# 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.
* Text

  Description automatically generatedjava.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:

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).

Table

Description automatically generatedThe 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).

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.

Text

Description automatically generated

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:

**Table

Description automatically generated**

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:

Table

Description automatically generated

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.

A picture containing timeline

Description automatically generated

**Standardization**

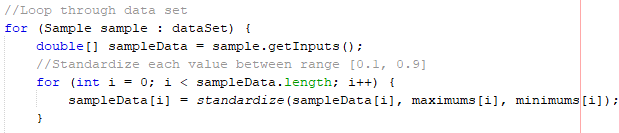
For the sake of avoiding floating point errors and avoiding certain inputs having more influence than others, every input value is standardized to the range [0.1,0.9] by the following methods:

Text

Description automatically generatedText, letter

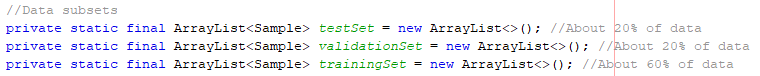
Description automatically generated

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:

**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.

****

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.

Text

Description automatically generated

**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:

Text

Description automatically generated

A picture containing text

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

Graphical user interface, text, application

Description automatically generated

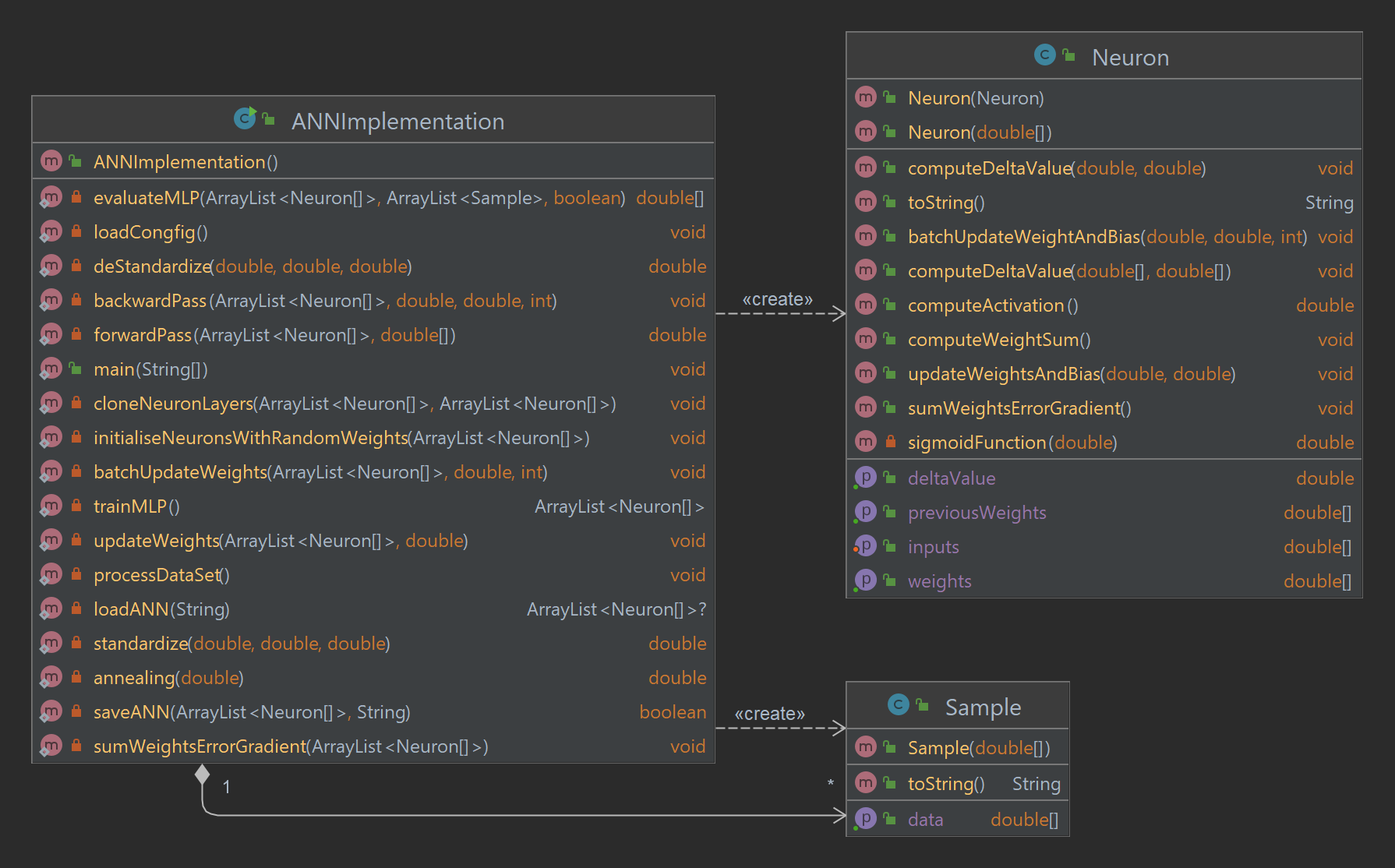
**Representation of the ANN**

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.

**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:

**Graphical user interface, text

Description automatically generated**

**Training the ANN**

A picture containing graphical user interface

Description automatically generatedThe 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.

Text

Description automatically generated

A picture containing text, clock

Description automatically generated**Activation** **function**

I have used the following function for the activation fucntion

**Text

Description automatically generated**

**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.

**Text, whiteboard

Description automatically generatedText, whiteboard

Description automatically generatedGraphical user interface, text, application

Description automatically generated**

**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.

**Text

Description automatically generatedGraphical user interface, text, application, email

Description automatically generated**

**Updating the weights**

Text, letter

Description automatically generatedFinally, updates the weights for every neuron in the ANN.

**A picture containing text, clock, watch, gauge

Description automatically generated**

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

Graphical user interface, application

Description automatically generatedOn 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.

