Report

Contents

[Introduction 2](#_Toc161764093)

[Data preprocessing 2](#_Toc161764094)

[Implementation of the MLP algorithm 2](#_Toc161764095)

[Training and network selection 2](#_Toc161764096)

[Evaluation of the final model 2](#_Toc161764097)

[Comparison with data driven model 2](#_Toc161764098)

[Conclusion 2](#_Toc161764099)

[Appendices 2](#_Toc161764100)

# Introduction

For this coursework, I developed a model to predict the Index Flood from eight catchment descriptors using a Multilayer Perceptron (MLP) Artificial Neural Network (ANN), trained by backpropagation.

The Index Flood is a measure for assessing flood risks, and is the median of the annual maximum series of catchment flow in m3/s.

The model’s predictors are Area (AREA), Base Flow Index (BFIHOST), Flood attenuation due to reservoirs and lakes (FARL), Flood plain extent (FPEXT), Lowest Drainage Path (LDP), Proportion of wet days (PROPWET), Median annual maximum 1-day rainfall (RMED-1D), and Standard Annual Average Rainfall (SAAR). The predictand is Index Flood. This data represents information from 597 UK river catchments.

The ANN refines its weights and biases through backpropagation, adjusting based on the interactions between catchment features and the Index Flood. This report covers the MLP ANN's preparation, design, training, and performance evaluation, to accurately predict flood risk.

## Project details

I chose to write this project using Python 3.12 because of its extensive support with mathematical operations such as matrix multiplication and data science libraries. The libraries I used are NumPy, Pandas, Matplotlib, and warnings.

I have implemented my model using an Object-Oriented approach, with a class called NeuralNetwork.

The methods in this class are the constructor \_\_init\_\_, get\_predicted\_results\_length, get\_rmse, forward­\_pass, back\_propogation, sigmoid\_prime, train\_network, test\_network, and plot\_rmse.

The class variables are self.momentum, self.num\_input\_nodes, self.num\_hidden\_nodes, self.num\_output\_nodes, self.rmse, validation\_rmse, self.predicted results, self.data, self.hidden\_nodes\_weights, self.hidden\_layer, self.output\_nodes\_weights, self.prev\_update\_output, and self.prev\_update\_hidden.

## Limitations

There are limitations to every project. In this project, a limitation is that there can only be a single hidden layer. I intend to implement code that allows for multiple hidden layers in the future.

# Data preprocessing

I used a separate python script to preprocess the data. The script takes in the original data file as a csv file. It removes values that are non-numerical or undefined values (e.g. -999). Then it removes anomalous values which are more than three times the standard deviation above or below the mean. Next the data is split into three data sets, training data, validation data, and testing data. The data is then standardised for each data set and then saved in separate csv files, called train.csv validate.csv, and test.csv.

# Implementation of the MLP algorithm

## Overview

A Multilayer Perceptron (MLP) is a type of feedforward artificial neural network, consisting of at least three layers of nodes: an input layer, one or more hidden layers, and an output layer. Each node, or neuron, in one layer connects with a certain weight to every node in the following layer, enabling the network to model complex relationships between input and output variables through these weighted connections.

In my implementation, the neural network is designed to predict outcomes based on input data by learning from the training dataset. The network architecture comprises an input layer sized according to the number of predictors in the dataset (8 here), one or more hidden layers which can be adjusted in number and size to optimize performance (1 hidden layer with between 4 to 16 nodes), and an output layer designed to output a single predictand.

The core learning mechanism of the MLP is the backpropagation algorithm, which iteratively adjusts the weights of the connections in the network to minimize prediction errors. This process involves two main phases: a forward pass and a backward pass. I made these two stages into two separate class methods.

