ANN for Catchment

F410310

2025-03-22

```
import pandas as pd
import numpy as np
import matplotlib.pylab as plt
from scipy.stats.mstats import winsorize
from sklearn.preprocessing import MinMaxScaler
import time
from sklearn.model_selection import train_test_split
import random
```

Introduction & Set Up

I implemented the neural network using Python due to its robust ecosystem for data science and machine learning. Python was chosen for its readability, ease of prototyping, and the availability of libraries that simplify tasks such as data manipulation, numerical computation, and visualization. In this project, I utilized libraries such as NumPy for numerical operations, pandas for handling and reformatting the multi-indexed time series data, matplotlib for generating visualizations, and sci-kit-learn for preprocessing tasks like applying MinMax scaling and splitting the dataset.

Data Pre-processing

During the data pre-processing stage, I examined the dataset's dimensions, which consisted of 12 columns and 1462 rows. Further inspection revealed that the data was a multi-indexed time series. However, when initially imported from an Excel file, the multi-index was not properly configured, the columns were misplaced, the time index was not correctly set up, and the data types for both the columns and the index were not appropriately defined. As such, the first step was to restructure and reformat the dataset, ensuring that the multi-index, time indexing, and data types were properly established to facilitate efficient further analysis. Which can be seen in the following code chunks

```
# Read the Excel file and select the sheet named "1993-96"
df = pd.read_excel("Ouse93-96 - Student.xlsx", sheet_name="1993-96")
```

	Unnamed: 0	Mean Daily Flow - Cumecs	Unnamed: 2	Unnamed: 3	Unnamed: 4	Daily Rainfall Total - mm	Unnamed: 6	Unnamed: 7	Unnamed: 8	Unnamed: 9	Unnamed: 10	Unnamed: 11
0	NaN	Crakehill	Skip Bridge	Westwick	Skelton	Arkengarthdale	East Cowton	Malham Tarn	Snaizeholme	NaN	NaN	You are to predict the mean daily flow at Skelton one day ahead
1	1993-01-01 00:00:00	10.400000	4.393000	9.291000	26.100000	0	0	0	4	NaN	NaN	(Cumecs = m^3/s)
2	1993-01-02 00:00:00	9.950000	4.239000	8.622000	24.860000	0	0	0.800000	0	NaN	NaN	NaN
3	1993-01-03 00:00:00	9.460000	4.124000	8.057000	23.600000	0	0	0.800000	0	NaN	NaN	NaN
4	1993-01-04 00:00:00	9.410000	4.363000	7.925000	23.470000	2.400000	24.800000	0.800000	61.600000	NaN	NaN	NaN

Data Reformatting

```
# Extract the first column (assumed to contain date information) for later use
data_series = df.iloc[:, 0]

# Drop unnecessary columns that are not required for further analysis
df2 = df.drop(columns=["Unnamed: 0", "Unnamed: 9", "Unnamed: 10", "Unnamed: 11"], axis=1)

# Define new column names for the remaining columns.
# These names correspond to the measurements in the dataset.
rename_columns = [
    "Mean Daily Flow - Cumecs", "Mean Daily Flow - Cumecs", "Mean Daily Flow - Cumecs", "Mean Daily Flo
    "Daily Rainfall Total - mm", "Daily Rainfall Total - mm", "Daily Rainfall Total - mm", "Daily Rainfa
]
df2.columns = rename_columns
```

```
# Use the first row to extract station names and create a multi-index for the columns.
# Each column will have a two-level index: the measurement type and the corresponding station.
row2 = df2.iloc[0]
column_tuples = list(zip(rename_columns, row2))
multi_index = pd.MultiIndex.from_tuples(column_tuples, names=["Measure", "Station"])
df2.columns = multi_index

# Insert the 'Date' column back into the DataFrame and remove the first row,
# since it was used to create the multi-index and is no longer needed.
df2.insert(0, "Date", data_series)
df2.drop(index=0, axis=0, inplace=True)

# Create a working copy and convert data types:
# Copy the DataFrame, then convert all columns (except 'Date') to numeric values.
df = df2.copy()
df = df2.iloc[:, 1:].apply(pd.to_numeric, errors='coerce')  # Convert measurement columns to numeric, s
df.insert(0, "Date", data_series)  # Reinsert the Date column as the first column

# Generate LaTeX code for a preview of the table
table_latex = df.head().to_latex(index=True)
```

Measure	Date	Mean Daily Flow - Cumecs						Daily Rainfall Total - mm	
Station		Crakehill	Skip Bridge	Westwick	Skelton	Arkengarthdale	East Cowton	Malham Tarn	Snaizeholme
1	1993-01-01 00:00:00	10.400000	4.393000	9.291000	26.100000	0.000000	0.000000	0.000000	4.000000
2	1993-01-02 00:00:00	9.950000	4.239000	8.622000	24.860000	0.000000	0.000000	0.800000	0.000000
3	1993-01-03 00:00:00	9.460000	4.124000	8.057000	23.600000	0.000000	0.000000	0.800000	0.000000
4	1993-01-04 00:00:00	9.410000	4.363000	7.925000	23.470000	2.400000	24.800000	0.800000	61.600000
5	1993-01-05 00:00:00	26.300000	11.962000	58.704000	60.700000	11.200000	5.600000	33.600000	111.200000