Diagram, schematic

Description automatically generatedAfter the first forward pass:

1st

2nd

3rd

Text, letter

Description automatically generated

Diagram, schematic

Description automatically generatedAfter the first backward pass:

1st

2nd

3rd

Text, letter

Description automatically generated

**Diagram

Description automatically generated**After first updating the weights:

1st

2nd

3rd

A screenshot of a computer

Description automatically generated with medium confidence

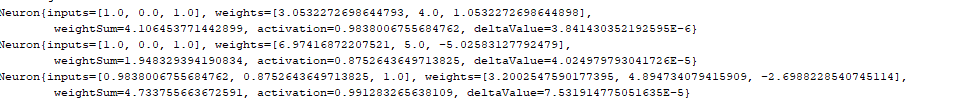
**A picture containing text, clock, watch

Description automatically generated**After 20,000 epochs:

1st

2nd

3rd

****

**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**

Text

Description automatically generatedLeads to more rapid progress along the valley compared with unmodified gradient descent.

A picture containing logo

Description automatically generated

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**

Graphical user interface, text, application, email

Description automatically generatedAlters the step size based on changes in the error function.

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:

**A picture containing text, clock

Description automatically generatedSimulated** **Annealing**

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



Chart, line chart

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

A screenshot of a computer

Description automatically generated with medium confidence

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.

A picture containing diagram

Description automatically generatedA picture containing text, clock

Description automatically generatedA picture containing text, seat

Description automatically generatedGraphical user interface, text

Description automatically generatedText, letter

Description automatically generated

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.

Graphical user interface, text, application, email

Description automatically generated

Text

Description automatically generated

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.

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.

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

# ANN Evaluation

Topics: Evaluation metrics, evaluation of final model.

**Evaluation Metrics**

A picture containing text

Description automatically generatedThere are four metrics that I use to evaluation the model

**RMSE** – Root Mean Squared Error

A picture containing text, indoor, screenshot

Description automatically generatedText

Description automatically generatedAverage absolute difference in real units

**MSRE** –Mean Squared Relative Error

Text

Description automatically generated with low confidenceAverage relative difference

**CE** – Coefficient of Efficiency

Direction of model correctness

A picture containing text

Description automatically generated**RSqr** – R-Squared (Coefficient of Determination)

Coincidence of the shape

Text

Description automatically generatedText

Description automatically generatedBelow is my implementation of model evaluation:

**Final Model Evaluation**

The final selected model had the following parameters:

Text

Description automatically generated

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**

1. package ann.implementation;

2.

3. import java.util.Arrays;

4.

5. public class Neuron {

6.

7. private double[] inputs;

8. private final double[] weights;

9. private final double[] previousWeights;

10. private final double[] weightsErrorGradient;

11.

12. private double weightSum = 0;

13. private double activation = 0;

14. private double deltaValue = 0;

15.

16. //Initialises neuron with initial wieghts

17. public Neuron(double[] weights) {

18. this.weights = weights;

19. this.previousWeights = Arrays.copyOf(weights, weights.length);

20. this.weightsErrorGradient = new double[weights.length];

21. for (int i = 0; i < weightsErrorGradient.length; i++) {

22. weightsErrorGradient[i] = 0;

23. }

24. weightSum = 0;

25. }

26.

27. //Clone neuron from exisitng neuron

28. public Neuron(Neuron neuron) {

29. this.weights = Arrays.copyOf(neuron.getWeights(), neuron.getWeights().length);

30. this.previousWeights = Arrays.copyOf(neuron.getPreviousWeights(), neuron.getPreviousWeights().length);

31. this.weightsErrorGradient = new double[weights.length];

32. for (int i = 0; i < weightsErrorGradient.length; i++) {

33. weightsErrorGradient[i] = 0;

34. }

35. weightSum = 0;

36. }

37.

38. public void setInputs(double[] inputs) {

39. this.inputs = inputs;

40. }

41.

42. public void computeWeightSum() {

43. //weightSum = ∑(weights[i] \* inputs[i])

44. weightSum = 0;

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

46. weightSum += weights[i] \* inputs[i];

47. }

48. }

49.

50. public double computeActivation() {

51. //activation, "output" = f(weightSum)

52. activation = sigmoidFunction(weightSum);

53. return activation;

54. }

55.

56.

57.

58.

59.

60.

61.

62.

63. public void computeDeltaValue(double sampleOutput, double upsilon) {

64. //For output neurons:

65. // Caculate omega (for wieght decay)

66. double sum = 0;

67. for (double weight : weights) {

68. sum += Math.pow(weight, 2);

69. }

70. double omega = sum / (2 \* weights.length);

71.

72. //deltaValue = (sampleOutput - activation, "modelled output" + upsilon \* omega) \* f'(weightSum)

73. deltaValue = (sampleOutput - activation + upsilon \* omega) \* (activation \* (1 - activation));

74. }

75.

76. public void computeDeltaValue(double[] nextWeights, double[] nextDeltaValues) {

77. //For non-output neurons:

78. //deltaValue = (sum of following weights \* following delta values ) \* f'(weightSum)

79. double sum = 0;

80. for (int i = 0; i < nextWeights.length; i++) {

81. sum += (nextWeights[i] \* nextDeltaValues[i]);

82. }

83. deltaValue = sum \* (activation \* (1 - activation));

84. }

85.

86. public void updateWeightsAndBias(double stepSize, double momentum) {

87. //Update each weight:

88. //new weight = old weight + stepSize \* deltaValue \* respective input

89. for (int i = 0; i < weights.length; i++) {

90. double tempWeight = weights[i];

91. weights[i] += (stepSize \* deltaValue \* inputs[i])

92. //Momentum

93. + (momentum \* (weights[i] - previousWeights[i]));

94. previousWeights[i] = tempWeight;

95. }

96. }

97.

98. public void sumWeightsErrorGradient() {

99. //Sum error gradient for each weight

100. for (int i = 0; i < weightsErrorGradient.length; i++) {

101. weightsErrorGradient[i] += deltaValue \* inputs[i];

102. }

103. }

104.

105. public void batchUpdateWeightAndBias(double stepSize, double momentum, int sampleCount) {

106. //Update each weight:

107. //new weight = old weight + stepSize \* (error gradient / sample count)

108. for (int i = 0; i < weights.length; i++) {

109. double tempWeight = weights[i];

110. weights[i] += (stepSize \* (weightsErrorGradient[i] / sampleCount))

111. //Momentum

112. + (momentum \* (weights[i] - previousWeights[i]));

113. previousWeights[i] = tempWeight;

114. }

115. //Reset error gradient sum for each weight

116. for (int i = 0; i < weightsErrorGradient.length; i++) {

117. weightsErrorGradient[i] += 0;

118. }

119. }

120.