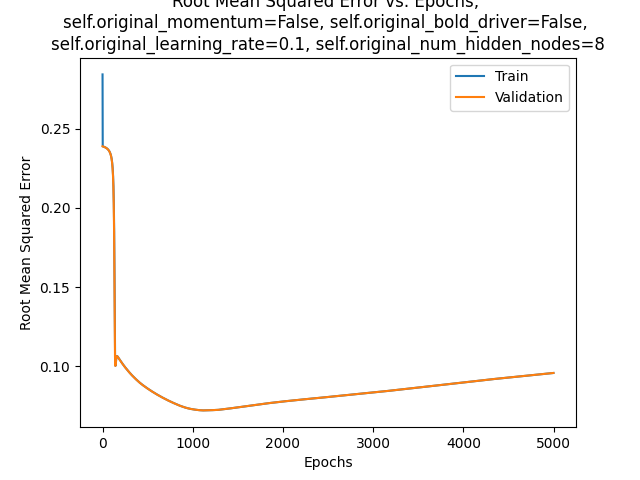
During the forward pass, input data is fed through the network layer by layer. Each neuron in the hidden and output layers sums its weighted inputs and applies an activation function to this sum, which is the sigmoid function in this MLP. Another possible choice of activation function was hyperbolic tangent (tanh). This activation function introduces non-linearity, enabling the network to learn complex patterns. The final layer produces the network's output, which is then compared to the actual target values to calculate the error using the Root Mean Square Error.

The backward pass, or backpropagation, starts with computing the gradient of the error function with respect to each weight. The error then propagates backward through the network to update the weights. This process leverages the derivative of the activation function used in the neurons (the derivative of the sigmoid function) to determine how much each weight contributed to the error.

To improve the learning process and improve convergence, modifications such as momentum and the bold driver method for adaptive learning rate adjustment are introduced. Momentum helps in smoothing out the updates and avoiding local minima by incorporating a fraction of the previous weight update into the current update. The bold driver method dynamically adjusts the learning rate based on the change in error from one iteration to the next, increasing it when the error decreases and decreasing it when the error increases, to fine-tune the learning process.

Through iterations of forward and backward passes, the network learns the optimal set of weights that minimizes the error between predictions and the actual outcomes. This trained model can then be used to make predictions on new, unseen data, offering insights or decisions based on its learned understanding of the relationships within the data.

To help see these differences, I have plotted some graphs.

The following graph shows Root Mean Square Error vs. Epochs

This graph shows how closely the RMSE for the validation data matches the RMSE for the training data. This shows the model generalises well.

The following graph shows the Predicted Results vs Actual Results

A graph with a red line and a line

Description automatically generated

The closer the line of best fit (red line) is to the perfect conditions line (y=x), the more accurate the model can predict the Flood Index.

The following graph shows Root Mean Square Error (Test data) vs. Epochs

A screen shot of a graph

Description automatically generated

This shows how closely the RMSE for the testing data matches the RMSE for the training data. This shows the model generalises well.

## Network architecture

There are three layers in my MLP, an input layer, a single hidden layer, and an output layer. Each layer is composed of nodes, or neurons, which are interconnected through weights that adjust during the training process.

Input Layer: The input layer serves as the entry point for the data into the neural network. The number of nodes in this layer corresponds to the number of features in the input dataset. For this project, the network is initialized to accept 8 inputs or predictors, making the input layer consist of 8 nodes. This layer does not apply any activation function to its inputs; it merely passes the values to the next layer. There is also a bias term added to the start of these 8 predictors with a value of 1.

Hidden Layers: Following the input layer is at least one hidden layer, which is responsible for transforming the inputs into something that the output layer can use. In the provided implementation, the network is configurable to contain a variable number of hidden nodes, set to 4 by default for the initial hidden layer. These hidden layers are where the majority of computation takes place through weighted inputs and activation functions. The activation function used in the hidden layers is the sigmoid function, defined as . This function introduces non-linearity to the network, enabling it to learn complex patterns and relationships in the data.

Output Layer: The final layer of the network is the output layer, which is designed to produce the network's prediction for each input. For this project, the output layer contains a single node, reflecting a single output predictand. The output node also utilizes the sigmoid activation function to ensure the output values fall within a specific range.

As highlighted, the sigmoid activation function is used both in the hidden layers and the output layer. This is because of the sigmoid function's ability to map any real-valued number into a value between 0 and 1, which is particularly useful for binary classification problems and for layers that need to normalize their outputs. The continuous and differentiable nature of the sigmoid function also facilitates the backpropagation process, where gradients of the error function are calculated and propagated backward through the network for weight updates.

## Improvements

Beyond the basic structure, the network allows toggleable options for advanced features such as momentum and a bold driver mechanism for adaptive learning rate adjustments. These enhancements aim to improve the efficiency and convergence of the training process. The momentum term helps in accelerating the gradients vectors in the right direction, leading to faster converging, while the bold driver adjusts the learning rate based on the progression of training to optimise the step sizes taken towards minimizing the error.

