ANN Implementation

AI Methods 21COB107

Nickolai Tchesnokov

March 30, 2022

Contents

1 Introduction

In this report we will be discussing the implementation of a multilayer perceptron to predict the mean daily flow at Skelton the next day. We will be using a dataset which provides us with time series data mean daily flow values at different locations (including Skelton) and also daily rainfall total values at different locations. We will first go over how we go about preparing the dataset to be passed to our algorithm and what steps must be taken before doing so. We will then be building our model to take the data and train it. Once we have our base implementation of the model, we will try to optimize it further using various optimisation techniques and compare our results to fit our data to the best possible model. Lastly, we will make a comparison of our model to a multivariate linear regression model which we will implement alongside our multilayer perceptron.

For my multilayer perceptron implementation, I have made the decision to utilize Python as the language to build my model on, as it is a great language for AI and machine learning development, and is the major language used in industry in this field. Ontop of being comfortable to use, easy to understand and very versatile, the main reason for its popularity is the libraries and modules provided to the user. This prevents users from having to code everything themselves from scratch and allows for cleaner, more efficient writing of code. Since machine learning involves a lot of data processing, such libraries are very useful, giving you a variety of options to handle and manipulate the data you are using, for instance Pandas and Matplotlib; two of the libraries I will be taking advantage of.

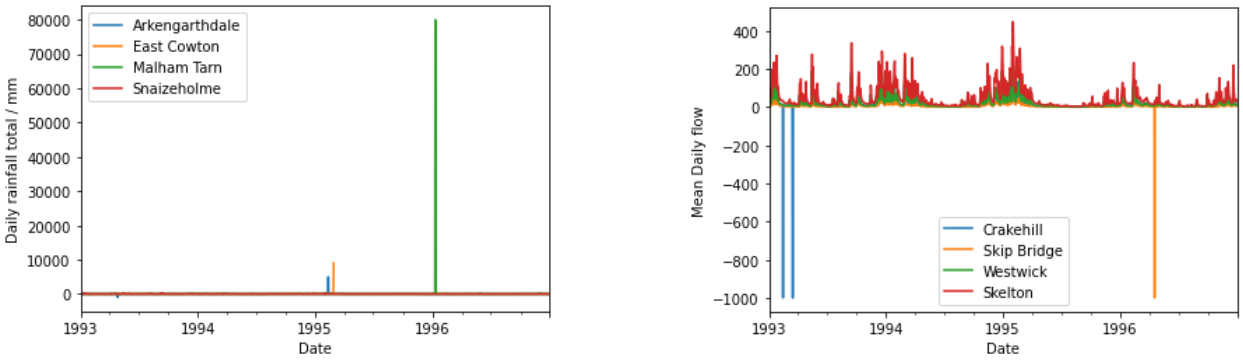
2 Data Preprocessing

2.1 Cleansing

The first thing I did after importing the dataset into python was to check for and remove any outliers. After converting all columns in the data frame to the same type (type float), I plotted the data, taking a look at the general structure of the data provided, but more specifically to check for anything that may look like an outlier. Since 4 of the columns record mean daily river flow, and the other 4 record daily rainfall total, we cannot simply plot a graph of the entire data frame. Separating the two and showing their data on separate graphs will allow us to spot their associated outliers. The two graphs used for this are shown below in figure 1.

If we take a look at figure 1b, we can clearly see that there are multiple negative values in our dataset. This should not be the case as it is impossible for the mean daily river flow to be a negative value.

In figure 1a, we can see that there are also a few outliers which need to be taken care of such as the record showing daily rainfall total of 80,000 mm in Malham Tarn.



(a) Daily Rainfall values

(b) Mean daily river flow values

Figure 1: Raw Data from our dataset

Looking at the excel file directly, there are also values in the data which are not of the correct type (ie. non integer/float types). Changing all columns in the data frame to the same type has now converted these incorrect type values to NaN (Not A Number). These can also be considered outliers and have to be dealt with as well.

