Artificial Neural Network (ANN) Implementation and Evaluation

F327599

March 24, 2025

Contents

1 Introduction

Predicting river flow is important for water management and flood control. This study uses an Artificial Neural Network (ANN) to predict the mean flow at Skelton for the next day. The model is trained on past flow data to identify patterns and improve forecasting accuracy. Results will help assess the effectiveness of ANNs for river flow prediction.

Some lines of code or text in this report have been split into two for formatting purposes. This does not affect the meaning or functionality of the content. Full code has been submitted along with this report.

There might also be some inconsistencies in the spelling as I accidentally changed from UK to USA English midway through the report and didn't manage to fix every spelling.

2 Data pre-processing

Data pre-processing is a crucial step in ensuring the quality of the input data before training the Artificial Neural Network (ANN). This involves:

- Handling missing values,
- Feature selection and engineering,
- Splitting data into training and testing sets.
- Normalization and standardization,

2.1 Python Code for Data Pre-processing

Below is the Python implementation for data pre-processing, including handling missing values, detecting and treating outliers, and adding lag features.

2.1.1 replace_negative Function

This function replaces negative values with NaN, as negative rainfall and negative flow do not exist in our context.

- The dataset is checked for any negative values, which are invalid in the context of rainfall and flow measurements. Rainfall and flow measurements can't be negative
- Any detected negative values are replaced with NaN to be handled in later stages of data processing, ensuring consistency and accuracy.

2.1.2 clean_data Function

This function is responsible for cleaning the dataset by applying multiple steps, including handling outliers and missing values.

- Iterates through all the location columns (excluding the Date column) and applies the `replace_negative` function to replace negative values, preventing invalid data from influencing further analysis.
- Converts all values to a numeric format, coercing non-numeric values into NaN to maintain data integrity and prevent errors in numerical operations later on in analysis.
- Detects outliers using the Z-score method by computing the Z-score of each column while excluding NaN values. Any data points with an absolute Z-score greater than 4 are flagged as outliers and replaced with NaN. This threshold is chosen since on a normal distribution, 99.99% of the data should lie within four standard deviations from the mean.So we are applying this to absolute extreme outliers for example a flood.
- Handles missing data using a rolling median technique, which smooths fluctuations and fills missing values in a robust manner. Linear interpolation is then applied to further refine the imputation of missing values, ensuring smooth transitions between data points.
- Applies an additional outlier detection method using the Interquartile Range (IQR), where values outside 1.5 times the IQR are identified as extreme outliers and replaced with NaN. This ensures that extreme deviations are accounted for using multiple statistical approaches.
- Uses forward fill and backward fill techniques to handle any remaining NaN values, ensuring no gaps remain in the dataset and allowing for a complete time series.

2.1.3 add_lags Function

This function adds lag features to the dataset, creating new columns for values shifted by a specified number of days (default lag is 1 day).

- Generates lagged features by shifting each column's values backward by a defined number of days, allowing past data points to be used as predictors in time series analysis.
- This process enhances the dataset for predictive modeling by incorporating temporal dependencies, making it suitable for forecasting applications.

2.1.4 load_and_clean_data Function

The main function, `load_and_clean_data`, is responsible for loading the dataset from an Excel file and applying the cleaning and feature engineering steps:

- Data are loaded from the Excel file by extracting the sheet `"1993-96"` and skipping the first row to ensure only relevant data are included.
- Rename columns to meaningful and standardized names for clarity and ease of use in further analysis.
- Drops unnecessary columns that do not contribute to the analysis, reducing noise, and improving computational efficiency.
- Converts the `Date` column to datetime format, ensuring the proper handling of time-based operations.
- Drops rows where all location values are NaN (excluding the date column), removing entirely missing observations and preserving data quality.
- Cleans the data by applying the `clean_data` function, which ensures robust handling of outliers and missing values before further processing, maintaining data integrity.
- Optionally shifts the `Skelton` column by one row if `shift_target` is set to `True`, enabling predictive modeling where past data is aligned with future values.
- Adds lag features using the `add_lags` function for the specified number of days, allowing historical trends to be incorporated into the dataset for better forecasting.
- Remove rows containing NaN values introduced by lagging to prevent incomplete records from affecting the analysis.
- Returns the fully cleaned and transformed DataFrame, optimised for predictive modeling or further statistical analysis.