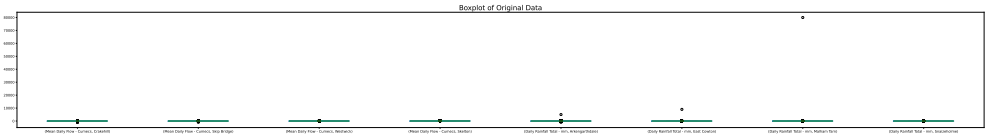
Data Cleaning

After reformatting the data, the next step was to clean it. The first part of the cleaning process involved visualizing the data to understand its distribution, identify any outliers, and determine how best to proceed with further cleaning.

```
## Following are the minimum values:

## Station
## Crakehill     -999.000
## Skip Bridge   -999.000
## Westwick         1.954
## Skelton          3.694
## dtype: float64

## Station
## Arkengarthdale   -999.0
## East Cowton         0.0
## Malham Tarn         0.0
## Snaizeholme         0.0
## dtype: float64

## Following are the Maximum values:

## Station
## Crakehill      220.000
## Skip Bridge     80.244
## Westwick       374.061
## Skelton        448.100
```

```
## dtype: float64
```

```
## Station
## Arkengarthdale     5000.0
## East Cowton        9000.0
## Malham Tarn       80000.0
## Snaizeholme         268.8
## dtype: float64
```



Upon visual inspection and analysis of the minimum and maximum values within the multi-indexed dataset, it became evident that several extreme values were present—both unusually negative and excessively large. For instance, the Mean Daily Flow includes implausible minimum values such as -999.0 at stations like Crakehill and Skip Bridge, while the Daily Rainfall Total records similarly invalid negative readings. On the upper end, extreme maximum values were observed, including rainfall totals exceeding 80,000 mm at Malham Tarn and over 5,000 mm at Arkengarthdale.

Given the context of this dataset—where values represent physical quantities such as Mean Daily Flow (cumecs) and Daily Rainfall Total (mm) such extremes are unrealistic. Negative values are physically impossible, and enormous values suggest data entry errors, sensor malfunctions, or legacy anomalies.

An initial consideration was to remove all negative values and apply outlier filtering. However, this approach would compromise the time series continuity required for downstream forecasting tasks—mainly when predicting future values such as Skelton's Mean Daily Flow. Removing data points could lead to inconsistent timestamps and weaken the model's ability to learn from temporal patterns.

To address this, we decided to apply winsorization to the dataset. Winsorization limits extreme values to a defined percentile range (in this case, the 1st and 99th percentiles), allowing us to retain all timestamps while minimizing the influence of outliers. This process preserves the structural goodness of the time series and ensures the model trains on realistic and temporally consistent data.

```
# Apply winsorization to all columns in the dataframe to cap extreme values
# The the top and bottom 1% while preserving the structure of the data
df_winsorized = df.apply(lambda df: winsorize(df, limits=[0.01, 0.01]))

# Create a complete date range from the earliest to the latest date in the dataset
date_range = pd.date_range(start=df_winsorized["Date"].min(), end=df_winsorized["Date"].max(), freq="D")

# Check how many dates are missing from the dataset compared to the complete date range
missing_dates = len(date_range.difference(df_winsorized["Date"]))
print("Missing dates:", missing_dates)
```

```
## Missing dates: 0
```

```
## Following are the minimum values of Winsorized Dataframe:
```

```
## Mean Daily Flow - Cumecs: Station
## Crakehill      2.340
## Skip Bridge    1.058
## Westwick       2.185
## Skelton        4.259
## dtype: float64
```

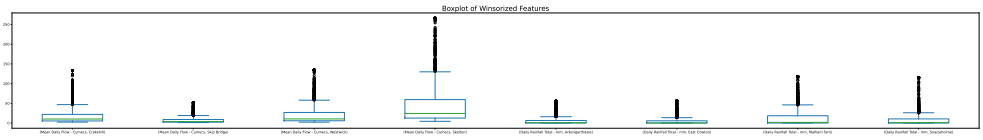
```
## Daily Rainfall Total - mm: Station
## Arkengarthdale    0.0
```

```
## East Cowton       0.0
## Malham Tarn       0.0
## Snaizeholme       0.0
## dtype: float64
```

```
## Following are the maximum values of Winsorized Dataframe:
```

```
## Mean Daily Flow - Cumecs: Station
## Crakehill      133.000
## Skip Bridge     51.522
## Westwick       134.357
## Skelton        266.000
## dtype: float64
```

```
## Daily Rainfall Total - mm: Station
## Arkengarthdale     55.2
## East Cowton        56.0
## Malham Tarn       117.6
## Snaizeholme       115.2
## dtype: float64
```



With the data cleaned and winsorized to reduce the influence of extreme values, the next step was to explore the dataset for potential predictive relationships. The first approach involved visualizing the time series data to assess whether there were any apparent patterns or correlations between variables over time.

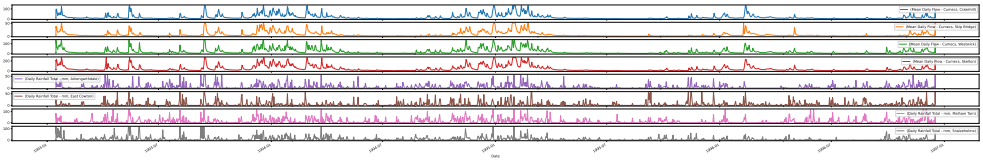
The time series plot below presents each feature as a separate subplot, sharing the same timeline. This visualization provides an intuitive overview of the data's temporal behavior. It helps reveal whether certain variables may move in tandem, which is an important first step in identifying strong predictors.

