AI Methods – ANN Report

Implementation

In the development of my MLP, I chose to code in Python, as I am most familiar with this language and I feel that python has some libraries that would make certain things easier when I'm coding.

The few libraries that I used are as follows:

- CSV
  - For reading the provided data from a CSV file to then standardise and split
- Random
  - Randrange – for selecting a random number between a range, to be used in splitting the data set
  - Uniform – for selecting a random float number between a range, to be used in randomly generating weights for the network
- Math
  - Exp – simply to be used for the sigmoid function

I decided to stick to a simple style of implementation for this project, with my data being stored in two separate two dimensional arrays, in the form of matrices. One to hold data on the hidden layer – weights and biases – and one to hold data on the output layer – weights and U values. This way I can easily and smoothly iterate through the matrices, accessing the data to use in the algorithm.

Here you can see what the matrices look like before they are filled with randomised weights:

HiddenLayer	Hidden 1	Hidden 2	Hidden 3	Hidden 4	Hidden 5
Bias	1	1	1	1	1
Input 1	1	1	1	1	1
Input 2	1	1	1	1	1
Input 3	1	1	1	1	1
Input 4	1	1	1	1	1
Input 5	1	1	1	1	1

When accessing the matrix, HiddenLayer[i][j] refers to the weight on the arc between Hidden node i and input node j.

If j is 0, then this is the bias for Hidden node i.

OutputLayer	Final Output	Bias	Hidden 1	Hidden 2	Hidden 3	Hidden 4	Hidden 5
Output node		1	1	1	1	1	1
U values	empty	1	0	0	0	0	0

When accessing this matrix, OutputLayer[i][0] refers to the weight between hidden node i and the output node, when i > 1.

If i = 0, then this refers to the final output of the network, and if i = 1 then this is the bias of the output node.

When j = 1, instead of weights, this refers to the "U" values of each node, which is then used to calculate the network output, and later the delta values.

I felt that using this approach to the project was a more fool proof way of creating a result, and I would hopefully encounter fewer issues. I respect that going the Object-Oriented approach would be potentially more efficient, but I think the added complexity would have made development trickier for me, especially in the limited time frame.

When I was developing the MLP, I hard coded the worked example into the data structures and made it able to calculate a forward pass, backward pass, and update the weights in accordance with the example provided. Once this was working fine, I enabled it to do this to several additional data points and have it loop (perform an epoch) as many times as I specified. The next step was to make the code more fluid so that the worked example could be substituted by the actual data provided, meaning that it can handle more inputs and therefore more weights.

When this was done, the code could calculate the Mean Squared Error (and later the Root Mean Squared Error) of a network with default values of 5 hidden nodes, and a learning parameter of 0.1. To make easier adjustments, I put the whole of the main algorithm within a function, and had the number of nodes, epochs and the learning parameter get passed into it from outside, giving me full control over the structure of the network. The only part of the algorithm that is hard coded, Is the number of hidden layers, and the number of output nodes.

In the "Training and network selection" section of this report I elaborate on some more of my implementation by talking about how I added code to analyse the accuracy of a network, and to prevent overfitting of the MLP to the training data.

<u>**Data Pre-processing**</u>

To prepare the provided data for use in the MLP, I made use of Weka to find all the outlying and spurious data, like extreme temperatures and negative wind speeds for example. For each of these outliers, I set them equal to the average of their respective field, so that no other valuable data from the record is lost.

Once the data was cleaned up, I stored it as a .CSV file to make it easier to read into my Python IDE (I had issues with the 'pandas' library, so decided to stick with the normal CSV library). Once the data was loaded in, I removed the 'Date' field and found the max and min of each field simply using Python's built in max and min functions. With this, I could then standardise the data between 0.1 and 0.9, using the formula below, to make it easier for the algorithm to work on.

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

Once the Data was fully standardised, I split it randomly into three sets of data, 60% for training, 20% for validation (to test for overfitting) and 20% for final testing of accuracy at the end. To do this, I randomly generated indexes from the data set and filled three separate arrays with this data. When an item was chosen from the data set, it was removed from that array so it couldn't be repeated elsewhere.