2.1.5 Key Data Processing Steps

- Replacement of negative values with NaN to ensure data robustness and prevent invalid values from affecting computations.
- Outlier detection using multiple statistical methods:
  - Z-score threshold of 4 to remove extreme deviations based on standard deviation.
  - IQR method (1.5 * IQR) to capture outliers in a distribution independent manner.

- Missing value imputation using:
  - Rolling median to smooth and fill missing data points with robust statistical estimates.
  - Linear interpolation to create continuous and logically consistent time series.
  - Forward and backfilling to ensure that no gaps remain in the dataset.
- Lag feature creation to enable time series prediction, incorporating historical data trends into the model to improve forecast accuracy.

```python
import numpy as np
import pandas as pd
from scipy.stats import zscore

# Function to replace negative values with NaN
def replace_negative(x):
    if isinstance(x, (int, float)) and x < 0:
        return np.nan
    return x

# Function to clean data with outlier handling and imputation
def clean_data(df):
    location_columns = df.columns[1:]  # All columns except the first one ("Date")

    for col in location_columns:
        df[col] = df[col].apply(replace_negative)
        df[col] = pd.to_numeric(df[col], errors='coerce')

        # Handle outliers using Z-score
        z_scores = zscore(df[col].dropna())
        outlier_indices = np.abs(z_scores) > 4
        outlier_indices = pd.Series(outlier_indices, index=df[col].dropna().index)
        df.loc[outlier_indices.index, col] =
        df.loc[outlier_indices.index, col].where(~outlier_indices, np.nan)
        # Fill missing data using rolling median
        df[col] = df[col].fillna(df[col].rolling(window=3, min_periods=1).median())

        # Linear interpolation
        df[col] = df[col].interpolate(method='linear', limit_direction='both')


        # Handle extreme outliers using IQR
        Q1 = np.percentile(df[col].dropna(), 25)
        Q3 = np.percentile(df[col].dropna(), 75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR
        df.loc[(df[col] < lower_bound) | (df[col] > upper_bound), col] = np.nan

        # Forward fill and backward fill
        df[col] = df[col].ffill().bfill()

    return df

# Function to add lag features
def add_lags(df, lag_days=1):
    for col in df.columns[1:]:
        for lag in range(1, lag_days + 1):
            df[f'{col}_lag{lag}'] = df[col].shift(lag)
    return df

def load_and_clean_data(file_path, lag_days=7, shift_target=True):
    xls = pd.ExcelFile(file_path)
    df = pd.read_excel(xls, sheet_name="1993-96", skiprows=1)

    df.columns = ["Date", "Crakehill", "Skip Bridge", "Westwick", "Skelton",
                  "Arkengarthdale", "East Cowton", "Malham Tarn", "Snaizeholme",
                  "Extra1", "Extra2", "Notes"]

    df = df.drop(columns=["Extra1", "Extra2", "Notes"], errors='ignore')
    df["Date"] = pd.to_datetime(df["Date"], errors='coerce')
    df = df.dropna(subset=df.columns[1:], how='all')

    df_cleaned = clean_data(df)

    if shift_target:
        df_cleaned['Skelton'] = df_cleaned['Skelton'].shift(-1)

    df_cleaned = add_lags(df_cleaned, lag_days=lag_days)
    df_cleaned = df_cleaned.dropna()

    return df_cleaned
```

2.2 Feature Selection and Engineering

Below is how I chose my features and the analysis I did to get there including the code for it.