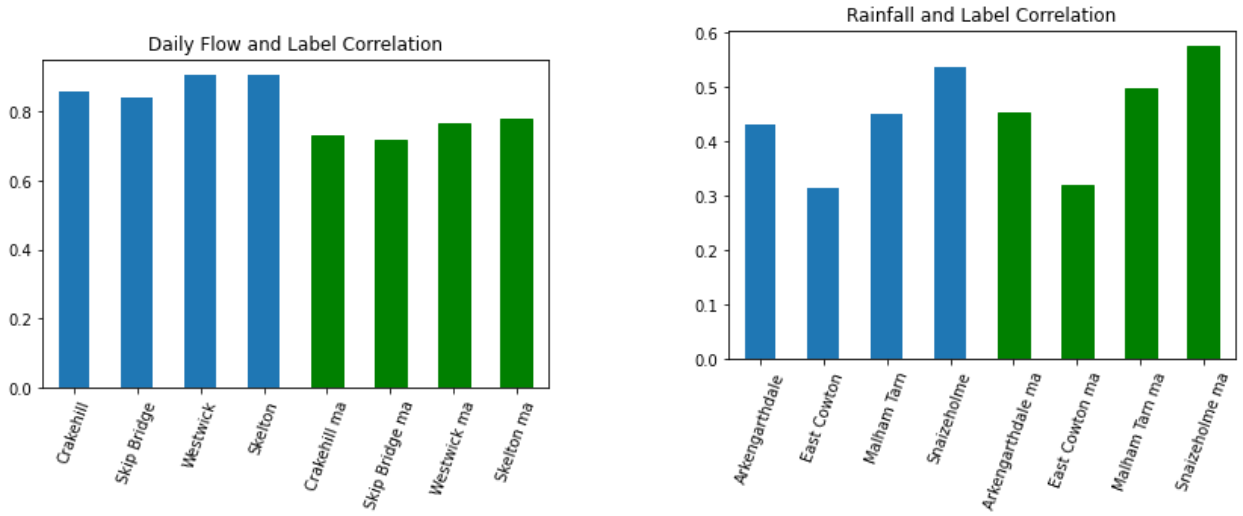
The method I have employed to take care of all outliers is via interpolation. My initial thought was to remove the rows from the data frame all together, however this is not a great solution as we are dealing with a time-series dataset, where we have not yet set our y-labels. The y-labels for our inputs are going to be the next days Skelton mean daily flow. This means we need to shift the Skelton column in the dataset up to align the next day's values with the current day predictor/input values (all other columns in the dataset). If we deal with outliers by removing rows, then the records of some days will be removed. Shifting the Skelton column would then not be possible as it would cause inconsistencies, as some y-labels will get shifted to the incorrect date.

To interpolate all outliers at once, I first identified and converted all outliers to NaN values, as these already existed in our dataset as mentioned above. The first step was to identify and change all negative values, as negatives should not exist with the type of data we are dealing with. I then checked for mean daily flow outliers, both lower bound and upper bound. However, for daily rainfall, I only checked for upper bound outliers, as the lowest a value could be is 0 (since negatives were removed) and 0 for daily rainfall is normal and should not be considered as an outlier. To check for upper/lower bound outliers, I looked for values that are over 3 standard deviations from the mean, or in other words:

$$\text{upper bound} = \frac{x - \mu}{\sigma} > 3 \qquad (1) \qquad\qquad \text{lower bound} = \frac{x - \mu}{\sigma} < 3 \qquad (2)$$

2.2 Moving Averages

Moving averages provide the means to smooth out data in time series forecasting. Calculating the moving average of our features we combine a certain number of consecutive days together into an average and use this as our new value. Specifically, in python, we will use the rolling() function on our dataset, which will group the values into a window size which we will set as 3. We can then take the mean values of these groups of data and get our transformed dataset. Now that we have our moving averages for all features, we have to decide which ones shall be used for training, and which should be left out. The method I employed to decide which features' moving average would be most useful for our model was to calculate the correlation between each feature and its moving average with the y labels. This will allow us to quantify the degree to which the labels are dependent on each feature in our dataset. To visualize these correlations, we plot two charts; one for daily river flow and one for total daily rainfall.



