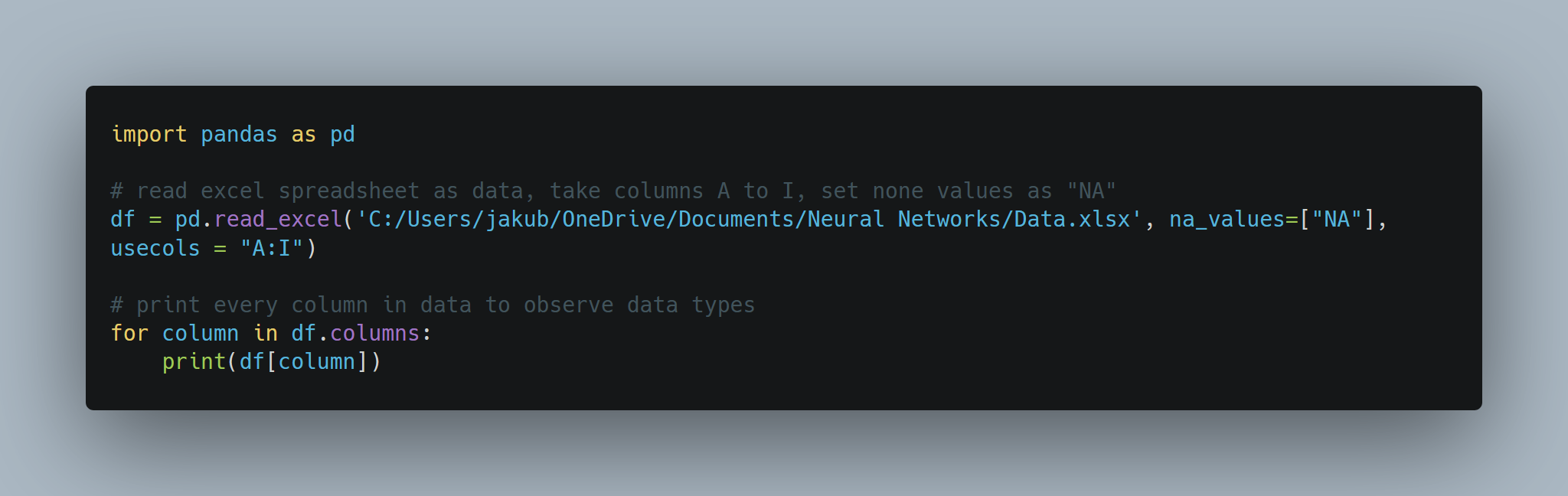
**Pre-building Plan:**

**Choice of Language:**

I will code my implementation in Java. The reasons for this are that despite Python being favoured in building neural networks, due to the wealth of libraries for the language specifically for neural works, for the purposes of this course I wish to reduce reliance in pre-built libraries wherever possible so as to understand the details of how the algorithms work themselves. In addition, I believe an object-oriented approach will be well suited to this project. Having a MLP class with neurons as attributes and algorithms like backpropagation and activation functions as methods of that class seems to me to be a natural fit for this problem. Finally, I will be coding in Java for my upcoming placement, so improving my proficiency in the language is beneficial.

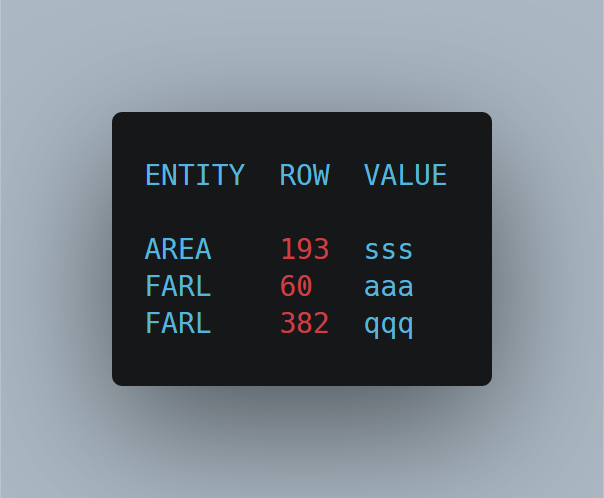
I will code my data cleaning software in Python, as the language has the support of many libraries built for data analysis and management that will make the process as frictionless as it can be. I will be using the Pandas library, as it has support for Excel sheets, allowing me to read and write directly to the .xlsx file.

**Cleaning the Data:**

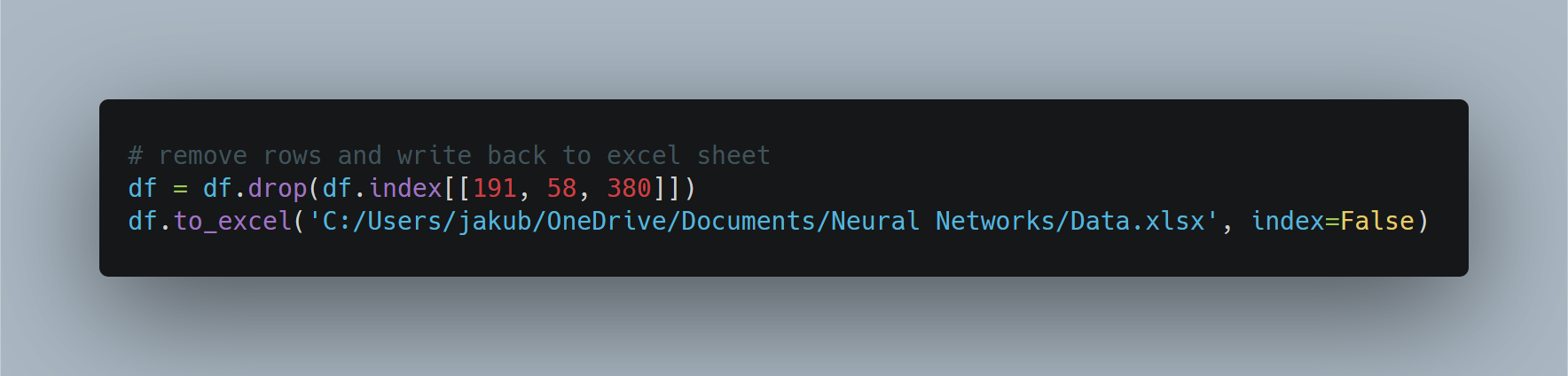
I will duplicate the Excel spreadsheet into two files, “Data.xlsx” and “Data Backup.xlsx”. The latter contains the original data that will not be modified in case an error is made in cleaning the data I have a backup I can revert to. The former will be the file that will be read from and written to the clean the data. I also will use a text file “Bin.txt” that will contain all modifications of data so we can obverse the changes.