First, I looked at all the locations and compared the correlations of them all against Skelton.I also compared the lagged versions against Skelton to see the effects of lagged data against Skelton.I added all the correlations into a file in ascending order.This analysis helped identify patterns and dependencies that could be useful for predictive modeling.

```
def correlation_analysis(df):
    df["Combined_Stations"] = df["Arkengarthdale"] + df["East Cowton"] + df["Snaizeholme"]
    df["Combined_Stations"] = pd.to_numeric(df["Combined_Stations"], errors="coerce")

    # Drop rows with NaN values
    df.dropna(inplace=True)

    df['Skelton_Lag1'] = df['Skelton'].shift(1)

    # Compute correlation matrix
    correlation_matrix = df.corr()

    # Save correlation results to a text file
    correlation_file = "correlation_results.txt"
    with open(correlation_file, "w") as f:
        f.write("Correlation with Skelton Flow:\n")
        f.write(correlation_matrix['Skelton_Lag1'].sort_values(ascending=True).to_string())

    print(f"Correlation results saved to {correlation_file}")
```

From the correlation analysis, the features that have the highest correlation with Skelton were put to one side as potential inputs for the model. This

made sure that only the most relevant data
points were considered, reducing noise and improving the efficiency of the model.

Table 1: Correlation values with Skelton Flow

Station	**Correlation**
East Cowton	0.052
East Cowton_lag3	0.0754
East Cowton_lag4	0.0949
East Cowton_lag2	0.0970
East Cowton_lag1	0.111
Arkengarthdale	0.217
Arkengarthdale_lag4	0.252
Malham Tarn	0.261
Arkengarthdale_lag3	0.275
Arkengarthdale_lag2	0.301
Malham Tarn_lag4	0.305
Arkengarthdale_lag1	0.308
Combined Stations	0.312
Malham Tarn_lag3	0.334
Snaizeholme	0.341
Snaizeholme_lag4	0.354
Malham Tarn_lag2	0.358
Malham Tarn_lag1	0.359
Snaizeholme_lag3	0.380
Snaizeholme_lag2	0.424
Snaizeholme_lag1	0.428
Westwick_lag4	0.694
Skip Bridge_lag4	0.701
Crakehill_lag4	0.705
Westwick_lag3	0.714
Skip Bridge_lag3	0.725
Crakehill_lag3	0.732
Skip Bridge_lag2	0.762
Skelton_lag4	0.762
Westwick_lag2	0.771
Crakehill_lag2	0.782
Skelton_lag3	0.817
Skip Bridge_lag1	0.826
Crakehill_lag1	0.857
Westwick_lag1	0.865
Skip Bridge	0.869
Westwick	0.895
Skelton_lag2	0.909
Skelton	0.909
Crakehill	0.915

