AI COURSEWORK REPORT

21COB107

MIR AYMAN ALI

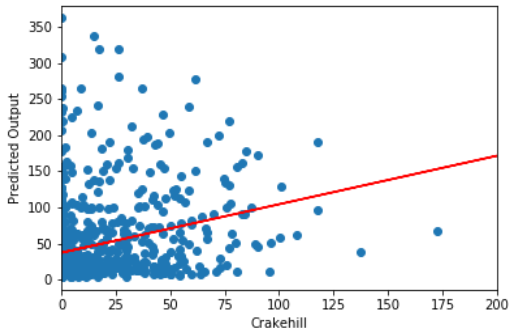
INTRODUCTION

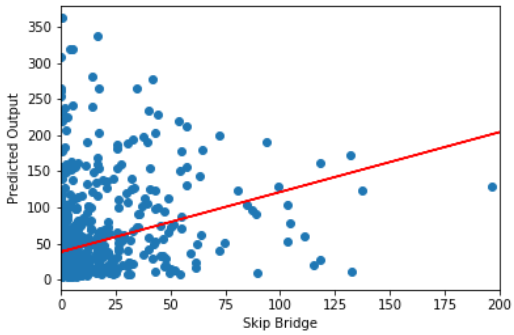
This report demonstrates my implementation of the Multi-Layer Perceptron with the backpropagation algorithm required to predict the mean daily flow at Skelton. Considering the numerous adjustments that can be made with many factors such as adjusting weights, standardizing data differently, etc, this report thoroughly goes through the effect of each adjustment on the algorithm and the possible discrepancies that may show up in predicting the output.

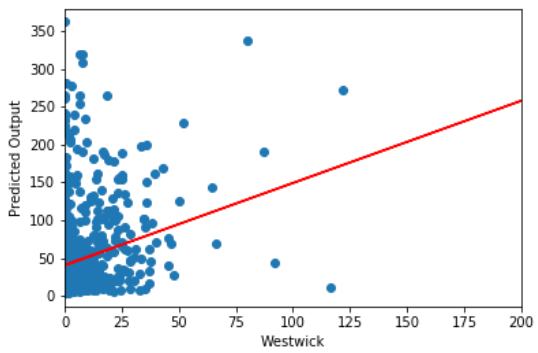
APPROACH

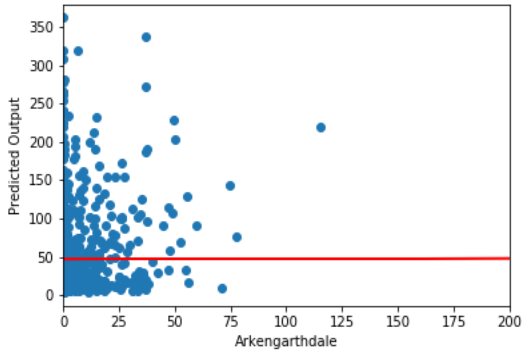
The language chosen to implement this program was **Python.** A flexible language with multiple features that allows me to manipulate and use data efficiently. The libraries used for the implementation of the algorithm were **'Pandas'**, '**NumPy', and 'matplotlib'**. Pandas was used to gather the data from Microsoft Excel and perform manipulation operations on it so that it is ready to be standardized. NumPy was used multiple things throughout the process, for aspects such as matrix multiplication, splitting of training, test and validation lists, and also in the creation of charts. Matplotlib was used primarily to create all the data points that have been displayed throughout this report. The approach that I took to coding this algorithm was an Object-Oriented approach, which I will go into more detail about when I talk about the implementation of the algorithm.

