**1. Data pre-processing (including cleansing, data splitting, identifying predictors) – 15%**

IDENTIFYING PREDICTORS

To identify predictors, I decided to use each value as provided in the original data set, lagged by one day. I found correlation values for each daily flow value and the observed output, suspecting that they would be different as they are not equidistant from Skelton, and therefore would not necessarily have the same amount of lag time to affect Skelton’s daily flow, or even the same effect after correct lagging, which could have been worked around by using weighted values for the inputs themselves. For example, Skip Bridge is much closer to Skelton than Crakehill and Westwick, meaning that there could be a case made for its flow reaching Skelton earlier. Even still, after looking at the correlation function, I decided to lag the flow values by one day as that was the highest correlation to the output. I had intentions of progressing towards the improvements below but was unable to. Even still, I was able to consider all of the inputs given and use them in creating my model.

By taking a look at the map, we can see that Malham Tarn is in the watershed for a tributary that has its confluence with the Ouse further downstream than Skelton, meaning that most (if not all) of the rainfall in Malham Tarn would not pass through Skelton . However, for rainfall I believe it would have been more beneficial to produce a weighted moving average of all of the locations’ rainfall, lagged by the appropriate amount of time for each input while also making sure the weights added to 1. For example, I would place a very low weight on Malham Tarn for the aforementioned reasons and I would lag it by a similar time as the others as they are similar in distance at around 60-80km from Skelton. East Cowton and Arkengarthdale both feed into the River Swale, so I would lag and weight them fairly equally, perhaps slightly more weighted towards East Cowton as the tributary it pours into has a more direct path.

Example for weighted moving average of rainfall values:

0.05\*Malham Tarn(3 days behind) + 0.2\*Snaizeholme(3 days behind) + 0.4\*Arkengarthdale (2 days behind) + 0.35\*East Cowton(2 days behind)

I would also, for all inputs, use lag times for different days and calculate the correlation coefficient in order to find the ideal lag times and weights for the inputs.

CLEANSING AND SPLITTING

All cleansing and splitting implementation was called from the main method of the backpropgationMain.java class, as well as the training and testing later.

To begin with, I copied all of the data from the original Excel file into a .csv file, so that it was easier to read in a comma-delimited format. I processed the data set within my java program, within a separate class called dataPreprocessing.java. I processed it by first instantiating a fileOperations object and using its method getValues() with the filename, which simply read the values into a 2-dimensional list of strings(List<List<String>>). The final step was deleting the date from each row of inputs.

The next function called was castingToDouble(), which cast all of the values into a two-dimensional array of doubles.

To cleanse the data, I then eliminated the non-numerical, negative and outlier values by deleting the row in which they were contained (eliminateOutliers()). My criteria for a value being an outlier was for it to be more than 2 standard deviations away from the mean of its other instances in the data.

With the outliers erased, I moved the Skelton daily flow value to the end of each row (repositionOutputToEnd()), so that it would not interfere with the indexing of the inputs. Keeping the inputs and outputs separate proved very helpful in navigating each row of data.

The next step was to move the next day’s Skelton daily flow to the output of each row, otherwise I would’ve been predicting values for the day it was, not the next day. I did this by taking the output value for each row and moving it to the previous day’s output value index.

For example (as the data is stored as a 2D array of doubles – referred to as DATA), if the output value for a row rested at DATA[n][m], that value would be replaced by DATA[n+1][m] (outputRepositionedFromNextDayArray()).

I needed to avoid issues from the data being trained in the order originally given, such as seasonal affections like training with more values from winter than summer leading to a model that was not able to adequately generalise for all seasons, for example. I did this by shuffling the repositioned array randomly, ensuring that the weights would not be affected (shuffleArray()).

Moving on, I standardised all of the input and output values in the range [0.1, 0.9]. Standardisation was key in the implementation of the training algorithm because with large input values, unless we use extremely small initial weights, changes made by the backpropagation algorithm would be insignificantly small. This would lead to training being very sluggish, as the gradient of the sigmoid function, for example, at extreme values would be approximately zero, since the transfer function only produces non-negligible values between 0 and 1. I chose the range [0.1, 0.9] because it allows the network to predict values outside of those it has been trained with. As mentioned in the lectures, if the range is [0, 1] in training at the highest observed value is 16, the system will not be able to cleanly predict a value for anything higher than this. I had an issue with achieving this, as I needed the minimum and maximum values to be returned so that I could destandardise the outputs later. To get around this, I changed the return type from a primitive type to a class which stored the minimum and maximum values for the outputs, as well as the entire data set (inputs and outputs) standardised. I did this by creating a class (standardisedPackager) within dataPreprocessing.java which had the standardised inputs and outputs as well as the minimum and maximum values for each value in the data set.

Next, I split the randomly shuffled data into a training, validation and testing set. As recommended by the lectures, I split this 60/20/20, but my code allows for any chosen distribution between the sets. I was able to do this by creating and returning an object of the dataSplitter class, which I had declared outside the function. It had 3 attributes: the training set, validation set, and test set. This gave me all of the sets separated properly.

This concludes the pre-processing of the data, which was all done within my java program, called from the main method of backpropagationMain.java. At this point all of the data is split, cleaned, and ready to begin training and testing.

**2. Implementation of the MLP algorithm (including modifications / improvements) – 35%**

I used java, as it allowed me to use OOP to break the program into functional sections, which made debugging it much easier, as I knew where an issue would be likely to stem from, and most importantly, what else it could have an effect on. I used 3 classes, backpropagationMain.java, dataPreprocessing.java, and fileOperations.java. By using these classes, I was able to break the coursework into 3 major sections: the main algorithm, where training and testing was done (backpropagationMain.java), pre-processing of data (dataPreprocessing.java), and reading/writing files using data (e.g. amount of epochs, amount of hidden layers, learning parameter, mean squared error, etc.) so that I could procure graphs from them (and initially read the file containing all raw data - fileOperations.java).