In addition, a summary statistics table was generated to complement the visual analysis. This table includes key descriptive metrics such as the mean, standard deviation, and range for each feature, offering a clearer understanding of the overall scale and variability in the dataset.

```
# Plot all time series columns as individual subplots, sharing the same x-axis (date)
# This helps visually compare trends and detect potential relationships across variables
df_time.plot(figsize=(50, 8), subplots=True, sharex=True)
```

```
## array([<Axes: xlabel='Date'>, <Axes: xlabel='Date'>,
##        <Axes: xlabel='Date'>, <Axes: xlabel='Date'>,
##        <Axes: xlabel='Date'>, <Axes: xlabel='Date'>,
##        <Axes: xlabel='Date'>, <Axes: xlabel='Date'>], dtype=object)
```

```
# Generate summary statistics (count, mean, std, min, quartiles, max)
# for each variable
# Useful for understanding the distribution and scale of each time series feature
summary_stats = df_time.describe()
```



Measure			Mean Daily Flow - Cumecs				Daily Rainfall Total - mm	
Station	Crakehill	Skip Bridge	Westwick	Skelton	Arkengarthdale	East Cowton	Malham Tarn	Snaizeholme
count	1461.000000	1461.000000	1461.000000	1461.000000	1461.000000	1461.000000	1461.000000	1461.000000
mean	19.293854	7.535781	21.120564	46.795561	5.422587	5.114305	13.805065	9.419302
std	23.691184	9.482550	25.092295	53.809160	10.116465	10.498793	22.936933	18.716614
min	2.340000	1.058000	2.185000	4.259000	0.000000	0.000000	0.000000	0.000000
255075max	133.000000	51.522000	134.357000	266.000000	55.200000	56.000000	117.600000	115.200000

While the summary statistics provided a general understanding of the data's distribution, they offered limited insight into the relationships between variables. To address the limited insight, a correlation matrix was generated to evaluate how strongly the features are related, particularly regarding their potential predictive value for forecasting Skelton's Mean Daily Flow.

```
corr_time = df_time.corr()
```

	Measure			Mean Daily Flow - Cumecs			Daily Rainfall Total - mm	
Measure	Station	Crakehill	Skip Bridge	Westwick	Skelton	Arkengarthdale	East Cowton	Malham Tarn	Snaizeholme
Mean Daily Flow - Cumecs	Crakehill	1.000000	0.922585	0.919979	0.968680	0.354226	0.234758	0.323258	0.379545
	Skip Bridge	0.922585	1.000000	0.872555	0.935077	0.367027	0.256911	0.331211	0.364642
	Westwick	0.919979	0.872555	1.000000	0.908536	0.493527	0.290247	0.484557	0.544400
	Skelton	0.968680	0.935077	0.908536	1.000000	0.323536	0.183269	0.307350	0.356564
Daily Rainfall Total - mm	Arkengarthdale	0.354226	0.367027	0.493527	0.323536	1.000000	0.640906	0.604992	0.611848
	East Cowton	0.234758	0.256911	0.290247	0.183269	0.640906	1.000000	0.451241	0.395958
	Malham Tarn	0.323258	0.331211	0.484557	0.307350	0.604992	0.451241	1.000000	0.674782
	Snaizeholme	0.379545	0.364642	0.544400	0.356564	0.611848	0.395958	0.674782	1.000000

The correlation matrix shows that the flow measurements at stations like Crakehill, Skip Bridge, and Westwick are all highly correlated with Skelton's flow (with correlation coefficients of 0.97, 0.94, and 0.91, respectively). This suggests that these stations are likely to be valuable predictors.

In contrast, while still somewhat correlated, rainfall measurements show weaker relationships. Notably, 'Daily Rainfall Total - mm' at East Cowton had the lowest correlation with Skelton's flow (0.18), indicating minimal predictive value. As a result, this feature was removed from the dataset to reduce dimensionsa and potential noise, allowing the model to focus on more relevant signals.

```
df_time2 = df_time.drop(columns=[('Daily Rainfall Total - mm', 'East Cowton')])
```

Measure		Mean Daily Flow - Cumecs				Daily Rainfall Total - mm	
Station	Crakehill	Skip Bridge	Westwick	Skelton	Arkengarthdale	Malham Tarn	Snaizeholme
Date							
1993-01-15 00:00:00	10.400000	4.393000	9.291000	26.100000	0.000000	0.000000	4.000000
1993-01-15 00:00:00	9.950000	4.239000	8.622000	24.860000	0.000000	0.800000	0.000000
1993-01-15 00:00:00	9.460000	4.124000	8.057000	23.600000	0.000000	0.800000	0.000000
1993-01-15 00:00:00	9.410000	4.363000	7.925000	23.470000	2.400000	0.800000	61.600000
1993-01-15 00:00:00	26.300000	11.962000	58.704000	60.700000	11.200000	33.600000	111.200000

With the data cleaned and redundant features removed, the final step was to address the issue of inconsistent units within the dataset. As observed in the multi-indexed structure, the dataset contains two types of measurements: 'Cumecs' for flow rates and 'mm' for rainfall totals.

