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 the complete process of designing and training a neural network for Ouse river flow forecasting at Skelton, England.

1 Introduction

This report aims to find and describe the best performing neural network model for predicting the 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.

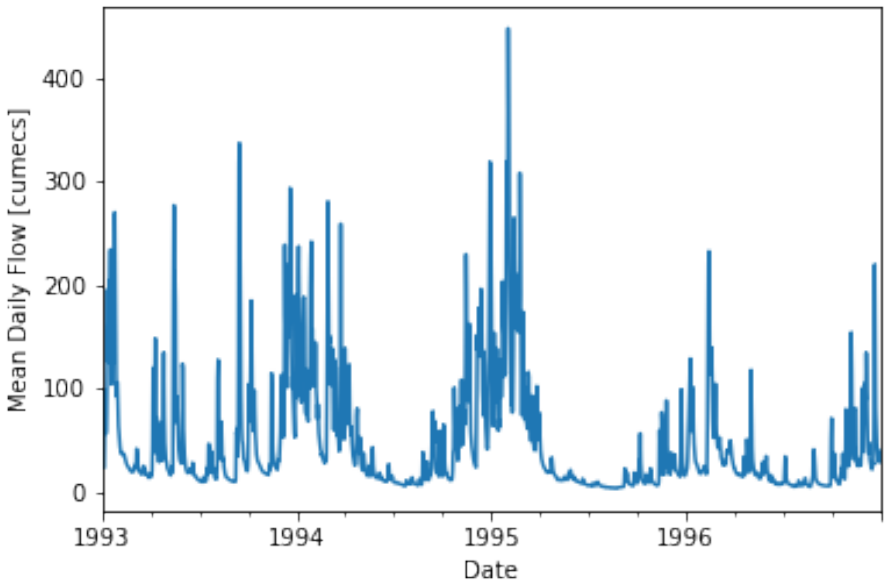

Figure 1: Ouse River flow at Skelton between 1993 and 1996.

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 point 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.

Table 1: Spurious Values.

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

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 the 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 the Mean Daily Flow columns.

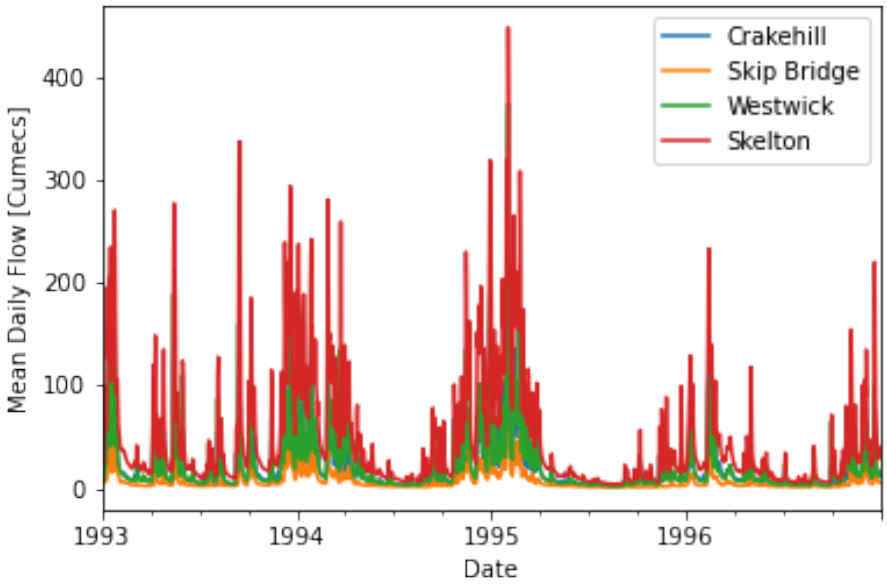

Figure 2: River Flow Data

2.2.2 Rainfall Columns

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

With the scale of the y-axis shown above, it is easy to detect that three 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, the linear interpolation method has been used.