One major advantage of using OOP in Java is that it allowed me to return multiple variables from a function by making use of Java’s ability to return classes. I did this by creating a class with the desired attributes and creating a function returning said class. For example, I was able to return an instance of the class trainingResults every time I ran training of a data set (backpropTraining()), returning the weights and biases in addition to the number of hidden nodes within the hidden layer. Then, using these attributes, I was easily able to perform validation and testing, which would have been much harder without having the aforementioned values readily available.

Another example is my function testing(), which takes as one of its inputs the aforementioned outputted class trainingResults, and returns its own instance of a class testingResults, which has mean squared error, destandardised modelled outputs, and destandardised observed outputs as attributes. By using a class as the return argument, testing() provides multiple measures I can use to draw graphs and evaluate the MLP with.

The other 2 examples of returning a class from a function are expanded upon in the pre-processing section.

**Momentum**

I chose to add momentum to my algorithm, which I did with a toggle so that its effect could be better demonstrated. I implemented this by adding a Boolean momentum in the arguments for my training function, as this meant I could choose whether I wanted this instance of the training function to use it or not. For each weight change, if momentum was true, it would add the improvement to the weight change using the alpha value and the previous weight change. Using this, along with a seeded random value that created the same random values, I could run the code twice, with the only variable being the use of momentum. This allowed me to see if momentum is useful, at changing levels of epochs and hidden nodes if I wanted to. I could also change the alpha values and see the effects on mean squared error.

**Transfer functions**

I added the option to train the MLP using the tanh transfer function. I did this to test if the transfer functions had different effects depending on the changing of other factors such as epoch count, as I suspected that there would be no concrete answer for which function performed best under all circumstances.

The results of training with tanh versus that of training with sigmoid can be seen in the graphs in section 4.

**Storage and structure of data**

I chose to store weights and biases within two-dimensional arrays of type double. They were structured such that, for [i][j], i always represented the index of previous layer node that the weighted path was coming from, while j represented the index of the path-ending node in the next layer . This made them easier to iterate through using simple for loops, as well as being easier to debug. If I noticed a problem, I did this by logging where the algorithm was at each point in the iteration, then checking to see at which point the problem occurred, and this would immediately show me which code was causing issues, allowing me to fix it much quicker.

Another reason I stored the is that they were simple to record as attributes of the trainingResults class and be used in conjunction with the testing data to find modelled outputs. This is because they are much easier to index through, once again making it easier to debug as it shows where the problems lie.

BACKPROPAGATIONMAIN.JAVA METHODS EXPLAINED

sigmoidActivation() – takes a double as input, returns the sigmoid activation transfer for it.

sigmoidActivationDiff()- takes a double (a sigmoid activation value) as input, returns the differential of the sigmoid activation transfer for it. Used to find delta values at nodes in the backwards pass, which enabled me to change the weights.

tanhActivation()- takes a double as input, returns the tanh activation transfer for it.

tanhActivationDiff()- takes a double (a sigmoid activation value) as input, returns the differential of the tanh activation transfer for it. Used to find delta values at nodes in the backwards pass, which enabled me to change the weights.

deltaHidden()- for each hidden node, takes as inputs the weight from it to the single output node, the delta value of the ‘next’ node (the output node), and the activation value of the node (all doubles), and a Boolean determining if the required transfer function is sigmoid (true) or tanh (false).

destandardisedValue()- takes as input the ‘raw’ value, which has been standardised in the pre-processing section and activated previously, the minimum and maximum values in the data for those values (all doubles). Returns the value destandardised, used to give a true modelled output to compare with the observed.

backpropTraining()- the most important function, used to train the MLP on the given data set. Takes as input the training set, the number of nodes in the hidden layer, the number of epochs, a Boolean detailing if the transfer function is Sigmoid(true) or tanh(false), and a Boolean detailing if momentum is used(true) or not (false).

Returns an object of class trainingResults, an aforementioned advantage of using the OOP approach in Java. The class contains the final weights and biases that have been trained, as well as the number of hidden nodes. These values are needed to perform testing of the MLP that has been trained.

testingResults()- used to test the MLP on the test set. Takes as input the test set

Returns an object of class trainingResults, an aforementioned advantage of using the OOP approach in Java. The class contains the final weights and biases that have been trained, as well as the number of hidden nodes. These values are all that is needed to perform testing of the MLP that has been trained.

Main algorithm implementation

Once the data has been preprocessed as mentioned in the previous section, the training set can be inputted into the backpropTraining(), along with parameters detailing the number of nodes in the hidden layer, the number of epochs, the desired transfer function (Sigmoid for true, or Tanh for false), and the desired use of momentum(true for yes) along with the desired alpha value. The output of this method is an object of class trainingResults, which has the final weights and biases of the ANN as its attributes, as well as the number of hidden nodes. This data is necessary for testing of the network in the next phase.

The trainingResults object created from training the weights can then be inputted into the testing function (testing()), along with the test set itself, 2 one-dimensional arrays containing the minimum and maximum values of every value in each row (used for destandardising outputs), and the activation function used in training. The function similarly returns an object, but this time of class testingResults with attributes mean squared error, destandardised modelled and observed outputs, for analysis of the ANN’s performance. This concludes the training and testing of the network.

**Libraries**

All libraries used were native java libraries

FileNotFoundException

Random

Scanner

File

ArrayList

List

Calendar

Date

FileWriter

IOException

DateFormat

SimpleDateFormat

The MLP can be programmed to have any amount of hidden layer nodes or inputs. The alpha value for momentum and the learning parameter itself can also be changed. However, only 1 hidden layer and 1 output can be modelled.

Furthermore, I added a feature (involving a variable IndependentCounter) that allows the user to select a feature (such as number of epochs or hidden nodes) that increments by a chosen amount and retrains and tests the network with the same starting weights (utilising seeded random values). This is useful for analysis of how that independent variable impacts the MSE of the network as it changes.

**3. Training and network selection – 20%**

After using a validation set against my MLP, I determined a final model, detailed below. The graphs below will illustrate how I came to my decisions. I was looking not to use too many epochs or as this could lead to over-fitting to the training set, which would have caused variance when testing.