121. private double sigmoidFunction(double value) {

122. return (1 / (1 + Math.pow(Math.E, -value)));

123. }

124.

125.

126. public double[] getWeights() {

127. return weights;

128. }

129.

130. public double[] getPreviousWeights() {

131. return previousWeights;

132. }

133.

134. public double getDeltaValue() {

135. return deltaValue;

136. }

137.

138. @Override

139. public String toString() {

140. return "Neuron{" + "inputs=" + Arrays.toString(inputs) + ", weights=" + Arrays.toString(weights) + ","

141. + "\n\tweightSum=" + weightSum + ", activation=" + activation + ", deltaValue=" + deltaValue + '}';

142. }

143. }

144.

**Sample Class**

1. package ann.implementation;

2.

3. public final class Sample {

4.

5. private final double[] data;

6.

7. public Sample(double[] data) {

8. this.data = data;

9. }

10.

11. public double[] getData() {

12. return data;

13. }

14.

15. @Override

16. public String toString() {

17. String str = "Sample:\t ";

18. for (int i = 0; i < data.length - 1; i++) {

19. str += String.format("\tpredictor %d: %.4f", i, data[i]);

20. }

21. str += String.format("\tpridictand: %.4f", data[data.length - 1]);

22. return str;

23. }

24.

25. }

26.

**ANN Implementation Class**

1. package ann.implementation;

2.

3. import java.io.\*;

4. import java.util.\*;

5. import static org.apache.poi.ss.usermodel.CellType.NUMERIC;

6. import org.apache.poi.ss.usermodel.FormulaEvaluator;

7. import org.apache.poi.xssf.usermodel.XSSFRow;

8. import org.apache.poi.xssf.usermodel.XSSFSheet;

9. import org.apache.poi.xssf.usermodel.XSSFWorkbook;

10.

11. public class ANNImplementation {

12.

13. private static final ArrayList<Sample> entireDataSet = new ArrayList<>();

14. //Data subsets

15. private static final ArrayList<Sample> testDataSet = new ArrayList<>(); //About 20% of data

16. private static final ArrayList<Sample> validationDataSet = new ArrayList<>(); //About 20% of data

17. private static final ArrayList<Sample> trainingDataSet = new ArrayList<>(); //About 60% of data

18.

19. private static double[] minimums;

20. private static double[] maximums;

21.

22. private static final XSSFWorkbook workbook = new XSSFWorkbook();

23. private static int dataPointsCount;

24.

25. //ANN parameters

26. private static int[] ANN\_SHAPE = new int[]{14, 12};

27. private static double MOMENTUM = 0.9d;

28. private static int MAX\_EPOCHS = 10000;

29. private static double START\_STEP\_SIZE = 0.1d;

30. private static double END\_STEP\_SIZE = 0.01d;

31. private static double MAX\_STEP\_SIZE = 0.5d;

32. private static double MIN\_STEP\_SIZE = 0.001d;

33.

34. //ANN improvments

35. private static boolean USE\_MOMENTUM = true;

36. private static boolean USE\_BOLD\_DRIVER = true;

37. private static boolean USE\_ANNEALING = false;

38. private static boolean USE\_WEIGHT\_DECAY = false;

39. private static boolean USE\_BATCH\_LEARING = false;

40.

41. private static final Random rand = new Random();

42.

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

44.

45. Scanner input = new Scanner(System.in);

46.

47. //Load model paramaters from config.txt

48. loadCongfig();

49. //Read and process data set from DataSet.xlsx - test it option 3

50. processDataSet();

51.

52. boolean askAgain = true;

53. //Give the user a list of options

54. while (askAgain) {

55. try {

56. System.out.println("\nWould you like to:\n"

57. + "(1) Train an new ANN on labelled data set?\n"

58. + "(2) Evaluate an existing ANN on labelled data set?\n"

59. + "(3) Execute existing ANN on unlabelled data set?\n"

60. + "(4) Save output and quit?");

61. int choice = input.nextInt();

62. ArrayList<Neuron[]> neuronLayers = null;

63. switch (choice) {

64. //If Train an new ANN

65. case (1):

66. //Train MLP with given parameters

67. neuronLayers = trainANN();

68. System.out.println("\nModel Trained.");

69. boolean success = false;

70. while (!success) {

71. //Attempt to load text file specifed by user

72. System.out.println("Enter file name (not file path), enter \"no\" to not save ANN: ");

73. String fileName = input.next();

74. if (fileName.equals("no") || fileName.equals("NO") || fileName.equals("No")) {

75. break;

76. }

77. success = saveANN(neuronLayers, fileName);

78. if (success) {

79. System.out.println("Model Saved.");

80. }

81. }

82. break;

83.

84. //If Evaluate an existing ANN

85. case (2):

86.

87. //Continue to ask until valid file chosen

88. while (neuronLayers == null) {

89. //Attempt to load text file specifed by user

90. System.out.println("\nEnter file name (not file path), enter \"no\" to cancel: ");

91. String fileName = input.next();

92. if (fileName.equals("no") || fileName.equals("NO") || fileName.equals("No")) {

93. break;

94. }

95. neuronLayers = loadANN(fileName);

96. //If file exist with correct format

97. if (neuronLayers != null) {

98. //If ANN compatible with data set

99. if (neuronLayers.get(0)[0].getWeights().length == dataPointsCount) {

100. //Evaluates data set using test set

101. double[] evaluationMetrics = evaluateANN(neuronLayers, testDataSet, true);

102. System.out.println("Evaluation Metrics: ");

103. System.out.printf("\nRMSE – Root Mean Squared Error: \t\t\t%.4f \t(closer to 0 is better)\n", evaluationMetrics[0]);

104. System.out.printf("SRE - Mean Squared Relative Error: \t\t%.4f \t(closer to 0 is better)\n", evaluationMetrics[1]);

105. System.out.printf("CE - Coefficient of Efficiency: \t\t%.4f \t(closer to 1 is better)\n", evaluationMetrics[2]);

106. System.out.printf("RSQR - R-Squared (Determination Coefficien): \t%.4f \t(closer to 1 is better)\n", evaluationMetrics[3]);

107. } else {

108. neuronLayers = null;

109. System.out.println("ANN not compatible with data set - incorrect number of inputs");

110. }

111. } else {

112. System.out.println("Incorrect file name");

113. }

114. }

115. break;

116.