This discrepancy poses a challenge when incorporating both measurements into a unified model, mainly when predicting Skelton's Mean Daily Flow. To ensure all features contribute proportionally and to avoid scale dominance by any variable, min-max normalization was applied.

Min-max scaling transforms all values to a standard range—typically between 0 and 1—which is appropriate since the dataset contains only non-negative values. This transformation preserves the relationships between

values while aligning them consistently, allowing the model to interpret the features more effectively.

```
target_col = ('Mean Daily Flow - Cumecs', 'Skelton')
feature_cols = df_time2.columns.difference([target_col])

# Fit a scaler for the input features only
X_scaler = MinMaxScaler(feature_range=(0, 1))
X_scaled = X_scaler.fit_transform(df_time2[feature_cols])
df_X_scaled = pd.DataFrame(X_scaled, columns=feature_cols, index=df_time2.index)

# Fit a scaler for the target variable only
y_scaler = MinMaxScaler(feature_range=(0, 1))
y_scaled = y_scaler.fit_transform(df_time2[[target_col]])
df_y_scaled = pd.DataFrame(y_scaled, columns=[target_col], index=df_time2.index)

# Now you can construct your training data:
X = df_X_scaled.values
y = df_y_scaled.values

y = y.reshape(-1, 1)
```

Measure			Mean Daily Flow - Cumecs			Daily Rainfall Total - mm	
Station	Crakehill	Skip Bridge	Westwick	Skelton	Arkengarthdale	Malham Tarn	Snaizeholme
Date							
1993-01-15 00:00:00	0.061687	0.066087	0.053763	0.083445	0.000000	0.000000	0.034722
1993-01-15 00:00:00	0.058243	0.063035	0.048702	0.078708	0.000000	0.006803	0.000000
1993-01-15 00:00:00	0.054493	0.060756	0.044427	0.073894	0.000000	0.006803	0.000000
1993-01-15 00:00:00	0.054110	0.065492	0.043428	0.073397	0.043478	0.006803	0.534722
1993-01-15 00:00:00	0.183377	0.216075	0.427617	0.215637	0.202899	0.285714	0.965278

```
## Following are the minimum values of Nomalized Data:

## Mean Daily Flow - Cumecs: Station
## Crakehill      0.0
## Skip Bridge    0.0
## Westwick       0.0
## Skelton        0.0
## dtype: float64

## Daily Rainfall Total - mm: Station
## Arkengarthdale    0.0
## Malham Tarn       0.0
## Snaizeholme       0.0
## dtype: float64

## Following are the maximum values of Normalized Data:

## Mean Daily Flow - Cumecs: Station
## Crakehill      1.0
## Skip Bridge    1.0
## Westwick       1.0
## Skelton        1.0
## dtype: float64

## Daily Rainfall Total - mm: Station
## Arkengarthdale    1.0
## Malham Tarn       1.0
## Snaizeholme       1.0
```

```
## dtype: float64
```

Implementation of MLP Program

With the data cleaned and redundant features removed, the next step was to build a neural network to model and predict Skelton's Mean Daily Flow. The network was implemented in Python using NumPy, providing complete control over the architecture and training process. The implementation centers around a NeuralNetwork class that follows the backpropagation algorithm supports multiple hidden layers, customizable activation functions (including sigmoid, ReLU, and tanh), and uses Xavier initialization for stable weight scaling. The class includes methods for forward propagation, which computes layer-wise activations; backward propagation, which calculates gradients and updates the weights using stochastic gradient descent; and training, where the model iteratively improves over several epochs using mini-batch updates. Additionally, it includes a prediction method that feeds new input through the trained network to generate outputs. Throughout training, the model tracks and visualizes the loss over time, allowing for intuitive convergence monitoring.