The transfer function used is Sigmoid unless stated otherwise. I did this to limit the independent variables to only those stated in the title, so as to ensure the test was fair. I also used the same random seedings so that the initial weights were the same.

Figures 1 and 2 demonstrate how learning parameter affects MSE.

Figures 3 and 4 look at the effect of changing the number of nodes in the hidden layer on MSE.

Figures 5 and 6 show how changing epochs in training changes MSE.

Figure 1

Figure 2

From the above two graphs, we can see that the learning parameter changing has very little effect on MSE. This is shown by the fact that for both instances of different epochs, the difference between minima and maxima are ≈ 0.0015. The graphs, especially figure 2, also indicate that ≈0.1 is the ideal value for the learning parameter.

The graphs below explore the effect of hidden nodes on MSE, at varying magnitudes of epochs.

Figure 3

Figure 4

While figures 3 and 4 seem quite erratic, we can at least infer that the number of nodes (higher than around 2-3) within a hidden layer has quite a small effect on performance overall, as the error at 30 hidden nodes in figure 4 (4000 epochs) is consistent with that of the error at just 4 hidden nodes. Similarly, figure 1 (1000 epochs), 8, 13, and 24 hidden nodes all produce approximately the same error. Following on from these results, I chose 11 hidden nodes for my network as it is at (figure 3) or near (figure 4) the error minima for varying amounts of training epochs

Figure 5

Figure 6

Despite figure 5 appearing to show much more variance in MSE by changing epochs, the overall range of MSE values is around 0.0002, whereas figure 6 shows a gradual increase in MSE, with around a 0.002 range overall. From this, we can deduce that increasing the number of epochs affects the performance by possibly causing over-fitting. This was unexpected, as I hypothesized that the number of epochs increasing would benefit the system’s performance, but as we can see, specifically for the hidden node value that best represents my final model, figure 6, more epochs by no means ensures higher performance

However, what the two graphs have in common is that the lowest MSE values came at approximately 4000 epochs, which is why I have chosen this number for my final model.

Below are the final parameters for my model as decided upon by the results from the above graphs. I chose to use the sigmoid transfer function for my final model, as I found it produces less MSE when compared to tanh for my network.

|  |  |
| --- | --- |
| Number of hidden nodes | 11 |
| Number of epochs | 4000 |
| Transfer function | Sigmoid |
| Learning parameter | 0.1 |

**4. Evaluation of final model (including comparisons between different modifications to the algorithm) – 20%;**

Figures 7 through 12 consider the use of momentum and its impact on MSE.

7 and 8 focus on comparing the use with the lack thereof in training, and 9, 10, 11, and 12 look at how different numbers of training epochs and hidden nodes’ performance are affected by a changing alpha value. I analysed these as because I wanted to see how I could modify my network to possibly increase its performance, however momentum’s main purpose is to increase the speed of the system and not necessarily the accuracy.

Figure 7

Figure 8

Figures 7 and 8 show that the minimal MSE is found when no momentum, but there was not much difference between the two, with no momentum being ≈0.00671 and momentum being ≈0.00675. This is understandable, as the use of momentum is not one that is aimed at increasing the actual performance of the ANN, but instead the speed at which it reaches minima.

Proving my point, the ANN using momentum reached its minima at ≈1800 epochs, whereas without it, the value was ≈3250 epochs. This shows that momentum, while not improving the overall accuracy of the MLP, will improve its pace in locating minima.

Finally, I noticed that increase in MSE after minima is gradual for the MLP with no momentum, whereas it was much steeper for the one using momentum, with MSE ≈0.00675 at 5000 epochs for no momentum, and ≈0.0073 for momentum use. This also makes sense as the use of momentum causes more drastic shifts in weight, pushing the error more sharply in the direction it was already travelling in.

jjjjjjj

Figure 9

Figure 10

Figure 11

Figure 12

Figures 9, 10, 11, and 12 illustrate the effect of changing alpha values during momentum on the overall MSE.

Figures 9 and 10 used just 5 hidden nodes, while 11 and 12 used 11 hidden nodes, according to my final model. Therefore, I took the latter into account more when selecting my alpha value. They do, however, illustrate that higher alpha values, coupled with higher epochs, can lead to very erratic results. This is demonstrated by the very steep rise in MSE after finding the minima in figure 10.

The increase after the minima was extremely steep for figure 11, which could indicate that erratic results are more likely with higher alpha values, as shown in figures 9 and 10. For this reason, I decided against using 4.7 as my alpha value, despite the fact that lowest overall MSE was found at alpha value ≈4.7, with the MSE being ≈0.00657.

Figure 12’s minima was when the alpha value was at its lowest, ≈0.05, perhaps showing that lower values are most stable, especially for numbers of epochs used in my model. For this reason, I chose 0.05 as my alpha value.

After analysing the graphs in section 4 and looking at the minima, my improved model to achieve minimum MSE is as such:

|  |  |
| --- | --- |
| Number of hidden nodes | 11 |
| Number of epochs | 4000 |
| Transfer function | Sigmoid |
| Learning parameter | 0.1 |
| Momentum | Yes |
| Alpha | 0.05 |

On the next page, along with the evaluation of my final model, I have included a graph of the observed outputs against the modelled outputs of both my final model from section 3 and the improved model above.

Figure 13

EVALUATE

IMPROVEMENTS NOT MADE – ANNEALING, BD, WEIGHT DECAY

**5. Comparison with another data driven model – 10%;**

I used LINEST within excel to produce a graph comparing the observed and modelled results for a linear model.