Observe that the data type for AREA and FARL are “object” instead of “float 64”. This introduces the possibility for letters and other non-numeric values to be within those columns. Therefore we must check and remove those characters from the values in those two columns.

Therefore, we write a try, except block to catch any items that cannot be parsed into a floating point.

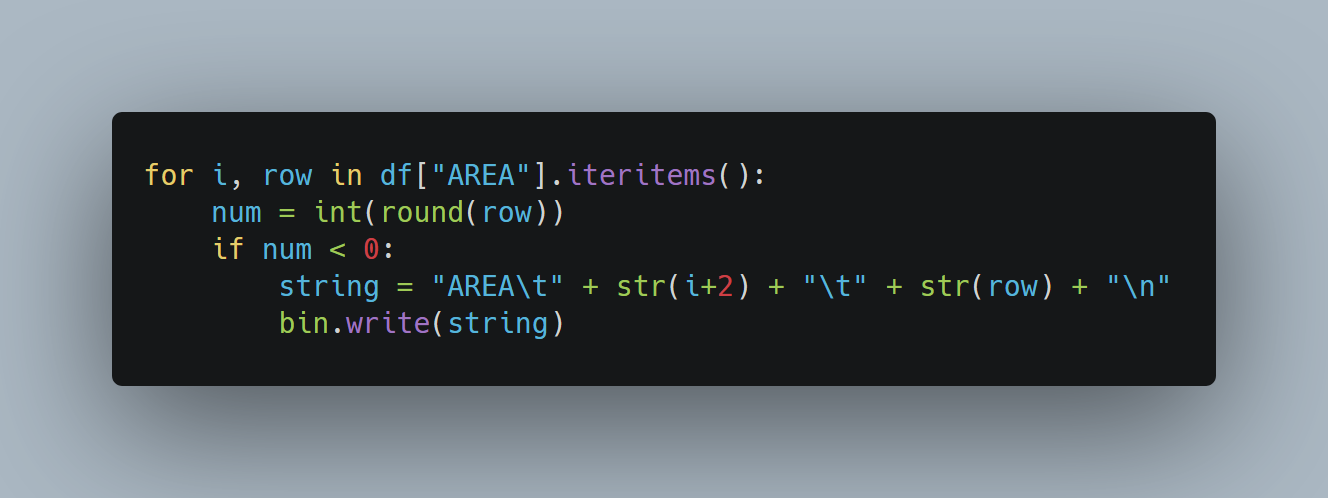
Result:

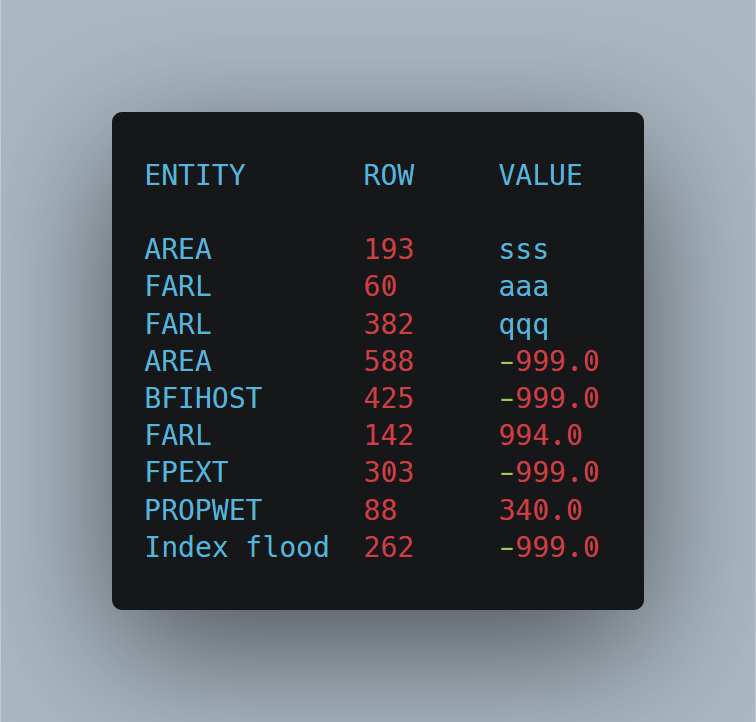
Now that we have found all values that are non-numeric and therefore clearly invalid, we simply delete these rows from our data, as there are plenty of other datapoints, and hence interpolating to try and predict what the correct was would yield little difference for a lot of effort. I built and ran code to remove these rows, then verified in the data that the deletion was correct.



It remains to perform validation on all the other values.

|  |  |  |  |
| --- | --- | --- | --- |
| Attribute | Minimum Value | Maximum Value | Justification |
| AREA | 0.00 | N/A | Area cannot be negative |
| BFIHOST | 0.00 | 1.00 | Ratio values |
| FARL | 0.00 | 1.00 | Ratio values |
| FPEXT | 0.00 | 1.00 | Ratio values |
| LDP | 0.00 | N/A | Distances cannot be negative |
| PROPWET | 0.00 | 1.00 | Ratio values |
| RMED-1D | 0.00 | 340 | Most rainfall in 24 hours (UK) |
| SAAR | 0.00 | N/A | Rainfall cannot be negative |
| Index Flood | 0.00 | N/A | Value cannot be negative |

Here is an example of how I handle validation of columns (AREA in this example. We check each row value against some condition and write to “Bin.txt” if its met.