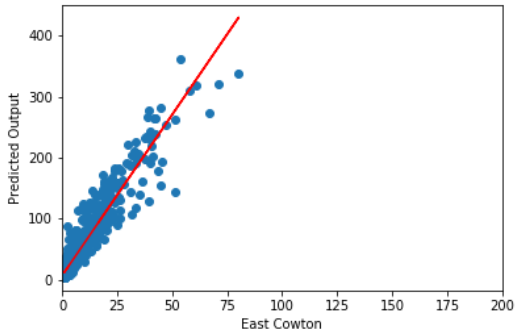
DATA COLLECTION AND STANDARDIZATION

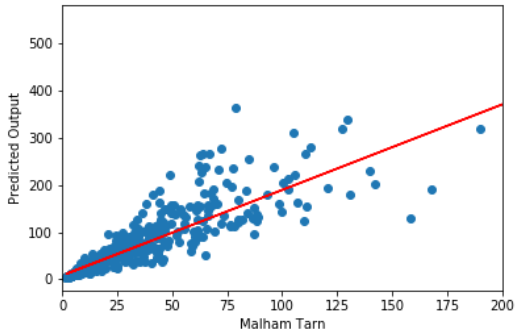


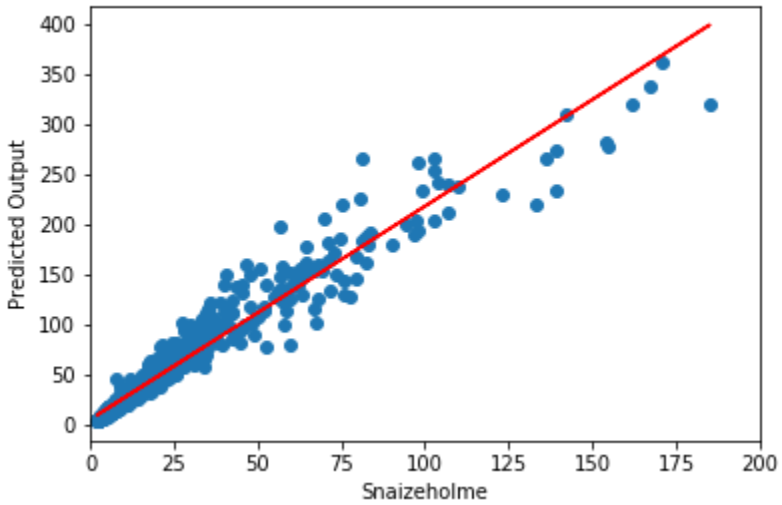








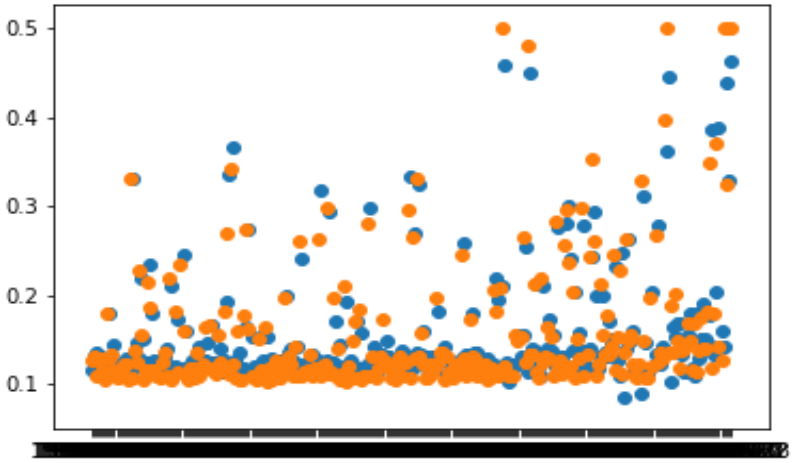




Before using the data, the predictors had to be selected to rightly predict the predictand (i.e. Skelton). For that, I first conducted linear regression for all the areas individually.

The scatter plots took the y-axis as the predicted output and for the x-axis, the individual potential predictor values were taken. For the line plotting, the x-axis was the same and the y axis was the actual output this time. As you can see in the graphs above, all of the areas are correlated with the predictand, hence why I chose all the areas to be the predictors except for Skelton.

After the predictors were identified, the data had to be prepared to be used. To do this, I first removed the outliers manually from Excel. I went through the data and removed the spurious data first. All the deleted rows are mentioned below in the table. After doing that, I went through all the columns and after doing that, I set a range of values for each of the columns that would be considered feasible to use. Once occurrences of values were found that are outside the range, the entire rows were removed. The disadvantage of such an approach is that the algorithm cannot take into account the extraordinary occurrences that might occur (Example: Highest rain flow ever seen in Skelton). After spurious data and outliers were removed manually from Excel, I shuffled the data in Excel to remove bias. The diagram below shows an example of the working of the network when data isn't shuffled. It can be seen that the network is well prepared for the same data set or similar consistencies in data, but when there is a sudden change or irregularity in the data, it is hard for the neural network to predict, which therefore makes it not feasible for this particular dataset. This is why shuffling of data was a crucial step of the process. After that, it was ready to be standardized in Python. The data was taken into the program by using 'Pandas'. After data was taken into Python, it was then written into a text file where it was then used to be split into lists of the individual predictors and predictand where it can be prepared to convert into the relevant data types, undergo standardization, and split into training, validation and testing sets where it is then used to evaluate the performance of the neural network.



DELETED ROWS

The rows with these values below were deleted in Excel before the dataset is used in Python. This is because the values are either spurious (false) or an impossible occurrence. For example, with a value of 80000, it implies that there should be an unrealistic amount of rainfall, which is therefore not feasible hence the row can be removed from the data.