117. //If Execute existing ANN on entire data set

118. case (3):

119. //Continue to ask until valid file chosen

120. while (neuronLayers == null) {

121. //Attempt to load text file specifed by user

122. System.out.println("\nEnter file name (not file path), enter \"no\" to cancel: ");

123. String fileName = input.next();

124. if (fileName.equals("no") || fileName.equals("NO") || fileName.equals("No")) {

125. break;

126. }

127. neuronLayers = loadANN(fileName);

128. //If file exist with correct format

129. if (neuronLayers != null) {

130. //If ANN compatible with data set

131. if (neuronLayers.get(0)[0].getWeights().length == dataPointsCount + 1) {

132. //Executes data set on all inputs in data set

133. executeANN(neuronLayers, entireDataSet);

134. System.out.println("Model Executed.");

135. } else {

136. neuronLayers = null;

137. System.out.println("ANN not compatible with data set - incorrect number of inputs");

138. }

139. } else {

140. System.out.println("Incorrect file name");

141. }

142. }

143. break;

144.

145. //If Save output and quit

146. case (4):

147. askAgain = false;

148. break;

149.

150. default:

151. System.out.println("Incorrect choice, try again");

152. askAgain = true;

153. break;

154. }

155. } catch (Exception e) {

156. System.out.println("Incorrect choice, try again");

157. input.next();

158. }

159. }

160.

161. //Save output to "Output.xlsx"

162. try {

163. FileOutputStream out = new FileOutputStream(new File("Output.xlsx"));

164. workbook.write(out);

165. out.close();

166. System.out.println("\nSuccessfully saved output file");

167. } catch (IOException e) {

168. System.out.println("\nError creating output file. Error: " + e);

169. }

170.

171. }

172.

173. private static void loadCongfig() {

174. String dir = System.getProperty("user.dir") + "\\config.txt";

175. File file = new File(dir);

176. boolean defaultValues = false;

177.

178. if (file.exists()) {

179. //Try get every value from config file

180. try {

181. Scanner reader = new Scanner(file);

182. //Read each line in text file

183. while (reader.hasNextLine()) {

184. //Convert each line to kay, value pair

185. String[] stringData = reader.nextLine().split("=");

186. if (stringData.length != 2) {

187. System.out.println("nahh");

188. continue;

189. }

190. String key = stringData[0].trim();

191. String value = stringData[1].trim();

192. //Set value for specified paramater

193. switch (key) {

194. case ("ANN\_SHAPE"):

195. if (value.equals("")) {

196. ANN\_SHAPE = new int[0];

197. } else {

198. String[] layerSizes = value.split(",");

199. ANN\_SHAPE = new int[layerSizes.length];

200. for (int i = 0; i < layerSizes.length; i++) {

201. ANN\_SHAPE[i] = Integer.valueOf(layerSizes[i].trim());

202. }

203. }

204. break;

205. case ("MOMENTUM"):

206. MOMENTUM = Double.valueOf(value);

207. break;

208. case ("MAX\_EPOCHS"):

209. MAX\_EPOCHS = Integer.valueOf(value);

210. break;

211. case ("START\_STEP\_SIZE"):

212. START\_STEP\_SIZE = Double.valueOf(value);

213. break;

214. case ("END\_STEP\_SIZE"):

215. END\_STEP\_SIZE = Double.valueOf(value);

216. break;

217. case ("MAX\_STEP\_SIZE"):

218. MAX\_STEP\_SIZE = Double.valueOf(value);

219. break;

220. case ("MIN\_STEP\_SIZE"):

221. MIN\_STEP\_SIZE = Double.valueOf(value);

222. break;

223. case ("USE\_MOMENTUM"):

224. USE\_MOMENTUM = Boolean.valueOf(value);

225. break;

226. case ("USE\_BOLD\_DRIVER"):

227. USE\_BOLD\_DRIVER = Boolean.valueOf(value);

228. break;

229. case ("USE\_ANNEALING"):

230. USE\_ANNEALING = Boolean.valueOf(value);

231. break;

232. case ("USE\_WEIGHT\_DECAY"):

233. USE\_WEIGHT\_DECAY = Boolean.valueOf(value);

234. break;

235. case ("USE\_BATCH\_LEARING"):

236. USE\_BATCH\_LEARING = Boolean.valueOf(value);

237. break;

238. }

239. }

240. reader.close();

241. } //If there is any errors (because of incorrect format) then use default values instead

242. catch (Exception e) {

243. System.out.println("An error occurred loading config file. Error: " + e);

244. defaultValues = true;

245. }

246. } //If there is no config file then use default values instead

247. else {

248. System.out.println("Config file does not exist, using default values.");

249. defaultValues = true;

250. }

251. //Reset model paramaters to default if necessary

252. if (defaultValues) {

253. ANN\_SHAPE = new int[]{10, 8};

254. MOMENTUM = 0.9;

255. MAX\_EPOCHS = 1000;

256. START\_STEP\_SIZE = 0.1;

257. END\_STEP\_SIZE = 0.01;

258. MAX\_STEP\_SIZE = 0.5;

259. MIN\_STEP\_SIZE = 0.001;

260. USE\_MOMENTUM = true;

261. USE\_BOLD\_DRIVER = true;

262. USE\_ANNEALING = false;

263. USE\_WEIGHT\_DECAY = false;

264. USE\_BATCH\_LEARING = false;

265. }

266. //Display used model paramaters

267. System.out.println("ANN parameters:");

268. System.out.println("ANN\_SHAPE = " + Arrays.toString(ANN\_SHAPE));

269. System.out.println("MOMENTUM = " + MOMENTUM);

270. System.out.println("MAX\_EPOCHS = " + MAX\_EPOCHS);

271. System.out.println("START\_STEP\_SIZE = " + START\_STEP\_SIZE);

272. System.out.println("END\_STEP\_SIZE = " + END\_STEP\_SIZE);

273. System.out.println("MAX\_STEP\_SIZE = " + MAX\_STEP\_SIZE);

274. System.out.println("MIN\_STEP\_SIZE = " + MIN\_STEP\_SIZE);

275. System.out.println("USE\_MOMENTUM = " + USE\_MOMENTUM);

276. System.out.println("USE\_BOLD\_DRIVER = " + USE\_BOLD\_DRIVER);

277. System.out.println("USE\_ANNEALING = " + USE\_ANNEALING);

278. System.out.println("USE\_WEIGHT\_DECAY = " + USE\_WEIGHT\_DECAY);

279. System.out.println("USE\_BATCH\_LEARING = " + USE\_BATCH\_LEARING);

280.