Figure 14

“The LINEST function calculates the statistics for a line by using the "least squares" method to calculate a straight line that best fits the training data.” – Microsoft Excel Help

explain

**Program listing (i.e. the code you have written)**

import java.io.FileNotFoundException;// throws error if file isn't found

import java.util.Random; // allows selection of random numbers to find global minima in weight space

import java.util.List; // data structure representing ordered sequence of objects

class backpropagationMain {

public double sigmoidActivation(double input) {// enter value, returns the sigmoid transfer

return 1 / (1 + Math.exp(-input));

}

public double sigmoidActivationDiff(double input) {// enter sigmoid-activated value, returns the differential

return input \* (1 - input);

}

public double tanhActivation(double input) {// enter value, returns the tanh transfer

return (Math.exp(input) - Math.exp(-input)) / (Math.exp(input) + Math.exp(-input));

}

public double tanhActivationDiff(double input) {// enter tanh-activated value, returns the differential

return 1 - (input \* input);

}

public double deltaHidden(double weight, double nextdelta, double value, boolean Sigmoid) {

// gives the delta value of a given hidden node

// needed for backwards passing to allow changing of weights

// gets a different differential based on the activation function

if (Sigmoid) {

return weight \* nextdelta \* sigmoidActivationDiff(value);

} else {

return weight \* nextdelta \* tanhActivationDiff(value);

}

}

class trainingResults {// used as the return type for the training function

// weights from input nodes to hidden layer

double[][] inputToHiddenWeights; // [i][j], where i = input value and j = hidden layer node

// weights from hidden layer nodes to the output node

double[] hiddenToOutputWeights; // [i], where i = hidden node value

// biases on hidden layer

double[] hiddenLayerBiases;// [i], where i = index of the hidden node within the layer

// bias on output layer

double[] outputBiases; // [i], where i = index of the output node within the layer

// number of hidden nodes

int numberOfHiddenNodes;

trainingResults(double[][] inputToHiddenWeights, double[] hiddenToOutputWeights, double[] hiddenLayerBiases,

double[] outputBiases, int numberOfHiddenNodes) {// constructor

this.inputToHiddenWeights = inputToHiddenWeights;

this.hiddenToOutputWeights = hiddenToOutputWeights;

this.hiddenLayerBiases = hiddenLayerBiases;

this.outputBiases = outputBiases;

this.numberOfHiddenNodes = numberOfHiddenNodes;

}

}

// MAIN TRAINING FUNCTION

public trainingResults backpropTraining(double[][] inputs, double learningParameter, int NumberOfHiddenNodes,

int epochs, boolean Sigmoid, boolean momentum, double Alpha) {

Random rand = new Random(67); // instance of random class

int epochCounter = 0;// updated each epoch

double p = learningParameter;// learning parameter

double prevWeight; // for momentum

double alpha = Alpha; // for momentum

int NoOfInputs = inputs[0].length - 1;// records number of inputs

// each array contains the weights from a given input node

double[][] inputToHiddenWeights = new double[NoOfInputs][NumberOfHiddenNodes];

// each array contains the last change in weights from a given input node

double[][] changeInInputToHiddenWeights = new double[NoOfInputs][NumberOfHiddenNodes];

for (int i = 0; i < NoOfInputs; i++) {// iterating through inputs

for (int j = 0; j < NumberOfHiddenNodes; j++) {// randomly assigning initial weights

inputToHiddenWeights[i][j] = rand.nextDouble();

}

}

// each index contains the weight from a given hidden layer node

double[] hiddenToOutputWeights = new double[NumberOfHiddenNodes];

double[] changeInHiddenToOutputWeights = new double[NumberOfHiddenNodes]; // initialising array

for (int i = 0; i < NumberOfHiddenNodes; i++) { // randomly assigning initial weights

hiddenToOutputWeights[i] = rand.nextDouble();

}

double[] hiddenLayerBiases = new double[NumberOfHiddenNodes];// initialising array

double[] changeInHiddenLayerBiases = new double[NumberOfHiddenNodes];// initialising array

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

hiddenLayerBiases[i] = rand.nextDouble(); // randomly assigning initial weights

}

double[] outputBiases = { rand.nextDouble() }; // randomly assigning initial weights

double[] changeInOutputBiases = { 0 }; // assigning initial weights

// initialising array sizes by the amount of nodes in their respective layers

double[] hiddenLayerWeightedSums = new double[NumberOfHiddenNodes];

double[] hiddenLayerActivation = new double[NumberOfHiddenNodes];

double[] outputLayerWeightedSums = new double[1];

double[] outputsActivation = new double[1];

double[] deltaValuesHidden = new double[NumberOfHiddenNodes];

double[] deltaValueOutput = new double[1];

// forward pass through all data by the inputted number of epochs

while (epochCounter < epochs) {

// iterates through all rows from inputted data

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

// iterating through all hidden layer nodes and declaring their initial values

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

hiddenLayerWeightedSums[i] = 0;

for (int j = 0; j < inputs[0].length - 1; j++) {

// iterating through all input values and creating weighted sums for the hidden

// layer nodes

hiddenLayerWeightedSums[i] += inputs[k][j] \* inputToHiddenWeights[j][i];

}

// adding biases to hidden layer weighted sums

hiddenLayerWeightedSums[i] += hiddenLayerBiases[i];

// depending on the selection in the method's arguments, an activation function

// is selected

if (Sigmoid) {

hiddenLayerActivation[i] = this.sigmoidActivation(hiddenLayerWeightedSums[i]);

} else {

hiddenLayerActivation[i] = this.tanhActivation(hiddenLayerWeightedSums[i]);

}

}

// finding the output's weighted sum

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

outputLayerWeightedSums[i] = 0;

for (int j = 0; j < hiddenLayerWeightedSums.length; j++) {

// adds the weights and values from the hidden layer

outputLayerWeightedSums[i] += hiddenLayerActivation[j] \* hiddenToOutputWeights[j];

}

// adding the bias

outputLayerWeightedSums[i] += outputBiases[i];

// depending on the selection in the method's arguments, an activation function

// is selected

if (Sigmoid) {

outputsActivation[i] = this.sigmoidActivation(outputLayerWeightedSums[i]);

} else {

outputsActivation[i] = this.tanhActivation(outputLayerWeightedSums[i]);

}

}

// backwards pass, changing weights

// starting with output node

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

// finding delta value of output node

deltaValueOutput[i] = (inputs[k][inputs[k].length - 1] - outputsActivation[i])

