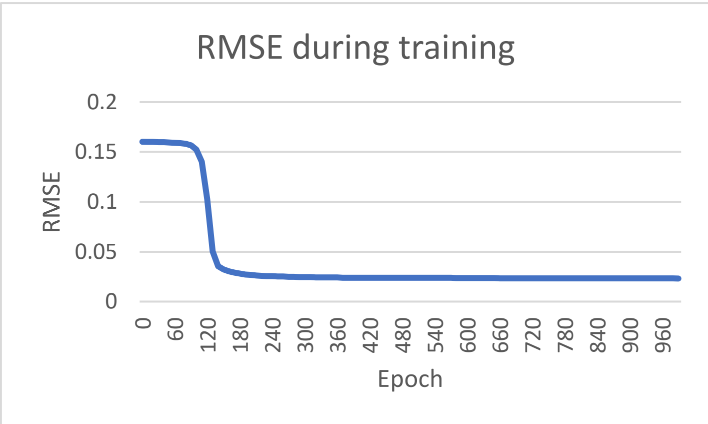
```
private static double annealing(double epoch) {
    //Decays stepSize (learning parameter) over time
    return END_STEP_SIZE + (START_STEP_SIZE - END_STEP_SIZE)
            * (1 - (1 / (1 + Math.pow(Math.E, (10 - 20 * (epoch / MAX_EPOCHS))))));
}
```

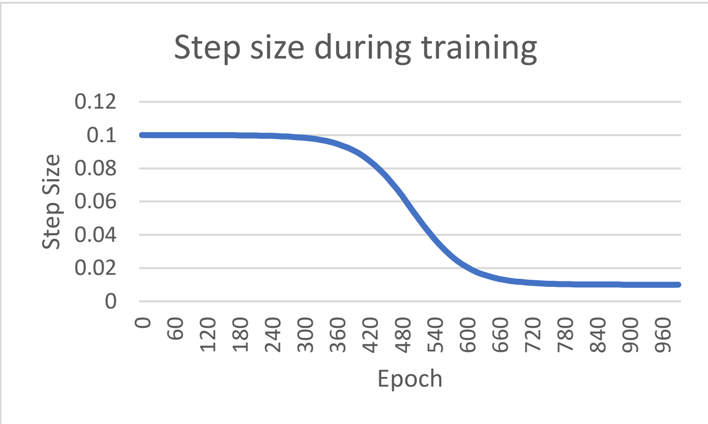



Evaluation metrics (measured on test set):

| | | |
|---|---|---|
| RMSE – Root Mean Squared Error | 0.0274 | (closer to 0 is better) |
| SRE - Mean Squared Relative Error | 0.0045 | (closer to 0 is better) |
| CE - Coefficient of Efficiency | 0.9729 | (closer to 1 is better) |
| RSQR - R-Squared (Determination Coefficient): | 0. 973 | (closer to 1 is better) |

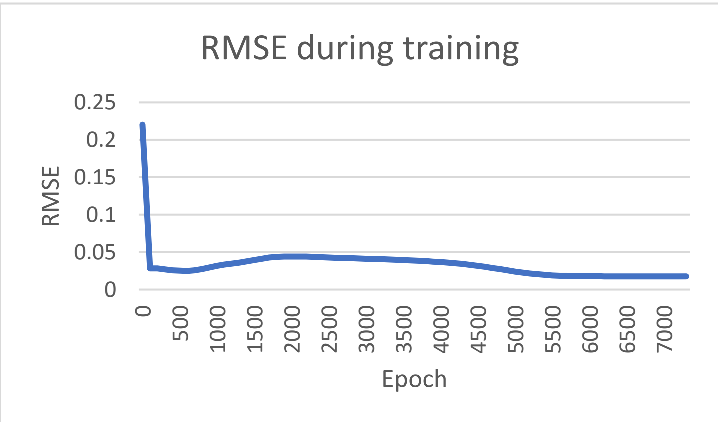
Training metrics (error calculated using validation set):

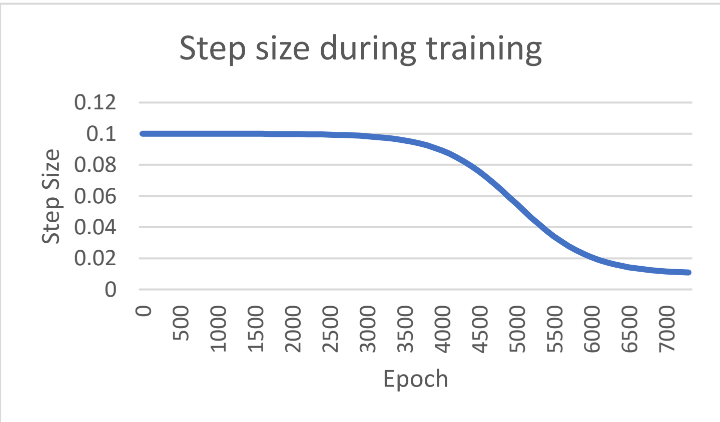



The error during training is near identical to when no improvements where implemented, this is because the local minima is reached while the step size is very close to 0.1 (the same step size when annealing is not implemented). However, unlike training with no improvements, when the step size decays you can see the error slowly reduce as the model descends into the local minima with smaller steps (causing it to descend instead of oscillate).

Below is momentum and annealing combined, you can clearly see the algorithm escaping a local minimum and finding a new minimum, the lowest error seen so far.



**Weight Decay**

Regularization technique that adds a penalty term to the error function which penalizes large weights

Large weights can cause small input changes to be too impactful. Penalizing larger weighs helps prevent this.

```java
//Compute delta value for output neuron
for (Neuron outputNeuron : neuronLayers.get(i)) {
    // Cacuiate upsilon (for weight decay)
    double upsilon = 1 / (stepSize * epoch);
    if (!USE_WEIGHT_DECAY || epoch < 100) {
        upsilon = 0;
    }
    //Compute delta value for output neuron
    outputNeuron.computeDeltaValue(sampleOutput, upsilon);
}
```

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

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

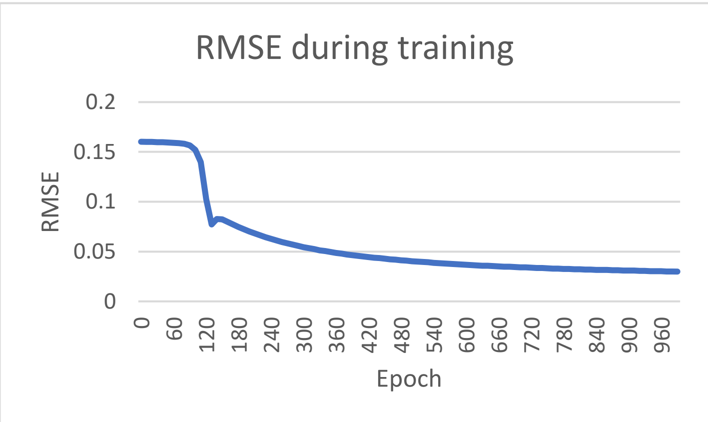
$$\widetilde{E} = E + \upsilon\Omega$$

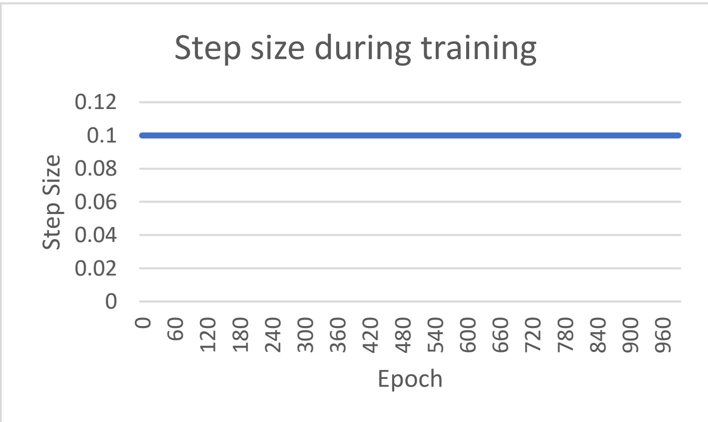
```java
public void computeDeltaValue(double sampleOutput, double upsilon) {
    //For output neurons:
    // Caculate omega (for wieght decay)
    double sum = 0;
    for (double weight : weights) {
        sum += Math.pow(weight, 2);
    }
    double omega = sum / (2 * weights.length);

    //deltaValue = (sampleOutput - activation, "modelled output" + upsilon * omega) * f'(weightSum)
    deltaValue = (sampleOutput - activation + upsilon * omega) * (activation * (1 - activation));
}
```

Evaluation metrics (measured on test set):

| | |
|---|---|
| RMSE – Root Mean Squared Error | 0.0326 (closer to 0 is better) |
| SRE - Mean Squared Relative Error | 0.0045 (closer to 0 is better) |
| CE - Coefficient of Efficiency | 0.9729 (closer to 1 is better) |
| RSQR - R-Squared (Determination Coefficient): | 0. 973 (closer to 1 is better) |

Training metrics (error calculated using validation set):




As you can see, the error decreases more uniformly and gradually compared to the other models. I assume this is because the other models put emphasis on some weights in particular which resulted in those weights becoming larger and also leading to a more rapid reduction in error while this model punishes larger weights causing it to take longer for the error to reduce.