281. if (!USE\_MOMENTUM) {

282. MOMENTUM = 0;

283. }

284. }

285.

286. private static void processDataSet() throws FileNotFoundException, IOException {

287.

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

289.

290. //Read data set from DataSet.xlsx

291. FileInputStream inputFile = new FileInputStream(new File("DataSet.xlsx"));

292. XSSFWorkbook dataSetWorkbook = new XSSFWorkbook(inputFile);

293. XSSFSheet sheet = dataSetWorkbook.getSheetAt(0);

294. FormulaEvaluator evaluator = dataSetWorkbook.getCreationHelper().createFormulaEvaluator();

295.

296. int sampleCount = sheet.getPhysicalNumberOfRows();

297. dataPointsCount = sheet.getRow(0).getPhysicalNumberOfCells();

298.

299. //Extra information for each column of data (used for data pre-processing)

300. minimums = new double[dataPointsCount];

301. maximums = new double[dataPointsCount];

302.

303. //Read every row in sheet

304. for (int row = 0; row < sampleCount; row++) {

305. //Set to false if a non-numeric value is detected in a sample

306. boolean validSample = true;

307. //Read row from data set

308. double[] sampleData = new double[dataPointsCount];

309. for (int col = 0; col < dataPointsCount; col++) {

310. if (sheet.getRow(row).getCell(col) != null && evaluator.evaluateInCell(

311. sheet.getRow(row).getCell(col)).getCellTypeEnum() == NUMERIC) {

312. sampleData[col] = sheet.getRow(row).getCell(col).getNumericCellValue();

313. //Calculate min and max values for input (column), used for standardization later

314. minimums[col] = Math.min(minimums[col], sampleData[col]);

315. maximums[col] = Math.max(maximums[col], sampleData[col]);

316. } else {

317. validSample = false;

318. break;

319. }

320. }

321. //Invalid samples are ignored

322. if (!validSample) {

323. continue;

324. }

325. //Convert each row of data into a Sample data object and add it to dataSet

326. dataSet.add(new Sample(sampleData));

327. }

328.

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

330.

331. //Loop through data set

332. for (Sample sample : dataSet) {

333. double[] sampleData = sample.getData();

334. //Standardize each value between range [0.1, 0.9]

335. for (int i = 0; i < sampleData.length; i++) {

336. sampleData[i] = standardize(sampleData[i], maximums[i], minimums[i]);

337. }

338. //Display sample data

339. System.out.println(sample);

340. //Randomly place sample between 3 data sets

341. switch (rand.nextInt(5)) {

342. case (0): //20% chance

343. testDataSet.add(sample);

344. break;

345. case (1): //20% chance

346. validationDataSet.add(sample);

347. break;

348. default: //60% chance

349. trainingDataSet.add(sample);

350. break;

351. }

352. entireDataSet.add(sample);

353. }

354. //Display data subset sizes

355. System.out.printf("\nTotal number of samples: %d\n", entireDataSet.size());

356. System.out.printf("Test set size: %.2f%%\n", (double) testDataSet.size() / dataSet.size() \* 100);

357. System.out.printf("Validation set size: %.2f%%\n", (double) validationDataSet.size() / dataSet.size() \* 100);

358. System.out.printf("Training set size: %.2f%%\n", (double) trainingDataSet.size() / dataSet.size() \* 100);

359. }

360.

361. private static double standardize(double value, double max, double min) {

362. //Standardizes value between range [0.1, 0.9]

363. return 0.8 \* ((value - min) / (max - min)) + 0.1;

364. }

365.

366. private static double deStandardize(double value, double max, double min) {

367. //Converts value back from range [0.1, 0.9] to normal range

368. return ((value - 0.1) / 0.8) \* (max - min) + min;

369. }

370.

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

372.

373. if (workbook.getSheetIndex("Training Data") != -1) {

374. workbook.removeSheetAt(workbook.getSheetIndex("Training Data"));

375. }

376. XSSFSheet spreadsheet = workbook.createSheet("Training Data");

377. //Save epoch data Headers

378. XSSFRow header = spreadsheet.createRow(0);

379. header.createCell(0).setCellValue("Epoch");

380. header.createCell(1).setCellValue("RMSE");

381. header.createCell(2).setCellValue("Step size");

382. int line = 1;

383.

384. double stepSize = START\_STEP\_SIZE;

385.

386. //Array list of neuron layers represents ANN

387. final ArrayList<Neuron[]> neuronLayers = new ArrayList<>();

388. //Create uninitialised hidden layers of neurons

389. for (int hiddenLayerSize : ANN\_SHAPE) {

390. if (hiddenLayerSize > 0) {

391. neuronLayers.add(new Neuron[hiddenLayerSize]);

392. }

393. }

394. //Create uninitialised output neuron

395. neuronLayers.add(new Neuron[1]);

396. //Initialise neurons with random weights

397. initialiseNeuronsWithRandomWeights(neuronLayers);

398.

399. //Copy of previous neural network instance (used for bold driver)

400. final ArrayList<Neuron[]> prevNeuronLayers = new ArrayList<>();

401. for (int hiddenLayerSize : ANN\_SHAPE) {

402. if (hiddenLayerSize > 0) {

403. prevNeuronLayers.add(new Neuron[hiddenLayerSize]);

404. }

405. }

406. prevNeuronLayers.add(new Neuron[1]);

407. cloneNeuronLayers(neuronLayers, prevNeuronLayers);

408.