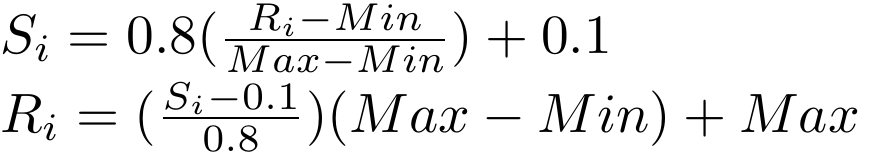
This is the final state of the “Bin.txt” file. It contains all values that are clearly not numeric as per previous examples, and every numerical value that is clearly of range. Deletion is simple, as per the previous example also. Take not however that previous deletion has changed the row numbers for some of these data points, and therefore would be different to the original data.

In the interest of avoiding hard coding my solution I have automated the cleaning process by introducing validation functions and looping over each column, applying the appropriate function to each column, and dropping rows that fail the validation process.

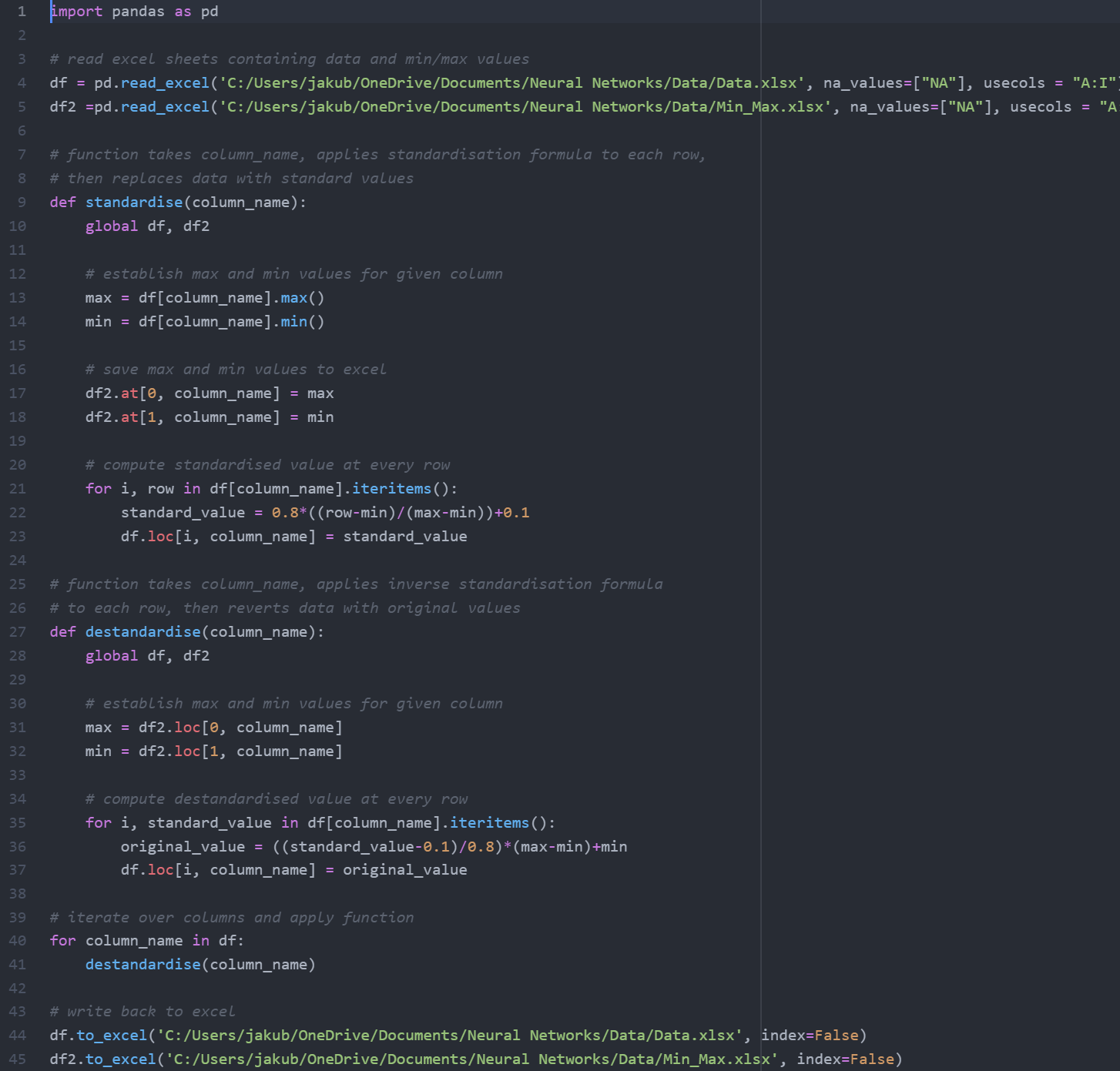
**Standardising and Segmenting the Data:**

It then remains to segment data in three subsets: a training subset, a validation subset and testing subset. In order to avoid any seasonal biases that obviously have an impact on rainfall levels, the data should be randomly segmented. However, the catchment data is not sequenced by date, and hence is already random. Therefore, we simply segment the first 60% for training, a subsequent 20% on validation and final 20% on testing.