### Batch Learning

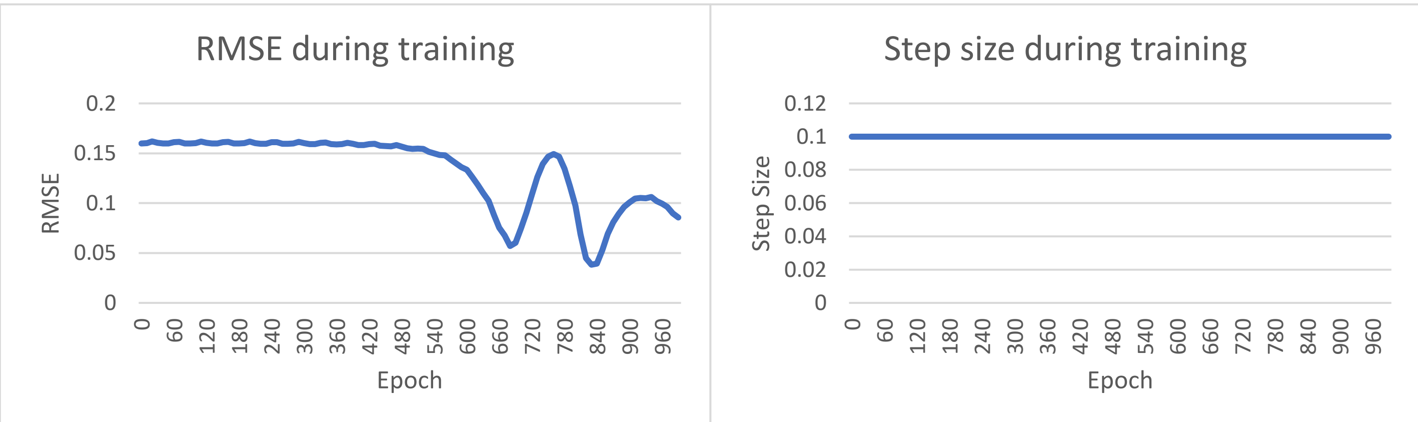
Weights and biases are only updated once per epoch.

```
    if (!USE_BATCH_LEARING) {
        //Update weights for every neuron in ANN
        updateWeights(neuronLayers, stepSize);
    } else {
        //Sum error gradient for all weights
        sumWeightsErrorGradient(neuronLayers);
    }
}

//Update weights at the end of the epoch
if (USE_BATCH_LEARING) {
    batchUpdateWeights(neuronLayers, stepSize, trainingDataSet.size());
}
public void sumWeightsErrorGradient() {
    //Sum error gradient for each weight
    for (int i = 0; i < weightsErrorGradient.length; i++) {
        weightsErrorGradient[i] += deltaValue * inputs[i];
    }
}

public void batchUpdateWeightAndBias(double stepSize, double momentum, int sampleCount) {
    //Update each weight:
    //new weight = old weight + stepSize * (error gradient / sample count)
    for (int i = 0; i < weights.length; i++) {
        double tempWeight = weights[i];
        weights[i] += (stepSize * (weightsErrorGradient[i] / sampleCount))
                //Momentum
                + (momentum * (weights[i] - previousWeights[i]));
        previousWeights[i] = tempWeight;
    }
    //Reset error gradient sum for each weight
    for (int i = 0; i < weightsErrorGradient.length; i++) {
        weightsErrorGradient[i] += 0;
    }
}
```

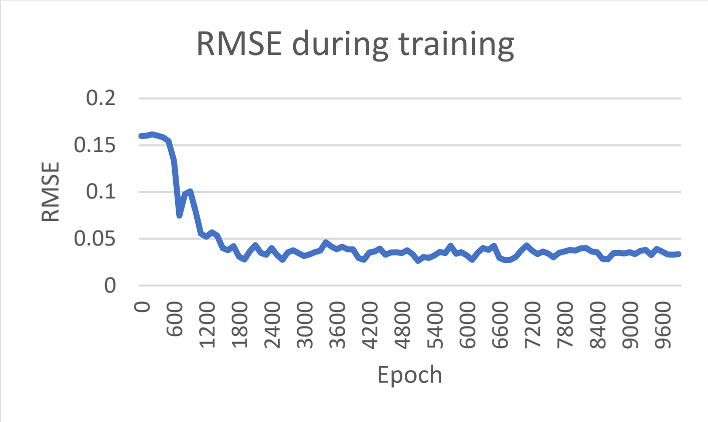
Evaluation metrics (measured on test set):

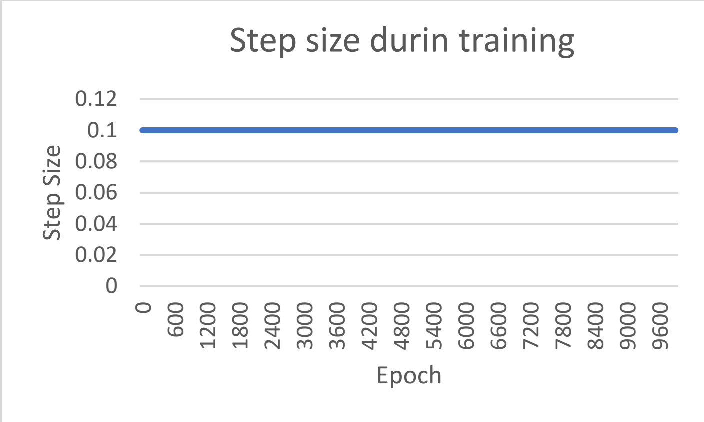
| | |
|---|---|
| RMSE – Root Mean Squared Error | 0.0770 (closer to 0 is better) |
| SRE - Mean Squared Relative Error | 0.0318 (closer to 0 is better) |
| CE - Coefficient of Efficiency | 0.7864 (closer to 1 is better) |
| RSQR - R-Squared (Determination Coefficient): | 0.8395 (closer to 1 is better) |

Training metrics (error calculated using validation set):




The error is clearly more variable, the algorithm is also significantly slower because weights are updated less frequently hence why a poor result was reached after only 1000 epochs. Below is an example with 10000 epochs.




# ANN selection

Topics: Best modifications, best ANN shape, best number of epochs.

**Best combination of modifications**

Below is every combination of modifications ordered ascendingly by RMSE measured using the validation set after training for 10000 epochs. Most results are similar and perfectly adequate but clearly some combinations are not compatible. I have selected the combination of Momentum and Annealing for the final model because that resulted in the lowest error.

| Momentum | Bold Driver | Annealing | Weight Decay | Batch Learning | RMSE |
|---|---|---|---|---|---|
| TRUE | FALSE | TRUE | FALSE | FALSE | 0.017618977 |
| TRUE | TRUE | TRUE | FALSE | FALSE | 0.018017004 |
| TRUE | TRUE | FALSE | FALSE | FALSE | 0.020617124 |
| FALSE | TRUE | TRUE | FALSE | TRUE | 0.021324943 |
| FALSE | FALSE | FALSE | FALSE | FALSE | 0.021613644 |
| FALSE | TRUE | TRUE | FALSE | FALSE | 0.021769251 |
| FALSE | FALSE | TRUE | FALSE | FALSE | 0.021771038 |
| FALSE | TRUE | FALSE | FALSE | FALSE | 0.021872063 |
| FALSE | FALSE | FALSE | TRUE | FALSE | 0.021932132 |
| FALSE | TRUE | FALSE | FALSE | TRUE | 0.022402073 |
| FALSE | TRUE | FALSE | TRUE | FALSE | 0.022823699 |
| FALSE | TRUE | TRUE | TRUE | TRUE | 0.026475065 |
| FALSE | FALSE | TRUE | TRUE | FALSE | 0.028486666 |
| FALSE | TRUE | TRUE | TRUE | FALSE | 0.028625731 |
| TRUE | FALSE | FALSE | TRUE | FALSE | 0.031600556 |
| TRUE | FALSE | FALSE | FALSE | FALSE | 0.031852734 |
| TRUE | FALSE | TRUE | TRUE | FALSE | 0.034271004 |
| TRUE | TRUE | TRUE | TRUE | FALSE | 0.034850028 |
| FALSE | FALSE | FALSE | FALSE | TRUE | 0.043067578 |
| FALSE | TRUE | FALSE | TRUE | TRUE | 0.167990002 |
| FALSE | FALSE | TRUE | FALSE | TRUE | 0.168714251 |
| FALSE | FALSE | FALSE | TRUE | TRUE | 0.583736065 |
| TRUE | FALSE | FALSE | FALSE | TRUE | 0.619221274 |
| TRUE | FALSE | TRUE | FALSE | TRUE | 0.619221274 |
| TRUE | FALSE | FALSE | TRUE | TRUE | 0.619221274 |
| TRUE | TRUE | TRUE | TRUE | TRUE | 0.619221274 |
| TRUE | TRUE | FALSE | TRUE | FALSE | 0.634625654 |
| FALSE | FALSE | TRUE | TRUE | TRUE | 0.63470564 |
| TRUE | TRUE | TRUE | FALSE | TRUE | 0.634706672 |
| TRUE | TRUE | FALSE | FALSE | TRUE | 0.634706672 |
| TRUE | TRUE | FALSE | TRUE | TRUE | 0.634706672 |
| TRUE | FALSE | TRUE | TRUE | TRUE | 0.634706672 |

**Best Neural Network shape**

Below is every combination of max epochs (100, 1000, 10000), Layer 1 neurons (0 – 20) and layer 2 neurons (0 – 20) ordered ascendingly by RMSE measured using the validation after training with only the Momentum and Annealing modifications (0 nodes indicates no hidden layer). Note, there is a one single output neuron in every case. In total there was 1323 total entries, here are the first 20:

| Maximum Epochs | Layer 1 neurons | Layer 2 neurons | RMSE |
|---|---|---|---|
| 10000 | 14 | 12 | 0.017444 |
| 10000 | 0 | 20 | 0.017547 |
| 10000 | 11 | 17 | 0.017557 |
| 10000 | 0 | 19 | 0.017597 |
| 10000 | 11 | 20 | 0.017647 |
| 10000 | 6 | 7 | 0.017649 |
| 10000 | 11 | 15 | 0.017681 |
| 10000 | 7 | 11 | 0.017687 |
| 10000 | 9 | 0 | 0.017691 |
| 10000 | 10 | 13 | 0.017696 |
| 10000 | 7 | 7 | 0.017697 |
| 10000 | 14 | 0 | 0.017717 |
| 10000 | 6 | 0 | 0.017731 |
| 10000 | 20 | 4 | 0.017735 |
| 10000 | 19 | 16 | 0.017737 |
| 10000 | 6 | 5 | 0.017738 |
| 10000 | 11 | 19 | 0.017742 |
| 10000 | 8 | 8 | 0.01775 |
| 10000 | 16 | 9 | 0.017781 |
| 10000 | 16 | 19 | 0.017783 |

Unsurprisingly, the best result was obtained with more epochs. As you can also see, with the best performing neural networks, the difference in error between the different neural network shapes is minimal. Most models above terminated training early because of change in error (RMSE) became minimal/negligible. I believe that more hidden layers will be unnecessary, especially given the high performance of some models with only one hidden layer. There is an argument to be made that the 2nd highest result with only one hidden layer and 20 neurons is superior because training this network would be significantly faster, so I have selected this network this network with 20 neurons in the first hidden layer, and no other hidden layers.