\* (outputsActivation[i] \* (1 - outputsActivation[i]));

prevWeight = outputBiases[i];

outputBiases[i] += p \* deltaValueOutput[i];

changeInOutputBiases[i] = outputBiases[i] - prevWeight;

// saving previous weight and change in weights for momentum if selected

if (momentum) {

outputBiases[i] += (alpha \* changeInOutputBiases[i]);

}

}

// changing path weights from hidden nodes

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

// calling function to find delta of a given hidden node, saving to array

deltaValuesHidden[i] = this.deltaHidden(hiddenToOutputWeights[i], deltaValueOutput[0],

hiddenLayerActivation[i], Sigmoid);

// saving previous weight and change in weights for momentum if selected

prevWeight = hiddenToOutputWeights[i];

// changing weights

hiddenToOutputWeights[i] += p \* deltaValueOutput[0] \* hiddenLayerActivation[i];

changeInHiddenToOutputWeights[i] = hiddenToOutputWeights[i] - prevWeight;

prevWeight = hiddenLayerBiases[i];

hiddenLayerBiases[i] += p \* deltaValuesHidden[i];

changeInHiddenLayerBiases[i] = hiddenLayerBiases[i] - prevWeight;

if (momentum) {

hiddenToOutputWeights[i] += (alpha \* changeInHiddenToOutputWeights[i]);

hiddenLayerBiases[i] += (alpha \* changeInHiddenLayerBiases[i]);

}

}

// changing path weights from input nodes

for (int i = 0; i < inputToHiddenWeights.length; i++) {// iterate through inputs

for (int j = 0; j < NumberOfHiddenNodes; j++) {// iterate through hidden nodes

// saving previous weight and change in weights for momentum if selected

prevWeight = inputToHiddenWeights[i][j];

// change weight

inputToHiddenWeights[i][j] += p \* deltaValuesHidden[j] \* inputs[k][i];

changeInInputToHiddenWeights[i][j] = inputToHiddenWeights[i][j] - prevWeight;

if (momentum) {

inputToHiddenWeights[i][j] += (alpha \* changeInInputToHiddenWeights[i][j]);

}

}

}

}

epochCounter++;// one pass through the data has been completed

}

// final weights have been found, so a new trainingResults object is constructed

// using them

// ready for testing

return new trainingResults(inputToHiddenWeights, hiddenToOutputWeights, hiddenLayerBiases, outputBiases,

NumberOfHiddenNodes);

}

// class that is returned from testing function, allows to use results for

// analysis

class testingResults {

double meanSquaredError;

double[] destandardisedModelledOutputs;

double[] destandardisedObservedOutputs;

testingResults(double meanSquaredError, double[] destandardisedModelledOutputs,

double[] destandardisedObservedOutputs) {

this.meanSquaredError = meanSquaredError;

this.destandardisedModelledOutputs = destandardisedModelledOutputs;

this.destandardisedObservedOutputs = destandardisedObservedOutputs;

}

}

// testing function

// takes test set and weights as inputs

public testingResults testing(double[][] testSet, trainingResults results, double[] mins, double[] maxes, boolean Sigmoid) {

dataPreprocessing tester = new dataPreprocessing();

// declaring array lengths and variables

double[] destandardisedObservedOutputs = new double[testSet.length];

double[] destandardisedModelledOutputs = new double[testSet.length];

double totalSquaredError = 0;// needed for mse later

double meanSquaredError;

double[] hiddenLayerWeightedSums = new double[results.numberOfHiddenNodes];

double[] hiddenLayerActivation = new double[results.numberOfHiddenNodes];

double[] outputLayerWeightedSums = new double[1];

double outputsActivation;

for (int k = 0; k < testSet.length; k++) {// iterate through every row in test set

// same as training but forward pass only

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

hiddenLayerWeightedSums[i] = 0;

for (int j = 0; j < testSet[0].length - 1; j++) {

hiddenLayerWeightedSums[i] += testSet[k][j] \* results.inputToHiddenWeights[j][i];

}

hiddenLayerWeightedSums[i] += results.hiddenLayerBiases[i];

if (Sigmoid) {

hiddenLayerActivation[i] = this.sigmoidActivation(hiddenLayerWeightedSums[i]);

} else {

hiddenLayerActivation[i] = this.tanhActivation(hiddenLayerWeightedSums[i]);

}

}

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

outputLayerWeightedSums[i] = 0;

for (int j = 0; j < hiddenLayerWeightedSums.length; j++) {

outputLayerWeightedSums[i] += hiddenLayerActivation[j] \* results.hiddenToOutputWeights[j];

}

outputLayerWeightedSums[i] += results.outputBiases[i];

if (Sigmoid) {

outputsActivation = this.sigmoidActivation(outputLayerWeightedSums[i]);

} else {

outputsActivation = this.tanhActivation(outputLayerWeightedSums[i]);

}

System.out.println("modelled(standardised): " + outputsActivation);

System.out.println("modelled(destandardised): " + tester.destandardisedValue(outputsActivation, mins[7], maxes[7]));

destandardisedModelledOutputs[k] = tester.destandardisedValue(outputsActivation, mins[7], maxes[7]);

System.out.println("observed(standardised): " + testSet[k][testSet[k].length - 1]);

System.out.println("observed(destandardised): " + tester.destandardisedValue(testSet[k][testSet[k].length - 1], mins[7], maxes[7]));

destandardisedObservedOutputs[k] = tester.destandardisedValue(testSet[k][testSet[k].length - 1],

mins[7], maxes[7]);

// incrementing total squared error between activated modelled and observed

// values

totalSquaredError += Math.pow(testSet[k][testSet[k].length - 1] - outputsActivation, 2);

}

}

meanSquaredError = totalSquaredError / testSet.length;// final mse calculation

return new testingResults(meanSquaredError, destandardisedModelledOutputs, destandardisedObservedOutputs);

}

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

dataPreprocessing dataPrep = new dataPreprocessing();