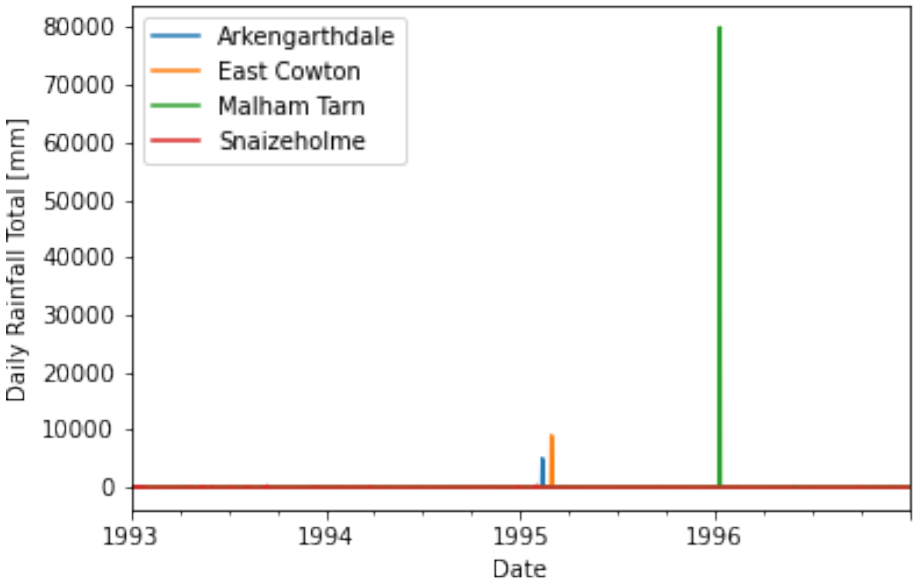

Figure 3: Rainfall Data

Table 2: Rainfall data outliers.

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

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 to the range $[0.1, 0.9]$ 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, all potential predictors have been lagged by at least one day.

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

Table 3: Correlations of lagged columns.

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

Subsequently, moving averages were explored. For each potential predictor, moving averages of 2, 3, and 4 days were tested. 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.

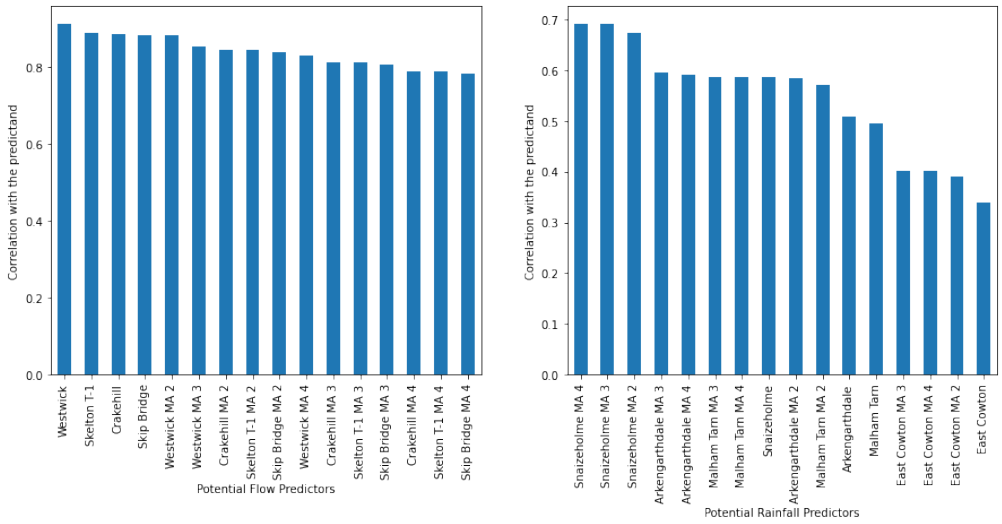

Figure 4: Potential moving average correlations.

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 a 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).