COLUMN	VALUE	REASON
East Cowton	#	Spurious Value
Skip Bridge	a	Spurious Value
Skelton	a	Spurious Value
Malham Tarn	80000	Impossible Value
Skip Bridge	-999	Impossible Value
Crakehill	-999	Impossible Value
Arkengarthdale	-999	Impossible Value

When it comes to standardization, **I chose to standardize my data between 0.1 and 0.4**. Before doing that, I explored three different methods of standardizations (i.e. 0 and 1, 0.1 and 0.9 and the general formula (Evaluation done in “Evaluation of Final Model”). The formula I used for standardization is displayed below:

```
Skelton[i] = 0.3*((Skelton[i]-
min(Skelton))/(max(Skelton)-min(Skelton)))+0.1
```

After standardizing all the data for data within a certain range, I put all the inputs/predictors in a single list **(i.e Malham Tarn, Snaizeholme, Arkengarthdale, East Cowton, Skip Bridge, Westwick, and Crakehill)** and prepared it to be split. One list element had a sublist with all seven inputs for a particular row. After the inputs were sorted, I split them into three different lists and added a percentage of the inputs to each list (i.e 60% Training, 20% Validation, and 20% Testing). The training data was used to train the Neural Network and prepare it with numerous inputs to take into account all the

possible scenarios in terms of the changing data. The validation data was used during training, it is run concurrently to see the error rate with a different set. This helped me understand which training parameters worked best for my model before overfitting occurred and also helped detect when overfitting would occur. Once both were passed as inputs, the final data set was the testing data which was purely used to confirm whether the neural network is working as it should.

IMPLEMENTATION OF THE ALGORITHM

The algorithm has been submitted as part of another file. With the implementation, I'll first talk about the base implementation and then go into detail about the modifications/improvements I made to the algorithm. All the code is in a separate file. In the report, I have just gone through bits of the code.

Base Algorithm

The base algorithm is in the class NeuralNetwork. This class consists of six attributes (i.e inputs, hidden_layers, outputs, weights, derivatives, and activations) and a few methods that are listed below:

1. **forward_propogation**

   This method goes about the forward propagation operation from input signals to produce output. The inputs that are passed to this method are the actual inputs that are used to predict outputs for those inputs.
   First, the inputs are assigned to activations,
   Then they are saved to a particular index which is used later on during backpropagation.

```
activ_operations = inputs
self.activ_operations[0] = activations
```

   After that, a loop is run through all the weights. In the code block below, w is the weights. In the loop, first, we undergo the dot product of each updated activation with the weights in that loop.

```
net_inputs = np.dot(activ_operations, w)
```

Then the activation function is applied to the result from the dot product operation which is then assigned to the activations list, basically overwriting with the updated inputs always at each iteration of the loop.

```
activ_operations = self._sigmoid(net_inputs)
```

After that's done, the activations are saved to the attributes which will be used for backpropagation.

```
self.activ_operations[i + 1] = activ_operations
```

2. **Back_propogation**

This is the method that conducts the backpropagation algorithm and calculates the error signal through each output.

First, there's a reversed loop run from the output layer to the input layer.
Then the activations are received from the saved values in forward propagation.

```
activ_operations = self.activ_operations[i+1]
```

First, there's a reversed loop run from the output layer to the input layer.
Then the activations are received from the saved values in forward propagation.
Delta is then calculated by multiplying the error rate (which is passed as input) with the derivative of the sigmoid function for the activations/updated inputs.

```
delta = error * self._sigmoid(activ_operations, deriv = True)
```

After that, it is very similar to forward propagations in the steps taken just in a reversed order.
The rest of the algorithm is highlighted and explained clearly in the file with the code.

3. **gradient_descent**

This method goes about the implementation of gradient descent to assist in Neural Network learning by descending the gradient. The method takes momentum and learning rate as input

First, it loops through the self.weights attribute just like in the forward_propogation method.

Below is the assignment of variables needed for the implementation of gradient descent and momentum

```
new_weights = self.weights[i]
deriv_operations = self.deriv_operations[i]
new_derivative = self.deriv_operations[i-1]
```

'new_weights' is basically where the assignment of the weights attribute takes place.

'deriv_operations' is where the assignment of the derivatives attribute takes place.

'new_derivatives' is where the assignment of the derivatives of a decremented index takes place.

The code below goes about updating the weights

```
new_weights+= derivatives * learningRate
```

The code below goes about implementing **momentum** by updating weights

```
new_weights = new_weights - (derivatives * learningRate) +
(momentum * new_derivative)
```

4. **softmax**

```
e_x = np.exp(x)
return e_x / e_x.sum()
```

The derivative function for softmax activation function

```
e_x = np.exp(x - np.max(x))
return e_x / e_x.sum()
```

The equation for softmax activation function

5. **tanh**

```
return (1.0 - (x**2))
```

The derivative function for tanh activation function

```
return np.tanh(x)
```

The equation for tanh activation function

6. **sigmoid**

The activation function used throughout the algorithm that calculates sigmoid derivative or sigmoid activation value depends on the user input.

```
return x * (1.0 - x)
```

The derivative function for sigmoid that was used in the algorithm

```
return 1.0 / (1 + np.exp(-x))
```

The equation for sigmoid that was used in the algorithm

7. **MSE**

   Mean Squared Error loss function is used for regression. The loss is the mean overseen data of the squared differences between true and predicted values, or writing it as a formula.

```
return np.average((target - output) ** 2)
```

   The error rate is calculated while training, by dividing the sum of all MSE's for all inputs by the number of inputs.