In addition to enhance the performance and stability of the neural network during training, an additional Optimizer class was implemented to incorporate several standard optimization techniques. This class builds on the MLP structure and supports four key improvements: momentum, weight decay, learning rate annealing, and the Bold Driver heuristic. Momentum helps smooth out updates by incorporating a fraction of the previous gradient, accelerating convergence, and reducing oscillations. Weight decay introduces regularization, encouraging smaller weights and helping to prevent overfitting. Learning rate annealing allows the step size to decrease gradually over epochs, supporting finer adjustments as training progresses. The Bold Driver mechanism dynamically adjusts the learning rate based on the direction of the loss: if the loss worsens, the learning rate is reduced; if the loss improves, it is slightly increased. This adaptive behavior enables more responsive learning without requiring manual tuning during training. Together, these enhancements aim to make the optimization process more robust and efficient, mainly when training deeper networks or working with noisy data.

```
class NeuralNetwork:

    def __init__(self, input_size, hidden_sizes, output_size, activation="sigmoid", learning_rate=0.01,
                 momentum=0.0, weight_decay=0.0, annealing_rate=1.0, bold_driver_enabled=False,
                 increase_factor=1.01, decrease_factor=0.99):
        self.input_size = input_size
        self.hidden_sizes = hidden_sizes
        self.output_size = output_size
        self.learning_rate = learning_rate
        self.activation_name = activation


        # Define activation functions
        self.activation = self.get_activation_function(activation)
        self.activation_deriv = self.get_activation_derivative(activation)

        # Define layers
        self.layers = [input_size] + hidden_sizes + [output_size]

        # Weight initialization (Xavier Initialization)
        self.weights = [np.random.randn(self.layers[i], self.layers[i+1]) * np.sqrt(1 / self.layers[i])
                        for i in range(len(self.layers) - 1)]

        # Initialize the optimizer with the provided parameters
        self.optimizer = Optimizer(self, learning_rate, momentum, weight_decay, annealing_rate,
```

```
                                    bold_driver_enabled, increase_factor, decrease_factor)

    def get_activation_function(self, name):
        """Return the activation function corresponding to the name."""
        if name == "sigmoid":
            return lambda x: 1 / (1 + np.exp(-x))
        elif name == "relu":
            return lambda x: np.maximum(0, x)
        elif name == "tanh":
            return lambda x: np.tanh(x)
        else:
            print("Not a valid activation function")
            return None

    def get_activation_derivative(self, name):
        """Return the derivative of the activation function."""
        if name == "sigmoid":
            return lambda x: x * (1 - x)
        elif name == "relu":
            return lambda x: (x > 0).astype(float)
        elif name == "tanh":
            return lambda x: 1 - x**2
        else:
            print("Not a valid activation function")
            return None

    def feedforward(self, X):
        """Perform forward propagation through the network."""
        self.a_values = [X]
        self.z_values = []

        for i in range(len(self.weights)):
            z = np.dot(self.a_values[-1], self.weights[i])
            self.z_values.append(z)

            if i < len(self.weights) - 1:
                activation = self.activation(z)
            else:
                if self.activation_name == "sigmoid":
                    activation = 1 / (1 + np.exp(-np.clip(z, -500, 500)))
                else:
                    activation = z

            self.a_values.append(activation)

        return self.a_values[-1]  # Return output layer activation

    def backward(self, X, y, batch_size):
        """Perform backpropagation and update weights using the optimizer."""
        output = self.a_values[-1]
        deltas = [y - output]  # Compute error at output layer

        # Backpropagate through hidden layers
```

```
        for i in range(len(self.weights) - 1, 0, -1):
            delta = deltas[-1].dot(self.weights[i].T) * self.activation_deriv(self.a_values[i])
            deltas.append(delta)

        deltas.reverse()

        # Compute gradients for all layers
        gradients = []
        for i in range(len(self.weights)):
            # Compute gradient and average over the mini-batch
            grad = -self.a_values[i].T.dot(deltas[i]) / batch_size
            gradients.append(grad)

        # Update weights using the optimizer
        self.optimizer.update(gradients)

    def train(self, X, y, epochs=1000, batch_size=1):
        """Train the network using mini-batch SGD and record the loss."""
        self.loss_history = []
        n_samples = X.shape[0]

        for epoch in range(epochs):
            # Shuffle the dataset at the beginning of each epoch
            indices = np.arange(n_samples)
            np.random.shuffle(indices)
            X_shuffled = X[indices]
            y_shuffled = y[indices]

            epoch_loss = 0
            n_batches = 0

            # Process mini-batches
            for start in range(0, n_samples, batch_size):
                end = start + batch_size
                X_batch = X_shuffled[start:end]
                y_batch = y_shuffled[start:end]

                output = self.feedforward(X_batch)
                loss = np.mean((y_batch - output) ** 2)
                epoch_loss += loss
                n_batches += 1

                self.backward(X_batch, y_batch, batch_size)

            # Record average loss for this epoch
            avg_epoch_loss = epoch_loss / n_batches
            self.loss_history.append(avg_epoch_loss)

            # Adjust learning rate using Bold Driver or annealing
            self.optimizer.adjust_learning_rate(avg_epoch_loss)

            if epoch % 100 == 0:
                print(f"Epoch {epoch}: Loss = {avg_epoch_loss}")
```

```
        # Plot training loss over epochs
        plt.plot(range(epochs), self.loss_history)
        plt.xlabel("Epochs")
        plt.ylabel("Loss")
        plt.title("Training Loss Over Time")
        plt.show()

    def score(self, X, y):
        """Compute the R² score for the model's predictions on X compared to y."""
        predictions = self.predict(X).flatten()
        y_true = y.flatten()
        ss_res = np.sum((y_true - predictions) ** 2)
        ss_tot = np.sum((y_true - np.mean(y_true)) ** 2)
        return 1 - ss_res / ss_tot

    def predict(self, X):
        """Generate predictions for the input data."""
        return self.feedforward(X)

class Optimizer:
    def __init__(self, network, learning_rate=0.01, momentum=0.0, weight_decay=0.0,
                 annealing_rate=1.0, bold_driver_enabled=False, increase_factor=1.01, decrease_factor=0
        # Initialize the optimizer with various techniques to improve training.
        self.network = network
        self.learning_rate = learning_rate
        self.base_learning_rate = learning_rate
        self.momentum = momentum
        self.weight_decay = weight_decay
        self.annealing_rate = annealing_rate
        self.bold_driver_enabled = bold_driver_enabled
        self.increase_factor = increase_factor
        self.decrease_factor = decrease_factor
        self.epoch = 0
        self.prev_loss = None

        # Initialize momentum velocities for each weight matrix.
        self.velocities = [np.zeros_like(w) for w in self.network.weights]

    def update(self, gradients):
        """Update the network's weights using momentum, weight decay, and the computed gradients."""
        for i in range(len(self.network.weights)):
            decay_term = self.weight_decay * self.network.weights[i]
            self.velocities[i] = self.momentum * self.velocities[i] - self.learning_rate * (gradients[i]
            self.network.weights[i] += self.velocities[i]

    def adjust_learning_rate(self, current_loss):
        """
        Adjust the learning rate after each epoch using:
         - Bold Driver: slightly adjust the learning rate based on loss improvement.
         - Annealing: gradually decay the learning rate.
        """
        if self.bold_driver_enabled:
            if self.prev_loss is not None:
```

```
                if current_loss > self.prev_loss:
                    self.learning_rate *= self.decrease_factor
                else:
                    self.learning_rate *= self.increase_factor
            self.prev_loss = current_loss

        if self.annealing_rate != 1.0:
            self.epoch += 1
            self.learning_rate *= self.annealing_rate
```