(a) Mean daily flow features

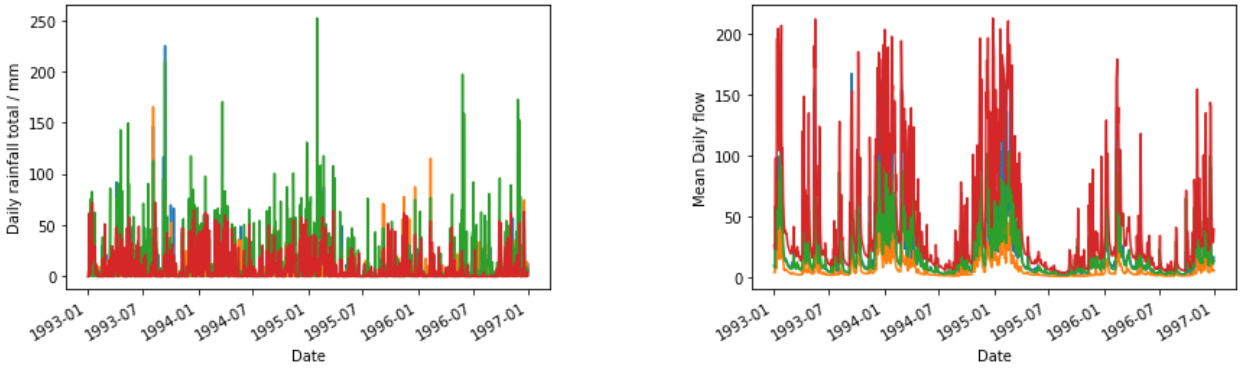
(b) Total Rainfall features

Figure 2: Correlation of features and moving averages with expected output

In figure 2, the blue bars show our original features, whereas the green bars are the moving averages of the original features. We can see from figure 2a that there is in fact a lower correlation between the y labels and the moving averages compared with the original data. Therefore, using these moving average features for daily flow is unnecessary as it could weaken our training model. However for total daily rainfall in figure 2b, the moving average correlation is slightly higher than the original data. Therefore, to optimize our training model, we will replace the original total daily rainfall data with the moving average data, but keep the original data for daily flow, and remove its moving average data.

2.3 Data Splitting

Having interpolated our data and configured our features we can now split our dataset into inputs x and labels y. We will have 8 inputs being the 8 columns we are provided with, and our labels being the shifted Skelton column. Since we are shifting the column upwards, the first Skelton y label can be removed as it is of no use anymore, as well as the last input row as this row has no y label associated with it, and so cannot be of use. Shuffling the data is also an important step that must be taken. This is because we want to avoid any chance of there being a bias/pattern when splitting our data into training, test and validation sets before training our model. In this case, we have a time-series dataset providing data on rainfall and river flow. One example of a potential bias/pattern that could exist is seasonal rainfall totals. The amount of rainfall that occurs in different times of the year could very well be drastically different, and so ignoring such a bias could result in our training set and testing set having very different rainfall daily totals to each other which would give us worse test results after training.



(a) Daily Rainfall values

(b) Mean daily river flow values

Figure 3: Correlation of features and moving averages with expected output

Figure 3 shows our unshuffled data after all outliers have been interpolated. Looking at figure 3b, there is clearly a pattern within the dataset, as at certain times in the year, the mean daily flow seems to be much lower than at other times of the year i.e. there is a correlation between the mean daily river flow and the time of the year.

After shuffling our data, we can move on to splitting our dataset into training and testing sets using an 80/20 split. We do this so we can estimate the performance of our algorithm when making predictions on data that has not been seen before. Therefore, the training set will be used to fit our model, whereas the test set will be used to evaluate our fit model. In practice, this is the expected usage of a machine learning model; fitting the model with known features/inputs and outputs, and then having the model make predictions on future inputs which have no associated outputs. We will also make a further split from our training set to get a validation set, which will be discussed in more detail later.

2.4 Standardization

Standardization is an important step to take as we want to remove any bias that may occur when using a dataset that contains data with different measurement types or units, or in our case, mean daily river flow and total daily rainfall.

To standardize our dataset we will use the following formula:

$$a + \frac{(x - min(x)) \cdot (b - a)}{max(x) - min(x)} \tag{3}$$

Where x is the dataset and a, b are the values we set to give our dataset values a range between a and b. In our case we will set these to be 0.1 and 0.9. We must apply this to both the training set and the test set. We must do this after splitting the training and test sets, as we cannot standardize the entire dataset at once. This is because standardizing all data at once would cause information leakage from the test set data, resulting in skewed estimates when testing the model. When standardizing the test set, parameters used for standardizing the training data should be used. In our case this is using the minimum and maximum values of the training set when standardizing the test set. This is because there may be different maximum and minimum values in the test set, and using these to standardize the test set would cause the same values from both the training and test set to be standardized to different values even though their raw value is the same.