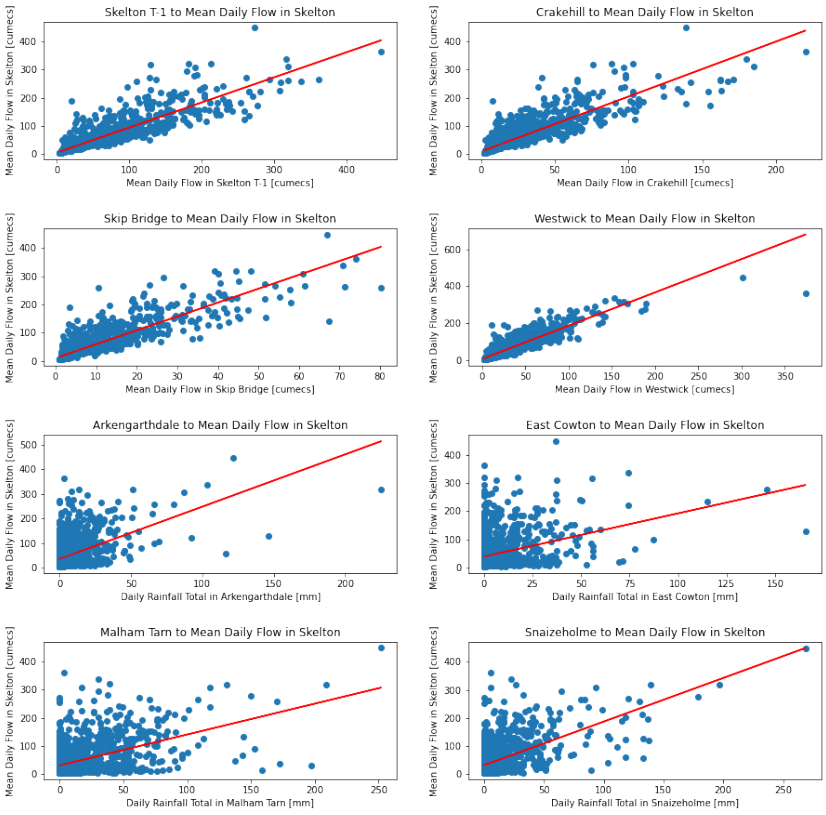

Figure 5: Correlations.

4 Implementation

4.1 Language and Library Selection

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

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

4.2.1 Layer class

The `Layer` class represents 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 the number of inputs, number of neurons, and an activation function as parameters. Passing an activation function directly to the `Layer`'s constructor allows each layer to have a 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 an $m \times n$ 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 a useful mechanism for restoring weights/biases after the validation error starts to increase, or after adjusting the learning rate when using bold driver.

4.2.2 NeuralNetwork class

`NeuralNetwork` class represents 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 the object's `list` attribute. The last layer in the list is treated as the output layer. Storing a network's layer in a list allows for creating neural networks with an arbitrary number of layers.

The `test` method tests the network on the passed test set. It is used for both validating and the final model testing. It calculates the *modelled* values for each example in the set. Subsequently, it destandardises those values using the `destandardise` function described in section 4.2.6, 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. Listing 1 shows an example of a saved network with two inputs and one hidden layer containing three hidden nodes.

```
{
  "layers": [
    {
      "weights": [
        [
          -3.4990896899022697,
          0.301524853238707,
          0.0019533297072200756,
        ],
        [
          -0.9256586183192762,
          -1.0809157995221792,
          -0.9519904948152225,
        ]
      ],
      "biases": [
        [
          -0.9860512995544151,
          0.23064075991603458,
          -0.0163933304460837,
        ]
      ],
      "activation_function": "Sigmoid"
    },
    {
      "weights": [
        [
          3.924996921291405
        ],
        [
          -1.3936761403978688
        ],
        [
          -1.7686185783019668
        ]
      ],
      "biases": [
        [
          0.907088384182177
        ]
      ],
      "activation_function": "Sigmoid"
    }
  ]
}
```

Listing 1: Example network.

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

The backpropagation algorithm has been abstracted into a separate class —`Backpropagation`. Hence, it can be broken down into several methods contained 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 the 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 methods described below. Additionally, `train` accepts a number of parameters related to different backpropagation extensions. If an extension 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 of the extensions on or off, without having to modify the method.