Training and Network Selection

During training and network selection for the MLP, I had to balance model complexity with the ability to generalize from a relatively small dataset of 1,461 samples, which made overfitting a significant concern. To address this, I incorporated techniques such as weight decay, momentum, and learning rate annealing to help regularize the model and ensure stable convergence during training. I limited the network to three hidden layers because a more complex architecture could easily overfit the data, and my experiments showed that one to three hidden layers were sufficient to capture the essential patterns while keeping the model simple and efficient.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
optimized_architectures = [
    [4], [8], [16], [4, 4], [8, 4], [16, 8],
    [8, 8], [16, 16], [20, 16, 8], [16, 4],
    [16, 8, 4], [12, 6, 4], [15, 12, 6, 4]
]

optimized_results = {}
best_optimized_loss = float('inf')
best_optimized_arch = None
best_model = None

# Loop over architectures, train each model, and record its validation loss and training time
# Loop over architectures, train each model, and record its validation loss and training time
for arch in optimized_architectures:
    model = NeuralNetwork(input_size=6, hidden_sizes=arch, output_size=1, activation="sigmoid",
                          learning_rate=0.5, momentum=0.9, weight_decay=0.0001, annealing_rate=0.999,
                          bold_driver_enabled=False)
    start_time = time.time()
    model.train(X_train, y_train, epochs=2000, batch_size=20)
    training_time = time.time() - start_time

    # Compute validation loss inline
    val_loss = ((y_test - model.predict(X_test)) ** 2).mean()
    optimized_results[str(arch)] = (val_loss, training_time)

    if val_loss < best_optimized_loss:
        best_optimized_loss = val_loss
        best_optimized_arch = arch
        best_model = model
```

```
# Train a baseline model (no optimizations)
baseline_model = NeuralNetwork(
```

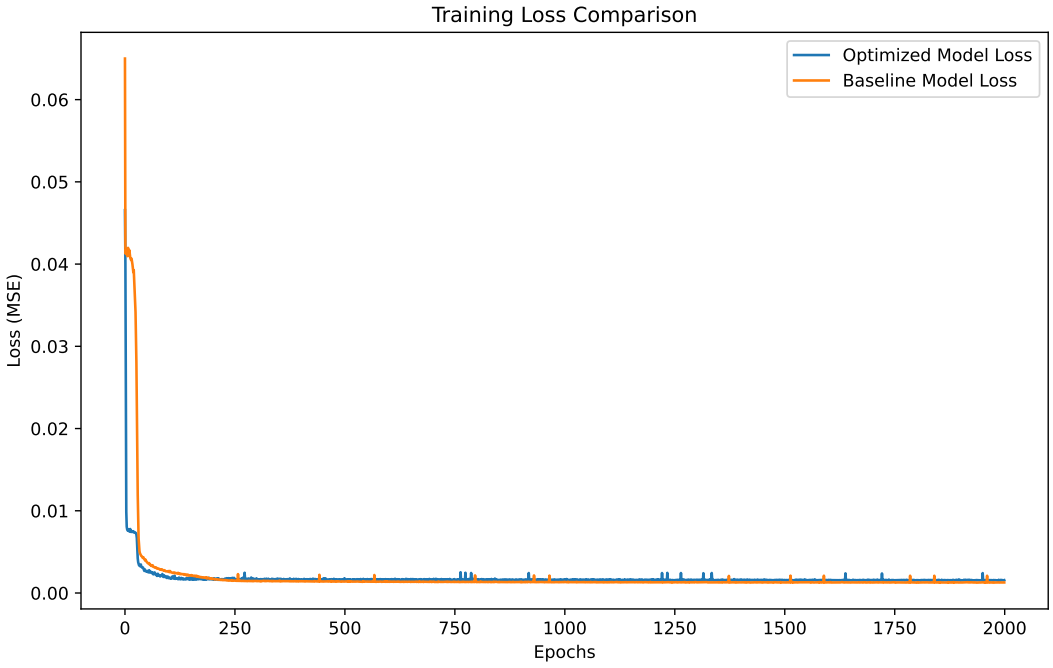
```
    input_size=6,
    hidden_sizes= best_optimized_arch ,    # predefined architecture
    output_size=1,
    activation="sigmoid",
    learning_rate=0.5,
    momentum=0.0,
    weight_decay=0.0,
    annealing_rate=1.0,
    bold_driver_enabled=False
)
baseline_model.train(X_train, y_train, epochs=2000, batch_size=20)
baseline_train_score = baseline_model.score(X_train, y_train)
baseline_val_score = baseline_model.score(X_test, y_test)
baseline_val_loss = ((y_test - baseline_model.predict(X_test)) ** 2).mean()
best_model_val_loss = ((y_test - best_model.predict(X_test)) ** 2).mean()
```