3 Implementation of MLP

After initializing our weights randomly (small random numbers) between the input, hidden and output layers and initializing our bias values in the same manner, there are four steps that must be taken within our neural network model to train our data:

- Forward Propagation
- Backward Propagation
- Gradient Descent
- Error calculation

3.1 Forward Propagation

Forward propagation involves calculating an output value by feeding an input value through each layer until it reaches the output layer. This will be used to generate predictions while

training our data. Firstly, we calculate the activation of a neuron given our input, which can be done by calculating the dot product between the inputs and weights. The next step is to use an activation function to get our output at each neuron. For our base implementation, we will use the sigmoid function (logistic function), which maps our values to a range between 0 and 1. We apply the sigmoid function to our activations and work our way through all weights between the neurons, continuously calculating the activations, eventually at the output layer, returning our final prediction. We use the following formula to calculate the activation output at each node:

$$a_i = \sigma(b_i + \sum_j w_{ij} \cdot a_j) \tag{4}$$

- a_i is the next neuron activation
- a_j is the current neuron activation
- $\sigma()$ is the sigmoid activation function
- w_{ij} is the weight between activations a_i and a_j
- b_i is the bias

3.2 Backward Propagation

Before implementing back propagation, we first need the error between our outputs from forward propagation, and the expected outputs (ie. our labels), calculated by getting the difference between them. These errors are then back propagated through the network, updating the subsequent weights by figuring out whether they should be higher or lower. Taking a look at the following equations, we can see exactly what we compute and how when backwards propagating:

Computing derivative:

$$\frac{\delta E}{\delta w_{ij}} = \frac{\delta E}{\delta a_i} \cdot \sigma^{'}(a_i) \cdot a_j \tag{5}$$

Computing error on next activation of neuron:

$$\frac{\delta E}{\delta a_j} = \sum_i \frac{\delta E}{\delta a_i} \cdot \sigma^{'}(a_i) \cdot w_{ij} \tag{6}$$

- $\frac{\delta E}{\delta w_{ij}}$ is the derivative

- $\frac{\delta E}{\delta a_j}$ is the error of the previous neuron activation
- $\sigma^{'}()$ is the sigmoid derivative function
- w_{ij} is the weight between activations a_i and a_j
- a_i is the current neuron activation
- a_j is the previous neuron activation

3.3 Gradient Descent

Gradient descent is an algorithm used to find the local minimum of a given function. To achieve this, we use the computed gradient (i.e. the derivatives we computed in back propagation), and multiply it by a value called the learning rate and make a step down the slope i.e opposite to the direction of the gradient:

$$w_{ij} = w_{ij} + \eta \cdot \frac{\delta E}{\delta w_{ij}} \tag{7}$$

- η is the learning rate

The learning rate is a parameter that we manually input to try to optimize the process of gradient descent. The value we choose for η will decide how far we move down the slope. Providing a high learning rate will make a big step. Since we are trying to reach the minimum point, a high learning rate could overshoot this point, and continue to do so without ever getting there. Providing a low learning rate will do the opposite; moving towards the minimum in very small steps. Just like with setting a high learning rate, we may never reach the minimum point If these steps taken towards it are too small. Therefore it is important to set the learning rate correctly so our model converges to the minimum point. Finally, we must also update our bias values as well, which is very similar to the way we update our weights:

$$b_i = b_i + \eta \cdot \frac{\delta E}{\delta a_i} \cdot \sigma^{'}(a_i) \tag{8}$$

3.4 Error Calculation

After forward propagation is complete, we get our output i.e. the predicted y value. Using this value, we want to calculate the error we have between this predicted value and the correct value. There are multiple ways to calculate the error. The first is mean squared

error, which calculates the average of a set of errors. We take the predicted values and the real values and apply the following equation to it:

$$\textbf{MSE} = \frac{1}{n} \cdot \sum_{i=1} (y_i - \hat{y}_i)^2 \tag{9}$$

- n is number of rows we are training in our dataset
- y_i is the actual value for the row (target value)
- $\hat{y}_i$ is the predicted value (the output)

Another type of error that we will be utilizing more often than the others is root mean square error (RMSE). We calculate the error rate using RMSE by simply taking the square root of the mean square error:

$$\textbf{RMSE} = \sqrt{\frac{1}{n} \cdot \sum_{i=1} (y_i - \hat{y}_i)^2} \tag{10}$$

We will be using RMSE to display our errors as the value is of the same unit as our features that we feed to our network, and therefore is simpler to interpret. We will apply this error function at every epoch and get our error rate at each epoch, for both the training set and validation set. We can then use these values to plot graphs representing how the errors change overtime and see what kind of fit we are making to the model, which will be very useful to optimize our algorithm.

