Predicting River Ouse Flow Using Neural Network

Arkadiusz Brunon Podkowa

March 2022

Contents

List of Figures

List of Tables

Listings

Abstract

Neural networks have been extensively used for many years helping find solutions for problems that are hard to define and not fully understood, including forecasting weather phenomena. This report describes complete process of designing and training neural network for Ouse river flow forecasting at Skelton, England.

1 Introduction

Aim of this report is to find and describe the best performing neural network model for predicting river flow of river Ouse at Skelton, England. The model will be trained on river flow and rainfall data recorded between January 1 1993 and December 31 1996. Figure 1 shows Meany Daily Flow in cumecs of river Ouse at Skelton for that date range.

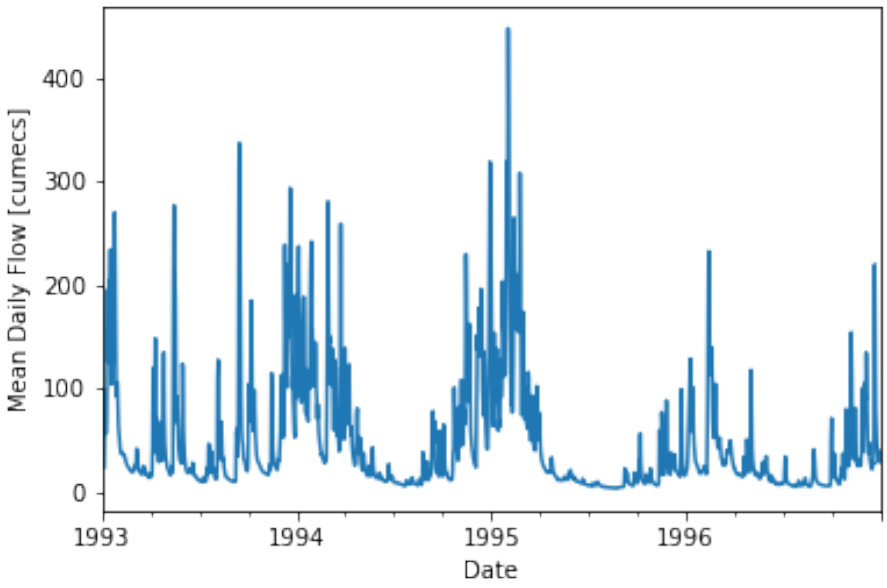

Figure 1: Ouse River flow at Skelton between 1993 and 1996.

For each date, values for mean daily flow in four locations are given —Crakehill, Skip Bridge, Westwick, and Skelton itself. Additionally, values for total daily rainfall for four locations are given —Arkengarthdale, East Cowton, Malham Tarn, Snaizeholme.

Since the given data is labelled, i.e., each data points has the associated observed, supervised learning techniques, with backpropagation being the training algorithm, will be used to construct the prediction model.

2 Data Preprocessing

2.1 Spurious Data

Since the given data is a time series, spurious data have been interpolated using linear interpolation, instead of being deleted. Table 1 shows rows that contain interpolated data (outliers are discussed separately in subsection 2.2). It also points out the reason for each data interpolation.

Row	Erroneous Column	Original Value	New Value	Reason
1993-02-13	Crakehill (Mean Daily Flow)	-999.00	12.65	negative data
1993-03-15	Crakehill (Mean Daily Flow)	-999.00	9.067	negative data
1993-03-16	Crakehill (Mean Daily Flow)	-999.00	9.123	negative data
1993-04-07	Skelton (Mean Daily Flow)	a	78.995	non-numerical data
1993-04-27	Arkengarthdale (Daily Rainfall Total)	-999.0	10.0	negative data
1995-03-01	East Cowton (Daily Rainfall Total)	#	0	non-numerical data
1996-02-09	Skip Bridge (Mean Daily Flow)	a	6.71	non-numerical data
1996-04-18	Skip Bridge (Mean Daily Flow)	-999	2.96	negative data

Table 1: Spurious Values.

2.2 Outliers

Outliers analysis has been divided into two parts: Mean Daily Flow columns and Rainfall columns.

2.2.1 Mean Daily Flow Columns

No outliers have been identified in Mean Daily Flow columns. The value that is the most distant from the mean (448.1 cumecs on February 1 1995) lies 7.2 standard deviations from the mean. However, given similar values in the Winter months (362.3 cumecs on 1995-02-02 or 337.2 cumecs on 1993-09-15) and the rising severity of the weather conditions, this value is considered to be accurate. Figure 2 shows the data for Mean Daily Flow columns.

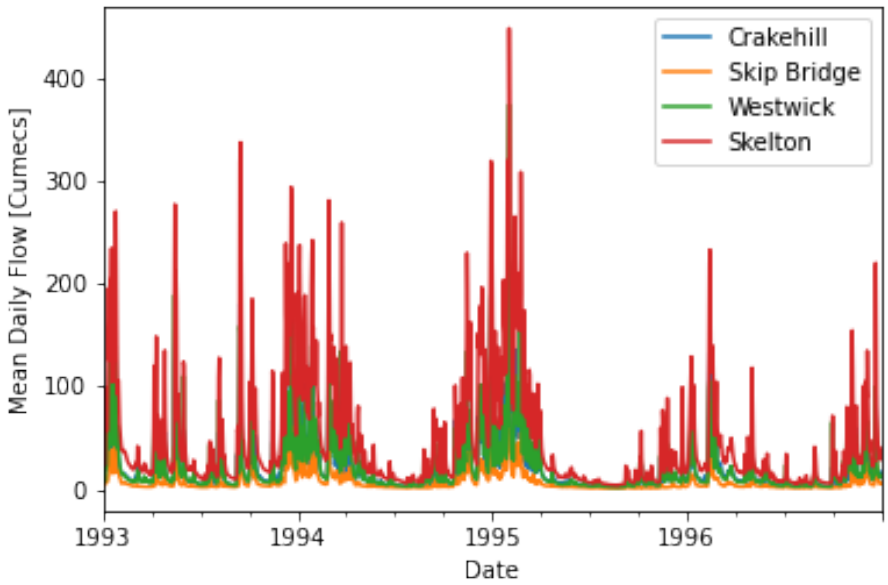

Figure 2: River Flow Data

2.2.2 Rainfall Columns

Three outliers have identified in Rainfall columns. Figure 2 shows data for Rainfall columns.