The `forward_pass` method performs a forward pass through the network calling `forward_pass` method of the 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 a backward pass through the network. Unlike `forward_pass`, it does not call corresponding layers' method, as the backward pass is specific to the backpropagation algorithm. `backward_pass` reverses the list of network 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, which stands for the value of α, is set to a floating-point number greater than zero, the difference between the old and the new weight/bias multiplied by α is added to each new weight/bias.

The `bold_driver` method adjusts the learning rate based on the change in the training error (MSE for 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 of 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 the number of epochs passed. It calculates and returns the annealed learning rate obtained using the formula below.

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

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, and $f(x)$ is 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 the 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`. The modified error function $\widetilde{E}$ is calculated using the formula below.

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

where $\upsilon\Omega$ is the weight decay term, $n \in \mathbb{N} \setminus \{0\}$ is the number of all weights and biases in the network, w_i is the ith weight/bias with $1 \leq i \leq n$, $e \in \mathbb{N}$ is the number of epochs passed, and ρ is the learning rate.

4.2.4 Activation Functions

Four activation functions have been implemented: sigmoid, tanh, ReLU, and Leaky ReLU. Each of them is represented as a 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 binding a 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 stand-alone functions. They accept 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 the number of hidden nodes, learning rate, and identifying the best performing ones.
- Testing a 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

Starting in this section and continuing in the remaining sections, a combination of the number of hidden nodes, learning rate, and an activation function will be referred to as a *configuration*. 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 the 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 numbers 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. Models were trained for 500 epochs or until the validation error starts to increase. The models were tested against the validation set. Table 4 below summarises the best configurations for each of the error measures. Bold values indicate the best values obtained for a particular error measure.

Table 4: Configurations with the best error measures.

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**

5.2 Error Measures

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

5.2.1 Mean Squared Error

The three best-performing configurations use sigmoid and have relatively low learning parameters and few hidden nodes (less than the number of predictors). The three worst-performing configurations use ReLU with more hidden nodes and a higher learning rate.

Table 5: The three best and three worst configurations for MSE.

Performance for MSE	Configuration	MSE
The best	(5, 0.2, sigmoid)	415.988647
The second best	(6, 0.2, sigmoid)	420.416983
The third best	(5, 0.3, sigmoid)	420.983047
The third worst	(8, 0.3, ReLU)	12825.760962
The second worst	(7, 0.6, ReLU)	12825.760962
The worst	(14, 0.8, ReLU)	12825.760962

5.2.2 Root Mean Squared Error

Since RMSE is the square root of MSE, the three best and three worst-performing configurations for RMSE are the same as for MSE. Table 6 shows RMSE values for those configurations.

Table 6: The three best and three worst configurations for RMSE.

Performance for RMSE	Configuration	RMSE
The best	(5, 0.2, sigmoid)	20.395800
The second best	(6, 0.2, sigmoid)	20.504072
The third best	(5, 0.3, sigmoid)	20.517871
The third worst	(8, 0.3, ReLU)	113.250876
The second worst	(7, 0.6, ReLU)	113.250876
The worst	(14, 0.8, ReLU)	113.250876

5.2.3 Mean Squared Root Error

Unlike for two previous error measures, not all the three best-performing models use sigmoid. The best-performing model uses Leaky ReLU with a considerably high number of hidden nodes and a high learning rate. The training of this model took 70 epochs compared to the training of the second and third best configurations, which took 500 epochs each. The three worst-performing configurations again use ReLU with a high learning rate.

5.2.4 Coefficient of Efficiency

The three best-performing models for Coefficient of Efficiency are the same as those for MSE and RMSE. They use sigmoid with a relatively low number of hidden nodes and learning rate.

Table 7: The three best and three worst configurations for MSRE.

Performance for MSRE	Configuration	MSRE
The best	(14, 0.6, LeakyRelu)	0.129091
The second best	(5, 0.2, sigmoid)	0.196227
The third best	(5, 0.3, sigmoid)	0.217949
The third worst	(6, 0.5, ReLU)	25.961618
The second worst	(10, 0.9, ReLU)	25.961618
The worst	(11, 0.8, ReLU)	25.961618

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

Table 8: The three best and three worst configurations for CE.