### Momentum

Momentum is a technique used to accelerate the convergence of gradient descent algorithms by adding a fraction of the previous weight update to the current update. This approach helps in smoothing out the updates and provides a way to overcome the issue of oscillations and slow convergence in steep gradients.

In my MLP implementation, the momentum term is incorporated into the weight update rule by

0.9 \* self.prev\_update\_hidden

in the line:

self.hidden\_nodes\_weights[h, j] += self.learning\_rate \* hidden\_layer\_delta[h] \* sig\_ws[h] + 0.9 \* self.prev\_update\_hidden

. This is designed to "carry over" a portion of the previous update, thereby imparting inertia to the optimisation process. This modification helps in navigating the error surface more effectively, allowing for faster convergence towards the global minimum by overcoming obstacles small local minima and flat regions.

The inclusion of momentum is justified by its ability to accelerate training, especially in complex landscapes of the error function. It enables the training process to break out of local minima and makes the convergence to the global minimum more efficient.

### Bold Driver Mechanism for Adaptive Learning Rate Adjustment

The bold driver mechanism is an adaptive learning rate technique that alters the learning rate by considering the progress of the training process. This strategy increases the learning rate when the model shows improvement (i.e., the error decreases) and decreases it otherwise.

In our network, the learning rate adjustment is performed as follows:

if self.bold\_driver:  
 if i % 1 == 0 and i != 0:  
 if validation\_RMSE\_array[-1] > validation\_RMSE\_array[-2]:  
 self.learning\_rate \*= 0.7  
 else:  
 self.learning\_rate \*= 1.05  
  
 # if the learning rate is too small, set it to 0.01  
 if self.learning\_rate < 0.001:  
 self.learning\_rate = 0.001  
 elif self.learning\_rate > 0.2: # if the learning rate is too large, set it to 0.15  
 self.learning\_rate = 0.2

The rationale behind the bold driver mechanism is to find a balance between fast convergence and the stability of the training process. By adaptively adjusting the learning rate, the network can exploit phases of rapid improvement while being cautious in situations where the optimization process might lead to an increase in error. This leads to more robust and efficient training, where the choice of learning rate can significantly impact the success of training the model.

## Code

Here I will detail the logic behind key parts of the code.

### Initialisation of parameters

def \_\_init\_\_(self, num\_input\_nodes=8, num\_hidden\_nodes=4, num\_output\_nodes=1, learning\_rate=0.01, momentum=False,  
 bold\_driver=False):  
  
 self.original\_learning\_rate = learning\_rate  
 self.original\_num\_hidden\_nodes = num\_hidden\_nodes  
 self.original\_momentum = momentum  
 self.original\_bold\_driver = bold\_driver  
  
 self.momentum = momentum  
 self.bold\_driver = bold\_driver  
  
 self.learning\_rate = learning\_rate # learning rate  
 self.num\_input\_nodes = num\_input\_nodes  
 self.num\_hidden\_nodes = num\_hidden\_nodes  
 self.num\_output\_nodes = num\_output\_nodes  
  
 self.rmse = list() # list that holds the root mean squared error for each row of data  
 validation\_rmse = list()  
  
 self.predicted\_results = list() # list that holds the final output for each row of data  
 self.data = pd.read\_csv("files/train.csv")  
  
 # create the hidden layer weights and add a bias weight  
 self.hidden\_nodes\_weights = np.random.uniform(-2 / num\_input\_nodes, 2 / num\_input\_nodes,  
 (num\_hidden\_nodes, num\_input\_nodes + 1))  
  
 self.hidden\_layer = list() # create a list to hold the output of the nodes in the hidden layer  
 self.final\_activation = 0  
  
 # create the output layer weights and add a bias weight  
 self.output\_nodes\_weights = np.random.uniform(-2 / num\_hidden\_nodes, 2 / num\_hidden\_nodes,  
 (num\_output\_nodes, num\_hidden\_nodes + 1))  
  
 self.prev\_update\_output = 0  
 self.prev\_update\_hidden = 0

This initializer sets up the neural network by defining its architecture and initializing the weights. Weights are initialized uniformly within a range that inversely scales with the number of nodes, a common practice to ensure initial activations are neither too high nor too low. The momentum and bold\_driver flags activate the respective algorithms during training.

# Training and network selection

# Evaluation of the final model

# Comparison with data driven model

# Conclusion

# Appendices