With the scale of y-axis shown above, it is easy to detect that 3 values lie considerably far from the rest of the values. Table 2 describes outlier values and the values that they have been replaced with. Again, linear interpolated method has been used.

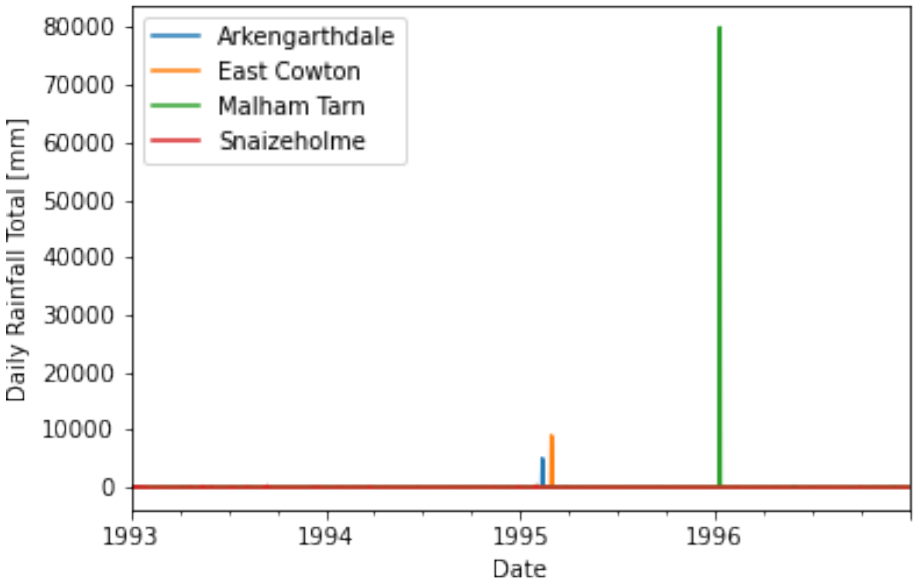

Figure 3: Rainfall Data

Row	Erroneous Column	Outlier Value	Interpolated Value
1995-02-11	Arkengarthdale	5000.0	15.6
1995-02-28	East Cowton	9000.0	0.0
1996-01-10	Malham Tarn	80000.0	4.4

Table 2: Rainfall Data Outliers.

2.3 Data Splitting

Data has been split into three subsets:

- Training set (60% of overall data),
- Validation set (20% of overall data), and
- Test set (20% of overall data).

2.4 Data Standardisation

Data has been standardised using the formula below:

$$S_i = 0.8\frac{R_i - Min}{Max - Min} + 0.1$$

where R_i is a raw value and S_i is the standardised value.

Min and Max values are, respectively, the minimum and maximum values of the training and validation sets combined.

3 Predictors Selection

Because both mean daily flow and daily rainfall total values cannot be obtained before a particular day ends, the predictor values have to be lagged by at least one day. Therefore, every potential predictor's values have been lagged by at least one day.

Initially, different number of lags were explored for all predictors in the data set. Table 3 shows correlation (Pearson correlation coefficient) between a predictor and the predictand for particular lag value (in days). It is clear that lagging columns by one day gives the best results for all potential predictors.

Predictor	T-1	T-2	T-3
Skelton	**0.889**	0.749	0.663
Crakehill	**0.885**	0.724	0.625
Skip Bridge	**0.884**	0.735	0.643
Westwick	**0.912**	0.733	0.627
Arkengarthdale	**0.507**	0.411	0.312
East Cowton	**0.338**	0.257	0.191
Malham Tarn	**0.495**	0.411	0.333
Snaizeholme	**0.584**	0.487	0.389

Table 3: Rainfall Data Outliers.

Subsequently, moving averages were explored. For each potential predictor, a moving average of 2, 3, and 4 days was tested. The figure 4 shows correlations between potential predictors (including different moving averages) and the predictand. It is split into two subplots, one for mean daily flow predictors and the other for rainfall predictors.

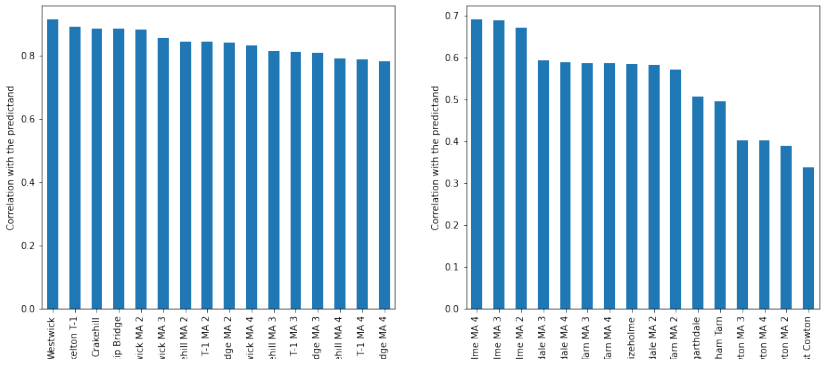

Figure 4: Potential moving average correlations.