Performance for CE	Configuration	CE
The best	(5, 0.2, sigmoid)	0.863
The second best	(6, 0.2, sigmoid)	0.861542
The third best	(5, 0.3, sigmoid)	0.861355
The third worst	(10, 0.9, ReLU)	-3.223982
The second worst	(11, 0.1, ReLU)	-3.223982
The worst	(4, 0.1, ReLU)	-3.223982

5.2.5 Coefficient of Determination - R-Sqr

Unlike for all four other error measures, no configuration using sigmoid is among the three best configurations for the coefficient of determination. The three best-performing models used either LeakyReLU or ReLU activation function with the number of hidden nodes greater than the number of predictors. On the other hand, the best configurations again use relatively low learning rates —0.2, 0.2, and 0.1, respectively. This observation strengthens the notion that small learning rates perform better.

The three worst-performing models again use ReLU with a relatively wide range of numbers of hidden nodes and learning rates. Since one of the three best performing configurations uses ReLU as well, this observation shows considerable differences in the performance of various configurations using ReLU.

Table 9: The three best and three worst configurations for RSqr.

Performance for RSqr	Configuration	RSqr
The best	(14, 0.2, LeakyReLU)	0.8721486
The second best	(9, 0.2, ReLU)	0.8719305
The third best	(11, 0.1, LeakyReLU)	0.8717774
The third worst	(8, 0.9, ReLU)	3.489269^{-32}
The second worst	(11, 0.1, ReLU)	3.489269^{-32}
The worst	(4, 0.1, ReLU)	3.489269^{-32}

5.2.6 Error Measures - Summary

In summary, the initial training of configurations has shown that sigmoid function coupled with low learning parameter and number of nodes less than the number of predictors performs best in most of the error measures - three out of five. In particular, configuration $(5, 0.2, \text{sigmoid})$ performs best in 3 of those error measures. It is also worth noting that LeakyReLU with 14 hidden nodes outperforms other configurations for the coefficient of determination and mean squared error. Considering the worst-performing models, for all 5 error measures, configurations using ReLU performed worst.

Subsequently, the two best-performing models with respect to each error measure were selected for further

training. Because certain models were among the two best performing models for more than one error measure, the number of configurations selected for further training was five. Additionally, since, tanh is not used in any of those five configurations, the best performing configuration using tanh for the MSE (and therefore RMSE) —(15, 0.1 Tanh) —was selected for further training. This action aims to prevent premature elimination of tanh function from further training, as the epoch limit for the initial training was only 500. Therefore, in total, six configurations were selected for further training. These configurations are:

- (5, 0.2, sigmoid)
- (6, 0.2, sigmoid)
- (14, 0.6, LeakyRelu)
- (14, 0.2, LeakyReLU)
- (9, 0.2, ReLU)
- (15, 0.1 Tanh)

5.3 Training Selected Model Configurations

The six best-performing models described in section 5.2.6 were trained for a maximum of 100,000 epochs. Training could also terminate if MSE on the validation set started to increase. Table 10 below shows their performance. Bold values indicate the best values for each of the performance measures.

Table 10: The results of training the selected configurations.

Configuration	Epochs	MSE	RMSE	MSRE	CE	RSqr
(5, 0.2, sigmoid)	5330	394.003953	19.849533	0.090473	0.870240	0.871538
(6, 0.2, sigmoid)	7390	**387.777161**	**19.692058**	**0.089893**	**0.872291**	**0.873590**
(14, 0.6, LeakyRelu)	70	722.103296	26.871980	0.129091	0.762186	0.860890
(14, 0.2, LeakyReLU)	130	487.396196	22.077051	0.225364	0.839483	0.872149
(9, 0.2, ReLU)	180	534.467440	23.118552	1.011510	0.823981	0.871930
(15, 0.1 Tanh)	1110	448.761269	21.183986	0.511380	0.852207	0.868388

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 10000 epochs.