409. //Previously measured RMSE

410. double error = Double.MAX\_VALUE;

411. int tinyErrorCount = 0;

412.

413. //Train for specified number of epochs

414. for (int i = 0; i < MAX\_EPOCHS; i++) {

415. //Loop through every sample in training set

416. for (Sample sample : trainingDataSet) {

417. //Inputs and bias passed to first layer of hidden neurons

418. double[] inputsAndBias = new double[sample.getData().length];

419. System.arraycopy(sample.getData(), 0, inputsAndBias, 0, inputsAndBias.length - 1);

420. inputsAndBias[inputsAndBias.length - 1] = 1;

421. //Correct output specified by sample

422. double sampleOutput = sample.getData()[sample.getData().length - 1];

423. //Forward pass through ANN

424. forwardPass(neuronLayers, inputsAndBias);

425. //Backward pass through ANN

426. backwardPass(neuronLayers, sampleOutput, stepSize, i + 1);

427.

428. if (!USE\_BATCH\_LEARING) {

429. //Update weights for every neuron in ANN

430. updateWeights(neuronLayers, stepSize);

431. } else {

432. //Sum error gradient for all weights

433. sumWeightsErrorGradient(neuronLayers);

434. }

435. }

436.

437. //Update weights at the end of the epoch

438. if (USE\_BATCH\_LEARING) {

439. batchUpdateWeights(neuronLayers, stepSize, trainingDataSet.size());

440. }

441.

442. //Simulated annealing

443. if (USE\_ANNEALING) {

444. stepSize = annealing(i + 1);

445. }

446.

447. //100 times while training (every 1% of training complete)

448. if (i % (Math.max(MAX\_EPOCHS / 100, 1)) == 0) {

449. //Get Root Mean Squared Error from evaluating model on validation set

450. double RMSE = evaluateANN(neuronLayers, validationDataSet, false)[0];

451.

452. if (USE\_BOLD\_DRIVER) {

453. double errorDiff = ((RMSE - error) / error);

454. //If the error increases by over 1 % then half the step size and revert the weights back to the prev weights

455. if (errorDiff > 0.01d) {

456. stepSize \*= 0.5;

457. stepSize = Math.max(MIN\_STEP\_SIZE, stepSize);

458. //revert model to last bold driver

459. cloneNeuronLayers(prevNeuronLayers, neuronLayers);

460. } else {

461. //If the error decreases by over 1 % then slightly increase step size

462. if (errorDiff < -0.01d) {

463. stepSize \*= 1.05;

464. stepSize = Math.min(MAX\_STEP\_SIZE, stepSize);

465. }

466. cloneNeuronLayers(neuronLayers, prevNeuronLayers);

467. }

468. }

469. //Display epoch data

470. System.out.printf("\nEpoch %d completed - %.2f%% \tStep Size:%f \tError:%f", i, (double) i / MAX\_EPOCHS \* 100, stepSize, RMSE);

471.

472. //Save epoch data

473. XSSFRow row = spreadsheet.createRow(line++);

474. row.createCell(0).setCellValue(i);

475. row.createCell(1).setCellValue(RMSE);

476. row.createCell(2).setCellValue(stepSize);

477.

478. //Terminate training early if error change is negligible 3 times in a row

479. if (Math.abs(RMSE - error) < 0.00001d) {

480. tinyErrorCount++;

481. if (tinyErrorCount >= 3) {

482. System.out.println(" - Change in Error minimal, stopping training");

483. break;

484. }

485. } else {

486. tinyErrorCount = 0;

487. }

488. error = RMSE;

489. }

490. }

491.

492. return neuronLayers;

493. }

494.

495. private static void initialiseNeuronsWithRandomWeights(ArrayList<Neuron[]> neuronLayers) {

496. //Initialise all neurons with random weights and bias

497. for (int i = 0; i < neuronLayers.size(); i++) {

498. int weightsCount;

499. //If first layer of hidden neurons then #weights = #sampleInputs + 1

500. if (i == 0) {

501. weightsCount = dataPointsCount;

502. } //Otherwise, #weights = #previousLayerNeurons + 1

503. else {

504. weightsCount = neuronLayers.get(i - 1).length + 1;

505. }

506. for (int j = 0; j < neuronLayers.get(i).length; j++) {

507. //Number of weights (#inputs + 1 for bias) for neuron

508. double[] weightsAndBias = new double[weightsCount];

509. //Randomise weights and bias values between [-2/n, 2/n] where n is the input size of the neuron

510. int range = weightsAndBias.length - 1;

511. for (int k = 0; k < weightsAndBias.length; k++) {

512. //Random value between [-2/n, 2/n] where n is the input size

513. weightsAndBias[k] = -2d / range + (2d / range - -2d / range) \* rand.nextDouble();

514. }

515. neuronLayers.get(i)[j] = new Neuron(weightsAndBias);

516. }

517. }

518. }

519.

520. private static void cloneNeuronLayers(ArrayList<Neuron[]> original, ArrayList<Neuron[]> clone) {

521. //Copy weights values from original to clone

522. for (int i = 0; i < original.size(); i++) {

523. for (int j = 0; j < original.get(i).length; j++) {

524. clone.get(i)[j] = new Neuron(original.get(i)[j]);

525. }

526. }

527. }

528.

529. private static double forwardPass(ArrayList<Neuron[]> neuronLayers, double[] sampleInputsAndBias) {

530. //Outputs from every neuron in the layer, used as inputs to the next layer

531. double[] layerOutputs = new double[0];

532.