**Best number of Epochs**

Using a neural network with 1 hidden layer consisting of 20 neurons, below is the results after training for a differing number of epochs ordered ascendingly by RMSE measured using the validation after training with only the Momentum and Annealing modifications.

| Maximum Epochs | Executed Epochs | RMSE |
|---|---|---|
| 10 | 10 | 0.054608 |
| 100 | 92 | 0.026329 |
| 1000 | 670 | 0.021191 |
| 10000 | 7000 | 0.017963 |
| 100000 | 75000 | 0.017697 |
| 1000000 | 230000 | 0.020749 |
| 10000000 | 1400000 | 0.028574 |
| 100000000 | 7500000 | 0.025358 |

Clearly, training beyond 100,000 epochs is unnecessary as a minima is found before then. I assume more epochs is leading to worse results because the model gets stuck oscillating while the step size is still large and this results in the training terminating early because the change in error is minimal (it takes too long for the step size to reduce which would lead to the model descending the hill). I have therefore selected 100,000 epochs for the final model.

# ANN Evaluation

Topics: Evaluation metrics, evaluation of final model.

**Evaluation Metrics**

There are four metrics that I use to evaluation the model

**RMSE** – Root Mean Squared Error

Average absolute difference in real units

**MSRE** –Mean Squared Relative Error

Average relative difference

**CE** – Coefficient of Efficiency

Direction of model correctness

**RSqr** – R-Squared (Coefficient of Determination)

Coincidence of the shape

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}\left(\hat{Q}_i - Q_i\right)^2}{n}}$$

$$MSRE = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{\hat{Q}_i - Q_i}{Q_i}\right)^2$$

$$CE = 1 - \frac{\sum_{i=1}^{n}\left(\hat{Q}_i - Q_i\right)^2}{\sum_{i=1}^{n}\left(Q_i - \bar{Q}\right)^2}$$

$$RSqr = \left(\frac{\sum_{i=1}^{n}(Q_i - \bar{Q})(\hat{Q}_i - \tilde{Q})}{\sqrt{\sum_{i=1}^{n}(Q_i - \bar{Q})^2 \sum_{i=1}^{n}(\hat{Q}_i - \tilde{Q})^2}}\right)^2$$

$Q_i$ is the observed value
$\hat{Q}_i$ is the modelled value
$\bar{Q}$ is the mean of the observed values
$\tilde{Q}$ is the mean of the modelled values

Below is my implementation of model evaluation:

```java
private static double[] evaluateANN(ArrayList<Neuron[]> neuronLayers, ArrayList<Sample> dataSubset) {

    //RMSE, MSRE, CE, RSQR
    double[] evaluationMetrics = new double[4];

    double[] sampleOutputs = new double[dataSubset.size()];
    double[] modelledOutputs = new double[dataSubset.size()];

    //Variables used for calculation
    double RMSEsum = 0, MSREsum = 0, CEnumerator = 0, CEdenominator = 0, RSQRnumerator = 0, RSQRdenominatorLeft = 0, RSQRdenominatorRight = 0;

    double sampleOutputsMean = 0;
    double modelledOutputsMean = 0;
    //Store sample output and modelled output for every sample in given data subset
    for (int i = 0; i < dataSubset.size(); i++) {
        Sample sample = dataSubset.get(i);
        //Inputs and bias passed to first layer of hidden neurons
        double[] inputsAndBias = new double[sample.getData().length];
        System.arraycopy(sample.getData(), 0, inputsAndBias, 0, inputsAndBias.length - 1);
        inputsAndBias[inputsAndBias.length - 1] = 1;

        //Correct output specified by sample
        double sampleOutput = sample.getData()[sample.getData().length - 1];
        //De-standardize sample output
        sampleOutputs[i] = deStandardize(sampleOutput, maximums[maximums.length - 1], minimums[minimums.length - 1]);
        //Forward pass through ANN
        double modelledOutput = forwardPass(neuronLayers, inputsAndBias);
        //De-standardize modelled output
        modelledOutputs[i] = deStandardize(modelledOutput, maximums[maximums.length - 1], minimums[minimums.length - 1]);

        //Calculate mean for both values
        sampleOutputsMean += sampleOutputs[i];
        modelledOutputsMean += modelledOutputs[i];
    }

    sampleOutputsMean /= dataSubset.size();
    modelledOutputsMean /= dataSubset.size();

    //Calculate evaluation metrics
        CEnumerator += Math.pow(modelledOutputs[i] - sampleOutputs[i], 2);
        CEdenominator += Math.pow(sampleOutputs[i] - sampleOutputsMean, 2);

        RSQRnumerator += (sampleOutputs[i] - sampleOutputsMean) * (modelledOutputs[i] - modelledOutputsMean);
        RSQRdenominatorLeft += Math.pow(sampleOutputs[i] - sampleOutputsMean, 2);
        RSQRdenominatorRight += Math.pow(modelledOutputs[i] - modelledOutputsMean, 2);
    }
    //RMSE - Root Mean Squared Error (closer to 0 is better)
    evaluationMetrics[0] = Math.sqrt(RMSEsum / dataSubset.size());
    //MSRE - Mean Squared Relative Error (closer to 0 is better)
    evaluationMetrics[1] = MSREsum / dataSubset.size();
    //CE - Coefficient of Efficiency (closer to 1 is better)
    evaluationMetrics[2] = 1 - CEnumerator / CEdenominator;
    //RSQR - R-Squared (Coefficient of Determination) (closer to 1 is better)
    evaluationMetrics[3] = Math.pow(RSQRnumerator / (Math.sqrt(RSQRdenominatorLeft * RSQRdenominatorRight)), 2);

    return evaluationMetrics;
}
```

### Final Model Evaluation

The final selected model had the following parameters:

config.txt - Notepad

File Edit Format View Help

```
ANN_SHAPE = 20
MOMENTUM = 0.9
MAX_EPOCHS = 100000
START_STEP_SIZE = 0.1
END_STEP_SIZE = 0.01
MAX_STEP_SIZE = 0.5
MIN_STEP_SIZE = 0.01
USE_MOMENTUM = true
USE_BOLD_DRIVER = false
USE_ANNEALING = true
USE_WEIGHT_DECAY = false
USE_BATCH_LEARING = false
```

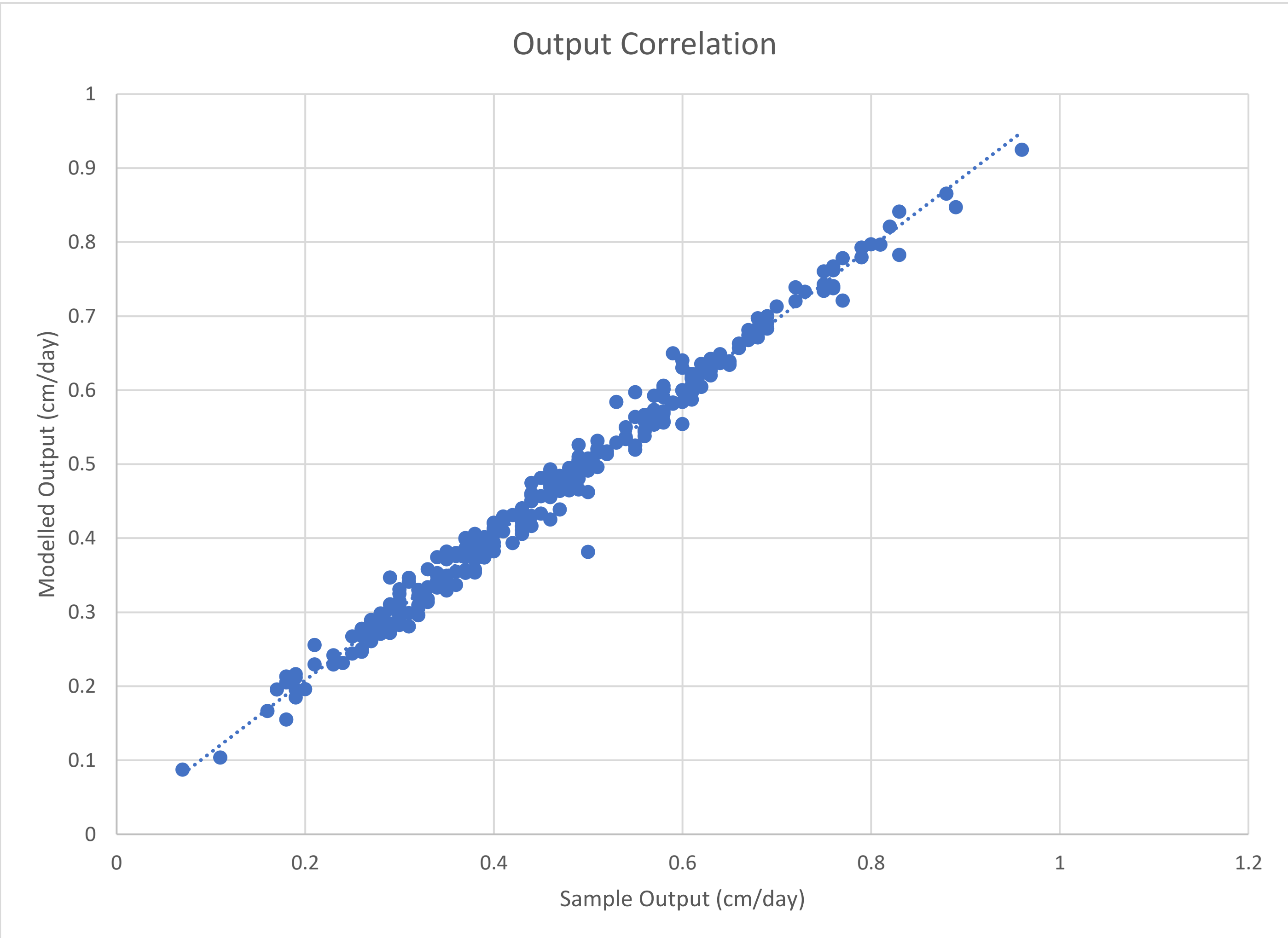
Evaluation metrics (measured on test set):