Figure 4 clearly shows that mean daily flow columns lagged by one day (labelled without "MA" in the *Potential Flow Predictors* subplot of the figure) have stronger correlations that moving averages of those columns, respectively. On the other hand, moving averages for rainfall columns have stronger correlation with the predictand that only lagged columns, respectively. For every rainfall column, moving averages from the last three days have the strongest correlation with the predictand. Therefore, eight final predictors have been identified for the mean daily flow in Skelton:

- Mean Daily Flow in Skelton on the previous day,
- Mean Daily Flow in Crakehill on the previous day,
- Mean Daily Flow in Skip Bridge on the previous day,

- Mean Daily Flow in Westwick on the previous day,
- Moving average of daily rainfall total in Arkengarthdale from three previous days,
- Moving average of daily rainfall total in East Cowton from three previous days,
- Moving average of daily rainfall total in Malham Tarn from three previous days, and
- Moving average of daily rainfall total in Snaizeholme from three previous days.

Figure 5 shows correlation between a particular predictor (x-axis) and the predictor (y-axis).

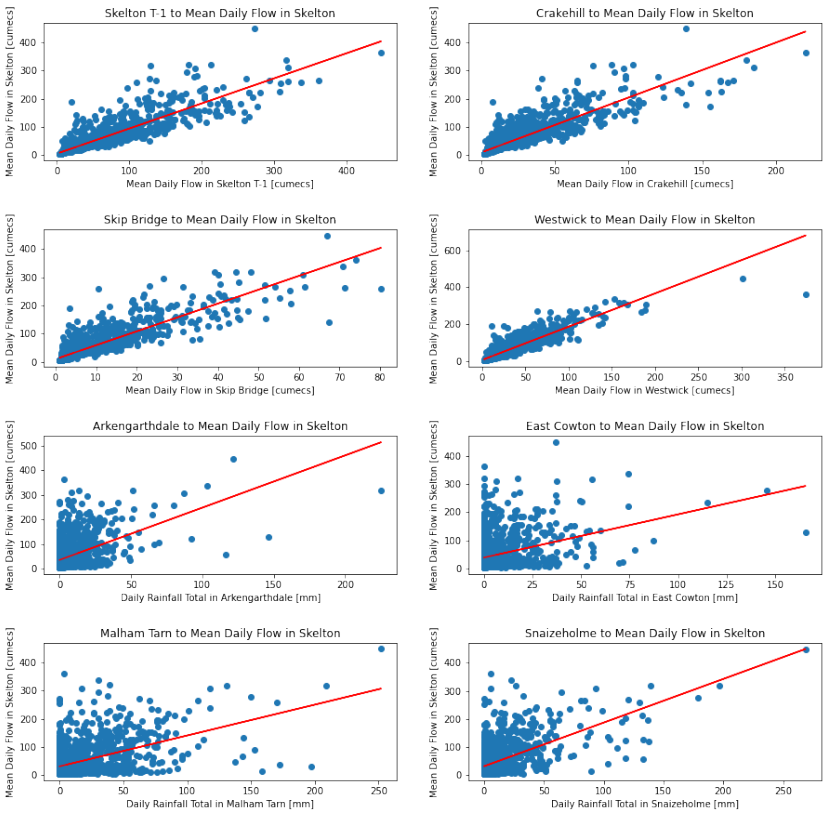

Figure 5: Correlations

4 Implementation

4.1 Language and Library Selection

The neural network has been implemented in Python using number of scientific and numerical libraries. Python has been used because of the wide range of scientific libraries available for this language as well as the ease of use and its popularity, which leads to a considerable online community.

The following is the list of libraries used and its main purpose in this project.

- Numpy: representation of weights and biases & operations on weights and biases
- Pandas: data pre-processing, data standardisation, reading and writing to files, model evaluation
- Matplotlib: plotting
- Sci-kit Learn: data splitting, linear regression

4.2 Design

The neural network has been implemented in Object-Oriented paradigm with a few stand-alone functions. The three main classes are

- `Layer`,
- `NeuralNetwork`, and
- `Backpropagation`

4.2.1 Layer class

`Layer` class represent a layer of a neural network. It implements `__init__`, `forward_pass`, `save_weights_and_biases`, and `restore_weights_and_biases` methods.

`__init__` method serves as a constructor and initialises a `Layer` object. It takes number of inputs, number of neurons, and an activation function as parameters. Passing activation function directly to the `Layer`'s constructor allows each layer to have different activation function, which makes the overall network more flexible. `__init__` initialises weights and biases of all nodes on the layer. Weights are stored in a $m \times n$ matrix (`numpy.ndarray`), where m is the number of inputs to the layer and n is the number of nodes on the layer. The biases are stored in a vector (`numpy.ndarray`) of length n. This representation of weights and biases allows for an arbitrary number of nodes on the layer and makes calculations efficient due to the use of matrix operations, especially the dot product.

The `forward_pass` method performs a forward pass through the layer and computes S_j and u_j for every node j on the layer. The values are stored as instance's attributes instead of being returned. This allows for easy retrieval of values in the later steps (i.e., backward pass).

`save_weights_and_biases` saves current weights and biases to the instance's attributes `saved_weights` and `saved_biases`, respectively.

Conversely, `restore_weights_and_biases` restores weights and biases (assigns to `weights` and `biases`) from `saved_weights` and `saved_biases`, respectively. Together with the `save_weights_and_biases`, these methods provide useful mechanism for restoring weights/biases after the validation error starts to increase, or after adjusting learning rate when using bold driver.

4.2.2 NeuralNetwork class

`NeuralNetwork` class represents the a neural network. It implements `__init__`, `test`, `save_network`, `load_network`, and `predict` methods.

The `__init__` method is a constructor and initialises a `NeuralNetwork` object. It creates `Layer` objects for all hidden layers and the output layer. The layers are stored in a list assigned to object's `list` attribute. The last layer in the list is treated as the output layer. Storing network's layer in a list allows for creating neural networks with an arbitrary number of layers.