3.5 Validation

When training our model we need to have a sample of data to provide an unbiased evaluation of the results from our training model so we can check and adjust our model's hyper parameters i.e a validation set. We can get this validation set by splitting the training data, in our case doing an 80/20 split, just like we did with the test set. For every epoch, our training data will go through all propagation steps, however we will only apply forward propagation to the validation set as all we need is our output to compare to the training data and are not training the validation set. At the end of every epoch, we can retrieve our error for the training data and for the validation data. These can then be compared to understand how our model is fitting the training data. When we have finished tuning our model and comparing different parameters and features, we can then train our final model on both the training and validation sets together before predicting on our test set.

3.6 Network Training

We now have everything required to start training our model. All that is needed is to set our hyper parameters. The output layer will have 1 node, leaving the hidden layers, which will be up to us to decide what works best for our model. The same applies with setting the learning rate. We will set the number of epoch to a very large number (around 10-50 thousand) as our strategy will be to cut off our training when the validation error starts to increase. This will prevent us from over fitting our training data and thus get the optimum validation error.

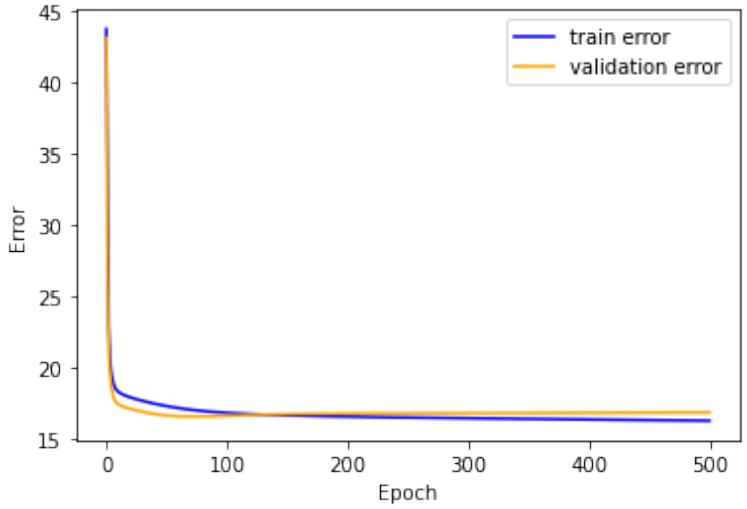


Figure 4: Example of our training model over fitting

Figure 4 compares the validation and training error throughout training. We can see how the validation error starts to increase after around 80 epoch and continues to do so while the training error continues going down. This shows that at this point we have begun over fitting our training set. Therefore we will stop training when this begins to happen to optimize our training.

When training our network, we would like to find the best possible hyper parameters to reduce the error as much as possible. The following tables show many different combinations of parameters with there corresponding root mean squared errors, ranging from 1 to 3 hidden layers.

Tables 1, 2, 3 show various combinations increasing in learning rate and hidden layers. In table 1, where we tested with only 1 hidden layer, we can see how our model starts over fitting relatively quickly as the number of epoch before the validation starts to increase is low. Ontop of this, many validation error results seem to be lower than the training error, which signifies an "unknown" fit. This seems to be the case as we are only training on a small number of epoch. Although the validation error is low, since the training error is higher, these do not seem to be a good fit for our model. Since our goal is to train a model that can predict unknown values, it does not make sense to predict the unknown values better than

what the model is being trained on.

Table 1: RMSE with 1 hidden layer.

Hidden Layers	Learning rate	Epoch	Time (s)	Training RMSE	Validation RMSE
5	0.2	275	12	16.62	16.56
5	0.5	110	4	16.86	16.57
5	0.8	72	3	17.06	16.60
10	0.2	243	10	16.67	16.72
10	0.5	92	4	17.05	16.77
10	0.8	60	2	17.37	16.77
20	0.2	240	10	16.72	17.01
20	0.5	93	4	17.14	16.99
20	0.8	66	3	17.37	17.03

Table 2: RMSE with 2 hidden layers.

Hidden Layers	Learning rate	Epoch	Time (s)	Training RMSE	Validation RMSE
5, 5	0.2	341	18	16.64	16.63
5, 5	0.5	140	7	16.87	16.63
5, 5	0.8	94	5	17.02	16.63
10, 10	0.2	267	14	16.73	16.81
10, 10	0.5	124	6	16.92	16.79
10, 10	0.8	96	5	16.95	16.77
20, 20	0.2	354	19	16.57	16.95
20, 20	0.5	580	32	16.21	16.87
20, 20	0.8	260	14	16.49	16.84

Table 3: RMSE with 3 hidden layers.