| | |
|---|---|
| RMSE – Root Mean Squared Error | 0.0189 (closer to 0 is better) |
| SRE - Mean Squared Relative Error | 0.0027 (closer to 0 is better) |
| CE - Coefficient of Efficiency | 0.9867 (closer to 1 is better) |
| RSQR - R-Squared (Determination Coefficient): | 0.9869 (closer to 1 is better) |

All these metrics are very close to ideal. On average, the modelled value is within less than +- 2% of the real value.

Because of this excellent performance, I believe that the Model is very accurate and suitable.

See the graph below displaying the correlation between the sample and modelled output for this model.




Measured Correlation = 0.993442. Clearly this model is very accurate.

# Simple multiple linear regression model comparison

Topics: LINEST linear model, single perceptron linear model, comparison to ANN model.

**LINEST**

LINEST function in Excel used to calculate multiple linear regression model (using standardized training set):

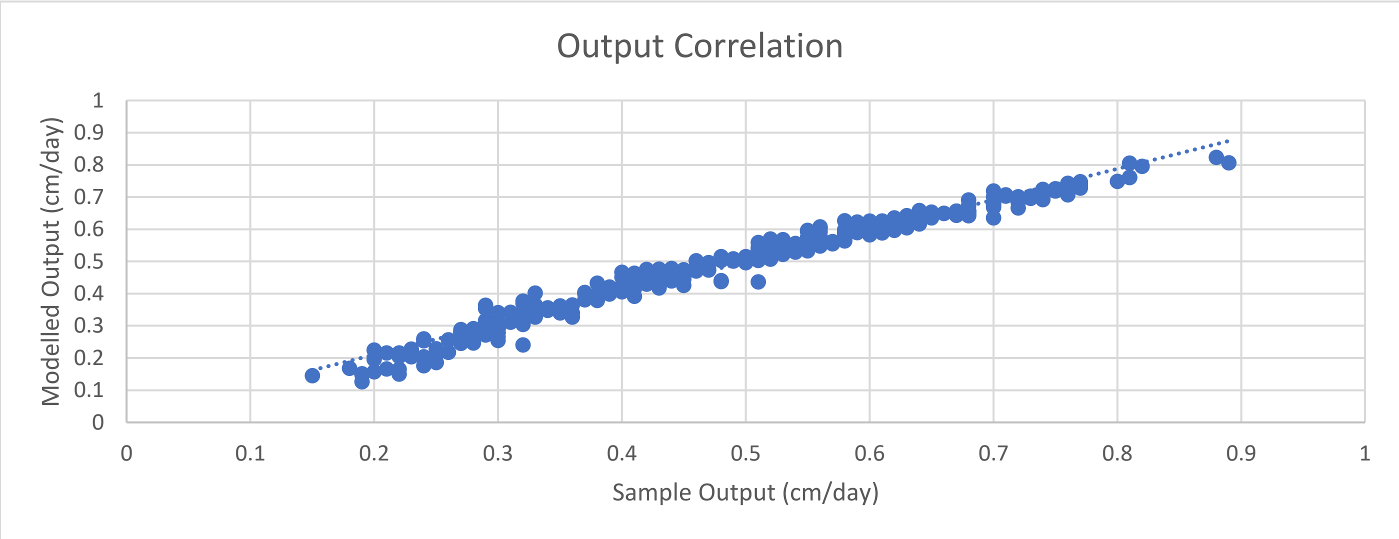
**PanE** = 0.414558 * **T** + 0.346879 * **W** + 0.554193 * **SR** + -0.5861 * **DSP** + -0.59098 * **DRH** + 0.624032

Evaluation metrics (measured on test set):

| | |
|---|---|
| RMSE – Root Mean Squared Error | 0.0276 (closer to 0 is better) |
| SRE - Mean Squared Relative Error | 0.0058 (closer to 0 is better) |
| CE - Coefficient of Efficiency | 0.9724 (closer to 1 is better) |
| RSQR - R-Squared (Determination Coefficient): | 0.9725 (closer to 1 is better) |

As you can see, in all evaluation metrics, the simple linear regression model performed slightly worse.

See the graph below displaying the correlation between the sample and modelled output for this model.




Measured Correlation = 0.986144. Again, this is slightly worse than the ANN.

As you can tell from the graph, this model is still very accurate but, unlike the ANN model, is notable less accurate at the tail ends of the data set. This model is presumable worse at predicting values outside the range it was trained on.

**Single Perceptron**

After training the MLP with only 1 neuron (sole output neuron). The weights from this neuron can be used to produce a similar linear equation, with the addition of the sigmoid function (using standardized training set):

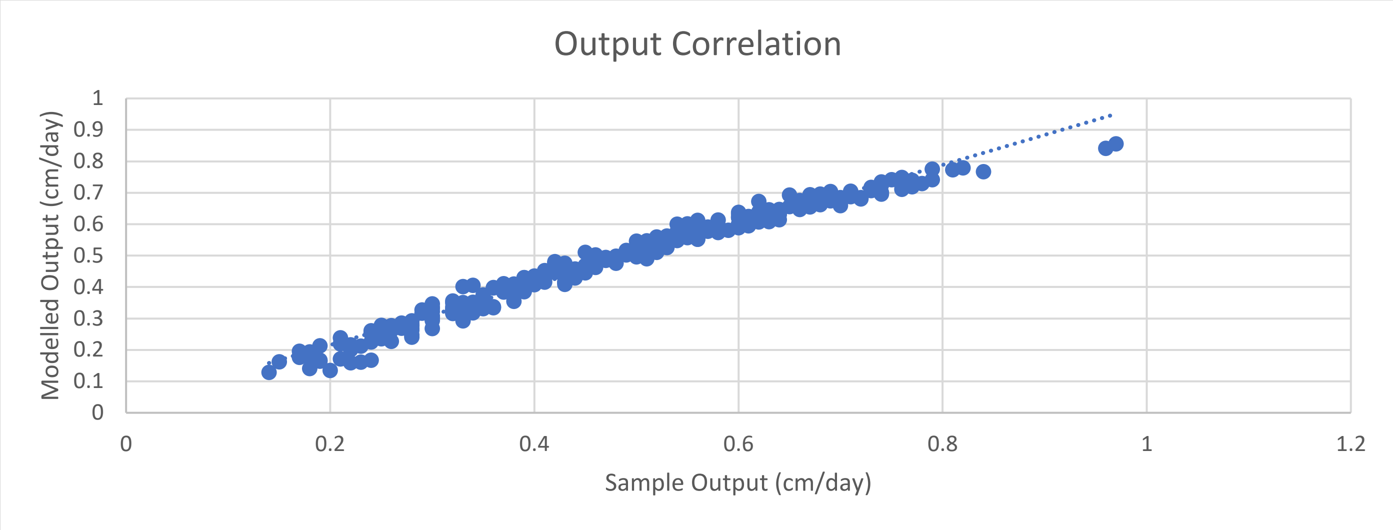
**PanE** = f(1. 726831 * **T** + 1. 525710 * **W** + 2.38017 * **SR** + -1.408130 * **DSP** + -2.500089 * **DRH** + -0.472245)

Evaluation metrics (measured on test set):

| | |
|---|---|
| RMSE – Root Mean Squared Error | 0.0274 (closer to 0 is better) |
| SRE - Mean Squared Relative Error | 0.0048 (closer to 0 is better) |
| CE - Coefficient of Efficiency | 0.9730 (closer to 1 is better) |
| RSQR - R-Squared (Determination Coefficient): | 0.9739 (closer to 1 is better) |

As you can see, in all evaluation metrics, this linear model performed slightly worse than the ANN but very similarly to the LINEST Simple multiple linear regression model.

See the graph below displaying the correlation between the sample and modelled output for this model.




Measured Correlation = 0.98796. Again, this is slightly worse than the ANN and near identical to the LINEST model.

As you can tell from the graph, this model is still very accurate and suffers in the same way as the LINEST model, it is notable less accurate at the tail ends of the data set. This model is presumable worse at predicting values outside the range it was trained on.

Overall, I believe this is a flaw associated with the linear model and not how the linear model was computed.