The `test` method tests network on the passed test set. It is used for both validating and for the final model testing. It calculates the *modelled* values for each example in the set. Subsequently, it destandardises

those value using the `destandardise` function described in section 4.2.6 and the `max_value` and `min_value` parameters passed to `test`. `test` returns a two-element list with the first element being *observed* values and the second being the *modelled* values.

The `forward_pass` method performs a forward pass through the network calling layers' `forward_pass` methods.

The `save_network` method saves the network to a file in JSON format. It stores weights (including biases) and activation function for each layer, both hidden and output layers. This allows for storing the trained models.

ADD EXAMPLE HERE

Conversely, the `load_network` method reads a network from a JSON file and returns it as a `NeuralNetwork` instance.

The `predict` method returns the predicted value for the given predictors. It simply performs a forward pass through the network calling `forward_pass` methods of the layers.

4.2.3 Backpropagation class

Backpropagation algorithm has been abstracted into a separate class —`Backpropagation`. Hence, it can be broken down into several methods coupled in a single object. `Backpropagation` implements `__init__`, `train`, `forward_pass`, `backward_pass`, `update_weights`, `bold_driver`, and `simulated_annealing`, and `weight_decay` methods.

The `__init__` method is the constructor. It initialises instance's attributes, including the `neural_network` attribute that stores the reference to a `NeuralNetwork` instance which is to be trained using backpropagation.

The `train` method trains a `NeuralNetwork` instance using backpropagation. It performs forward pass, backward pass, and weights/biases update for each training example for the specified number of epochs, or until error on an independent validation set starts to increase (used only for selecting the best performing model). Forward pass, backward pass, and updating weights are implemented as separate method described below. Additionally, `train` accepts number of parameters related to different backpropagation extensions. If an extensions is considered to be set, `train` performs additional operations related to the corresponding extension. If `momentum` is set to a floating point number greater than 0, momentum is used. If `bold_driver` is an integer (not infinity), bold driver is used. If `simulated_annealing` or `weight_decay` is set to `True`, simulated annealing or weight decay are used, respectively. This design allows for seamless switching the extensions on or off, without having to modify the method.

The `forward_pass` method performs forward pass through the network calling `forward_pass` method of network's layers. It passes inputs to the first hidden layer. For all other hidden layers and the output layer, outputs of the previous layer are passed to the layer.

The `backward_pass` method performs backward pass through the network. Unlike `forward_pass`, it does not call corresponding layers' method, as backward pass is specific to the backpropagation algorithm. `backward_pass` reverses the list of network's layers and starts the pass from the output layer. For each layer it calculates the delta values that are later used for weights/biases update. The delta value for a node j is calculated using one of the formulas given below.

$$\delta_j = \begin{cases} (C_j - u_j)\ f'(S_j) & \text{if } j \text{ is an output node and} \\ (\sum_{m:m>j} w_{j,m}\delta_m)\ f'(S_j) & \text{otherwise} \end{cases}$$

If the `weight_decay` flag is set to `True`, the method adds the weight decay term to the error function in δ calculation for the output layer. This is done by calling the `weight_decay` method described below.

The `update_weights` updates weights and biases for each hidden layer and the output layer. It uses `layer.sum` and `output` calculated during the previous forward pass and `layer.delta` calculated during the previous backward pass. Each weight/bias is calculated during the equation below.

$$w^*_{i,j} = w_{i,j} + \rho\delta_j u_i$$

where $w_{i,j}$ is the weight from neuron i to neuron j, $w_{i,j}$ is the updated weight from neuron i to neuron j, ρ is the learning rate, $delta_j$ is the delta value of neuron j, u_i is the value of the activation function of node i. If i is an input node, u_i is simply the input to the network.

If the `momentum` parameter is set to a floating point number greater than zero —α, the `update_weights` method adds the difference between the old and the new weight/bias multiplied by α to each new weight/bias.

The `bold_driver` method adjusts the learning rate based on the change in the training error (MSE with respect to the training set). If the training error increased by 4%, the learning rate is decreased by 30%. If the training error has decreased by 4%. The learning rate is increased by 5%. Additionally, the `bold_driver` method ensures that the learning rate stays in the range 0.01 to 0.5.

The `simulated_annealing` method anneals the learning rate over time (epochs passed). The method accepts start and end values of the learning rate, epoch limit, and number of epochs passed. It calculates and returns the annealed learning rate. If the `simulated_annealing` flag in the `train` method is set to `True`, `simulated_annealing` is called at the beginning of each epoch.

The `weight_decay` method calculates a penalty term for the error function. It accepts number of epochs passed and the learning rate as parameters. It returns the penalty term, which is added to the error function for the hidden layer inside the `backward_pass` method if `weight_decay` parameter to the `train` method is set to `True`.

4.2.4 Activation Functions

Four activation functions have been implemented: sigmoid, tanh, ReLU, and Leaky ReLU. Each of them is represented as class containing five methods:

- `__init__` initialises an activation function instance;
- `func` computes the output of an activation function;
- `der` computes the output of the first derivative of the activation function;
- `vectorised_func` is a vectorised version of the `func` method and computes the output of the activation function for every element of an array;
- `vectorised_der` is a vectorised version of the `der` method and computes the output of the first derivative of the activation function for every element of an array.

Representing activation functions as classes instead of functions allows to bind function and its first order derivative together, which simplifies the overall structure of a program.

4.2.5 Error Measures

Error Measures have been implemented as a stand-alone functions. They accepts two `numpy.ndarray` parameters —`observed` values and `modelled` values.