8. **RMSE**

   The Root Mean Squared Error loss function is used when calculating the error rate for destandardized outputs. The calculation is pretty similar to MSE, just the square root is taken from the MSE calculation.

9. **train:** Method that runs all the above methods and performs the training of the neural network

LIMITS

When it comes to passing inputs in my algorithm, the fields that are editable have to be edited manually before running the program:

- Number of inputs
- Hidden layers
- Number of outputs
- Learning rate
- Epochs
- Momentum

All these fields are editable and the MLP is created with any change in inputs of the correct data type. In the gradient descent method where momentum is carried out, two inputs are used (i.e. learning rate and momentum). With any change in any of these parameters, the MLP is still created. The performance of the model can vary because these parameters were tested to produce the best results hence when they are changed, it is uncertain how the model performs.

TRAINING AND NETWORK SELECTION

1. TRAINING

1. EPOCHS AND LEARNING RATE

In the final algorithm, the number of epochs used was 750 and the learning rate used was 0.7. When using this set of specifications with RMSE as the error measure, the error measures for each of the destandardized values are 11.518632 for the training set error, 12.40486 for the validation set error and 24.104785 for the testing set error is. As you can see in the table below, the error rate of the training, validation, and test set for the last ten epochs is displayed which showcases how well the model does at predicting outputs. I chose not to train the model for more epochs as the validation error starts increasing which indicated overfitting. A more thorough analysis of all the specifications and how these were the ones decided to be used will be shown in the evaluation section below.

TRAINING ERROR	VALIDATION ERROR	TEST ERROR	EPOCH
11.580109	12.476644	24.1094	740
11.573926	12.469422	24.108759	741

11.567751	12.462211	24.108158	742
11.561584	12.455008	24.107598	743
11.555424	12.447815	24.107077	744
11.549272	12.440632	24.106597	745
11.543128	12.433458	24.106155	746
11.536992	12.426294	24.105754	747
11.530864	12.41914	24.105391	748
11.524744	12.411995	24.105068	749
11.518632	12.40486	24.104785	750

2. NETWORK SELECTION

When training the network, my algorithm takes as input three parameters when training (i.e. No of inputs, hidden layers, and no of outputs).

- **number of inputs -** An integer input that outlines the number of inputs that are needed for the algorithm. In my algorithm, I used the length of each index in the list consisting of the predictors I used
- **neurons/perceptrons in the hidden layer -** An integer input that outlines the number of perceptrons for each of the hidden layers. The input is passed as a list (For example -> [3,4,5] means 3 perceptrons in the first hidden layer, 4 in the second, and 5 in the third). In my algorithm, I chose to use [7] -> 7 perceptrons in the first hidden layer.
- **number of outputs -** An integer input that outlines the number of outputs the algorithm is supposed to return. In my algorithm, the number of outputs I required was 1, so I entered 1 as the input.
- **Weights -** The number of weights selected for the algorithm depends on the parameters above. After the values above are entered as input, all those values are added to the same list and the length of that list (i.e 3) would make up the number of weights that would be created. The values of those weights are randomly created using the numpy.random.rand function which takes the value of the above inputs to create random weights in those matrices. For example -> If perceptrons in the hidden layer are 3 and the number of outputs is 1, the weights that would be created randomly are in the form of a 3 x 1 matrix.

- **Activations -** The number of activations for each layer selected for the algorithm depends on the parameters above. After the attribute values above are entered as input, all those values are added to the same list and the length of that list (i.e 3) would make up the number of activation arrays that would be created. The values of the activations will all be zeros to start with and will be done using the numpy.zeros function. The number of zeros to be added will be dependent on the number of neurons in each layer. Essentially in the end, after overwriting is done, activations will be a list of arrays containing activations for each layer. It will be overwritten and returned during the forward propagation operation.
- **Derivatives** - The derivatives are the derivatives for the error function with respect to the weights. After the attribute values above are entered as input, all those values are added to the same list and the length of that list (i.e 3) would make up the amount of derivatives arrays that would be created. The values of the derivatives will all be zeros to start with and will be done using the numpy.zeros function. The number of zeros to be added will be dependent on the number of neurons in each layer. The rows will be the number of neurons in each layer 'i' (will be run in a loop) and columns will be the number of neurons in the subsequent layer. Essentially in the end, after overwriting is done, derivatives will be a list of arrays containing derivatives of the error function with respect to weight for each layer. It will be overwritten during the backpropagation algorithm.