533. //Forward pass through every layer in ANN

534. for (int i = 0; i < neuronLayers.size(); i++) {

535. //Inputs and bias for every neuron in the layer

536. double[] inputsAndBias;

537. //If first layer then inputs equal to inputs from the sample

538. if (i == 0) {

539. inputsAndBias = sampleInputsAndBias;

540. } //Otherwise, inputs equal to outputs of previous layer

541. else {

542. inputsAndBias = layerOutputs;

543. }

544. //Clears layer outputs and sets bias input to 1

545. layerOutputs = new double[neuronLayers.get(i).length + 1];

546. layerOutputs[layerOutputs.length - 1] = 1;

547. //Set inputs, calculate weight sum, compute activation for every neuron in layer

548. for (int j = 0; j < neuronLayers.get(i).length; j++) {

549. neuronLayers.get(i)[j].setInputs(inputsAndBias);

550. neuronLayers.get(i)[j].computeWeightSum();

551. layerOutputs[j] = neuronLayers.get(i)[j].computeActivation();

552. }

553. }

554. //Returns the output of the final neuron (output neuron), the output of the entire ANN

555. return layerOutputs[layerOutputs.length - 2];

556. }

557.

558. private static void backwardPass(ArrayList<Neuron[]> neuronLayers, double sampleOutput, double stepSize, int epoch) {

559. //Backward pass through every layer in ANN

560. for (int i = neuronLayers.size() - 1; i >= 0; i--) {

561. //If last layer (output layers)

562. if (i == neuronLayers.size() - 1) {

563. //Compute delta value for output neuron

564. for (Neuron outputNeuron : neuronLayers.get(i)) {

565. // Cacuiate upsilon (for weight decay)

566. double upsilon = 1 / (stepSize \* epoch);

567. if (!USE\_WEIGHT\_DECAY || epoch < 100) {

568. upsilon = 0;

569. }

570. //Compute delta value for output neuron

571. outputNeuron.computeDeltaValue(sampleOutput, upsilon);

572. }

573. } //Otherwise, if hidden layer

574. else {

575. //Loop through neurons in hidden layer

576. for (int j = 0; j < neuronLayers.get(i).length; j++) {

577. double[] forwardWeights = new double[neuronLayers.get(i + 1).length];

578. double[] forwardDeltaValues = new double[neuronLayers.get(i + 1).length];

579. //Loop through forward neurons

580. for (int k = 0; k < neuronLayers.get(i + 1).length; k++) {

581. Neuron forwardNeuron = neuronLayers.get(i + 1)[k];

582. //Store weights between current neuron and all forward neurons

583. forwardWeights[k] = forwardNeuron.getWeights()[j];

584. //Store delta values of every forward neuron

585. forwardDeltaValues[k] = forwardNeuron.getDeltaValue();

586. }

587. //Compute delta value for hidden neuron

588. neuronLayers.get(i)[j].computeDeltaValue(forwardWeights, forwardDeltaValues);

589. }

590. }

591. }

592. }

593.

594. private static void updateWeights(ArrayList<Neuron[]> neuronLayers, double stepSize) {

595. //Calculates new weights for every neuron in ANN

596. for (Neuron[] neuronLayer : neuronLayers) {

597. for (Neuron neuron : neuronLayer) {

598. neuron.updateWeightsAndBias(stepSize, MOMENTUM);

599. }

600. }

601. }

602.

603. private static void batchUpdateWeights(ArrayList<Neuron[]> neuronLayers, double stepSize, int sampleCount) {

604. //Calculates new weights for every neuron in ANN

605. for (Neuron[] neuronLayer : neuronLayers) {

606. for (Neuron neuron : neuronLayer) {

607. neuron.batchUpdateWeightAndBias(stepSize, MOMENTUM, sampleCount);

608. }

609. }

610. }

611.

612. private static void sumWeightsErrorGradient(ArrayList<Neuron[]> neuronLayers) {

613. //sum every weights error gradient for every neuron in ANN

614. for (Neuron[] neuronLayer : neuronLayers) {

615. for (Neuron neuron : neuronLayer) {

616. neuron.sumWeightsErrorGradient();

617. }

618. }

619. }

620.

621. private static double annealing(double epoch) {

622. //Decays stepSize (learning parameter) over time

623. return END\_STEP\_SIZE + (START\_STEP\_SIZE - END\_STEP\_SIZE)

624. \* (1 - (1 / (1 + Math.pow(Math.E, (10 - 20 \* (epoch / MAX\_EPOCHS))))));

625. }

626.

627. private static double[] evaluateANN(ArrayList<Neuron[]> neuronLayers, ArrayList<Sample> dataSubset, boolean store) {

628. XSSFSheet spreadsheet = null;

629. int line = 1;

630. if (store) {

631. if (workbook.getSheetIndex("Evaluation Data") != -1) {

632. workbook.removeSheetAt(workbook.getSheetIndex("Evaluation Data"));

633. }

634. spreadsheet = workbook.createSheet("Evaluation Data");

635. //Save evaluation data Headers

636. XSSFRow header = spreadsheet.createRow(0);

637. header.createCell(0).setCellValue("Sample outputs");

638. header.createCell(1).setCellValue("Modelled output");

639. }

640.

641. //RMSE, MSRE, CE, RSQR

642. double[] evaluationMetrics = new double[4];

643.

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

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

646.

647. //Variables used for calculation

648. double RMSEsum = 0, MSREsum = 0, CEnumerator = 0, CEdenominator = 0, RSQRnumerator = 0, RSQRdenominatorLeft = 0, RSQRdenominatorRight = 0;

649.

650. double sampleOutputsMean = 0;

651. double modelledOutputsMean = 0;

652. //Store sample output and modelled output for every sample in given data subset

653. for (int i = 0; i < dataSubset.size(); i++) {

654. Sample sample = dataSubset.get(i);

655. //Inputs and bias passed to first layer of hidden neurons

656. double[] inputsAndBias = new double[sample.getData().length];

657. System.arraycopy(sample.getData(), 0, inputsAndBias, 0, inputsAndBias.length - 1);

658. inputsAndBias[inputsAndBias.length - 1] = 1;

659.

660. //Correct output specified by sample

661. double sampleOutput = sample.getData()[sample.getData().length - 1];

662. //De-standardize sample output

663. sampleOutputs[i] = deStandardize(sampleOutput, maximums[maximums.length - 1], minimums[minimums.length - 1]);

664. //Forward pass through ANN

665. double modelledOutput = forwardPass(neuronLayers, inputsAndBias);

666. //De-standardize modelled output

667. modelledOutputs[i] = deStandardize(modelledOutput, maximums[maximums.length - 1], minimums[minimums.length - 1]);

668.