(6, 0.2, sigmoid) performed best in all five performance measures (MSE, RMSE, MSRE, CE). It stands in contrast to the results of initial testing where (5, 0.2, sigmoid) performed best in MSE, RMSE, and CE. Hence, (6, 0.2, sigmoid) is chosen as the best performing model and tested on different backpropagation extensions. The 6 shows mean squared error on both training and validation set for this model.

5.4 Training Selected Model Using Extended Backpropagation

The trial of the below extensions aimed to accelerate the training process. It did not aim, however, to improve the accuracy of the model. The extensions were tested on the configuration identified in section 5.3 as the best performing configuration —(6, 0.2, sigmoid). Each extension and its implementation are described in section 4.2.3.

5.4.1 Momentum

Momentum aims to speed up the learning process by adding the momentum term to the new values of weights and biases.

Training the model with Momentum reduced the training time from 7390 to 4010 epochs. However, the model performed worse for all five performance measures.

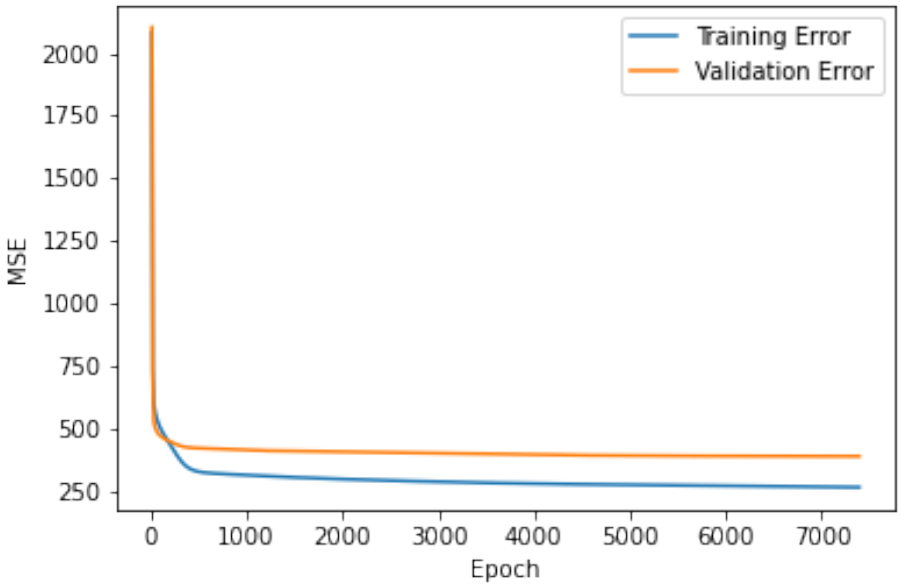

Figure 6: Training and validation Mean Squared Error for the best performing model.

5.4.2 Bold Driver

Bold driver adjusts learning rate based on the 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 than the model using only momentum. Table 11 summarises those values.

Table 11: The results of training the selected configurations.

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

However, bold driver coupled with momentum

5.4.3 Simulated Annealing

Simulated Annealing mimics process of annealing a material. Simulated Annealing significantly increased training time (in epochs). However, the model with this extensions improved in all five performance measures.

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.

5.4.5 Extensions Summary

Figure 7 contains separate bar charts for each error measure showing performance of different extensions.

Based on Figure 7, it is clear 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 —coefficient of determination —are marginal. Therefore, **(6, 0.2, sigmoid) with simulated annealing** has been chosen to be the final model.

6 Evaluation

As indicated in section 5.4.5, (6, 0.2, sigmoid) with simulated annealing performed best and was chosen as the final model. Therefore, it was trained for 15460 epochs and evaluated on the test set. Table 12 shows Mean Squared Error, Root Mean Squared Error, Mean Squared Root Error, Coefficient of Efficiency, and Coefficient of Determination for the final model and the test set.

Table 12: The final model's performance.

MSE	RMSE	MSRE	CE	RSqr
358.058438	18.922432	0.083190	0.890414	0.892726

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

The value of the coefficient of determination (RSqr) is considerably high. It means that the model captured the overall tendency in changes of the values. It is further confirmed by Figure 8, which shows that points tend to lie along the diagonal of the plot.