EVALUATION OF FINAL MODEL

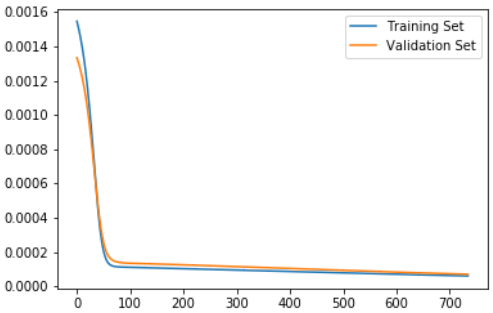
In this section, we will see how the decision to use the training parameters that were used came about.

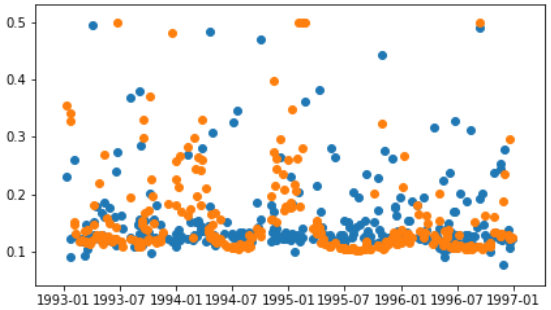
PS: In all scatter plots and line graphs, orange points are the actual output and blue points are the predicted output.
In all line charts showing MSE, the blue line is the error rate for the validation set and the red line is the error rate for the training set.

1. EPOCHS AND LEARNING RATE

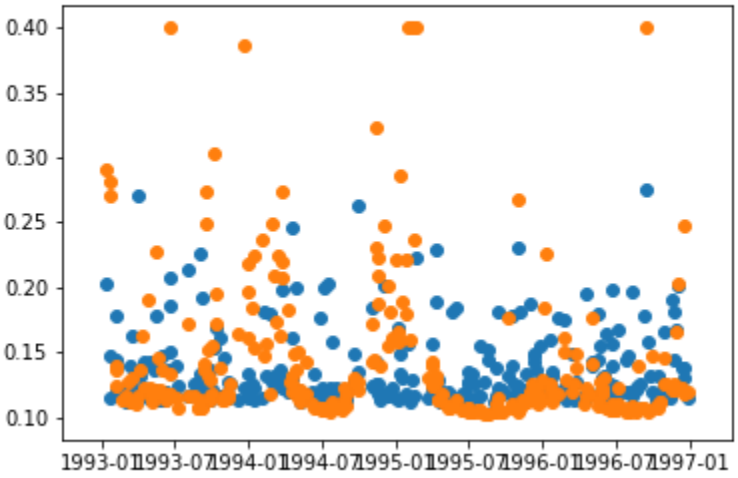
a) In the final algorithm, the number of epochs used was 750 and the learning rate used was 0.7. When using this set of specifications, the error rate using MSE is seen below. After a couple of epochs, it becomes a straight line pretty much, the

training error being around 0.0001 (Standardized outputs). The scatter plot below shows the predicted outputs when compared to the expected output.





b) Using any smaller number of epochs would cause irregularity in training as seen below. This is because the algorithm hasn't gone through enough training to reduce the error rate to a minimum which causes the predicted data to appear as shown below. It predicts outputs correctly sometimes, but not as consistently because it hasn't been trained enough.

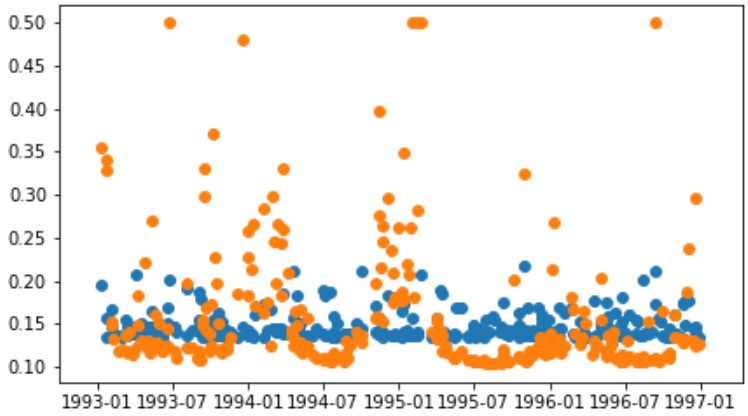


c) Using a larger number of epochs would produce more accurate outputs. But, the error rate from the validation set starts increasing after 750 epochs thus indicating that there is a high chance overfitting will occur hence why I chose to stop it at 750 epochs.