Hidden Layers	Learning rate	Epoch	Time (s)	Training RMSE	Validation RMSE
5, 5, 5	0.2	408	27	16.74	16.58
5, 5, 5	0.5	197	13	16.97	16.58
5, 5, 5	0.8	538	35	16.45	16.47
10, 10, 10	0.2	324	21	16.80	16.79
10, 10, 10	0.5	172	11	16.96	16.78
10, 10, 10	0.8	462	30	16.35	16.55
20, 20, 20	0.2	464	31	16.57	16.91
20, 20, 20	0.5	782	54	16.17	16.69
20, 20, 20	0.8	647	44	16.25	16.69

In table 3, although the training time is longer, comparing the training and validation errors, we can see that we get a good fit, where the validation errors are slightly above the training error, with the validation errors being very similar to those with 1 or 2 hidden layers.

Table 4: RMSE with learning rate of 0.05.

Hidden Layers	Learning rate	Epoch	Time (s)	Training RMSE	Validation RMSE
5	0.05	1075	46	16.54	16.57
10	0.05	1016	43	16.50	16.72
20	0.05	1119	48	16.48	17.04
10, 10	0.05	1059	58	16.53	16.82
20, 20	0.05	1268	71	16.42	16.99
20, 20, 20	0.05	1473	101	16.42	16.97

Table 4 shows some of the lowest errors we can get when using a small learning rate. We can see how the training time is significantly longer, as it takes the network alot longer to start over fitting as we are taking much smaller steps. In this case, increasing the number of hidden layers seems to do less good, as the error is higher, even though the number of epochs used for training before over fitting is larger. Unlike with higher learning rates, a lower learning rate fits the model much better when there are less nodes and hidden layers.

In almost all cases, no matter how many hidden layers there are, it seems to be best to always go with a small number of nodes in each layer, as the error tends to increase. Therefore, for our base implementation of the algorithm, we get the optimum results by either using 3 hidden layers (with a small number of nodes) with a high learning rate, or by using 1 hidden layer (with a small number of nodes) and a much lower learning rate.

4 Evaluation

After implementing our base implementation of our algorithm and discovering how to optimize our parameters to get the best model fit, we now are going to try implement further optimizations to our algorithm to see if we can improve our results even further.

Table 5: Best results for the base implementation

Hidden Layers	Learning rate	Epoch	Time (s)	Training RMSE	Validation RMSE
5	0.05	1075	46	16.54	16.57
5, 5, 5	0.8	538	35	16.45	16.47
10, 10, 10	0.8	462	30	16.35	16.55

4.1 Activation Function

In our initial implementation we used the sigmoid activation function. Another activation function we can use in our model is the tanh function. Just like the sigmoid function, tanh is a S shaped curve, however tanh has a range -1 to 1, whereas sigmoid is between 0 and 1, making tanh a stretched and shifted version of sigmoid. We use the following equations for the corresponding activation function:

Sigmoid activation function and its derivative:

$$\sigma(x) = \frac{1}{1 + \epsilon^{-x}} \tag{11}$$

$$\sigma^{'}(x) = \sigma(x) \cdot (1 - \sigma(x)) \tag{12}$$

Tanh activation function and its derivative:

$$\tanh(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{13}$$

$$\tanh^{'}(x) = 1 - \tanh^{2}(x) \tag{14}$$

The main difference between tanh and sigmoid are their gradients. Tanh has a gradient which is 4 times larger than sigmoids gradient, which means using tanh during training will result in greater weight updates, which causes our model to have greater steps towards the minimum.

After implementing tanh and adjusting our learning rate and hidden layer parameters for the tanh activation function we get the following results as some of the best:

Table 6: RMSE with tanh activation function

Hidden Layers	**Learning rate**	**Epoch**	**Time (s)**	**Training RMSE**	**Validation RMSE**
5	0.01	1904	73	16.46	16.47
5, 5	0.1	1398	67	16.66	16.66
10, 10, 10	0.05	698	33	16.24	16.70

We can see that we get very similar error rates to the sigmoid function, however unfortunately tanh does not converge as quick towards this error like the sigmoid function does. Therefore the sigmoid function is a better fit for our model.