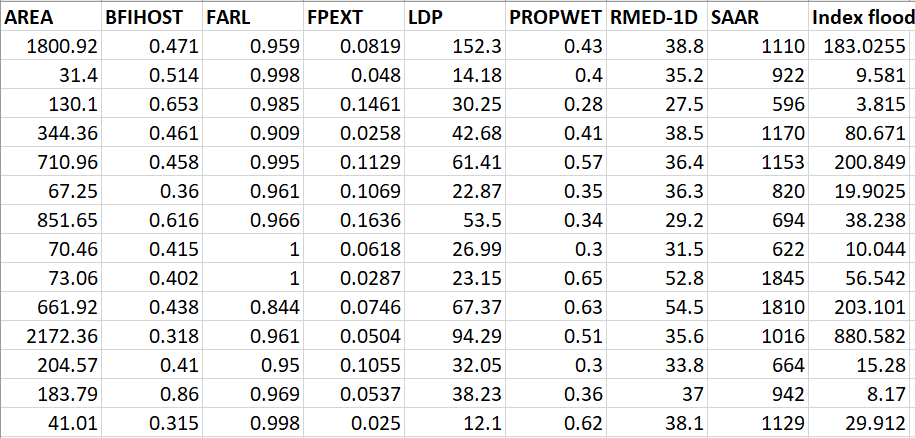
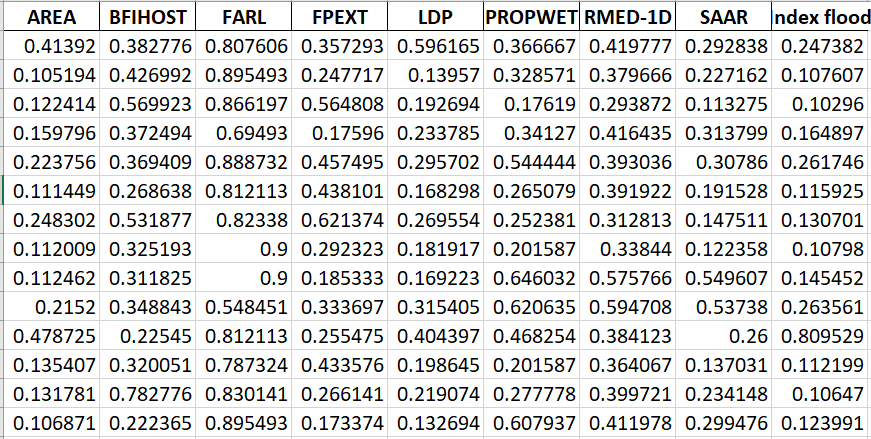
I will standardise and revert the data with their respective formulae:



These formulae can be used on the data by writing functions that compute Si and Ri values for each row. Then we can iterate over each column and apply the function to fully standardise or revert the data:



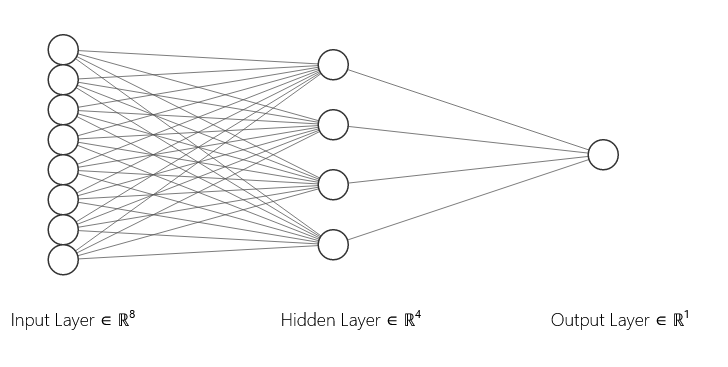
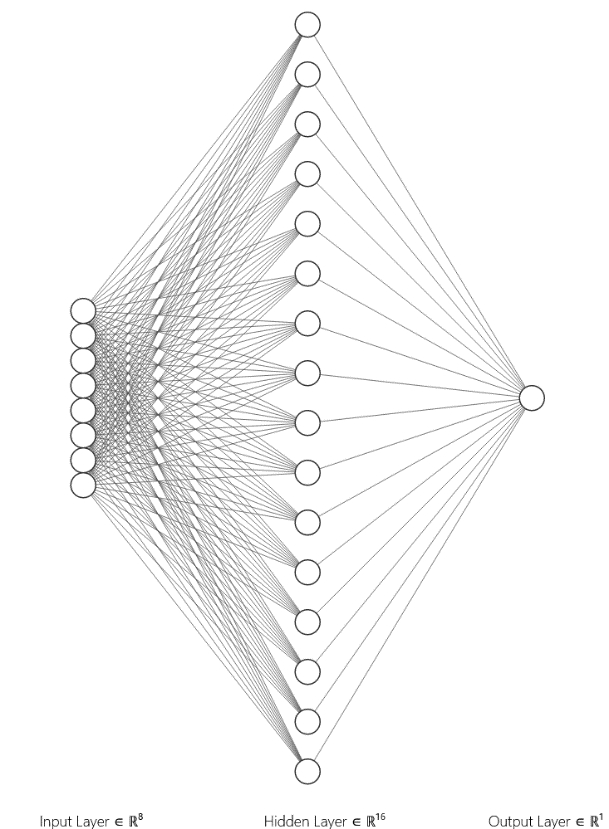
Here we can observe a sample of the difference between standardised and original data:



**Constructing the Artificial Neural Network:**

We can see from the dataset that we have 8 predictors and one single predictand. Therefore, we know that our Neural Network will contain 8 input nodes, and one output node. By suggestion in the lecture videos, I will only construct a single hidden layer, since it has been said in those videos that for simple problems, i.e., problems with a single predictand, a single hidden layer is said to be sufficient.

It is claimed in the lecture slides and videos that:

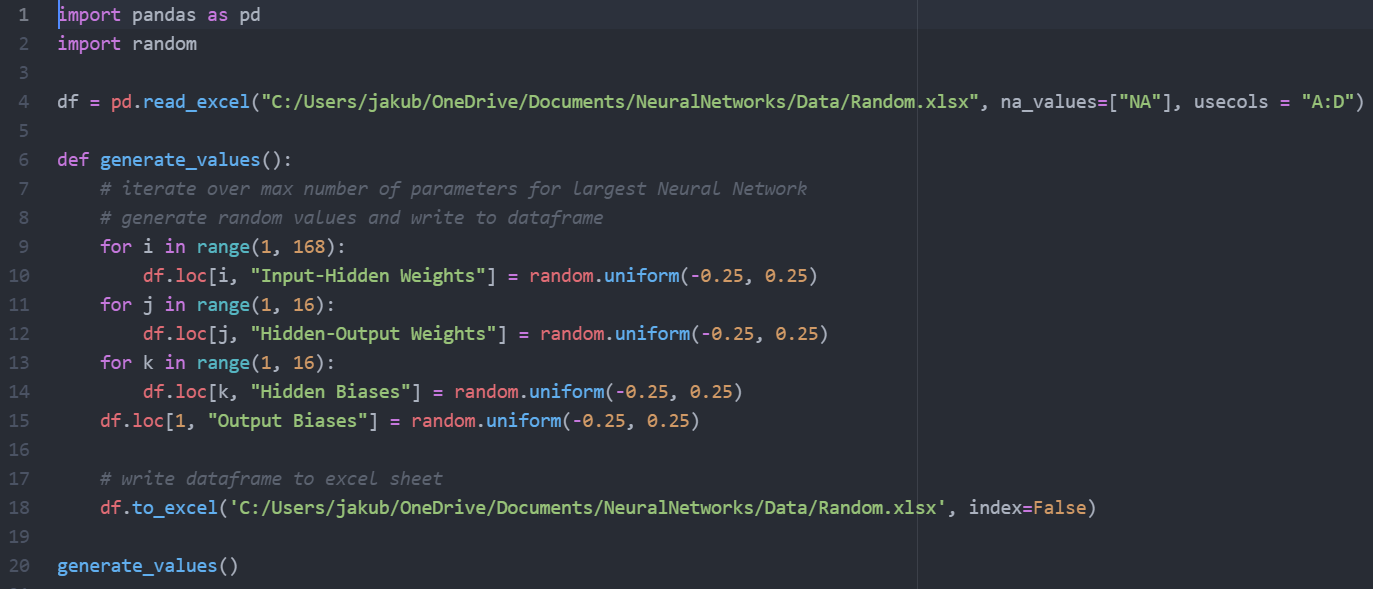
Therefore, the number of hidden nodes in our Neural Network will range from 4 to 16 hidden nodes, with the final number depending on whichever network performs the best on our validation set, i.e., what number produces the smallest error.

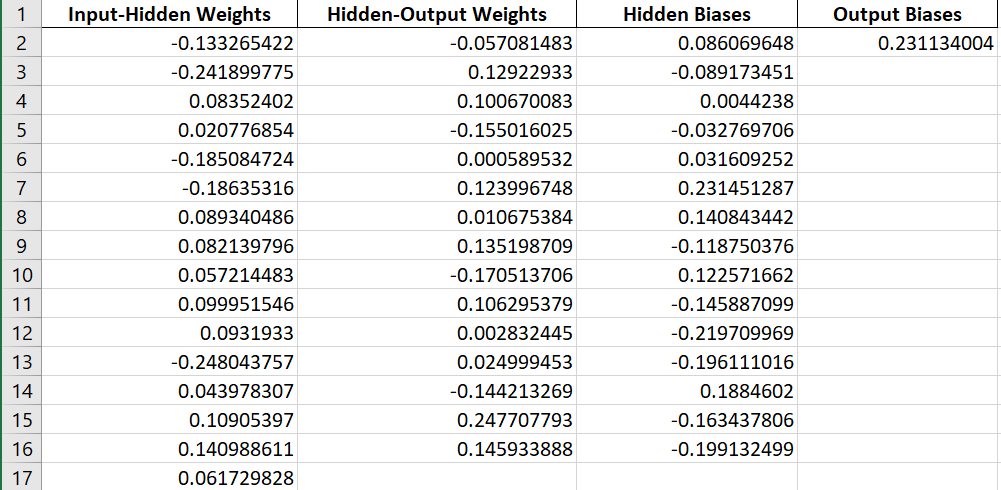
Here is just a simple comparison of the two extremes in the range of possible hidden nodes. We can see that the number of weights and biases and hence the dimension of the search-space will vary significantly across the different networks.

|  |  |  |
| --- | --- | --- |
| Parameters | 4 Hidden Node Network | 16 Hidden Node Network |
| Hidden Node Biases | 4 | 16 |
| Hidden Node Weights | 8x4 = 32 | 8x16 = 128 |
| Output Node Biases | 1 | 1 |
| Output Node Weights | 4 | 16 |
| Total Parameters | 41 | 161 |
| Global Minima | 4! x 24 = 384 | 16! x 216 = 1018 |



For the sake of simplicity and ease of testing, I will generate a random set of 161 numbers within that range, then each multi-layer perceptron will take its initial weights and biases from this pool. The code for this is trivial when using Pandas for Python:

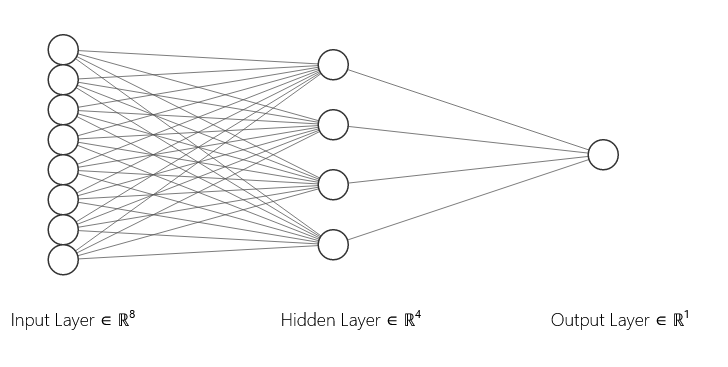




**Coding the Multi-Layer Perceptron:**

To code the Neural Network, I will construct a MLP class with multi-dimensional arrays to store weights, biases and outputs of neurons, and methods to initialise and modify those weights and biases accordingly. The three key arrays we will use are:

* Double outputs[layer][neuron]
* Double biases[layer][neuron]
* Double weights[layer][neuron][prevNeuron]