Training and network selection (and some implementation)

When I reached the point where I could easily make adjustments to the structure of a network, I began to test to see which combination creates the most accurate MLP. Therefore I tested three different number of nodes, each with four different learning parameters, and at varying epochs, to populate the table of results below:

Nodes	**Learning Parameter**	**Epochs**	**Mean Squared Error**	**Root Mean Squared Error**
5	**0.1**	**1000**	**0.00021**	**0.01451**
		2500	**0.00017**	**0.01334**
		10000	**0.00015**	**0.01239**
	0.2	**1000**	**0.00017**	**0.01336**
		2500	**0.00015**	**0.01250**
		10000	**0.00014**	**0.01217**
	0.3	**1000**	**0.00016**	**0.01276**
		2500	**0.00015**	**0.01225**
		10000	**0.00013**	**0.01146**
	0.4	**1000**	**0.00016**	**0.01283**
		2500	**0.00015**	**0.01245**
		10000	**0.00012**	**0.01098**
7	**0.1**	**1000**	**0.00025**	**0.01597**
		2500	**0.00022**	**0.01494**
		10000	**0.00019**	**0.01387**
	0.2	**1000**	**0.00022**	**0.01501**
		2500	**0.00021**	**0.01451**
		10000	**0.00019**	**0.01375**
	0.3	**1000**	**0.00022**	**0.01485**
		2500	**0.00020**	**0.01439**
		10000	**0.00016**	**0.01293**
	0.4	**1000**	**0.00022**	**0.01488**
		2500	**0.00020**	**0.01432**
		10000	**0.00016**	**0.01294**
10	**0.1**	**1000**	**0.00025**	**0.01609**
		2500	**0.00022**	**0.01511**
		10000	**0.00018**	**0.01356**
	0.2	**1000**	**0.00022**	**0.01500**
		2500	**0.00020**	**0.01427**

		10000	**0.00017**	**0.01320**
	0.3	**1000**	**0.00022**	**0.01493**
		2500	**0.00020**	**0.01426**
		10000	**0.00016**	**0.01299**
	0.4	**1000**	**0.00022**	**0.01487**
		2500	**0.00020**	**0.01417**
		10000	**0.00016**	**0.01297**

With this table, I could choose the best variation to get the most accurate MLP. As the table shows, the most accurate MLP I tested was with 5 nodes, a learning parameter of 0.4 and a total of 10,000 epochs (highlighted).

Once I populated this table, I implemented the code to check accuracy against the validation set every 500 epochs, this is to stop the MLP from being overfit to the training data. This was done by simply performing a forward pass through the network with each point from the validation set and taking the Root Mean Squared Error from the results. If the accuracy got worse between two of these checks, then that means the MLP is starting to overfit, so it would stop the code cycling through the epochs. When this was applied, surprisingly it regularly stopped at about 1000 epochs, meaning that at this point the MLP is the most accurate against unseen data.

Because of this realisation, if we look at the previous table it would seem like the best fit MLP at 1000 epochs is with 5 nodes, a learning parameter of 0.3. But to fully test this I needed to train them up to this 'optimum' point, and then test the accuracy against the third data set – The Test data. This was done in exactly the same way as with the validation set, but only once and outside of any epoch loop, at the end of the algorithm.

Therefore, I entered the inputs and recorded the data in the below table:

Number of Nodes	**Learning Parameter**	**Epochs**	**Test Data Accuracy (RMSE)**
5	0.1	1000	0.01390
	0.2	1000	0.01281
	0.3	1000	0.01272
	0.4	1000	0.01239
7	0.1	1000	0.01573
	0.2	1000	0.01523
	0.3	1000	0.01397
	0.4	1000	0.01366
10	0.1	1000	0.01427
	0.2	1000	0.01399
	0.3	1000	0.01274
	0.4	1000	0.01298

As you can see, each of them was stopped at the same number of epochs by the validation set comparison, and the most accurate result was from 5 nodes and a learning parameter of 0.4. Therefore, for what I tested, this is the most accurate my MLP can get, a Root Mean Squared Error of **0.01239**.

Evaluation

With my implementation of the MLP I didn’t add any extra modifications like annealing or bold driver, and because of this I feel it reached a limit to how accurate it could get. Therefore if I were to go back and implement one or more of these modifications, I would be interested to see how they improve (or even worsen) the accuracy of my MLP.

When I was testing which network was the best, I noticed generally that adding more nodes into the MLP didn’t make much of a noticeable difference, or it made the network less accurate, therefore I found that 5 nodes was usually more reliable.

I also noticed that the learning parameter made the most difference in terms of both initial accuracy and final accuracy values. I found that a learning parameter of 0.01 was far to small and would land me in a local optimum that wasn’t ideal. I also found that using 0.3 or 0.4 would provide me with the most accurate MLP, and if I had performed more in depth testing I could of found the exact value that is perfect for my model, but I decided that for the sake of this project, 0.4 would be precise enough.

As well as this, I noticed that using any larger than this value would start to show a reduction in accuracy, with a test of 5 node and a learning parameter of 0.6 resulting with a final RMSE of 0.01416.