4.2 Momentum

The first modification we will make is to implement momentum into our model, specifically within our gradient descent. Momentum is used to increase the rate at which the training model moves towards the minimum, and helps get out of, or pass through any local minimum that your model may come across, decreasing the error further. To implement momentum into gradient descent, we first set a momentum value between 0 and 1 and then update our weights and bias values using the updated gradient descent equation we have already mentioned:

$$w_{ij} = w_{ij} + \eta \cdot \frac{\delta E}{\delta w_{ij}} + m \cdot \Delta w_{ij} \tag{15}$$

$$b_i = b_i + \eta \cdot \frac{\delta E}{\delta a_i} \cdot \sigma^{'}(a_i) + m \cdot \Delta b_i \tag{16}$$

- m is the momentum (value we set between 0 and 1)
- Δw_{ij} is the difference between the updated weight and original weight
- Δb_i is the difference between the updated bias and original bias

Lets apply momentum to our model and take a look at the results.

Table 7: RMSE with a momentum of 0.9

Hidden Layers	Learning rate	Epoch	Time (s)	Training RMSE	Validation RMSE
5	0.05	543	28	16.57	16.57
5, 5, 5	0.8	259	22	16.78	16.37
10, 10, 10	0.8	214	18	16.77	16.47

Table 7 shows that we have made an improvement to our algorithm because we have reached the same or even slightly better error rates than without momentum in table 5. The number of epoch it takes to reach these errors has decreased almost by double and as a result so has the time.

4.3 Simulated Annealing

Simulated annealing is an optimization algorithm which helps our training model reach a global minimum and avoid getting stuck in any local minimum. We utilize simulated annealing in our model by using the following equation:

$$f(x) = p + (q - p) \cdot (1 - \frac{1}{1 + e^{10 - \frac{20x}{r}}}) \tag{17}$$

- p is the end parameter
- q is the start parameter i.e. learning rate we begin with
- r is the number of epochs we are running
- x is the epoch we are currently on

We use this to calculate the new learning rate after every epoch which we then feed to our model's back propagation.

Table 8: RMSE with Simulated Annealing with end parameter 0.01

Hidden Layers	**Learning rate**	**Epoch**	**Time (s)**	**Training RMSE**	**Validation RMSE**
5	0.05	1131	50	16.53	16.57
5, 5, 5	0.8	541	37	16.44	16.47
10, 10, 10	0.8	467	32	16.34	16.55

In Table 8 we can see that after implementing annealing, the errors have decreased by a very small amount. However such an amount is negligable and only caused for extra training time which we can see by comparing these results with table 5. We can deduce that we have already reached a point very close to the global minimum and so annealing is not necessary for our model as the improvements would be too small compared to the time taken to make that extra improvement.

4.4 Final Model

After testing our model with different optimizations, we can conclude that, for our final model, it is best to use the sigmoid activation function with momentum (with a value of 0.9), which maintains a very similar, if not better validation error whilst also reducing the time of training significantly which can be seen from looking at our improved results in table 7.

Since we now have our final optimized model, we can merge our training and validation sets back together and retrain on the entire set. The final step is to now feed forward our test set and calculate predictions for our test set. We will select the following combination to train our combined training and validation set from our 3 best combinations in table 7:

Once we train using these parameters, we can pass our test data and plot the results:

Table 9: Training parameters with momentum of 0.9

Hidden Layers	Learning rate	Epoch
5	0.05	543

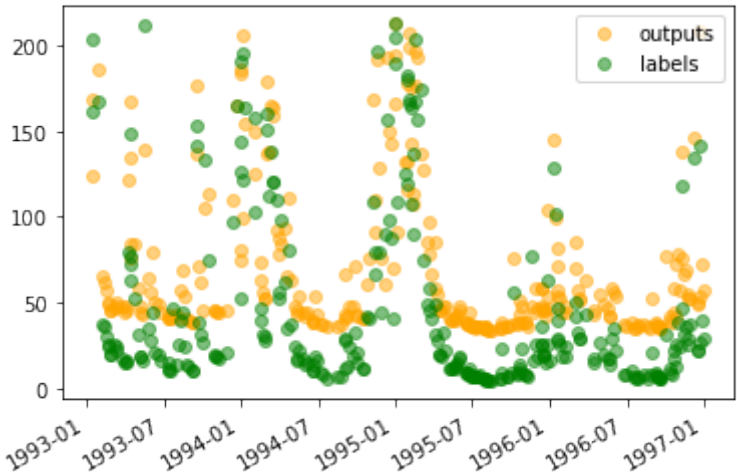


Figure 5: Outputs and Labels comparison

In figure 5 we can see a visual representation of how well we predicted the next day's Skelton mean daily flow (the outputs) against the actual daily flow (the labels). The RMSE of our test set is 17.77. Considering our mean daily flow values can be within a range of 0 to 200, an error of 17.77 indicates that we predict each value with 91% accuracy on average.