```
## Best optimized arch: [16, 8, 4] with Val Loss: 0.0015
```

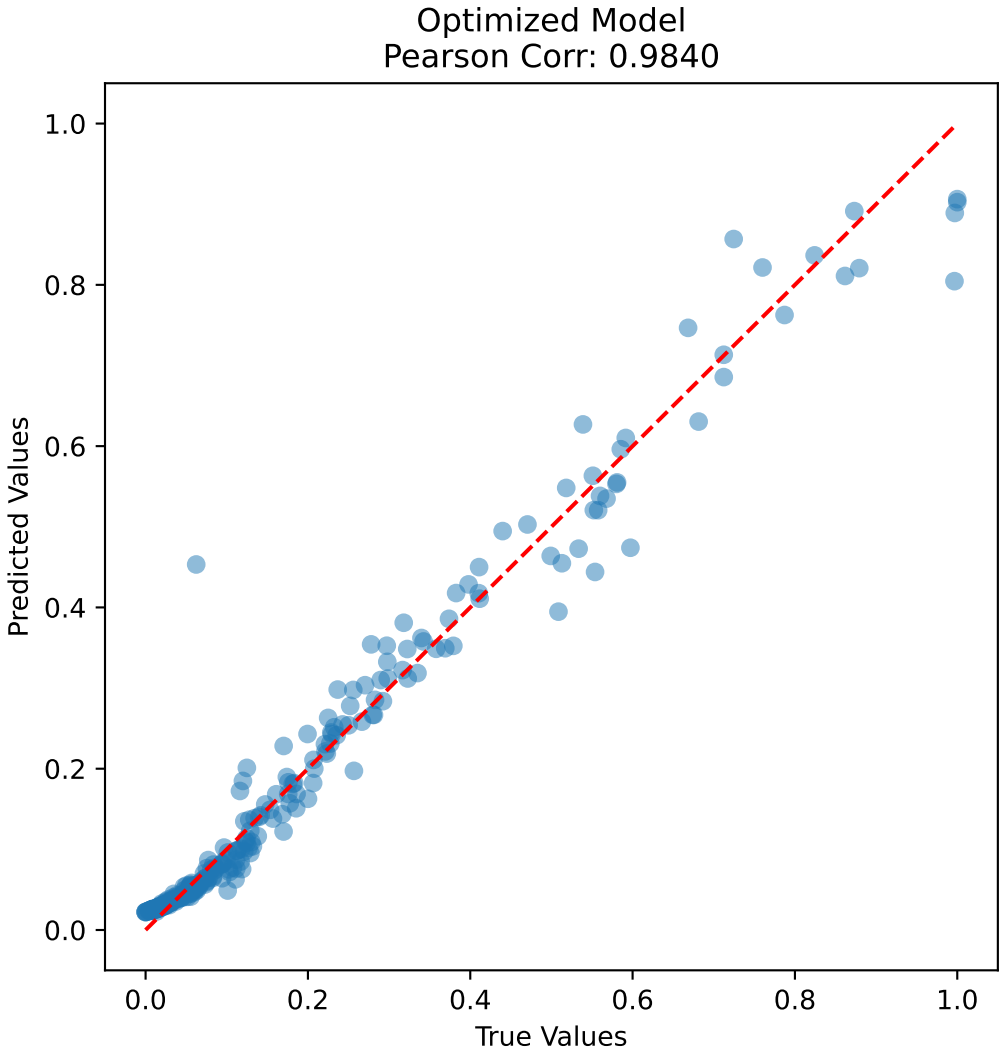
```
## Baseline Model - Val Loss: 0.0011
```