// deleteDates will instantiate class dataPreprocessing within itself

// in order to get the original data from a csv file I created containing the

// raw values

// delete the dates at the beginning of each row

List<List<String>> deleteddates = dataPrep.deleteDates("arda.csv");

// casts the values from List<List<String>> to double

double[][] cast = dataPrep.castingToDouble(deleteddates);

// eliminates outliers, non-numerical, and negative values

double[][] cleanedData = dataPrep.eliminateOutliers(cast);

// make the output the last item in each row

// this made it a lot easier to work with

double[][] repositionedArray = dataPrep.repositionOutputToEnd(cleanedData, 3);

// data restructured such that the next day's flow at skelton is the observed

// answer for each row of data

double[][] outputRepositionedFromNextDayArray = dataPrep

.getOneDayAheadOutputInTheRawAsOutput(repositionedArray);

// shuffle all values so that they can be split properly, without seasonal

// affections

double[][] shuffledArray = dataPrep.shuffleArray(outputRepositionedFromNextDayArray);

// standardise all values in range [0.1,0.9], and return mins and maxes for

// destandardisation

dataPreprocessing.standardisedPackager standardizedPack = dataPrep.standardiseInputs(shuffledArray);

// split into 60/20/20 for training, validation, and testing (attributes of

// splitData)

dataPreprocessing.dataSplitter splitData = dataPrep.splitData(standardizedPack.inputsStandardised, 0.6, 0.2,

0.2);

// instantiate backpropagation object to begin training and testing

backpropagationMain test = new backpropagationMain();

// creating .csv files to make analysis graphs

// declare new fileOperations object to make new files with

fileOperations fileOps = new fileOperations();

// independent counter stands for the independent variable that will be changed

// to form the x-axis of the graph, in the case where y is mean squared error

// in this example, i did not use it as i had to find observed vs modelled values

// but it can be used to replace any continuous value in the arguments for backpropTraining (line 332)

int IndependentCounterStart = 0;

int IndependentCounterEnd = 0;

int IndependentCounterStep = 1;

// declaring size of result arrays - must be the difference between start and end

// divided by step size

int arraySize = (int) Math.floor((IndependentCounterEnd - IndependentCounterStart) / IndependentCounterStep)

+ 1;

double[] IndependentCountArrayForGraph = new double[splitData.testSet.length];

double[] mseArrayForGraph = new double[splitData.testSet.length];

String[] merged = new String[splitData.testSet.length];

int indexForGraph = 0;

// use createUniqueIdentifier to automatically record a unique filename prefix

String fileName;

fileName = fileOps.createUniqueIdentifier();

for (int IndependentCounter = IndependentCounterStart; IndependentCounter <= IndependentCounterEnd; IndependentCounter += IndependentCounterStep) {

// train the weights using 60% of the shuffled standardised values

trainingResults readyfortesting = test.backpropTraining(splitData.trainingSet, 0.1, 11, 4000,

true, true, 0.05);

// (double[][] inputs, double learningParameter, int NumberOfHiddenNodes,

// int epochs, boolean Sigmoid, boolean momentum, double Alpha)

// test the weights using the test set and find the mean squared error

testingResults tested = test.testing(splitData.testSet, readyfortesting, standardizedPack.mins, standardizedPack.maxes, true);

System.out.println("modelled at " + IndependentCounter + ":\n" + tested.destandardisedModelledOutputs[23]);

IndependentCountArrayForGraph = tested.destandardisedObservedOutputs;

mseArrayForGraph = tested.destandardisedModelledOutputs;

indexForGraph++;

}

indexForGraph = 0;

// merge the epochCounter and mse values into an array as this is an easier

merged = fileOps.mergeTwoArraysAsIsAndReturnAsStringArray(IndependentCountArrayForGraph, mseArrayForGraph);

// update filename so that file created will have key configuration data in its

// name

fileName += ".csv";

// create a new csv file with the modelled and observed values, so they can be

// made into a graph in excel

fileOps.createFile(fileName);

fileOps.writeArrayToFileAsLines(merged, fileName);

}

}

import java.io.FileNotFoundException;//catches error if no file found

import java.util.ArrayList;//adaptable-length array

import java.util.List;

import java.util.Random;

public class dataPreprocessing {

public List<List<String>> deleteDates(String filename) throws FileNotFoundException {

// function to delete the dates at the beginning of every row of inputs

// return is given in form List<List<String>> as this is how it is read from

// the .csv file

fileOperations reader = new fileOperations();

List<List<String>> originals = reader.getValues(filename);// reads values from file into a list

originals.remove(0);// takes away the place names of each input

for (List list : originals) {

list.remove(0);// removes first value - the date

}

return originals;

}

public double[][] castingToDouble(List<List<String>> inputs) {

// make each list an array, cast the whole wider list of lists to an array, then

// go through the arrays and convert all the values to doubles, weeding out

// non-numerical values

List<String> tempI;// declare a variable to temporarily hold each list within inputs

double tempDouble;// declare a variable to temporarily hold each value within a list in inputs

boolean rowHasOnlyNumbers;// flag indicating if the row is free of non-numerical values

boolean noNegatives;// flag indicating the row has no negative values

int initialSize = inputs.size();

List<List<String>> inputsNumerical = new ArrayList<>();

for (int i = 0; i < initialSize; i++) {// iterating through inputs

rowHasOnlyNumbers = true;// assumes true until proven otherwise

tempI = inputs.get(i);

for (int j = 0; j < tempI.size(); j++) {// iterating through an individual row of inputs

try {

tempDouble = Double.parseDouble(tempI.get(j));// attempts to cast numerical value to type double

} catch (Exception e) {// if value non-numerical, row not used

rowHasOnlyNumbers = false;

}

}

noNegatives = true;

if (rowHasOnlyNumbers) {// no non-numerical values detected

for (int k = 0; k < tempI.size(); k++) {

// checks if the value, when parsed to a double, is negative

if (Double.parseDouble(tempI.get(k)) < 0) {

noNegatives = false;