7 Baseline Comparison

The final model has been compared against multivariate linear regression. The equation below shows the coefficients for the linear regression model:

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

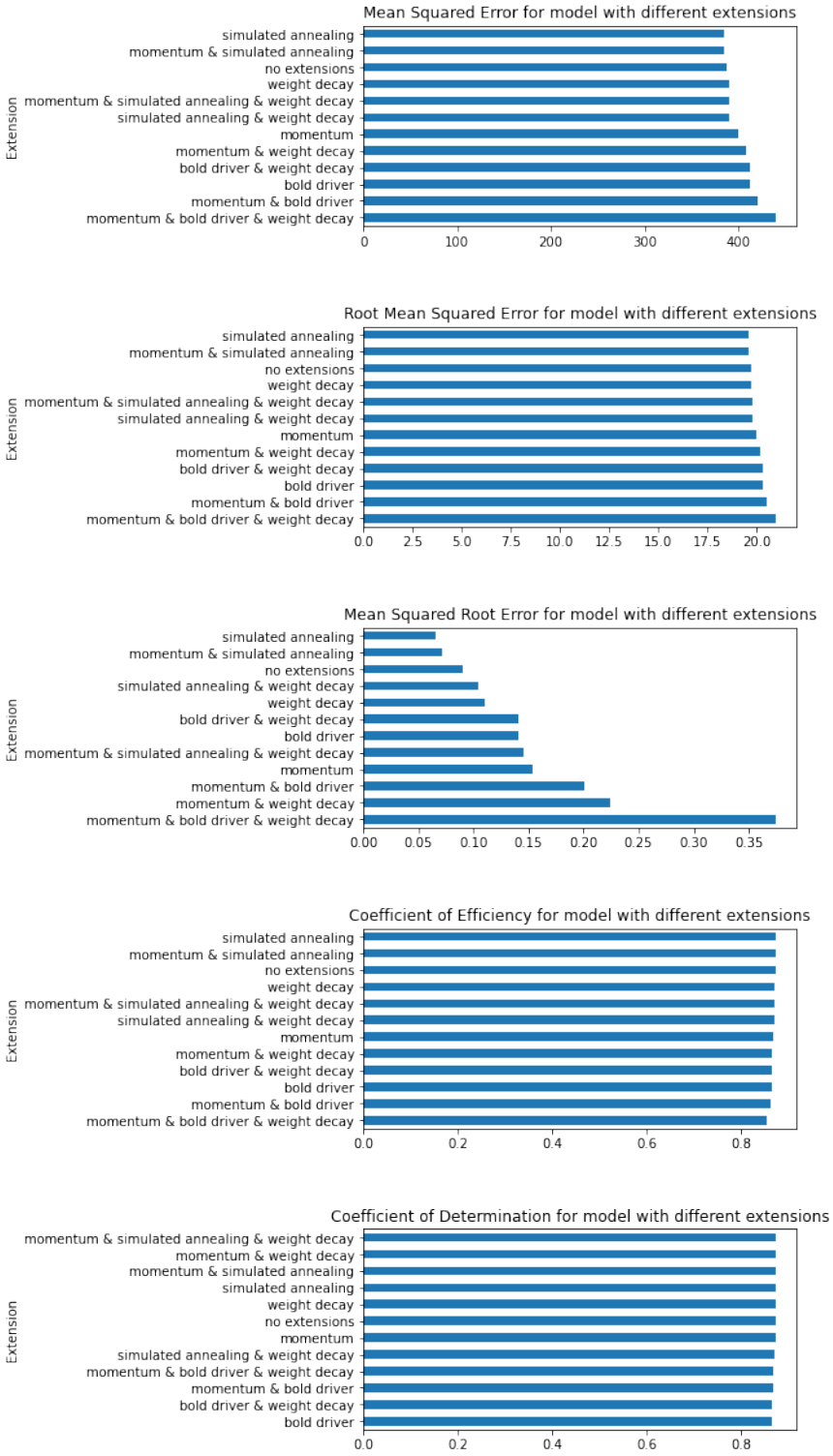

Figure 7: Error measures for model with extensions.