2. HIDDEN LAYERS

a) When training the network, my algorithm takes as input three parameters when training (i.e. No of inputs, hidden layers, and no of outputs). The hidden layer input can also be a list as for example [3,1] means 3 neurons in the first hidden layer and 1 in the second. For my algorithm, I chose my input as [7] which means 7 neurons in the first hidden layer. The output is the same as the output as 1a) scatterplot shown above.

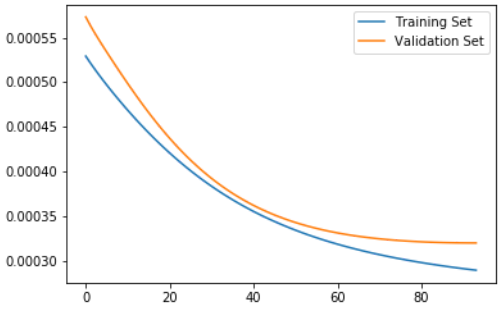
b) When I increase the number of neurons or add another hidden layer. For example [3,4], the output is shown below. This is a pretty similar pattern to 1b) as changing the hidden layer causes overfitting of data again

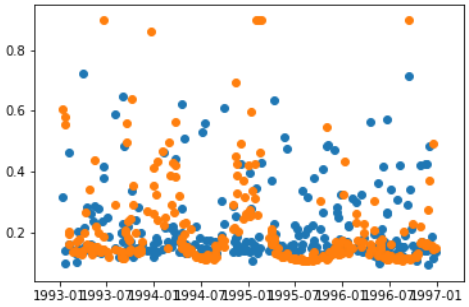


3. DATA STANDARDIZATION

When it comes to the range of standardization, I used a range of 0.1 to 0.4. But, before that, I explored a couple of other ranges as well (0 to 1 and 0.1 to 0.9). Charts for all the standardizations can be seen below:

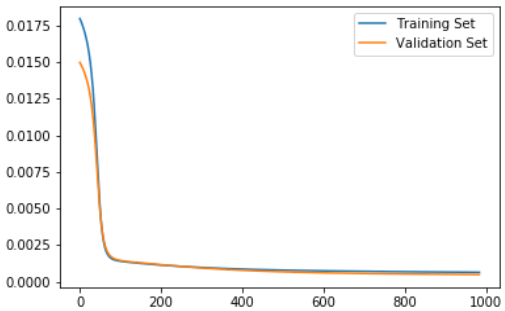
a) 0.1 to 0.9

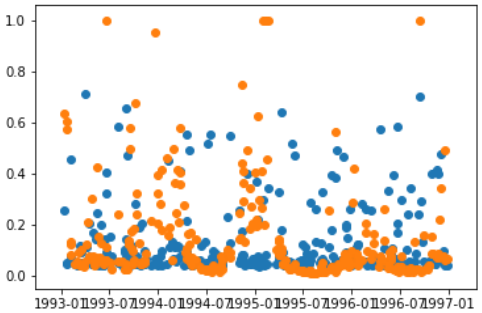




As you can see in the figure above, the error rate for the training set and validation set starts to diverge after a certain number of epochs thus indicating that there is a threat of overfitting occurring due to which the validation set error starts increasing. Also, when it comes to predicted output, the max in the range of predicted output is nearing 0.7 which is nowhere near to predicting large outputs that are nearing 0.9. The learning rate and the epochs have been adjusted to the best specification that doesn't cause overfitting.

b) 0 to 1





As you can see in the figure above, the error rate is pretty good for both validation and training set and overfitting is at a minimum. But, when it comes to predicted output, the max in the range of predicted output is nearing 0.7 again like 3 a) which is nowhere near to predicting large outputs that are nearing 1. The

learning rate and the epochs have been adjusted to the best specification that doesn't cause overfitting.

c) 0.1 to 0.4 - Used in algorithm

The figure is the same as 1 a). This particular range of standardization was chosen as it sets the Mean Squared Error to a consistent graph and also allows the predicted outputs to reach the max actual outputs and therefore, be able to predict all values.

4. ACTIVATION FUNCTION

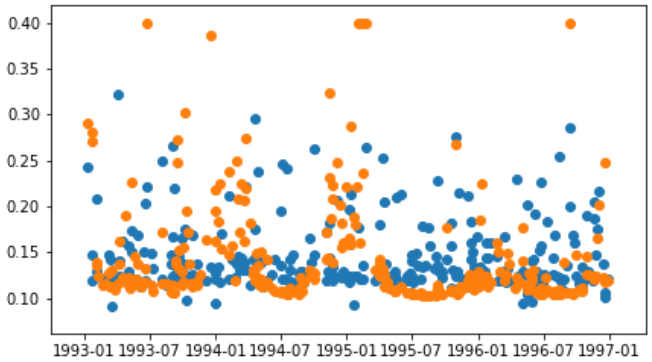
In the algorithm, I considered having an activation function to be crucial. Without this function, the weights and bias would simply do a linear transformation. A linear equation is simple to solve but is limited in its capacity to solve complex problems and has less power to learn complex functional mappings from data. A neural network without an activation function is just a linear regression model.

a) Sigmoid

I decided to use sigmoid as it was the most consistent in terms of getting the lowest error rate when it comes to training and validation set without a risk of overfitting which leads to better accuracy in predicting outputs for any datasets. As seen in figure 1 a), the predicted output is very close to the actual output and much more accurate than any other activation function. Hence, I could conclude that sigmoid would be the best activation function for my model.