**Comparison to ANN model**

The ANN model performed better in all metrics. While the linear models are very accurate and are likely perfectly suitable, the ANN model is clearly superior and leads to better predicted results. Although, it is worth noting that the difference in performance is not sufficient as to warrant the additional resources needed to develop the ANN model.

# Code listings

## Neuron Class

```java
package ann.implementation;

import java.util.Arrays;

public class Neuron {

    private double[] inputs;
    private final double[] weights;
    private final double[] previousWeights;
    private final double[] weightsErrorGradient;

    private double weightSum = 0;
    private double activation = 0;
    private double deltaValue = 0;

    //Initialises neuron with initial wieghts
    public Neuron(double[] weights) {
        this.weights = weights;
        this.previousWeights = Arrays.copyOf(weights, weights.length);
        this.weightsErrorGradient = new double[weights.length];
        for (int i = 0; i < weightsErrorGradient.length; i++) {
            weightsErrorGradient[i] = 0;
        }
        weightSum = 0;
    }

    //Clone neuron from exisitng neuron
    public Neuron(Neuron neuron) {
        this.weights = Arrays.copyOf(neuron.getWeights(), neuron.getWeights().length);
        this.previousWeights = Arrays.copyOf(neuron.getPreviousWeights(), neuron.getPreviousWeights().length);
        this.weightsErrorGradient = new double[weights.length];
        for (int i = 0; i < weightsErrorGradient.length; i++) {
            weightsErrorGradient[i] = 0;
        }
        weightSum = 0;
    }

    public void setInputs(double[] inputs) {
        this.inputs = inputs;
    }

    public void computeWeightSum() {
        //weightSum = ∑(weights[i] * inputs[i])
        weightSum = 0;
        for (int i = 0; i < inputs.length; i++) {
            weightSum += weights[i] * inputs[i];
        }
    }

    public double computeActivation() {
        //activation, "output" = f(weightSum)
        activation = sigmoidFunction(weightSum);
        return activation;
    }

    public void computeDeltaValue(double sampleOutput, double upsilon) {
        //For output neurons:
        // Caculate omega (for wieght decay)
        double sum = 0;
        for (double weight : weights) {
            sum += Math.pow(weight, 2);
        }
        double omega = sum / (2 * weights.length);

        //deltaValue = (sampleOutput - activation, "modelled output" + upsilon * omega) * f'(weightSum)
        deltaValue = (sampleOutput - activation + upsilon * omega) * (activation * (1 - activation));
    }

    public void computeDeltaValue(double[] nextWeights, double[] nextDeltaValues) {
        //For non-output neurons:
        //deltaValue = (sum of following weights * following delta values ) * f'(weightSum)
        double sum = 0;
        for (int i = 0; i < nextWeights.length; i++) {
            sum += (nextWeights[i] * nextDeltaValues[i]);
        }
        deltaValue = sum * (activation * (1 - activation));
    }

    public void updateWeightsAndBias(double stepSize, double momentum) {
        //Update each weight:
        //new weight = old weight + stepSize * deltaValue * respective input
        for (int i = 0; i < weights.length; i++) {
            double tempWeight = weights[i];
            weights[i] += (stepSize * deltaValue * inputs[i])
                    //Momentum
                    + (momentum * (weights[i] - previousWeights[i]));
            previousWeights[i] = tempWeight;
        }
    }

    public void sumWeightsErrorGradient() {
        //Sum error gradient for each weight
        for (int i = 0; i < weightsErrorGradient.length; i++) {
            weightsErrorGradient[i] += deltaValue * inputs[i];
        }
```

```java
    }

    public void batchUpdateWeightAndBias(double stepSize, double momentum, int sampleCount) {
        //Update each weight:
        //new weight = old weight + stepSize * (error gradient / sample count)
        for (int i = 0; i < weights.length; i++) {
            double tempWeight = weights[i];
            weights[i] += (stepSize * (weightsErrorGradient[i] / sampleCount))
                    //Momentum
                    + (momentum * (weights[i] - previousWeights[i]));
            previousWeights[i] = tempWeight;
        }
        //Reset error gradient sum for each weight
        for (int i = 0; i < weightsErrorGradient.length; i++) {
            weightsErrorGradient[i] += 0;
        }
    }

    private double sigmoidFunction(double value) {
        return (1 / (1 + Math.pow(Math.E, -value)));
    }


    public double[] getWeights() {
        return weights;
    }

    public double[] getPreviousWeights() {
        return previousWeights;
    }

    public double getDeltaValue() {
        return deltaValue;
    }

    @Override
    public String toString() {
        return "Neuron{" + "inputs=" + Arrays.toString(inputs) + ", weights=" + Arrays.toString(weights) + ","
                + "\n\tweightSum=" + weightSum + ", activation=" + activation + ", deltaValue=" + deltaValue + '}';
    }
}

```

## Sample Class

```java
package ann.implementation;

public final class Sample {

    private final double[] data;

    public Sample(double[] data) {
        this.data = data;
    }

    public double[] getData() {
        return data;
    }

    @Override
    public String toString() {
        String str = "Sample:\t ";
        for (int i = 0; i < data.length - 1; i++) {
            str += String.format("\tpredictor %d: %.4f", i, data[i]);
        }
        str += String.format("\tpridictand: %.4f", data[data.length - 1]);
        return str;
    }

}