}

}

if (noNegatives) {// no negatives detected

inputsNumerical.add(tempI);// new 2d array of the numerical non-negative inputs

}

}

}

double[][] inputsAsDoubles = new double[inputsNumerical.size()][inputsNumerical.get(0).size()];

// allocates space in memory for the inputs, as a 2d array of type double

for (int i = 0; i < inputsNumerical.size(); i++) {// iterating through inputs

tempI = inputsNumerical.get(i);

for (int j = 0; j < tempI.size(); j++) {// iterating through an individual list

tempDouble = Double.valueOf(tempI.get(j));// attempts to cast numerical value to type double

inputsAsDoubles[i][j] = tempDouble;// assigns to index

}

}

return inputsAsDoubles;

}

public double[][] eliminateOutliers(double[][] inputsWithOutliers) {// finding and eliminating outliers

double[] totals = new double[inputsWithOutliers[0].length];// 1d array of the running totals for each predictor

double[] means = new double[inputsWithOutliers[0].length];// 1d array of the means for each predictor

double[] totalDeviationsSquared = new double[inputsWithOutliers[0].length];// 1d array of the total deviation

// from the mean squared for each

// predictor

double[] standardDeviations = new double[inputsWithOutliers[0].length];// 1d array of the standard deviations

// for each predictor

double[] upperBounds = new double[means.length];// 1d array of all the means for each predictor

double[] lowerBounds = new double[means.length];// 1d array of all the means for each predictor

boolean validRow;// flag that is true if the row of inputs contains no outliers (more than 2

// standard deviations from the mean)

// go through every value for each predictor and add it to its respective total

for (int i = 0; i < inputsWithOutliers[0].length; i++) {

for (int j = 0; j < inputsWithOutliers.length; j++) {

totals[i] += inputsWithOutliers[j][i];

}

}

// go through every value for each predictor and add it to its respective total

// squared deviation

for (int i = 0; i < inputsWithOutliers[0].length; i++) {

// finding means

means[i] = totals[i] / inputsWithOutliers.length;

for (int j = 0; j < inputsWithOutliers.length; j++) {

totalDeviationsSquared[i] += Math.pow(inputsWithOutliers[j][i] - means[i], 2);// total squared deviation

}

// final standard deviations

standardDeviations[i] = Math.sqrt(totalDeviationsSquared[i] / inputsWithOutliers.length);

}

// establishing upper and lower bounds and weeding outliers out

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

upperBounds[i] = means[i] + (2 \* standardDeviations[i]);// 2 SD from mean

lowerBounds[i] = means[i] - (2 \* standardDeviations[i]);// 2 SD from mean

}

// eliminating outliers

int numberOfValidRows = 0;

boolean[] validityRegister = new boolean[inputsWithOutliers.length];

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

validRow = true;

for (int j = 0; j < inputsWithOutliers[0].length; j++) {

// checking if there are any outliers in each row

if (inputsWithOutliers[i][j] > upperBounds[j] || inputsWithOutliers[i][j] < lowerBounds[j]) {

validRow = false;

}

}

if (validRow) {

numberOfValidRows++;

}

validityRegister[i] = validRow;

}

double[][] outliersEliminated = new double[numberOfValidRows][inputsWithOutliers[0].length];

int outliersEliminated2Index = 0;

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

if (validityRegister[i]) {

outliersEliminated[outliersEliminated2Index] = inputsWithOutliers[i];

outliersEliminated2Index++;

}

}

return outliersEliminated;

}

public standardisedPackager standardiseInputs(double[][] unstandardisedInputs) {

// standardising inputs in range [0.1,0.9]

// for final return

double[][] inputsStandardised = new double[unstandardisedInputs.length][unstandardisedInputs[0].length];

// initial minimum and maximum values, to be changed

double[] mins = { 1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000 };

double[] maxes = { 0, 0, 0, 0, 0, 0, 0, 0 };

// finding max and min values for each input

for (int j = 0; j < unstandardisedInputs.length; j++) {

for (int i = 0; i < unstandardisedInputs[0].length; i++) {

if (unstandardisedInputs[j][i] > maxes[i]) {

maxes[i] = unstandardisedInputs[j][i];

}

if (unstandardisedInputs[j][i] < mins[i]) {

mins[i] = unstandardisedInputs[j][i];

}

}

}

// standardising inputs

for (int j = 0; j < inputsStandardised.length; j++) {

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

if (maxes[i] == mins[i]) {

inputsStandardised[j][i] = 0.9;

} else {

inputsStandardised[j][i] = (0.8 \* (((unstandardisedInputs[j][i] - mins[i]) / (maxes[i] - mins[i]))))

+ 0.1;

}

}

}

return new standardisedPackager(inputsStandardised, mins, maxes);

}

public double[][] shuffleArray(double[][] inputArray) {

// randomly shuffles the data so that seasonal affections do not occur

Random rand = new Random(42);

int randomIndextoSwap;

double[] tempArray = new double[inputArray[0].length];

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

randomIndextoSwap = rand.nextInt(inputArray.length);

tempArray = inputArray[randomIndextoSwap];

inputArray[randomIndextoSwap] = inputArray[i];

inputArray[i] = tempArray;

}

return inputArray;

}

public double[][] repositionOutputToEnd(double[][] inputArray, int targetColumn) {

// the skelton value is in the middle of each row

// much easier to deal with at the end

double[][] repositionedArray = new double[inputArray.length][inputArray[0].length];

int runningIndex;

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

runningIndex = 0;

for (int j = 0; j < inputArray[0].length; j++) {

if (j != targetColumn) {

repositionedArray[i][runningIndex] = inputArray[i][j];

runningIndex++;

}

}

repositionedArray[i][repositionedArray[0].length - 1] = inputArray[i][targetColumn];

}

return repositionedArray;

}

public double[][] getOneDayAheadOutputInTheRawAsOutput(double[][] inputArray) {

// changes the output value to that of the next day

// essentially lags all predictors by 1 day

double[][] outputRepositionedFromNextDayArray = new double[inputArray.length - 1][inputArray[0].length];