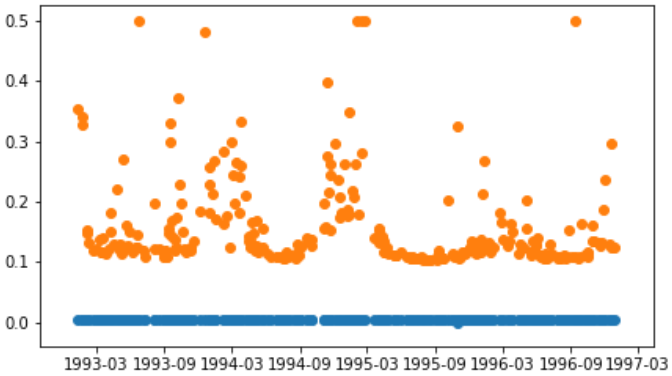
b) Tanh

For the algorithm, I decided not to go with Tanh as it was not very consistent in predicting the output. It would be able to predict outputs if the epochs were higher but this comes at the expense of overfitting of data as the validation and training error are far apart. Even when the epochs are increased to the maximum without overfitting, the predicted outputs form a similar pattern. Hence, I concluded that the tanh activation function would not be the best fit for my model.



c) Softmax

Another activation function I experimented with is the Softmax function. As seen from the results below, all of the predicted outputs are completely wrong and way off. Originally, I thought using softmax would be beneficial as it is what's generally used when dealing with multiple inputs. But, seeing the performance of the activation function, I can conclude that this isn't the best fit for my model.



5. ERROR MEASURES

In my algorithm, I chose to implement only two error measures (ie MSE and RMSE)

a) Mean Squared Error

To execute MSE, the formula I used was:

```
return np.average((target - output) ** 2)
```

When the MSE function is called, the subtraction of predicted output from expected output takes place and then is squared for each of the outputs that are passed as inputs. Then the average of the function is taken.

```
sum_errors / len(items)
```

After the function is called, the output is added to a variable “sum_errors” when training the model. This is then divided by the number of inputs/number of times the MSE function is called. This basically returns the training and validation error after each iteration.
The graph for error rate and performance is the same as the one in 1 a)

b) Root Mean Squared Error

To execute RMSE, the formula I used was the same as the one used in MSE. The only thing added is the “math.sqrt()” function which is used as below:

```
math.sqrt(sum_errors / len(items))
```

The function above basically conducts the square root of MSE which calculates the RMSE to get the error rates of both the training, validation, and test set. I used RMSE to calculate the error rate for destandardized outputs, which has been displayed earlier.

6. OPTIMIZATIONS

a) Momentum

The first optimization method I added to my algorithm is momentum. The formula used for updating the weights was $W'i = Wi - \alpha\Delta Wi + \mu\Delta Wi-1$

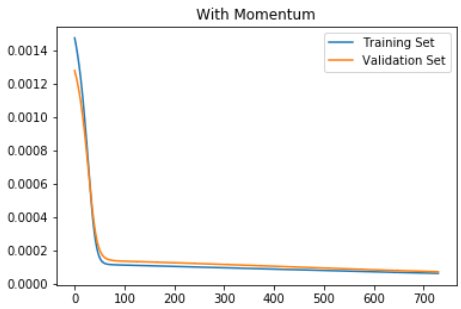
The individual components of the formula are as follows:

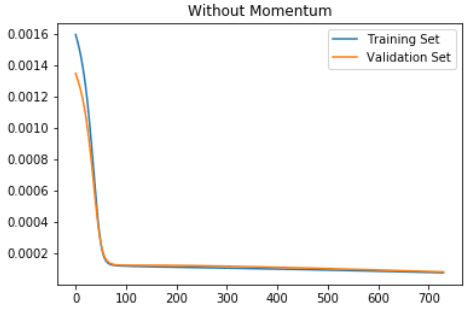
- Wi: The weight matrix at i that’s being overwritten
- α: The learning rate
- ΔWi: The derivative matrix at i
- μ: momentum value

This was implemented in the gradient descent function. The analysis of how using momentum affected my algorithm is shown below,

ERROR_RATE

When it comes to error rate, the error rate of the training set is 0.007637 whereas the error rate of the validation set is 0.007602 at epoch 750 with a learning rate of 0.7. This is marginally better than the error rate without momentum. Without momentum, the error rate of the training set is 0.008825 and the error rate of the validation set is 0.009133. The diagram on the left is the MSE with momentum and the one on the right is the one without momentum.