4.2.6 Miscellaneous Functions

Currently, there is only one additional function implemented —`destandardise`. It destandardises passed value(s) (parameter `x`) using the formula given below.

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

where R_i is the destandardised value and S_i is the standardised value.

Min and Max values are, respectively, the minimum and maximum values of the training and validation sets combined. They are passed as parameters to the `destandardise` function.

5 Model Selection

Model selection has been split into three parts:

- Testing multiple combinations of number of hidden nodes, learning rate, and identifying the best performing ones.
- Testing handful of best performing models and identifying the best performing model
- Testing the best performing model with different backpropagation extensions and identifying extension(s) that improve performance.

5.1 Initial Configuration Training

Let $\mathcal{C} = (n, \rho, a)$ be the network's configuration where $n \in \mathbb{N}$ is the number of hidden nodes, $\rho \in \mathbb{C}$ is the learning rate, and $a \in \{\text{sigmoid}, \tanh, \text{ReLU}, \text{Leaky ReLU}\}$ is the activation function. Different combinations of those three hyperparameters have been tested to obtain the best hyperparameter values.

The network has been tested with a number of hidden neurons ranging from $\frac{2}{n}$ to $2n$, where n is the number of inputs to the network. For the set of 8 inputs these values are 4 and 16, respectively. Since, the range is considered to be inclusive, it gives 13 different number of hidden nodes.

The network has been tested with the learning rate $\rho \in (0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9)$. This gives 9 different possibilities.

The network has been tested with four different activation functions: sigmoid, tanh, ReLU, and Leaky ReLU. This gives 4 different possibilities.

Thus, the total number of possible combinations is 468.

$$13 \times 9 \times 4 = 468$$

The best performing configurations have been selected by running a script testing all 468 configurations of the network. Table 4 below summarises the best configurations for each of the error measures with respect to the validation set. Bold values indicate the best values obtained for particular error measure.

5.2 Error Measures

The subsections below identify 3 the best performing and three the worst performing models with respect to each error measure.

Configuration	MSE	RMSE	MSRE	CE	RSqr
(5, 0.2, sigmoid)	**415.988647**	**20.395800**	0.196227	**0.863**	0.865647
(14, 0.6, LeakyRelu)	722.103296	26.871980	**0.129091**	0.762186	0.860890
(14, 0.2, LeakyRelu)	487.396196	22.077051	0.225364	0.839483	**0.872149**

Table 4: Configurations with the best error measures.

5.2.1 Mean Squared Error

The three best performing configurations use sigmoid and have relatively low learning parameter and few hidden nodes (less than number of predictors). The three worst performing configurations use ReLU and have higher learning parameter.

5.2.2 Root Mean Squared Error

The three best performing configurations use sigmoid and have relatively low learning parameter and few hidden nodes (less than number of predictors). The three worst performing configurations use ReLU and have higher learning parameter.

5.2.3 Mean Squared Root Error

Unlike for two previous error measures, not all the three best performing models use sigmoid. The second best performing model uses Leaky ReLU with considerably high number of hidden nodes and learning parameter. The training of this model took 7.7 seconds compared to > 50 seconds for the other two best performing models.

The three worst performing configurations again use ReLU with high learning parameter.

5.2.4 Coefficient of Efficiency

The three best performing models use sigmoid with relatively low number of hidden nodes and learning parameter equal to 0.2.

The three worst performing model used ReLu activation function. Interestingly for coefficient of efficiency three worst performing configurations used small (the two worst use the minimal) learning parameter. This contrasts to MSE, RMSE, and MSRE.

5.2.5 Coefficient of Determination - R-Sqr

Unlike for all four other error measures, all the three best performing models with regards to the coefficient of determination, used tanh activation function with number of hidden nodes equal to 4 in all three cases. Learning parameters, 0.4, 0.5, and 0.3, respectively, are higher than for best models with respect to the other error measures.

The three worst performing models again use ReLU with relatively wide range of numbers of hidden nodes and learning parameters.

5.2.6 Error Measures - Summary

In summary, the initial training of configurations has shown that sigmoid functions coupled with low learning parameter and number of nodes less than number of predictors, performs best in most of the error measures - four out of five. In particular, configuration $(4, 0.2, \text{sigmoid})$ performs best in 3 of those error measures. It is also worth noting that tanh with 4 hidden nodes outperforms other configurations with respect to the coefficient of determination.

Considering the worst performing models, for all 5 error measures, configurations using ReLU performed worst.

5.3 Training Selected Model Configurations

After testing all 468 combinations, 2 best performing models with respect to each error measure were selected for further training. Because certain models were among two best performing models for more than one error measure, the final number of configurations selected for further training is 6. Since, tanh is not used in any of the six best performing configurations, the best performing configuration using tanh with respect to the MSE (and therefore RMSE) was selected for further training.

The six best performing models described earlier were trained for the maximum of 100,000 epochs. Table 5 below shows their performance. Bold values indicate the best values for each of the performance measures.

Configuration	Epochs	MSE	RMSE	MSRE	CE	RSqr
(4, 0.2, sigmoid)	6690	370.455746	19.247227	0.104959	0.886620	0.886769
(6, 0.2, sigmoid)	7390	**363.344594**	**19.061600**	**0.099198**	**0.888796**	0.888927
(5, 0.2, sigmoid)	5330	369.915055	19.233176	0.099453	0.886785	0.886959
(14, 0.6, leaky ReLU)	70	682.463099	26.123995	0.167948	0.791128	0.892317
(4, 0.4, tanh)	320	469.354988	21.664602	1.130747	0.856351	**0.913161**
(4, 0.5, tanh)	250	570.377495	23.882577	1.465154	0.825432	0.912897

Table 5: The Results of Training the Selected Configurations.

All model configurations stopped due to validation error increase to prevent overfitting. The termination occurred relatively quickly in all cases, as all configurations terminated before passing 1000 epochs.