B1, O1

B2, O2

W1

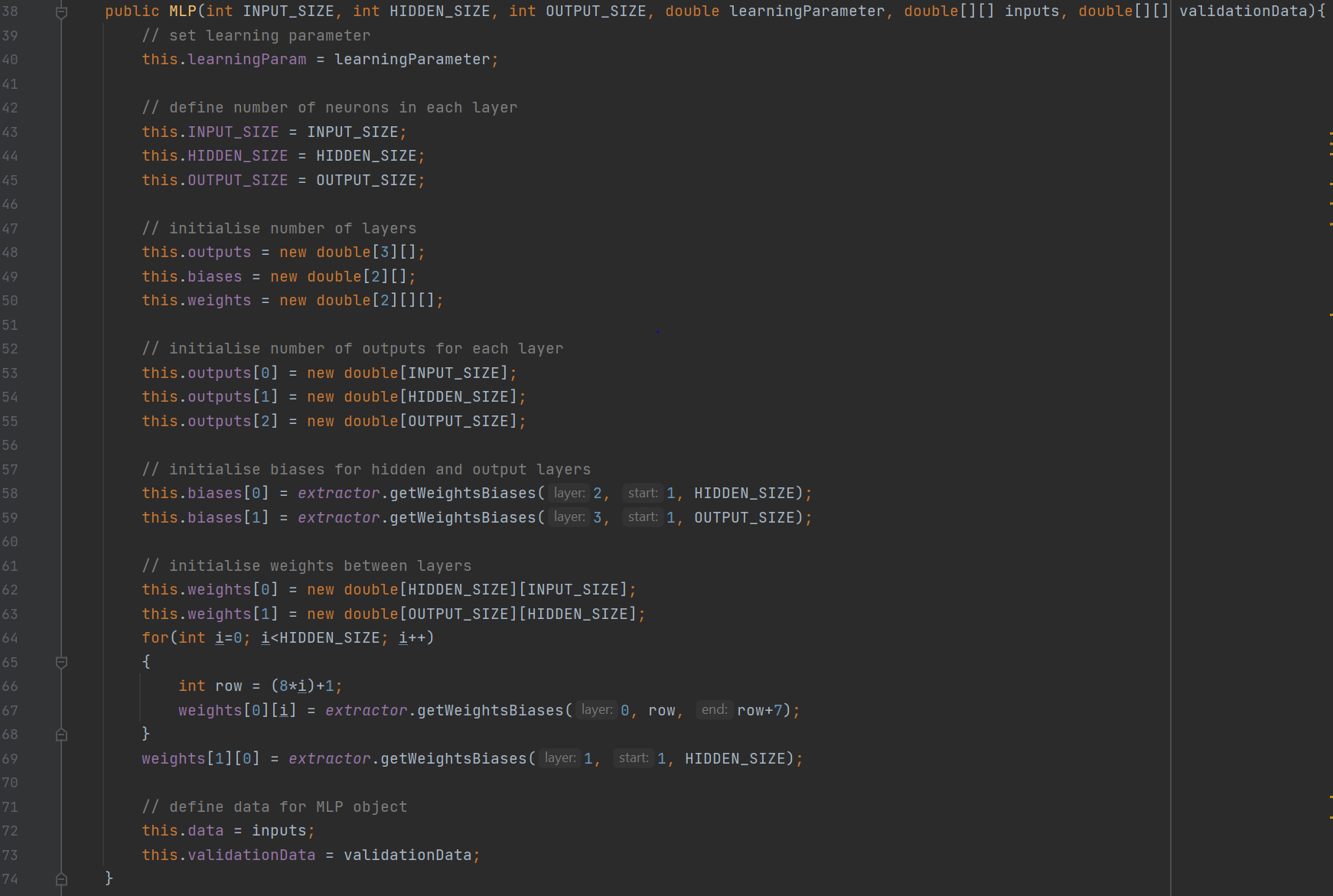
W2

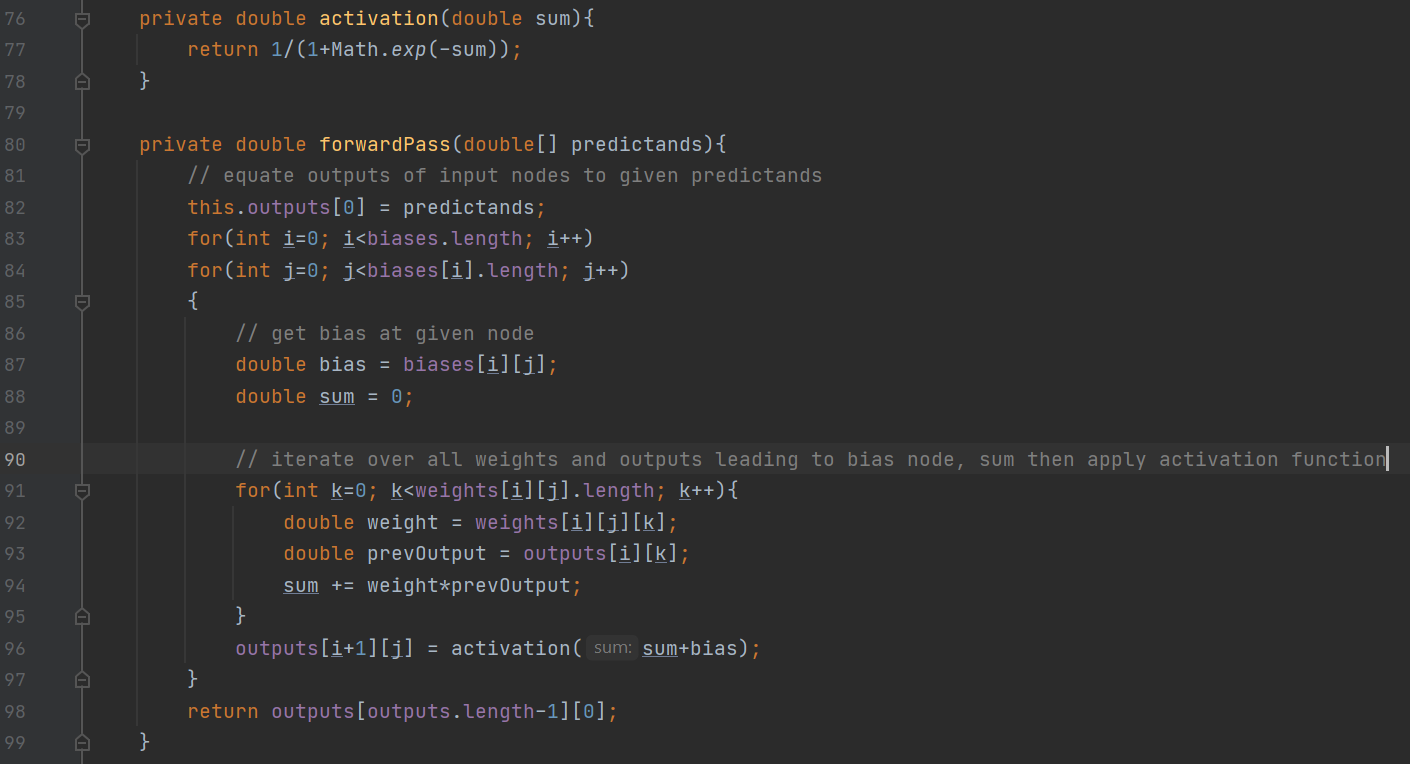
For examples on how outputs, biases are accessed for the example on the left:

B1 = biases[1][0], B2 = biases[2][0]

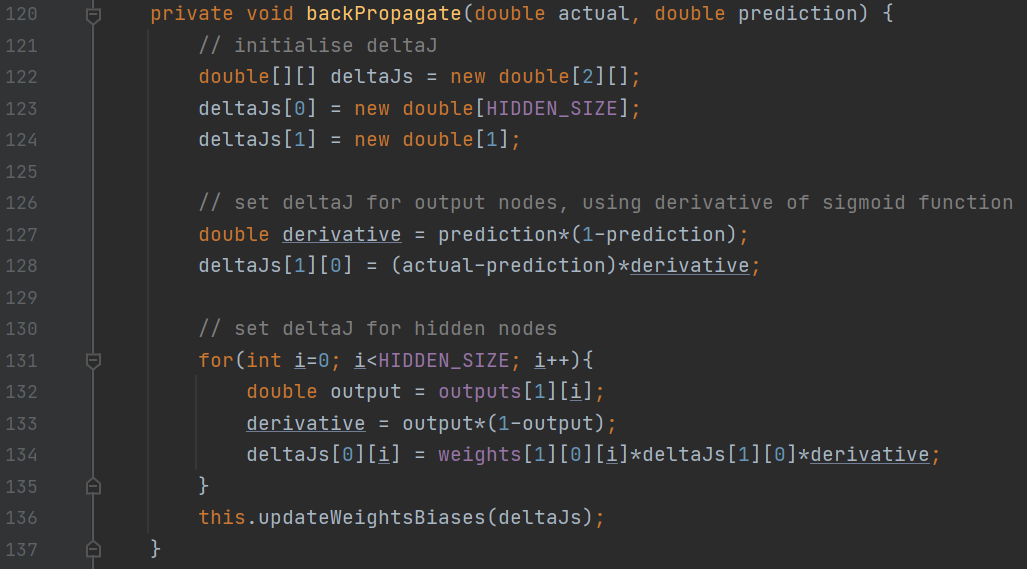
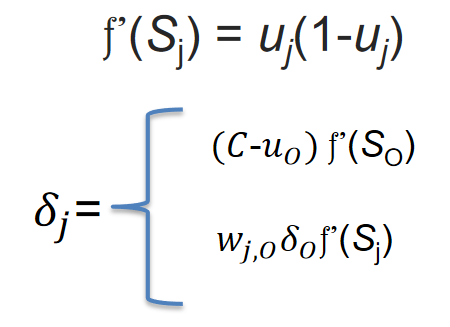
O1 = outputs[1][0], O2 = outputs[2][0]

W1 = weights[1][0][0], W2 = weights[0,3,7]

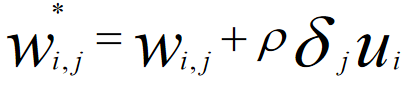
Below is the constructor for the MLP class. As you can see the primary purpose of the constructor is the initialise the various arrays with appropriate sizes for each layer. The sizes themselves are determined by the number of input nodes, hidden nodes, and output nodes, which are taken as inputs for the construction of the MLP object. Weights and biases are taken from the excel file containing initial values using a rudimentary class called Extractor that reads the Excel file.

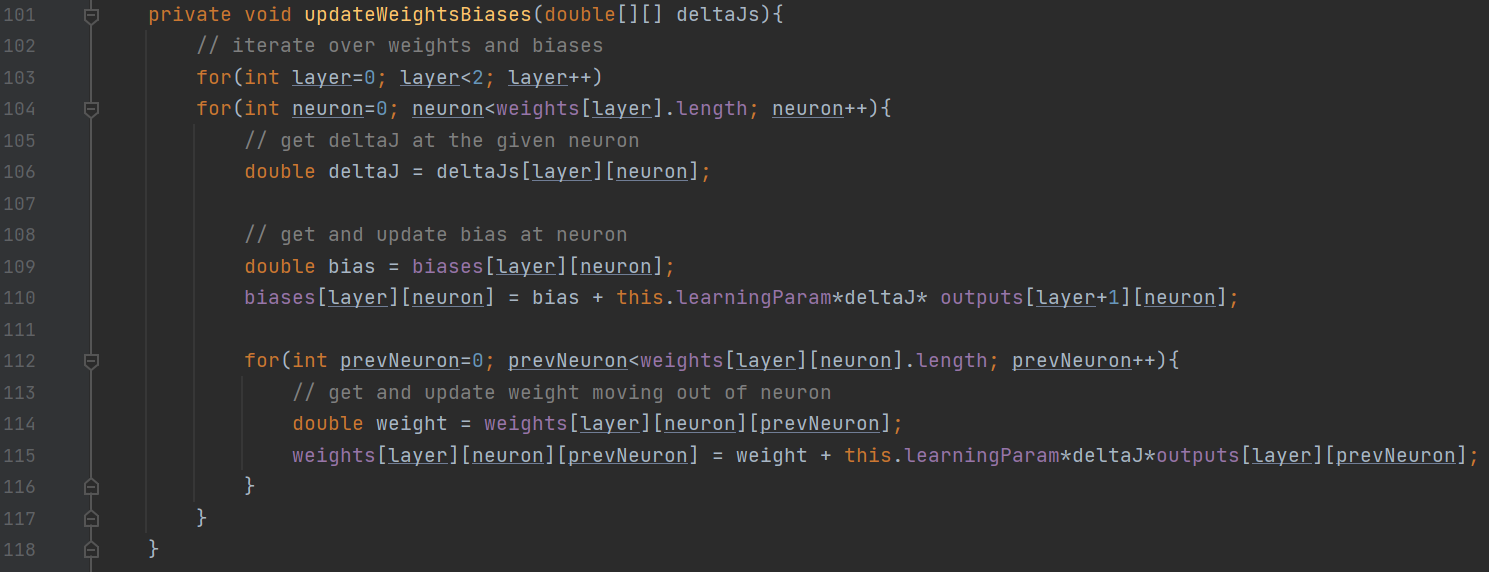
The next step is to code the algorithm computing the forward pass. Fundamentally, this algorithm is a process of taking a sequence of predictands, setting the first layer of outputs to the predictands (inputs for the forward pass can be taken as outputs of the first layer of nodes.). We then take a node, get the bias from that node, then get the sum of products of weights and outputs leading to that node. Add the bias to that sum, then apply the activation function. Take that as the output for that node and repeat the process.