```

## ANN Implementation Class

```java
package ann.implementation;

import java.io.*;
import java.util.*;
import static org.apache.poi.ss.usermodel.CellType.NUMERIC;
import org.apache.poi.ss.usermodel.FormulaEvaluator;
import org.apache.poi.xssf.usermodel.XSSFRow;
import org.apache.poi.xssf.usermodel.XSSFSheet;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;

public class ANNImplementation {

    private static final ArrayList<Sample> entireDataSet = new ArrayList<>();
    //Data subsets
    private static final ArrayList<Sample> testDataSet = new ArrayList<>(); //About 20% of data
    private static final ArrayList<Sample> validationDataSet = new ArrayList<>(); //About 20% of data
    private static final ArrayList<Sample> trainingDataSet = new ArrayList<>(); //About 60% of data

    private static double[] minimums;
    private static double[] maximums;

    private static final XSSFWorkbook workbook = new XSSFWorkbook();
    private static int dataPointsCount;

    //ANN parameters
    private static int[] ANN_SHAPE = new int[]{14, 12};
    private static double MOMENTUM = 0.9d;
    private static int MAX_EPOCHS = 10000;
    private static double START_STEP_SIZE = 0.1d;
    private static double END_STEP_SIZE = 0.01d;
    private static double MAX_STEP_SIZE = 0.5d;
    private static double MIN_STEP_SIZE = 0.001d;
```

```java

    //ANN improvments
    private static boolean USE_MOMENTUM = true;
    private static boolean USE_BOLD_DRIVER = true;
    private static boolean USE_ANNEALING = false;
    private static boolean USE_WEIGHT_DECAY = false;
    private static boolean USE_BATCH_LEARING = false;

    private static final Random rand = new Random();

    public static void main(String[] args) throws IOException {

        Scanner input = new Scanner(System.in);

        //Load model paramaters from config.txt
        loadCongfig();
        //Read and process data set from DataSet.xlsx - test it option 3
        processDataSet();

        boolean askAgain = true;
        //Give the user a list of options
        while (askAgain) {
            try {
                System.out.println("\nWould you like to:\n"
                        + "(1) Train an new ANN on labelled data set?\n"
                        + "(2) Evaluate an existing ANN on labelled data set?\n"
                        + "(3) Execute existing ANN on unlabelled data set?\n"
                        + "(4) Save output and quit?");
                int choice = input.nextInt();
                ArrayList<Neuron[]> neuronLayers = null;
                switch (choice) {
                    //If Train an new ANN
                    case (1):
                        //Train MLP with given parameters
                        neuronLayers = trainANN();
                        System.out.println("\nModel Trained.");
                        boolean success = false;
                        while (!success) {
                            //Attempt to load text file specifed by user
                            System.out.println("Enter file name (not file path), enter \"no\" to not save ANN: ");
                            String fileName = input.next();
                            if (fileName.equals("no") || fileName.equals("NO") || fileName.equals("No")) {
                                break;
                            }
                            success = saveANN(neuronLayers, fileName);
                            if (success) {
                                System.out.println("Model Saved.");
                            }
                        }
                        break;

                    //If Evaluate an existing ANN
                    case (2):

                        //Continue to ask until valid file chosen
                        while (neuronLayers == null) {
                            //Attempt to load text file specifed by user
                            System.out.println("\nEnter file name (not file path), enter \"no\" to cancel: ");
                            String fileName = input.next();
                            if (fileName.equals("no") || fileName.equals("NO") || fileName.equals("No")) {
                                break;
                            }
                            neuronLayers = loadANN(fileName);
                            //If file exist with correct format
                            if (neuronLayers != null) {
                                //If ANN compatible with data set
                                if (neuronLayers.get(0)[0].getWeights().length == dataPointsCount) {
                                    //Evaluates data set using test set
                                    double[] evaluationMetrics = evaluateANN(neuronLayers, testDataSet, true);
                                    System.out.println("Evaluation Metrics: ");
                                    System.out.printf("\nRMSE - Root Mean Squared Error: \t\t\t%.4f \t(closer to 0 is better)\n", evaluationMetrics[0]);
                                    System.out.printf("SRE - Mean Squared Relative Error: \t\t%.4f \t(closer to 0 is better)\n", evaluationMetrics[1]);
                                    System.out.printf("CE - Coefficient of Efficiency: \t\t%.4f \t(closer to 1 is better)\n", evaluationMetrics[2]);
                                    System.out.printf("RSQR - R-Squared (Determination Coefficien): \t%.4f \t(closer to 1 is better)\n",
evaluationMetrics[3]);
                                } else {
                                    neuronLayers = null;
                                    System.out.println("ANN not compatible with data set - incorrect number of inputs");
                                }
                            } else {
                                System.out.println("Incorrect file name");
                            }
                        }
                        break;

                    //If Execute existing ANN on entire data set
                    case (3):
                        //Continue to ask until valid file chosen
                        while (neuronLayers == null) {
                            //Attempt to load text file specifed by user
                            System.out.println("\nEnter file name (not file path), enter \"no\" to cancel: ");
                            String fileName = input.next();
                            if (fileName.equals("no") || fileName.equals("NO") || fileName.equals("No")) {
                                break;
                            }
                            neuronLayers = loadANN(fileName);
                            //If file exist with correct format
                            if (neuronLayers != null) {
                                //If ANN compatible with data set
                                if (neuronLayers.get(0)[0].getWeights().length == dataPointsCount + 1) {
                                    //Executes data set on all inputs in data set
                                    executeANN(neuronLayers, entireDataSet);
                                    System.out.println("Model Executed.");
                                } else {
                                    neuronLayers = null;
                                    System.out.println("ANN not compatible with data set - incorrect number of inputs");
                                }
                            } else {
                                System.out.println("Incorrect file name");
```

```java
                            }
                        }
                        break;

                    //If Save output and quit
                    case (4):
                        askAgain = false;
                        break;

                    default:
                        System.out.println("Incorrect choice, try again");
                        askAgain = true;
                        break;
                }
            } catch (Exception e) {
                System.out.println("Incorrect choice, try again");
                input.next();
            }
        }

        //Save output to "Output.xlsx"
        try {
            FileOutputStream out = new FileOutputStream(new File("Output.xlsx"));
            workbook.write(out);
            out.close();
            System.out.println("\nSuccessfully saved output file");
        } catch (IOException e) {
            System.out.println("\nError creating output file. Error: " + e);
        }

    }

    private static void loadCongfig() {
        String dir = System.getProperty("user.dir") + "\\config.txt";
        File file = new File(dir);
        boolean defaultValues = false;

        if (file.exists()) {
            //Try get every value from config file
            try {
                Scanner reader = new Scanner(file);
                //Read each line in text file
                while (reader.hasNextLine()) {
                    //Convert each line to kay, value pair
                    String[] stringData = reader.nextLine().split("=");
                    if (stringData.length != 2) {
                        System.out.println("nahh");
                        continue;
                    }
                    String key = stringData[0].trim();
                    String value = stringData[1].trim();
                    //Set value for specified paramater
                    switch (key) {
                        case ("ANN_SHAPE"):
                            if (value.equals("")) {
                                ANN_SHAPE = new int[0];
                            } else {
                                String[] layerSizes = value.split(",");
                                ANN_SHAPE = new int[layerSizes.length];
                                for (int i = 0; i < layerSizes.length; i++) {
                                    ANN_SHAPE[i] = Integer.valueOf(layerSizes[i].trim());
                                }
                            }
                            break;
                        case ("MOMENTUM"):
                            MOMENTUM = Double.valueOf(value);
                            break;
                        case ("MAX_EPOCHS"):
                            MAX_EPOCHS = Integer.valueOf(value);
                            break;
                        case ("START_STEP_SIZE"):
                            START_STEP_SIZE = Double.valueOf(value);
                            break;
                        case ("END_STEP_SIZE"):
                            END_STEP_SIZE = Double.valueOf(value);
                            break;
                        case ("MAX_STEP_SIZE"):
                            MAX_STEP_SIZE = Double.valueOf(value);
                            break;
                        case ("MIN_STEP_SIZE"):
                            MIN_STEP_SIZE = Double.valueOf(value);
                            break;
                        case ("USE_MOMENTUM"):
                            USE_MOMENTUM = Boolean.valueOf(value);
                            break;
                        case ("USE_BOLD_DRIVER"):
                            USE_BOLD_DRIVER = Boolean.valueOf(value);
                            break;
                        case ("USE_ANNEALING"):
                            USE_ANNEALING = Boolean.valueOf(value);
                            break;
                        case ("USE_WEIGHT_DECAY"):
                            USE_WEIGHT_DECAY = Boolean.valueOf(value);
                            break;
                        case ("USE_BATCH_LEARING"):
                            USE_BATCH_LEARING = Boolean.valueOf(value);
                            break;
                    }
                }
                reader.close();
            } //If there is any errors (because of incorrect format) then use default values instead
            catch (Exception e) {
                System.out.println("An error occurred loading config file. Error: " + e);
                defaultValues = true;
            }
        } //If there is no config file then use default values instead
        else {
            System.out.println("Config file does not exist, using default values.");
            defaultValues = true;
```

```java
        }
        //Reset model paramaters to default if necessary
        if (defaultValues) {
            ANN_SHAPE = new int[]{10, 8};
            MOMENTUM = 0.9;
            MAX_EPOCHS = 1000;
            START_STEP_SIZE = 0.1;
            END_STEP_SIZE = 0.01;
            MAX_STEP_SIZE = 0.5;
            MIN_STEP_SIZE = 0.001;
            USE_MOMENTUM = true;
            USE_BOLD_DRIVER = true;
            USE_ANNEALING = false;
            USE_WEIGHT_DECAY = false;
            USE_BATCH_LEARING = false;