(6, 0.2, sigmoid) performed best in four out of five performance measures (MSE, RMSE, MSRE, CE). This stands in contrast to the results of initial testing where (4. 0.2, sigmoid) performed best in MSE, RMSE, and CE.

Considering the coefficient of determination, (4, 0.4, tanh) performed best. However, this configuration performed considerably worse to (6, 0.2, sigmoid) in other performance measures.

Interestingly, (14, 0.6, leaky ReLU) terminated after only 70 epochs. It performed worst out of all six configurations in all performance measures except for the coefficient of determination, for which it is the second best activation function.

The figure below compares (6, 0.2, sigmoid) and (4, 0.4, tanh) with the observed values.

Considering all the above, (6, 0.2, sigmoid) is chosen as the best performing model. The 6 shows mean squared error on both training and validation set for this model.

The training of (6, 0.2, sigmoid) terminated after 7390 epochs. Therefore, that number of epochs is being used in the final training on both validation and training set.

5.4 Training Selected Model Using Extended Backpropagation

Trial of the below extensions aimed to accelerate the training process. It did not aim, however, to improve accuracy of the model. The extensions were tested on the configuration identified in the previous section as the best performing configuration - (6, 0.2, sigmoid) running for 7390 epochs.

5.4.1 Momentum

Momentum aims to speed up the learning process by adding weight delta to the new weight values.

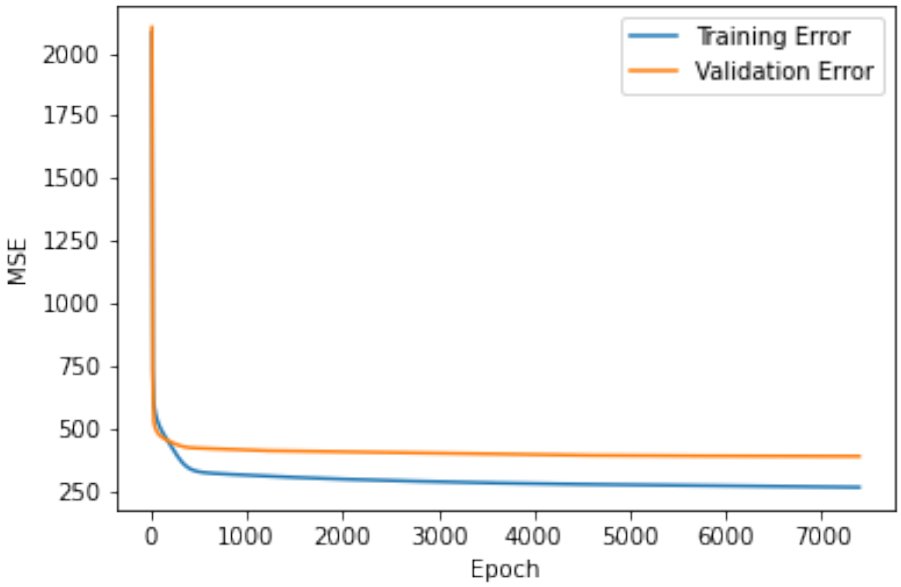

Figure 6: Training and Validation Mean Squared Error for the best performing model.

Training the model with Momentum reduced the time of training from 7390 to 4010 epochs. However, the model performed worse with respect to MSE, RMSE, MSRE, and CE. Coefficient of Determination has slightly improved.

The code listing below shows `update_weights` method with implemented Momentum. Lines 8, 9, 12, 13, 14, and 15 were added to implement this extension.

```
1  def update_weights(self, learning_parameter, inputs):
2          """Updates the layer's weights.
3
4          Args:
5              learning_parameter: Learning parameter of the network.
6              inputs: Inputs to the layer.
7          """
8          previous_weights = self.weights.copy()
9          previous_biases = self.biases.copy()
10         self.weights = self.weights + learning_parameter * np.dot(inputs.T, self.delta)
11         self.biases = self.biases + learning_parameter * self.delta
12         weights_delta = self.weights - previous_weights
13         biases_delta = self.biases - previous_biases
14         self.weights = self.weights + 0.9 * weights_delta
15         self.biases = self.biases + 0.9 * biases_delta
```

Listing 1: update_weights method with Momentum.

5.4.2 Bold Driver

Bold driver adjusts learning rate based on change in training error. The model with bold driver as the only extension, performed worse in all error measures. Moreover, the model with both momentum and bold driver performed worse thant the model using only momentum. Table 6 summarises those values.

However, bold driver coupled with momentum

Extensions	MSE	RMSE	MSRE	CE	RSqr
bold driver	411.993922	20.297633	0.140497	0.864316	0.866376
momentum & bold driver	419.92797	20.492144	0.201175	0.861703	0.867886

Table 6: The Results of Training the Selected Configurations.

5.4.3 Simulated Annealing

Simulated Annealing mimics process of annealing a material. It updates the learning rate every epoch using the following equation.

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

where p is the start value of the learning parameter, q is the start value of the learning parameter, r is the epoch limit, x is the number of epochs elapsed so far.

The code listing below shows `simulated_annealing` implementing Simulated Annealing.

```
def simulated_annealing(
    start_param: float,
    end_param: float,
    epoch_limit: int,
    epochs_passed: int
) -> float:
    """Returns annealed value of the learning parameter.

    Args:
        start_param: Initial learning parameter value.
        end_paramr: Final learning parameter value.
        epoch_limit: Limit of epochs.
        epochs_passed: Number of epochs that have elapsed.

    Returns:
        A float representing annealed learning parameter.
    """
    divisor = 1 + np.e ** (10 - (20 * epochs_passed) /  epoch_limit)
    return end_param + (start_param - end_param) * (1 - (1 / divisor))
```

Listing 2: The Simulated Annealing implementation in Python..