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

outputRepositionedFromNextDayArray[i] = inputArray[i];

outputRepositionedFromNextDayArray[i][inputArray[0].length - 1] = inputArray[i + 1][inputArray[0].length

- 1];

}

return outputRepositionedFromNextDayArray;

}

class standardisedPackager {// contains all data needed to (de)standardise a value

double[][] inputsStandardised;

double[] mins;

double[] maxes;

standardisedPackager(double[][] inputsStandardised, double[] mins, double[] maxes) {

this.inputsStandardised = inputsStandardised;

this.mins = mins;

this.maxes = maxes;

}

}

class dataSplitter {//returned by splitData() method

double[][] trainingSet;

double[][] validationSet;

double[][] testSet;

dataSplitter(double[][] v1, double[][] v2, double[][] v3) {

trainingSet = v1;

validationSet = v2;

testSet = v3;

}

}

public double destandardisedValue(double Si, double Min, double Max) {

// destandardises a value to the raw using its minimum and maximum values

return ((((Si - 0.1) / 0.8) \* (Max - Min)) + Min);

}

public dataSplitter splitData(double[][] inputArray, double percentTrainingSet, double percentvalidationSet,

double percentTestSet) {

// separates the data into training, validtion and testing sets

// arguments are taken as multipliers to find percentages, e.g. 0.6 = 60%

int trainingDataCount = (int) Math.floor(inputArray.length \* percentTrainingSet);

int validationDataCount = (int) Math.floor(inputArray.length \* percentvalidationSet);

int testDataCount = (int) Math.floor(inputArray.length \* percentTestSet);

double[][] trainingSet = new double[trainingDataCount][inputArray[0].length];

double[][] validationSet = new double[validationDataCount][inputArray[0].length];

double[][] testSet = new double[testDataCount][inputArray[0].length];

for (int trainingIndex = 0; trainingIndex < trainingDataCount; trainingIndex++) {

trainingSet[trainingIndex] = inputArray[trainingIndex];

}

for (int validationIndex = 0; validationIndex < validationDataCount; validationIndex++) {

validationSet[validationIndex] = inputArray[trainingDataCount + validationIndex];

}

for (int testIndex = 0; testIndex < testDataCount; testIndex++) {

testSet[testIndex] = inputArray[trainingDataCount + validationDataCount + testIndex];

}

return new dataSplitter(trainingSet, validationSet, testSet);

}

}

import java.util.Scanner; // Import the Scanner class

import java.io.File;

import java.util.ArrayList; // import the ArrayList class

import java.util.Calendar;

import java.util.Date;

import java.util.List;

import java.io.FileNotFoundException;

import java.io.FileWriter;

import java.io.IOException;

import java.text.DateFormat;

import java.text.SimpleDateFormat;

class fileOperations {

private static List<String> getRecordFromLine(String line) {

// used in getValues, gets values from the file line by line

String COMMA\_DELIMITER = ",";

List<String> values = new ArrayList<String>();

try (Scanner rowScanner = new Scanner(line)) {

rowScanner.useDelimiter(COMMA\_DELIMITER);

while (rowScanner.hasNext()) {

values.add(rowScanner.next());

}

}

return values;

}

public List<List<String>> getValues(String filename) throws FileNotFoundException {

// opens given file and reads values

List<List<String>> records = new ArrayList<>();

try (Scanner scanner = new Scanner(new File(filename));) {

while (scanner.hasNextLine()) {

records.add(getRecordFromLine(scanner.nextLine()));

}

}

return records;

}

public static void createFile(String fileName) {

// creates a file with the inputted filename

try {

File myObj = new File("./reportfiles/" + fileName);

if (myObj.createNewFile()) {

System.out.println("File created: " + myObj.getName());

} else {

System.out.println("File already exists.");

}

} catch (IOException e) {

System.out.println("An error occurred.");

e.printStackTrace();

}

}

public static void writeArrayToFileAsLines(String[] inputArray, String fileName) {

// writes given string array into the given file, line by line

try {

FileWriter myWriter = new FileWriter("./reportfiles/" + fileName);

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

myWriter.write((inputArray[i]));

if (i != inputArray.length - 1) {

myWriter.write("\n");

}

}

myWriter.close();

System.out.println("Successfully wrote to the file.");

} catch (IOException e) {

System.out.println("An error occurred.");

e.printStackTrace();

}

}

public String[] convertArrayToStringArray(double[] inputArray) {

// converts a double[] into a string[]

// to make for easier writing to file

String[] stringArray = new String[inputArray.length];

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

stringArray[i] = Double.toString(inputArray[i]);

}

return stringArray;

}

public String[] mergeTwoArraysAsIsAndReturnAsStringArray(double[] inputArray1, double[] inputArray2) {

// merges the 2 given double[], separating with commas

// used primarily to make graphs in excel e.g. hidden layers/mse

String[] mergedArray = new String[inputArray1.length];

for (int i = 0; i < inputArray1.length; i++) {

mergedArray[i] = Double.toString(inputArray1[i]) + "," + Double.toString(inputArray2[i]);

}

return mergedArray;

}

public String createUniqueIdentifier() {

// creates unique identifier to be able to differentiate files apart based on date and time produced

Date date = Calendar.getInstance().getTime();

DateFormat dateFormat = new SimpleDateFormat("MMdd-hhmmss-");

String strDate = dateFormat.format(date);

System.out.println("Converted String: " + strDate);

return strDate;

}

public String[] convert2dDoubleArrayToOneDimensionStringArray(double[][] inputArray) {//converts a 2d double array to one dimension string array

String[] stringArray = new String[inputArray.length];

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

stringArray[i]="";

for (int j = 0; j < inputArray[0].length; j++) {

stringArray[i] = stringArray[i] + Double.toString(inputArray[i][j]);

if (j != inputArray[0].length-1) {

stringArray[i] = stringArray[i] + ",";

}

}

}

return stringArray;

}

}