        }
        //Display used model paramaters
        System.out.println("ANN parameters:");
        System.out.println("ANN_SHAPE = " + Arrays.toString(ANN_SHAPE));
        System.out.println("MOMENTUM = " + MOMENTUM);
        System.out.println("MAX_EPOCHS = " + MAX_EPOCHS);
        System.out.println("START_STEP_SIZE = " + START_STEP_SIZE);
        System.out.println("END_STEP_SIZE = " + END_STEP_SIZE);
        System.out.println("MAX_STEP_SIZE = " + MAX_STEP_SIZE);
        System.out.println("MIN_STEP_SIZE = " + MIN_STEP_SIZE);
        System.out.println("USE_MOMENTUM = " + USE_MOMENTUM);
        System.out.println("USE_BOLD_DRIVER = " + USE_BOLD_DRIVER);
        System.out.println("USE_ANNEALING = " + USE_ANNEALING);
        System.out.println("USE_WEIGHT_DECAY = " + USE_WEIGHT_DECAY);
        System.out.println("USE_BATCH_LEARING = " + USE_BATCH_LEARING);

        if (!USE_MOMENTUM) {
            MOMENTUM = 0;
        }
    }

    private static void processDataSet() throws FileNotFoundException, IOException {

        final ArrayList<Sample> dataSet = new ArrayList<>();

        //Read data set from DataSet.xlsx
        FileInputStream inputFile = new FileInputStream(new File("DataSet.xlsx"));
        XSSFWorkbook dataSetWorkbook = new XSSFWorkbook(inputFile);
        XSSFSheet sheet = dataSetWorkbook.getSheetAt(0);
        FormulaEvaluator evaluator = dataSetWorkbook.getCreationHelper().createFormulaEvaluator();

        int sampleCount = sheet.getPhysicalNumberOfRows();
        dataPointsCount = sheet.getRow(0).getPhysicalNumberOfCells();

        //Extra information for each column of data (used for data pre-processing)
        minimums = new double[dataPointsCount];
        maximums = new double[dataPointsCount];

        //Read every row in sheet
        for (int row = 0; row < sampleCount; row++) {
            //Set to false if a non-numeric value is detected in a sample
            boolean validSample = true;
            //Read row from data set
            double[] sampleData = new double[dataPointsCount];
            for (int col = 0; col < dataPointsCount; col++) {
                if (sheet.getRow(row).getCell(col) != null && evaluator.evaluateInCell(
                        sheet.getRow(row).getCell(col)).getCellTypeEnum() == NUMERIC) {
                    sampleData[col] = sheet.getRow(row).getCell(col).getNumericCellValue();
                    //Calculate min and max values for input (column), used for standardization later
                    minimums[col] = Math.min(minimums[col], sampleData[col]);
                    maximums[col] = Math.max(maximums[col], sampleData[col]);
                } else {
                    validSample = false;
                    break;
                }
            }
            //Invalid samples are ignored
            if (!validSample) {
                continue;
            }
            //Convert each row of data into a Sample data object and add it to dataSet
            dataSet.add(new Sample(sampleData));
        }

        System.out.println("\nReading Data Set:");

        //Loop through data set
        for (Sample sample : dataSet) {
            double[] sampleData = sample.getData();
            //Standardize each value between range [0.1, 0.9]
            for (int i = 0; i < sampleData.length; i++) {
                sampleData[i] = standardize(sampleData[i], maximums[i], minimums[i]);
            }
            //Display sample data
            System.out.println(sample);
            //Randomly place sample between 3 data sets
            switch (rand.nextInt(5)) {
                case (0): //20% chance
                    testDataSet.add(sample);
                    break;
                case (1): //20% chance
                    validationDataSet.add(sample);
                    break;
                default: //60% chance
                    trainingDataSet.add(sample);
                    break;
            }
            entireDataSet.add(sample);
        }
        //Display data subset sizes
        System.out.printf("\nTotal number of samples: %d\n", entireDataSet.size());
        System.out.printf("Test set size: %.2f%%\n", (double) testDataSet.size() / dataSet.size() * 100);
        System.out.printf("Validation set size: %.2f%%\n", (double) validationDataSet.size() / dataSet.size() * 100);
        System.out.printf("Training set size: %.2f%%\n", (double) trainingDataSet.size() / dataSet.size() * 100);
```

```
    }

    private static double standardize(double value, double max, double min) {
        //Standardizes value between range [0.1, 0.9]
        return 0.8 * ((value - min) / (max - min)) + 0.1;
    }

    private static double deStandardize(double value, double max, double min) {
        //Converts value back from range [0.1, 0.9] to normal range
        return ((value - 0.1) / 0.8) * (max - min) + min;
    }

    private static ArrayList<Neuron[]> trainANN() {

        if (workbook.getSheetIndex("Training Data") != -1) {
            workbook.removeSheetAt(workbook.getSheetIndex("Training Data"));
        }
        XSSFSheet spreadsheet = workbook.createSheet("Training Data");
        //Save epoch data Headers
        XSSFRow header = spreadsheet.createRow(0);
        header.createCell(0).setCellValue("Epoch");
        header.createCell(1).setCellValue("RMSE");
        header.createCell(2).setCellValue("Step size");
        int line = 1;

        double stepSize = START_STEP_SIZE;

        //Array list of neuron layers represents ANN
        final ArrayList<Neuron[]> neuronLayers = new ArrayList<>();
        //Create uninitialised hidden layers of neurons
        for (int hiddenLayerSize : ANN_SHAPE) {
            if (hiddenLayerSize > 0) {
                neuronLayers.add(new Neuron[hiddenLayerSize]);
            }
        }
        //Create uninitialised output neuron
        neuronLayers.add(new Neuron[1]);
        //Initialise neurons with random weights
        initialiseNeuronsWithRandomWeights(neuronLayers);

        //Copy of previous neural network instance (used for bold driver)
        final ArrayList<Neuron[]> prevNeuronLayers = new ArrayList<>();
        for (int hiddenLayerSize : ANN_SHAPE) {
            if (hiddenLayerSize > 0) {
                prevNeuronLayers.add(new Neuron[hiddenLayerSize]);
            }
        }
        prevNeuronLayers.add(new Neuron[1]);
        cloneNeuronLayers(neuronLayers, prevNeuronLayers);

        //Previously measured RMSE
        double error = Double.MAX_VALUE;
        int tinyErrorCount = 0;

        //Train for specified number of epochs
        for (int i = 0; i < MAX_EPOCHS; i++) {
            //Loop through every sample in training set
            for (Sample sample : trainingDataSet) {
                //Inputs and bias passed to first layer of hidden neurons
                double[] inputsAndBias = new double[sample.getData().length];
                System.arraycopy(sample.getData(), 0, inputsAndBias, 0, inputsAndBias.length - 1);
                inputsAndBias[inputsAndBias.length - 1] = 1;
                //Correct output specified by sample
                double sampleOutput = sample.getData()[sample.getData().length - 1];
                //Forward pass through ANN
                forwardPass(neuronLayers, inputsAndBias);
                //Backward pass through ANN
                backwardPass(neuronLayers, sampleOutput, stepSize, i + 1);

                if (!USE_BATCH_LEARING) {
                    //Update weights for every neuron in ANN
                    updateWeights(neuronLayers, stepSize);
                } else {
                    //Sum error gradient for all weights
                    sumWeightsErrorGradient(neuronLayers);
                }
            }

            //Update weights at the end of the epoch
            if (USE_BATCH_LEARING) {
                batchUpdateWeights(neuronLayers, stepSize, trainingDataSet.size());
            }

            //Simulated annealing
            if (USE_ANNEALING) {
                stepSize = annealing(i + 1);
            }

            //100 times while training (every 1% of training complete)
            if (i % (Math.max(MAX_EPOCHS / 100, 1)) == 0) {
                //Get Root Mean Squared Error from evaluating model on validation set
                double RMSE = evaluateANN(neuronLayers, validationDataSet, false)[0];

                if (USE_BOLD_DRIVER) {
                    double errorDiff = ((RMSE - error) / error);
                    //If the error increases by over 1 % then half the step size and revert the weights back to the prev weights
                    if (errorDiff > 0.01d) {
                        stepSize *= 0.5;
                        stepSize = Math.max(MIN_STEP_SIZE, stepSize);
                        //revert model to last bold driver
                        cloneNeuronLayers(prevNeuronLayers, neuronLayers);
                    } else {
                        //If the error decreases by over 1 % then slightly increase step size
                        if (errorDiff < -0.01d) {
                            stepSize *= 1.05;
                            stepSize = Math.min(MAX_STEP_SIZE, stepSize);
                        }
                        cloneNeuronLayers(neuronLayers, prevNeuronLayers);
                    }
```

```java
            }
            //Display epoch data
            System.out.printf("\nEpoch %d completed - %.2f%% \tStep Size:%f \tError:%f", i, (double) i / MAX_EPOCHS * 100, stepSize, RMSE);

            //Save epoch data
            XSSFRow row = spreadsheet.createRow(line++);
            row.createCell(0).setCellValue(i);
            row.createCell(1).setCellValue(RMSE);
            row.createCell(2).setCellValue(stepSize);

            //Terminate training early if error change is negligible 3 times in a row
            if (Math.abs(RMSE - error) < 0.00001d) {
                tinyErrorCount++;
                if (tinyErrorCount >= 3) {
                    System.out.println(" - Change in Error minimal, stopping training");
                    break;
                }
            } else {
                tinyErrorCount = 0;
            }
            error = RMSE;
        }
    }

    return neuronLayers;
}

private static void initialiseNeuronsWithRandomWeights(ArrayList<Neuron[]> neuronLayers) {
    //Initialise all neurons with random weights and bias
    for (int i = 0; i < neuronLayers.size(); i++) {
        int weightsCount;
        //If first layer of hidden neurons then #weights = #sampleInputs + 1
        if (i == 0) {
            weightsCount = dataPointsCount;
        } //Otherwise, #weights = #previousLayerNeurons + 1
        else {
            weightsCount = neuronLayers.get(i - 1).length + 1;
        }
        for (int j = 0; j < neuronLayers.get(i).length; j++) {
            //Number of weights (#inputs + 1 for bias) for neuron
            double[] weightsAndBias = new double[weightsCount];
            //Randomise weights and bias values between [-2/n, 2/n] where n is the input size of the neuron
            int range = weightsAndBias.length - 1;
            for (int k = 0; k < weightsAndBias.length; k++) {
                //Random value between [-2/n, 2/n] where n is the input size
                weightsAndBias[k] = -2d / range + (2d / range - -2d / range) * rand.nextDouble();
            }
            neuronLayers.get(i)[j] = new Neuron(weightsAndBias);
        }
    }
}

private static void cloneNeuronLayers(ArrayList<Neuron[]> original, ArrayList<Neuron[]> clone) {
    //Copy weights values from original to clone
    for (int i = 0; i < original.size(); i++) {
        for (int j = 0; j < original.get(i).length; j++) {
            clone.get(i)[j] = new Neuron(original.get(i)[j]);
        }
    }
}

private static double forwardPass(ArrayList<Neuron[]> neuronLayers, double[] sampleInputsAndBias) {
    //Outputs from every neuron in the layer, used as inputs to the next layer
    double[] layerOutputs = new double[0];

    //Forward pass through every layer in ANN
    for (int i = 0; i < neuronLayers.size(); i++) {
        //Inputs and bias for every neuron in the layer
        double[] inputsAndBias;
        //If first layer then inputs equal to inputs from the sample
        if (i == 0) {
            inputsAndBias = sampleInputsAndBias;
        } //Otherwise, inputs equal to outputs of previous layer
        else {
            inputsAndBias = layerOutputs;
        }
        //Clears layer outputs and sets bias input to 1
        layerOutputs = new double[neuronLayers.get(i).length + 1];
        layerOutputs[layerOutputs.length - 1] = 1;
        //Set inputs, calculate weight sum, compute activation for every neuron in layer