The list below shows change in performance measure after adding Simulated Annealing with respect to a model without any extensions.

- MSE: 0.84% improvement,
- RMSE: 0.42% improvement,
- MSRE: 16.34% **deterioration**,
- CE: 0.11% improvement,
- RSqr: 0.15% improvement.

The improvement is negligible. On the other hand, Mean Squared Root Error deteriorated by over 16%.

The list below shows change in performance measure when using only Simulated Annealing, without implementing Momentum.

- MSE: 0.08% improvement,

- RMSE: 0.04% improvement,
- MSRE: 23.85% improvement,
- CE: 0.01% improvement,
- RSqr: 0.03% improvement.

Simulated Annealing being the only extension improves every performance measure. Moreover, Mean Squared Root Error has improved (decreased) by over 23%. However, it increases number of epochs to 15460.

5.4.4 Weight Decay

Weight Decay aims penalise large weight values by adding an additional term to the error function. Let $\widetilde{E}$ be the new error function with the weight decay term.

$$\widetilde{E} = E + \upsilon\Omega, \text{ where}$$
$$\Omega = \frac{1}{2n}\sum_{i=1}^{n} w_i^2 \text{ and}$$
$$\upsilon = \frac{1}{\rho e}$$

for number of weights and biases $n \in \mathbb{N}$, all weights and biases $w_i \wedge i \leq n \leq n$, and number of epochs passed $e \in \mathbb{N}$.

The listing below shows the `weight_decay` method of the `Backpropagation` class that calculates the weight decay term.

```
def weight_decay(
        self,
        epochs_passed: int,
        learning_rate: float
    ) -> float:
        """Calculates the penalty term for the error function.

        Args:
            epochs_passed: Number of epochs that have elapsed.
            learning_rate: Learning rate value.

        Returns:
            The penalty term for the error function.
        """
        weights_and_biases_sum = 0
        n = 0
        for layer in self.neural_network.layers:
            weights_and_biases_sum += np.sum(
                np.power(
                    np.vstack((layer.weights, layer.biases)),
                    2
                )
            )
            n += layer.weights.shape[0] * layer.weights.shape[1] + layer.biases.shape[1]
        omega = (1 / (2 * n)) * weights_and_biases_sum
        regularisation_parameter = 1 / (learning_rate * epochs_passed)
        return regularisation_parameter * omega
```

Listing 3: weight_decay method of the Backpropagation class.

The figure 7 shows graphs for different error measures for model with different extensions.

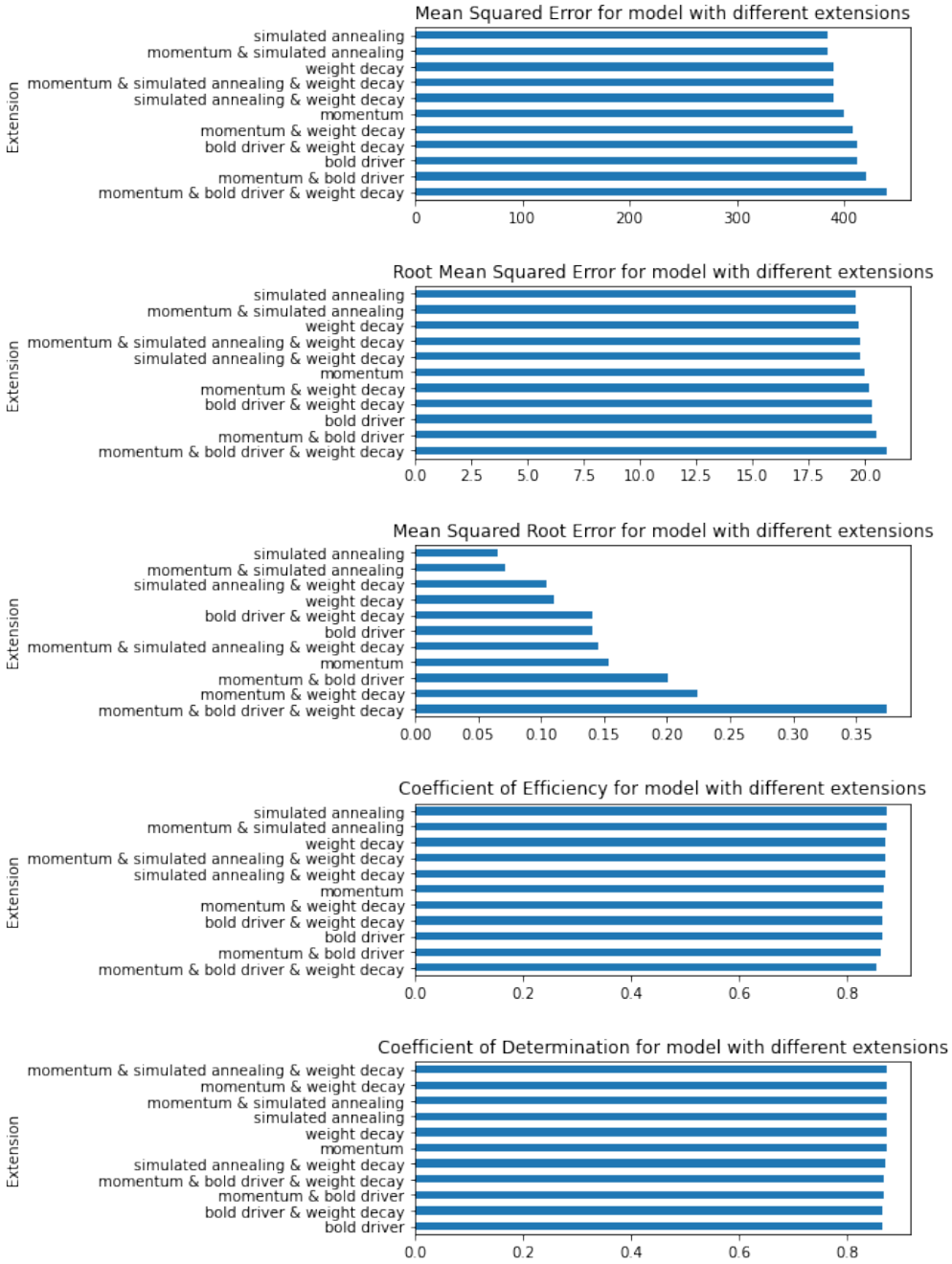

Figure 7: Error measures for model with extensions.

