**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 DATA array randomly…… (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 EXPLAIN . 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 FINISH

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 or Tanh), and the desired use of momentum(yes or no).

FINISH

**Libraries**

All libraries used were native java libraries

FileNotFoundException

Random

Scanner

File

ArrayList

List

Calendar

Date

FileWriter

IOException

DateFormat

SimpleDateFormat

**Are there limits on your code (e.g. have things been hard-coded or can it create any MLP with any number of inputs, hidden layers, outputs, etc).**

The MLP can be programmed to have any amount of hidden layer nodes, 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.

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

After using the validation set against my MLP, I determined a final model, detailed below. I made sure not to use too many epochs or hidden nodes, as this could lead to over-fitting to the training set, which would have caused variance when testing. Furthermore, I decided not to use a higher amount of epochs as the presence of momentum when changing weights means that the same point in error space is reached in less epochs than without momentum.

Figure

Figure

From these 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 6, also indicate that ≈0.1 is the ideal value for the learning parameter

Figure 3

Figure 4

Figure 5

Figure 6

|  |  |
| --- | --- |
| Number of hidden nodes | 11 |
| Number of epochs | 4000 |
| Transfer function | Sigmoid |
| Learning parameter | 0.1 |
| Momentum | No |

GRAPH

USE MODELLED V OBSERVED

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 through 5 consider the use of momentum and its impact on MSE.

1 and 2 focus on comparing the use with the lack thereof in training, and 3, 4, and 5 look at how different number of training epochs and hidden nodes re affected by a changing alpha value.

Figures 6 and 7 demonstrate how learning parameter affects MSE.

Figures 8 and 9 look at the effect of changing the number of nodes in the hidden layer on MSE.

Figures 10 and 11 show how changing epochs in training changes MSE.

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

Figure 7

Figure 8

Figures 1 and 2 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 11 HIDDEN 4000 EPOCHS

Figures 3, 4, and 5 illustrate the effect of changing alpha values during momentum on the overall MSE.

Figure 3’s minima was when the alpha value was at its lowest, ≈0.05, perhaps showing that

The lowest overall MSE was found at alpha value ≈4.7, with the MSE being ≈0.00657

The increase after the minima was extremely steep for

IMPROVEMENTS NOT MADE – ANNEALING, BD, WEIGHT DECAY

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 |  |
| Number of epochs |  |
| Transfer function |  |
| Learning parameter |  |
| Momentum |  |
| Alpha | 0.8 |

MODELLED V OBSERVED GRAPH

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

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