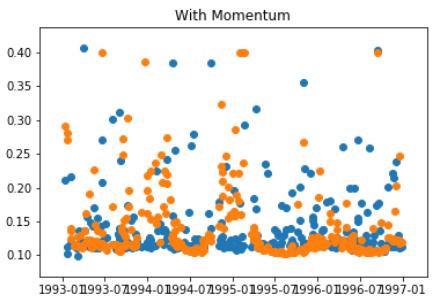


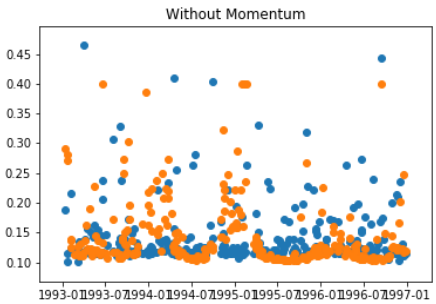


As you can see in the diagram, there is a very marginal but better error rate in the graph where momentum is used.

PREDICTED OUTPUT

When it comes to predicting outputs after momentum is added, as you can see from the line graphs below, the chart with momentum is much better and more consistent at predicting outputs than the chart without momentum. When using momentum, the model can predict outputs that are a sudden increase/decrease from the current output which means it can predict out-of-the-ordinary instances such as higher than usual rainfall in Skelton. Due to the reasons mentioned above, I chose to use momentum as it makes my model marginally better at predicting outputs.

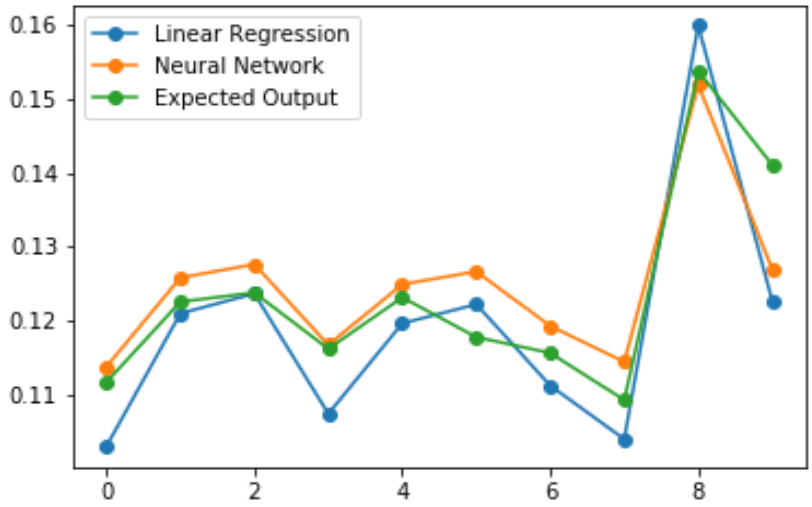




COMPARISON WITH DATA-DRIVEN MODEL

In this section, I compare the results of my neural network in predicting outputs when compared to a data-driven model. In this instance, I'm going to be using Linear Regression. To do this, I created an instance of the Linear Regression class in Scikit-learn. To compare if the outputs of my network or the predicted outputs using Linear Regression are better, I created multiple diagrams to analyze. In the line graph below, the x-axis is the expected output and the y-axis is the predicted output of the Linear Regression instance. This graph helps show the correlation between both the outputs to the expected output.

With the chart below, we can see a more detailed analysis of which model is better at predicting results. For this chart, I haven't used all the outputs, just about ten from each of the lists. As you can see, the neural network output is better at predicting outputs than the Linear Regression Instance. It is overall a lot more consistent when it comes to predicting outputs as well.



ERROR RATES

When comparing error rates of both predicted outputs of test sets with the Neural Network, the Linear Regression model has a Root Mean Squared Error of 0.284929 and for my neural network model, it is 0.011777. Seeing the massive difference in the error

rates, it can easily be seen that my neural network model is much better at predicting outputs than the Linear Regression model.

Seeing the data points displayed above, we can conclude that in this instance, neural network prediction of the outputs is better than linear regression. Hence, the neural network is the better model to use.

CONCLUSION

That is my entire implementation of the Multi-Layer Perceptron with the backpropagation algorithm and its optimizers. All analysis of the multiple data points led me to believe that the specifications/parameters that were used, produced the best outputs and have covered all possible cases and discrepancies.