Based on Figure 7, it is clearly visible that the model with simulated annealing performs best in four out of five error measures. Additionally, differences in the values for the measure for which model with simulated annealing does not perform best, are marginal. Therefore, (6, 0.2, sigmoid) with simulated annealing as the only extensions has been chosen to be the final model.

6 Evaluation

As indicated in the last section, (6, 0.2, sigmoid) performed best and was chosen as the final model. Therefore, it was trained on the full training set (training set and validation set) for 7390 epochs. The table 7 shows Root Mean Squared Error, Mean Squared Root Error, Coefficient of Efficiency, and Coefficient of Determination for the final model.

RMSE	MSRE	CE	RSqr
20.795357	0.111751	0.867647	0.879144

Table 7: The final model performance.

Figure 8 shows values predicted by the final model and the observed values.

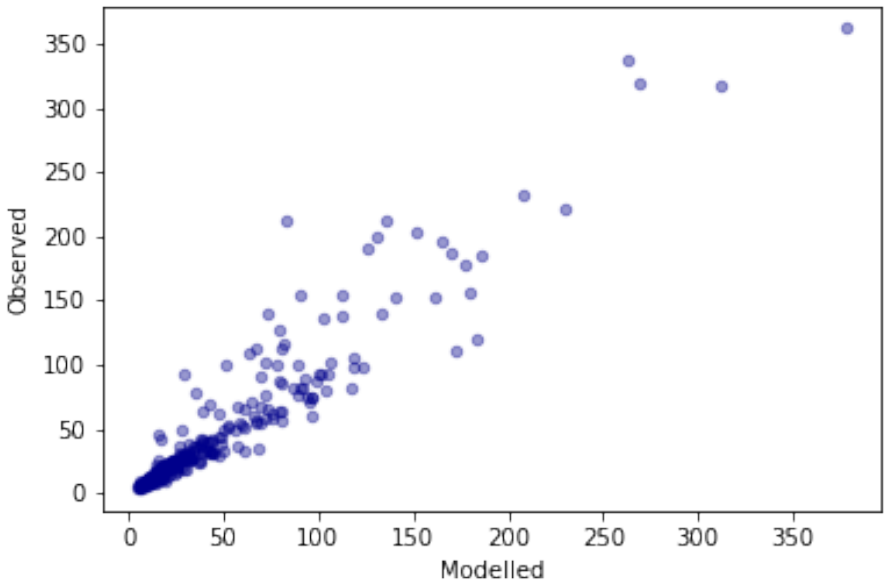

Figure 8: Final Model's Predictions.

7 Baseline Comparison

The final model has been compared with multivariate linear regression.

The equation below shows the coefficients for the linear regression model:

$$y = 0.141342x_1 + 0.058616x_2 + 0.029235x_3 + 0.044469x_4 + 0.010804x_5 - 0.004593x_6 + 0.523017x_7 + 0.294751x_8$$

where

- x_1: Snaizeholme MA
- x_2: Malham Tarn MA
- x_3: East Cowton MA

- x_4: Arkengarthdale MA
- x_5: Skelton T-1
- x_6: Crakehill
- x_7: Skip Bridge
- x_8: Westwick
- y: linear regression prediction for Skelton

The figure 9 shows the modelled values for both the best performing neural network and linear regression model.

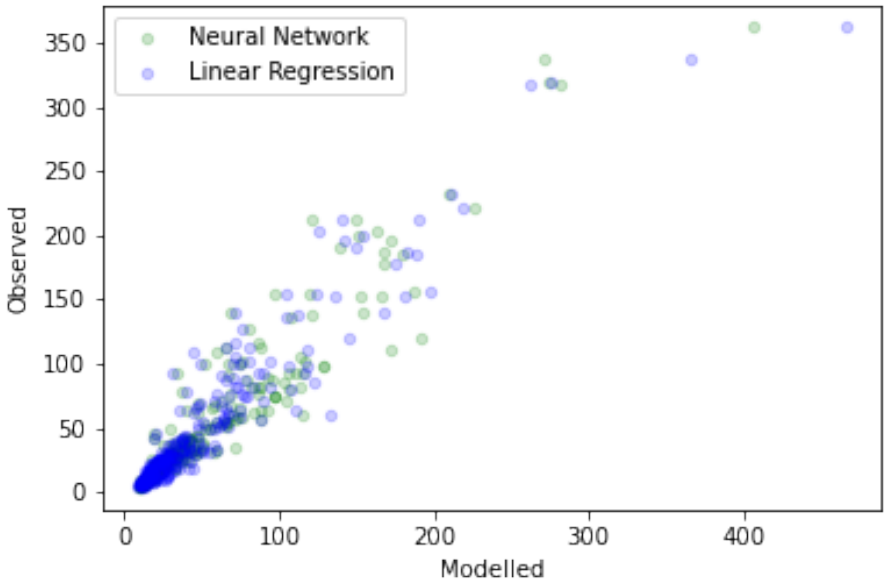

Figure 9: The best performing neural network vs linear regression.

A Code