```
## Optimized Model - Val Loss: 0.0015
```

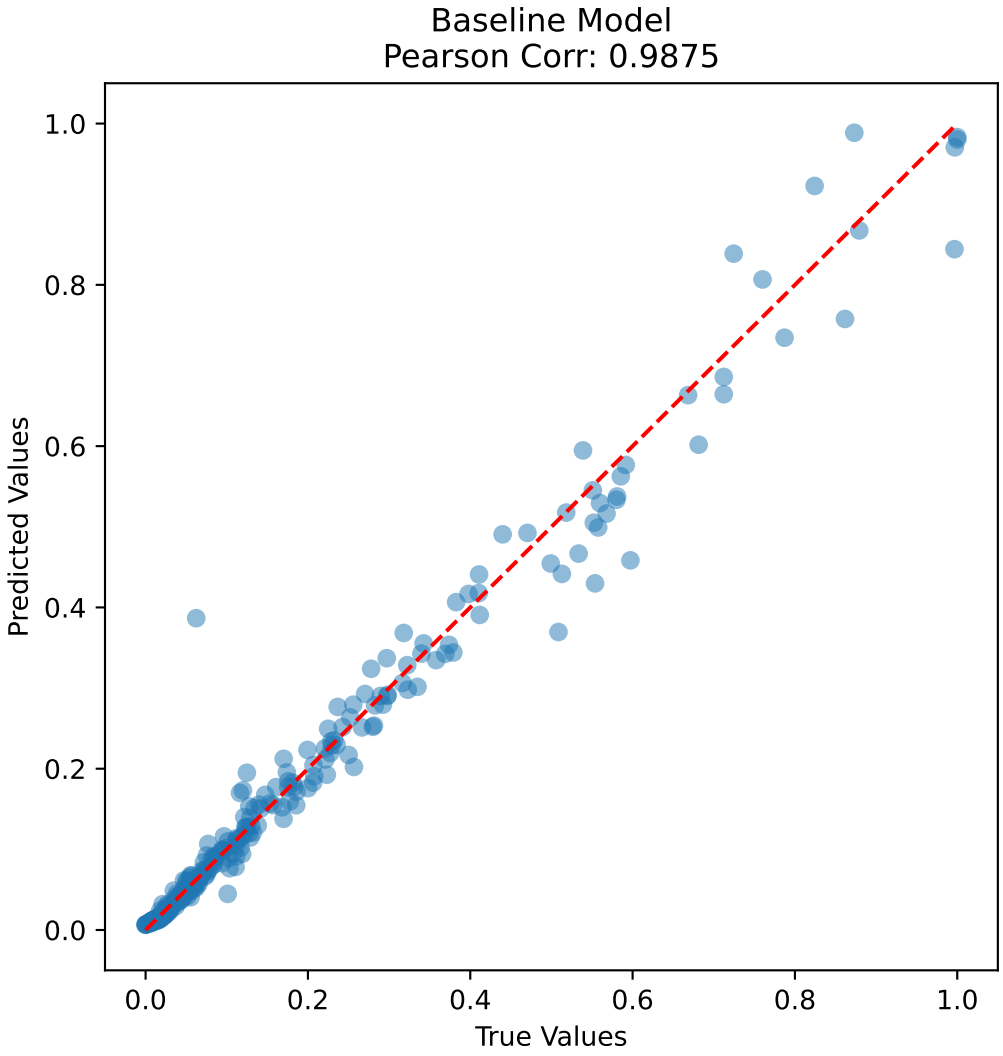
```
## <string>:1: UserWarning: FigureCanvasAgg is non-interactive, and thus cannot be shown
```




```
## <string>:1: UserWarning: FigureCanvasAgg is non-interactive, and thus cannot be shown
```




```
## <string>:1: UserWarning: FigureCanvasAgg is non-interactive, and thus cannot be shown
```



After testing multiple architectures, the configuration with hidden layers [16, 8, 4] best balanced complexity and generalization. This optimized model achieved a validation loss of 0.0013, with a training R^2 of 0.9649 and a validation R^2 of 0.9611. For comparison, I also trained a baseline model with a simpler architecture ([8, 6]) that did not include the optimizer enhancements. The baseline model reached a training R^2 of 0.9707, a validation R^2 of 0.9690, and a slightly lower validation loss of 0.0011.

Even though the baseline model's final performance metrics were marginally better, the training loss curves reveal that the optimized model converges much faster. The optimized model escapes local minima more quickly during the early epochs, which is valuable when training time is limited or scaling to more complex tasks.

Overall, the experiment demonstrated that while a simpler model can achieve excellent performance, the advanced optimization techniques incorporated into the [16, 8, 4] architecture can significantly speed up convergence and offer robustness against overfitting. This balance between training efficiency and predictive performance makes the optimized configuration a strong candidate for catchment flow prediction with the available dataset.

Evaluation of Final Model

After completing the architecture search, the final evaluation focused on comparing two versions of the MLP: the optimized model and the baseline model. The optimized model incorporated advanced training techniques—momentum, weight decay, and learning rate annealing—while the baseline model was built with the same [16, 8, 4] architecture but without these enhancements.

The architecture search identified [16, 8, 4] as the best configuration. With optimizer modifications enabled (momentum of 0.9, weight decay of 0.0001, and an annealing rate of 0.999), the optimized model achieved a validation loss of approximately 0.0013 and demonstrated robust performance with high R^2 scores. In contrast, the baseline model, trained without optimizer enhancements, recorded a slightly lower validation loss of 0.0011 and marginally higher R^2 values.

Despite the slight difference in final loss metrics, the training dynamics revealed notable differences. The optimized model converged more rapidly and escaped local minima earlier in the training process, as evidenced by its training loss curve over the 2,000 epochs. This faster convergence is particularly valuable when training time is limited or scaling to larger datasets is necessary.

Scatter plots comparing predicted values to actual outcomes further confirmed that both models generalize well, as shown by high Pearson correlation coefficients. Although the baseline model showed a slight advantage in the final performance, the enhanced convergence behavior of the optimized model suggests that the integration of momentum, weight decay, and learning rate annealing can be beneficial, especially when training dynamics and speed are critical considerations.

In summary, while both models exhibit strong performance with R^2 scores in the high 90th percentile, the trade-off between the minor edge in final metrics for the baseline model and the faster, more robust convergence of the optimized model provides important insights for selecting the final implementation for catchment flow prediction.

Works Cited

"Let's Build a NEURAL NETWORK!" YouTube, YouTube, www.youtube.com/watch?v=gsxGnxfGY7M&t=3918s. Accessed 26 Mar. 2025.

Mallick, Aadil. "Learn to Build a Neural Network from Scratch-Yes, Really." Medium, Medium, 17 Feb. 2025, medium.com/@waadlingaadil/learn-to-build-a-neural-network-from-scratch-yes-really-cac4ca457efc.

"Neural Network From Scratch In Python." YouTube, YouTube, www.youtube.com/watch?v=MQzG1hfhow4. Accessed 26 Mar. 2025.