        for (int j = 0; j < neuronLayers.get(i).length; j++) {
            neuronLayers.get(i)[j].setInputs(inputsAndBias);
            neuronLayers.get(i)[j].computeWeightSum();
            layerOutputs[j] = neuronLayers.get(i)[j].computeActivation();
        }
    }
    //Returns the output of the final neuron (output neuron), the output of the entire ANN
    return layerOutputs[layerOutputs.length - 2];
}

private static void backwardPass(ArrayList<Neuron[]> neuronLayers, double sampleOutput, double stepSize, int epoch) {
    //Backward pass through every layer in ANN
    for (int i = neuronLayers.size() - 1; i >= 0; i--) {
        //If last layer (output layers)
        if (i == neuronLayers.size() - 1) {
            //Compute delta value for output neuron
            for (Neuron outputNeuron : neuronLayers.get(i)) {
                // Caculate upsilon (for weight decay)
                double upsilon = 1 / (stepSize * epoch);
                if (!USE_WEIGHT_DECAY || epoch < 100) {
                    upsilon = 0;
                }
                //Compute delta value for output neuron
                outputNeuron.computeDeltaValue(sampleOutput, upsilon);
            }
        } //Otherwise, if hidden layer
        else {
            //Loop through neurons in hidden layer
            for (int j = 0; j < neuronLayers.get(i).length; j++) {
```

```java
                    double[] forwardWeights = new double[neuronLayers.get(i + 1).length];
                    double[] forwardDeltaValues = new double[neuronLayers.get(i + 1).length];
                    //Loop through forward neurons
                    for (int k = 0; k < neuronLayers.get(i + 1).length; k++) {
                        Neuron forwardNeuron = neuronLayers.get(i + 1)[k];
                        //Store weights between current neuron and all forward neurons
                        forwardWeights[k] = forwardNeuron.getWeights()[j];
                        //Store delta values of every forward neuron
                        forwardDeltaValues[k] = forwardNeuron.getDeltaValue();
                    }
                    //Compute delta value for hidden neuron
                    neuronLayers.get(i)[j].computeDeltaValue(forwardWeights, forwardDeltaValues);
                }
            }
        }
    }

    private static void updateWeights(ArrayList<Neuron[]> neuronLayers, double stepSize) {
        //Calculates new weights for every neuron in ANN
        for (Neuron[] neuronLayer : neuronLayers) {
            for (Neuron neuron : neuronLayer) {
                neuron.updateWeightsAndBias(stepSize, MOMENTUM);
            }
        }
    }

    private static void batchUpdateWeights(ArrayList<Neuron[]> neuronLayers, double stepSize, int sampleCount) {
        //Calculates new weights for every neuron in ANN
        for (Neuron[] neuronLayer : neuronLayers) {
            for (Neuron neuron : neuronLayer) {
                neuron.batchUpdateWeightAndBias(stepSize, MOMENTUM, sampleCount);
            }
        }
    }

    private static void sumWeightsErrorGradient(ArrayList<Neuron[]> neuronLayers) {
        //sum every weights error gradient for every neuron in ANN
        for (Neuron[] neuronLayer : neuronLayers) {
            for (Neuron neuron : neuronLayer) {
                neuron.sumWeightsErrorGradient();
            }
        }
    }

    private static double annealing(double epoch) {
        //Decays stepSize (learning parameter) over time
        return END_STEP_SIZE + (START_STEP_SIZE - END_STEP_SIZE)
                * (1 - (1 / (1 + Math.pow(Math.E, (10 - 20 * (epoch / MAX_EPOCHS))))));
    }

    private static double[] evaluateANN(ArrayList<Neuron[]> neuronLayers, ArrayList<Sample> dataSubset, boolean store) {
        XSSFSheet spreadsheet = null;
        int line = 1;
        if (store) {
            if (workbook.getSheetIndex("Evaluation Data") != -1) {
                workbook.removeSheetAt(workbook.getSheetIndex("Evaluation Data"));
            }
            spreadsheet = workbook.createSheet("Evaluation Data");
            //Save evaluation data Headers
            XSSFRow header = spreadsheet.createRow(0);
            header.createCell(0).setCellValue("Sample outputs");
            header.createCell(1).setCellValue("Modelled output");
        }

        //RMSE, MSRE, CE, RSQR
        double[] evaluationMetrics = new double[4];

        double[] sampleOutputs = new double[dataSubset.size()];
        double[] modelledOutputs = new double[dataSubset.size()];

        //Variables used for calculation
        double RMSEsum = 0, MSREsum = 0, CEnumerator = 0, CEdenominator = 0, RSQRnumerator = 0, RSQRdenominatorLeft = 0, RSQRdenominatorRight = 0;

        double sampleOutputsMean = 0;
        double modelledOutputsMean = 0;
        //Store sample output and modelled output for every sample in given data subset
        for (int i = 0; i < dataSubset.size(); i++) {
            Sample sample = dataSubset.get(i);
            //Inputs and bias passed to first layer of hidden neurons
            double[] inputsAndBias = new double[sample.getData().length];
            System.arraycopy(sample.getData(), 0, inputsAndBias, 0, inputsAndBias.length - 1);
            inputsAndBias[inputsAndBias.length - 1] = 1;

            //Correct output specified by sample
            double sampleOutput = sample.getData()[sample.getData().length - 1];
            //De-standardize sample output
            sampleOutputs[i] = deStandardize(sampleOutput, maximums[maximums.length - 1], minimums[minimums.length - 1]);
            //Forward pass through ANN
            double modelledOutput = forwardPass(neuronLayers, inputsAndBias);
            //De-standardize modelled output
            modelledOutputs[i] = deStandardize(modelledOutput, maximums[maximums.length - 1], minimums[minimums.length - 1]);

            //Calculate mean for both values
            sampleOutputsMean += sampleOutputs[i];
            modelledOutputsMean += modelledOutputs[i];

            //Save evaluation data
            if (store && spreadsheet != null) {
                XSSFRow row = spreadsheet.createRow(line++);
                row.createCell(0).setCellValue(sampleOutputs[i]);
                row.createCell(1).setCellValue(modelledOutputs[i]);
            }
        }

        sampleOutputsMean /= dataSubset.size();
        modelledOutputsMean /= dataSubset.size();

        //Calculate evaluation metrics
        for (int i = 0; i < dataSubset.size(); i++) {
```

```
            RMSEsum += Math.pow(modelledOutputs[i] - sampleOutputs[i], 2);

            MSREsum += Math.pow((modelledOutputs[i] - sampleOutputs[i]) / sampleOutputs[i], 2);

            CEnumerator += Math.pow(modelledOutputs[i] - sampleOutputs[i], 2);
            CEdenominator += Math.pow(sampleOutputs[i] - sampleOutputsMean, 2);

            RSQRnumerator += (sampleOutputs[i] - sampleOutputsMean) * (modelledOutputs[i] - modelledOutputsMean);
            RSQRdenominatorLeft += Math.pow(sampleOutputs[i] - sampleOutputsMean, 2);
            RSQRdenominatorRight += Math.pow(modelledOutputs[i] - modelledOutputsMean, 2);
        }
        //RMSE – Root Mean Squared Error (closer to 0 is better)
        evaluationMetrics[0] = Math.sqrt(RMSEsum / dataSubset.size());
        //MSRE - Mean Squared Relative Error (closer to 0 is better)
        evaluationMetrics[1] = MSREsum / dataSubset.size();
        //CE - Coefficient of Efficiency (closer to 1 is better)
        evaluationMetrics[2] = 1 - CEnumerator / CEdenominator;
        //RSQR - R-Squared (Coefficient of Determination) (closer to 1 is better)
        evaluationMetrics[3] = Math.pow(RSQRnumerator / (Math.sqrt(RSQRdenominatorLeft * RSQRdenominatorRight)), 2);

        return evaluationMetrics;
    }

    private static void executeANN(ArrayList<Neuron[]> neuronLayers, ArrayList<Sample> dataSubset) {
        int line = 1;
        if (workbook.getSheetIndex("Execution Data") != -1) {
            workbook.removeSheetAt(workbook.getSheetIndex("Execution Data"));
        }
        XSSFSheet spreadsheet = workbook.createSheet("Execution Data");
        //Save evaluation data Headers
        XSSFRow header = spreadsheet.createRow(0);
        header.createCell(0).setCellValue("Modelled output (standardized)");

        //Store modelled output for every sample in given data subset
        for (int i = 0; i < dataSubset.size(); i++) {
            Sample sample = dataSubset.get(i);
            //Inputs and bias passed to first layer of hidden neurons
            double[] inputsAndBias;
            inputsAndBias = new double[sample.getData().length + 1];
            System.arraycopy(sample.getData(), 0, inputsAndBias, 0, inputsAndBias.length - 1);
            inputsAndBias[inputsAndBias.length - 1] = 1;

            //Forward pass through ANN
            double modelledOutput = forwardPass(neuronLayers, inputsAndBias);

            //Save execution data
            XSSFRow row = spreadsheet.createRow(line++);
            row.createCell(0).setCellValue(modelledOutput);
        }
    }

    private static boolean saveANN(ArrayList<Neuron[]> neuronLayers, String fileName) {
        String dir = System.getProperty("user.dir") + "\\ANN Models\\" + fileName;

        try {
            File file = new File(dir);
            //Overwrites file if it already exits
            if (!file.createNewFile()) {
                file.delete();
            }

            FileWriter writeToFile = new FileWriter(dir, true);
            PrintWriter printToFile = new PrintWriter(writeToFile);

            String ANNweights = "";
            for (Neuron[] neuronLayer : neuronLayers) {
                //Write ANN structure to fist line
                printToFile.print(neuronLayer.length + ",");
                //Write weights for each neuron on seperate lines
                for (Neuron neuron : neuronLayer) {
                    String neuronWeights = "";
                    for (double weight : neuron.getWeights()) {
                        neuronWeights += weight + ",";
                    }
                    ANNweights += "\n" + neuronWeights;
                }
            }
            printToFile.println(ANNweights);
            printToFile.close();
            writeToFile.close();
        } catch (IOException e) {
            System.out.println("An error occurred saving ANN. Error: " + e);
            return false;
        }
        return true;
    }

    private static ArrayList<Neuron[]> loadANN(String fileName) {

        ArrayList<double[]> fileNumericData = new ArrayList<>();

        String dir = System.getProperty("user.dir") + "\\ANN Models\\" + fileName;

        File file = new File(dir);
        if (file.exists()) {
            try {
                Scanner reader = new Scanner(file);
                //Read each line in text file
                while (reader.hasNextLine()) {
                    //Convert each line to array of doubles
                    String[] stringData = reader.nextLine().split(",");
                    double[] numericData = new double[stringData.length];
                    for (int i = 0; i < stringData.length; i++) {
                        numericData[i] = Double.valueOf(stringData[i]);
                    }
                    fileNumericData.add(numericData);
                }
                reader.close();
                //Get ANN structure (number of layers and neurons per layer from first line)
```

```java
                int[] hiddenLayerSizes = new int[fileNumericData.get(0).length];
                for (int i = 0; i < fileNumericData.get(0).length; i++) {
                    hiddenLayerSizes[i] = (int) fileNumericData.get(0)[i];
                }
                //Create array list of neuron layers represents ANN
                final ArrayList<Neuron[]> neuronLayers = new ArrayList<>();
                //Create uninitialised hidden layers of neurons
                for (int hiddenLayerSize : hiddenLayerSizes) {
                    neuronLayers.add(new Neuron[hiddenLayerSize]);
                }
                int line = 1;
                //Initialise all neurons with weights from the file
                for (int i = 0; i < neuronLayers.size(); i++) {
                    for (int j = 0; j < neuronLayers.get(i).length; j++) {
                        neuronLayers.get(i)[j] = new Neuron(fileNumericData.get(line++));
                    }
                }
                //Return loaded ANN
                return neuronLayers;
            } catch (IOException e) {
                System.out.println("An error occurred loading ANN. Error: " + e);
            }
        } else {
            System.out.println("The file does not exist.");
        }

        return null;
    }
}
```