5 Model Comparison

5.1 Multivariate Linear Regression

The model we will be comparing our optimized MLP algorithm to is a simple multivariate linear regression model.

We will implement the multivariate linear regression model by making use of the SKLearn library that python provides us with and importing the linear_model specifically from SKLearn. We first fit our training data to the to the model, and then we feed our test set, which gives us the following outputs:

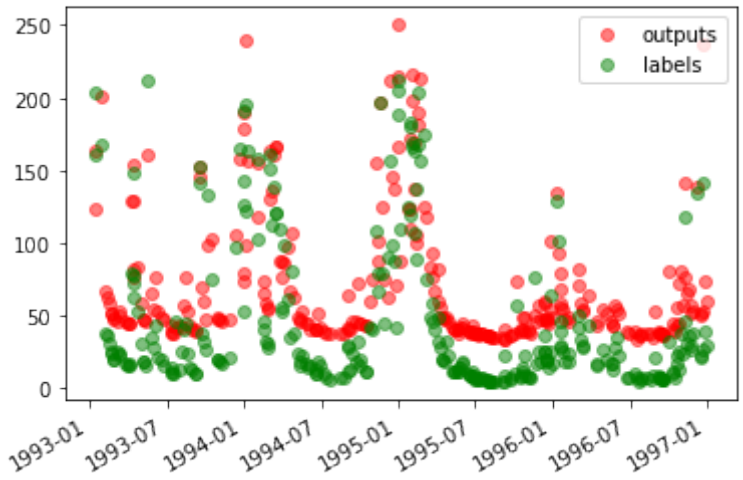


Figure 6: Linear Regression Outputs and Labels comparison

The results from the linear regression model displayed in figure 6 are actually very similar to the outputs that we predicted using our MLP model shown in figure 5. To take a closer look at both outputs and compare both models, we can plot both outputs on the same plot:

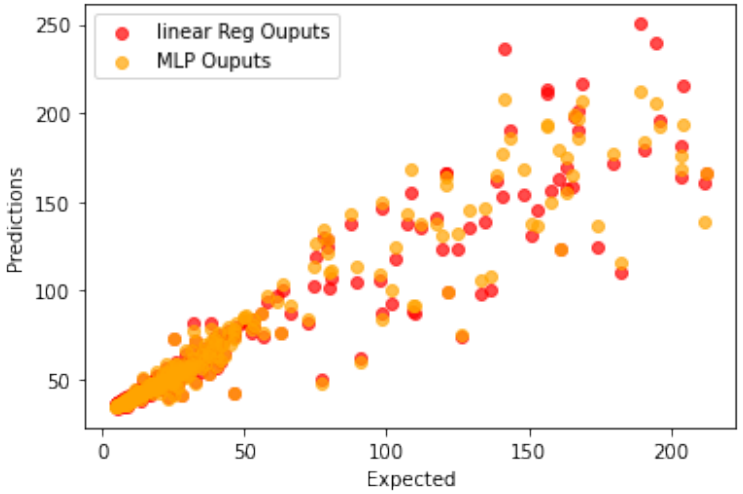


Figure 7: Linear Regression vs MLP model outputs

Figure 7 seems to show us that both the linear regression and MLP models outputs results very similar to eachother, however if we calculate the RMSE of the linear regression outputs, we get a value of 30.60, almost double the error compared to our MLP model.

5.2 Conclusion

After taking many steps to preprocess our data using various methods and techniques, we successfully implemented a multilayer perceptron, taking any number of hidden layers and

nodes, and trained our processed data. By trying many different combinations of learning rates and hidden layers, we found the best parameters that provided us with the lower error rate. Using these parameters we tried different optimization extensions out such as momentum and simulated annealing to try see if we could optimize our model even further, and found that momentum was a good fit for our model in improving the validation error but more importantly reducing our training time by a significant amount as well. Finally, by comparing our optimized model with a multivariate linear regression model, we can see how we managed to produce a better model with an RMSE of 17.8 compared to the RMSE of 30 which it produced, giving us almost double the accuracy.