where

- x_1: Snaizeholme MA

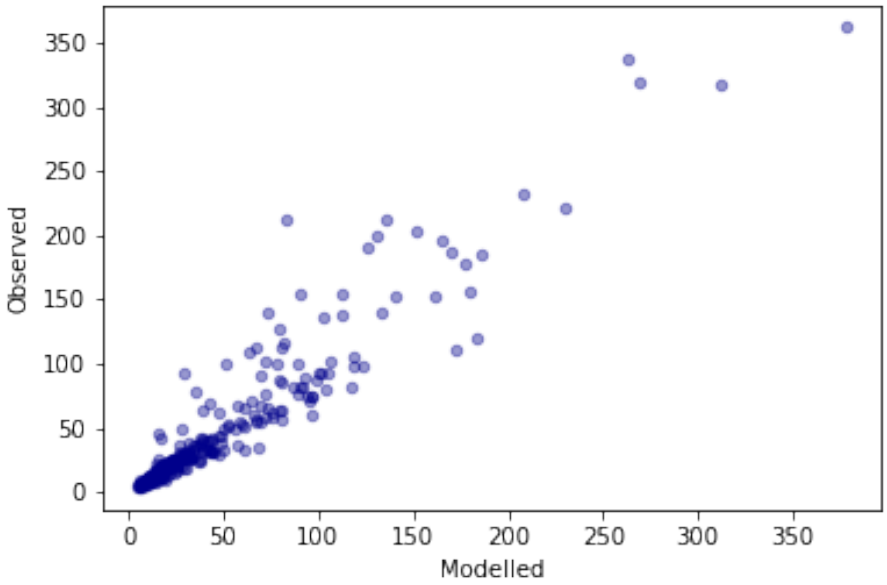

Figure 8: The final model's predictions.

- 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

Table 13 shows the performance of the final model and linear regression and the improvement of the final model over linear regression with respect to each error measure. Note that, in contrast to MSE, RMSE, and MSRE, an increase in CE or RSqr value indicates improvement. The final model's improvement is expressed in percentages.

Table 13: The best model vs linear regression.

Error Measure	**Final Model**	**Linear Regression**	**Model's improvement over LR**
MSE	358.058438	521.789648	31.38%
RMSE	18.922432	22.842715	17.16%
MSRE	0.083190	0.139255	40.26%
CE	0.890414	0.840303	5.96%
RSqr	0.892726	0.847852	5.29%

Figure 13 clearly shows that the final neural network model performs better than multivariate linear regression with respect to all five error measures. Notably, the final model performs better with respect to Mean Squared

Root Error by over 40%. Since MSRE represents error relative to the observed value, it means that the final model gives considerably closer results to the observed values than linear regression.

Figure 9 shows values modelled by both the best performing neural network and linear regression model plotted against the observed values.

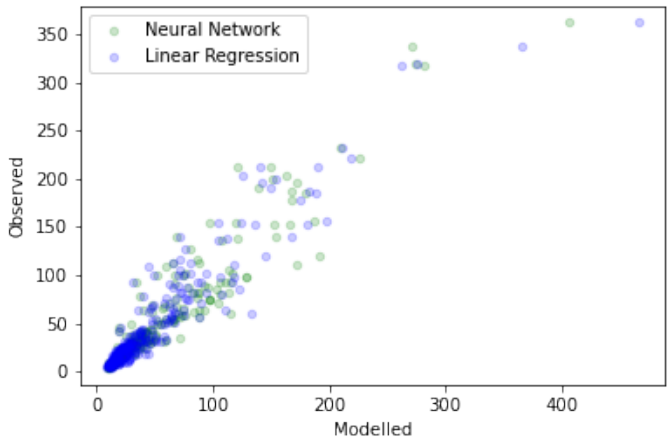

Figure 9: The best performing neural network versus linear regression.