669. //Calculate mean for both values

670. sampleOutputsMean += sampleOutputs[i];

671. modelledOutputsMean += modelledOutputs[i];

672.

673. //Save evaluation data

674. if (store && spreadsheet != null) {

675. XSSFRow row = spreadsheet.createRow(line++);

676. row.createCell(0).setCellValue(sampleOutputs[i]);

677. row.createCell(1).setCellValue(modelledOutputs[i]);

678. }

679. }

680.

681. sampleOutputsMean /= dataSubset.size();

682. modelledOutputsMean /= dataSubset.size();

683.

684. //Calculate evaluation metrics

685. for (int i = 0; i < dataSubset.size(); i++) {

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

687.

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

689.

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

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

692.

693. RSQRnumerator += (sampleOutputs[i] - sampleOutputsMean) \* (modelledOutputs[i] - modelledOutputsMean);

694. RSQRdenominatorLeft += Math.pow(sampleOutputs[i] - sampleOutputsMean, 2);

695. RSQRdenominatorRight += Math.pow(modelledOutputs[i] - modelledOutputsMean, 2);

696. }

697. //RMSE – Root Mean Squared Error (closer to 0 is better)

698. evaluationMetrics[0] = Math.sqrt(RMSEsum / dataSubset.size());

699. //MSRE - Mean Squared Relative Error (closer to 0 is better)

700. evaluationMetrics[1] = MSREsum / dataSubset.size();

701. //CE - Coefficient of Efficiency (closer to 1 is better)

702. evaluationMetrics[2] = 1 - CEnumerator / CEdenominator;

703. //RSQR - R-Squared (Coefficient of Determination) (closer to 1 is better)

704. evaluationMetrics[3] = Math.pow(RSQRnumerator / (Math.sqrt(RSQRdenominatorLeft \* RSQRdenominatorRight)), 2);

705.

706. return evaluationMetrics;

707. }

708.

709. private static void executeANN(ArrayList<Neuron[]> neuronLayers, ArrayList<Sample> dataSubset) {

710. int line = 1;

711. if (workbook.getSheetIndex("Execution Data") != -1) {

712. workbook.removeSheetAt(workbook.getSheetIndex("Execution Data"));

713. }

714. XSSFSheet spreadsheet = workbook.createSheet("Execution Data");

715. //Save evaluation data Headers

716. XSSFRow header = spreadsheet.createRow(0);

717. header.createCell(0).setCellValue("Modelled output (standardized)");

718.

719. //Store modelled output for every sample in given data subset

720. for (int i = 0; i < dataSubset.size(); i++) {

721. Sample sample = dataSubset.get(i);

722. //Inputs and bias passed to first layer of hidden neurons

723. double[] inputsAndBias;

724. inputsAndBias = new double[sample.getData().length + 1];

725. System.arraycopy(sample.getData(), 0, inputsAndBias, 0, inputsAndBias.length - 1);

726. inputsAndBias[inputsAndBias.length - 1] = 1;

727.

728. //Forward pass through ANN

729. double modelledOutput = forwardPass(neuronLayers, inputsAndBias);

730.

731. //Save execution data

732. XSSFRow row = spreadsheet.createRow(line++);

733. row.createCell(0).setCellValue(modelledOutput);

734. }

735. }

736.

737. private static boolean saveANN(ArrayList<Neuron[]> neuronLayers, String fileName) {

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

739.

740. try {

741. File file = new File(dir);

742. //Overwrites file if it already exits

743. if (!file.createNewFile()) {

744. file.delete();

745. }

746.

747. FileWriter writeToFile = new FileWriter(dir, true);

748. PrintWriter printToFile = new PrintWriter(writeToFile);

749.

750. String ANNweights = "";

751. for (Neuron[] neuronLayer : neuronLayers) {

752. //Write ANN structure to fist line

753. printToFile.print(neuronLayer.length + ",");

754. //Write weights for each neuron on seperate lines

755. for (Neuron neuron : neuronLayer) {

756. String neuronWeights = "";

757. for (double weight : neuron.getWeights()) {

758. neuronWeights += weight + ",";

759. }

760. ANNweights += "\n" + neuronWeights;

761. }

762. }

763. printToFile.println(ANNweights);

764. printToFile.close();

765. writeToFile.close();

766. } catch (IOException e) {

767. System.out.println("An error occurred saving ANN. Error: " + e);

768. return false;

769. }

770. return true;

771. }

772.

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

774.

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

776.

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

778.

779. File file = new File(dir);

780. if (file.exists()) {

781. try {

782. Scanner reader = new Scanner(file);

783. //Read each line in text file

784. while (reader.hasNextLine()) {

785. //Convert each line to array of doubles

786. String[] stringData = reader.nextLine().split(",");

787. double[] numericData = new double[stringData.length];

788. for (int i = 0; i < stringData.length; i++) {

789. numericData[i] = Double.valueOf(stringData[i]);

790. }

791. fileNumericData.add(numericData);

792. }

793. reader.close();

794. //Get ANN structure (number of layers and neurons per layer from first line)

795. int[] hiddenLayerSizes = new int[fileNumericData.get(0).length];

796. for (int i = 0; i < fileNumericData.get(0).length; i++) {

797. hiddenLayerSizes[i] = (int) fileNumericData.get(0)[i];

798. }

799. //Create array list of neuron layers represents ANN

800. final ArrayList<Neuron[]> neuronLayers = new ArrayList<>();

801. //Create uninitialised hidden layers of neurons

802. for (int hiddenLayerSize : hiddenLayerSizes) {

803. neuronLayers.add(new Neuron[hiddenLayerSize]);

804. }

805. int line = 1;

806. //Initialise all neurons with weights from the file

807. for (int i = 0; i < neuronLayers.size(); i++) {

808. for (int j = 0; j < neuronLayers.get(i).length; j++) {

809. neuronLayers.get(i)[j] = new Neuron(fileNumericData.get(line++));

810. }

811. }

812. //Return loaded ANN

813. return neuronLayers;

814. } catch (IOException e) {

815. System.out.println("An error occurred loading ANN. Error: " + e);

816. }

817. } else {

818. System.out.println("The file does not exist.");

819. }

820.

821. return null;

822. }

823. }

824.