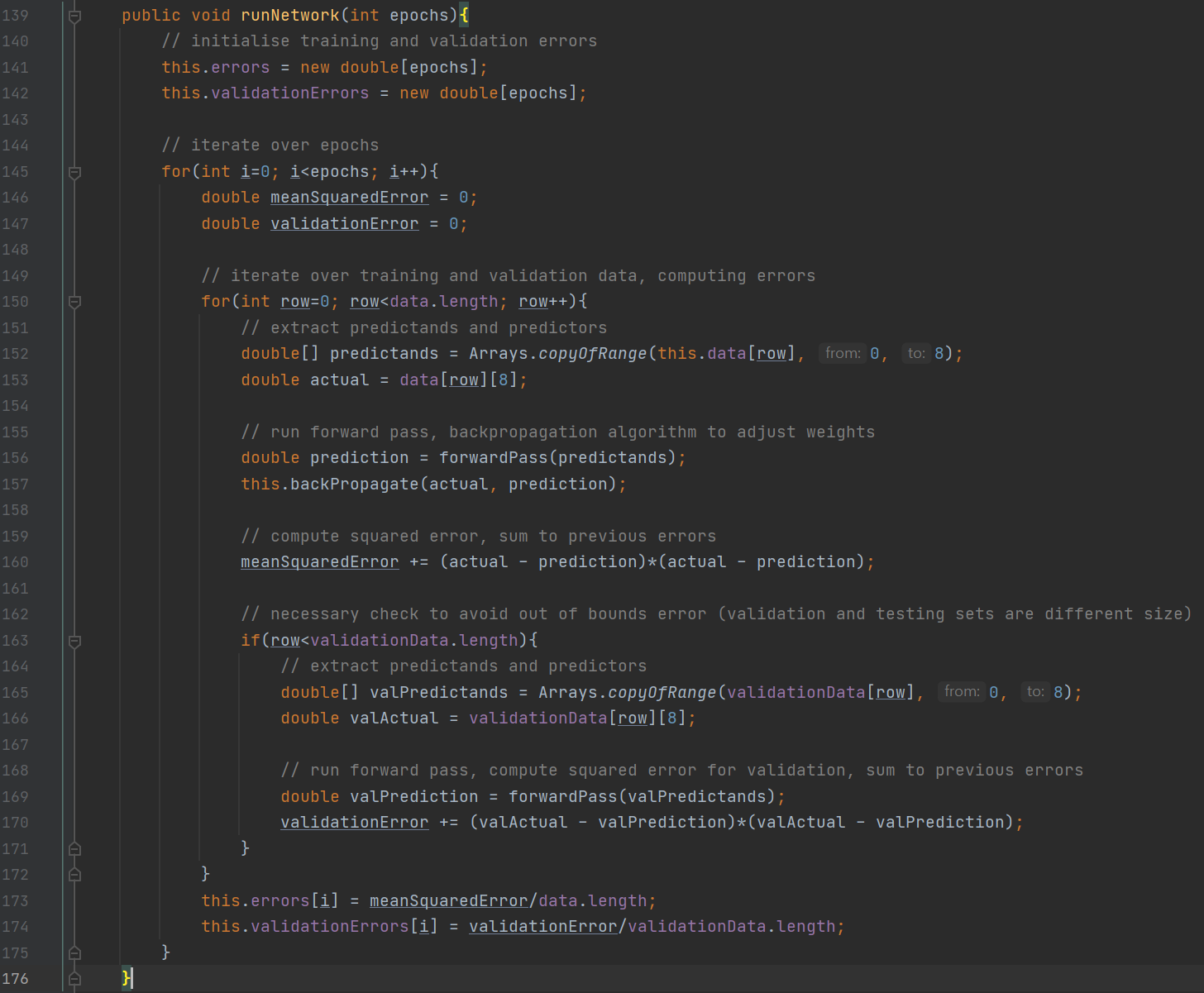
Now this algorithm functions for a single output node at the moment, which is somewhat problematic as the MLP class is supposed to take any arbitrary amount of input, hidden and outputs nodes. However, for the purpose of this problem, the algorithm will suffice, so I will not be making the necessary modifications.

After the forward pass, we obtain the prediction from the network. In order to backpropagate, we need to set the necessary δJ’s at output and hidden nodes. Since we are using the sigmoid activation, the derivative and δJ’s are as follows (taken from slides):

The algorithm takes the predicted value derived from the forward pass algorithm, compares it with the actual value, and computes the δJ’s accordingly. You may note that once again this algorithm is dependant on there being a single output node and a single hidden layer, as the δJ for hidden nodes are dependent on deltaJs[1][0] which is the δJ for the output node. Once again, it suffices for the problem we have, and hence I will keep it in its current state.

After computation of δJ’s has been completed for all hidden and output nodes, it remains to adjust all the weights and biases according to those δJ’s. This is done accordingly to the following formula in the slides:

Where Wi,j\* is the new weight/ bias, Wi,j is the current weight/bias, ρ is the learning parameter, Ui is the output out of that node.

We have everything required to run an epoch of the MLP. Therefore, I will construct a public method, called run Network(), that takes a given number of epochs as input, and calls the functions previously defined iteratively, storing errors for the training and validation sets.