To improve the predictive capability of the model, hydrological features were engineered. These features included variables that describe water-related dynamics, such as flow accumulation, precipitation patterns, soil moisture levels, and groundwater storage.By including context-specific knowledge, these features provided additional insights into hydrological processes affecting Skelton.

```python
def add_hydrological_features(df):
    """
    Adds hydrological features to the given DataFrame.

    Parameters:
    df (pd.DataFrame): Input DataFrame with required columns.

    Returns:
    pd.DataFrame: DataFrame with new features added.
    """

    # Combine rain station data from multiple locations
    df["RainStations"] = df["Arkengarthdale"] + df["East Cowton"] + df["Snaizeholme"]
    df["RainStations"] = pd.to_numeric(df["Combined_Stations"], errors="coerce")

    # Cumulative sum of Skelton flow to track total water volume over time
    df['CumulativeFlow'] = df['Skelton'].cumsum()

    # Ratio of Crakehill flow to Westwick flow to measure relative flow contribution
    df['Flow_Ratio'] = df['Crakehill'] / (df['Westwick'] + 1e-6)  # Avoid division by zero

    # Change in Skelton flow compared to the previous time step
    df['Flow_Change'] = df['Skelton'].diff()

    # 7-day exponentially weighted moving average of combined rainfall stations (API-7)
    df["API_7"] = df["Combined_Stations"].ewm(alpha=1/7, adjust=False).mean()

    # Ratio of Skelton flow to Crakehill flow to analyze water propagation
    df["Flow_Propagation_Ratio"] = df["Skelton"] / (df["Crakehill"] + 1e-5)
    # Avoid division by zero

    # Change in Skelton flow compared to three time steps ago
    df["Flow_Change_Skelton_3"] = df["Skelton"] - df["Skelton"].shift(3)

    # Change in Skelton flow compared to the previous time step
    df["Flow_Change_Skelton_Lag1"] = df["Skelton"] - df["Skelton"].shift(1)
```

```
# Recession coefficient, indicating how Skelton flow decreases over time
df["Recession_K"] = (df["Skelton"].shift(1) - df["Skelton"]) / df["Skelton"].shift(1)

# 3-day rolling average of combined Crakehill, Westwick, and Skip Bridge flows
df['CWSB'] = (df['Crakehill'] +
df['Westwick'] +
df['Skip Bridge']).rolling(window=3).mean()

# Skelton flow lagged by one time step
df['Skelton_Lag1'] = df['Skelton'].shift(1)

# Percentage change in flow at Crakehill
df['Crakehill_Pct_Change'] = df['Crakehill'].pct_change()

# Percentage change in flow at Westwick
df['Westwick_Pct_Change'] = df['Westwick'].pct_change()

# Percentage change in flow at Skip Bridge
df['SkipBridge_Pct_Change'] = df['Skip Bridge'].pct_change()

# Interaction term: Crakehill and Westwick flow percentage changes
df['Crakehill_Westwick_Interaction'] =
df['Crakehill_Pct_Change'] * df['Westwick_Pct_Change']



# Interaction term: Crakehill and Skip Bridge flow percentage changes
df['Crakehill_SkipBridge_Interaction'] =
df['Crakehill_Pct_Change'] * df['SkipBridge_Pct_Change']

# Interaction term: Westwick and Skip Bridge flow percentage changes
df['Westwick_SkipBridge_Interaction'] =
df['Westwick_Pct_Change'] * df['SkipBridge_Pct_Change']

# Interaction between Crakehill, Westwick, and Skip Bridge percentage changes
df['Crakehill_Westwick_SkipBridge_Interaction'] =
df['Crakehill_Pct_Change'] * df['Westwick_Pct_Change'] * df['SkipBridge_Pct_Change']

# Mixed interaction of Crakehill percentage change
# with Westwick and Skip Bridge interaction
df['Crakehill_Pct_Change * Westwick_SkipBridge_Interaction'] =
df['Crakehill_Pct_Change'] * df['Westwick_SkipBridge_Interaction']

# Interaction of Westwick and Skip Bridge percentage changes
df['Westwick_Pct_Change * SkipBridge_Pct_Change'] =
df['Westwick_Pct_Change'] * df['SkipBridge_Pct_Change']

# Sum of percentage changes in Crakehill and Westwick flows
df['Crakehill_Westwick_Pct_Change_Sum'] =
df['Crakehill_Pct_Change'] + df['Westwick_Pct_Change']

# 7-day rolling mean of Crakehill percentage change to smooth fluctuations
df['Crakehill_Rolling_Mean'] =
df['Crakehill_Pct_Change'].rolling(window=7).mean()

# Interaction of rainfall with Crakehill flow percentage change
df['Rainfall_Crakehill_Interaction'] =
df["RainStations"] * df['Crakehill_Pct_Change']

# Interaction of rainfall with Skip Bridge flow percentage change
df['Rainfall_SkipBridge_Interaction'] =
df["RainStations"] * df['SkipBridge_Pct_Change']

# Interaction of rainfall with Westwick flow percentage change
df['Rainfall_Westwick_Interaction'] =
df["RainStations"] * df['Westwick_Pct_Change']

# Lagged interaction of Crakehill percentage change with Westwick percentage change
df['Crakehill_Lag_Interaction'] =
df['Crakehill_Pct_Change'].shift(1) * df['Westwick_Pct_Change']

# Lagged interaction of Westwick percentage change with Crakehill percentage change
df['Westwick_Lag_Interaction'] =
df['Westwick_Pct_Change'].shift(1) * df['Crakehill_Pct_Change']



# Lagged interaction of Skip Bridge percentage change with Crakehill percentage change
df['SkipBridge_Lag_Interaction'] =
df['SkipBridge_Pct_Change'].shift(1) * df['Crakehill_Pct_Change']

# Cumulative rainfall over 7 days interacting with Crakehill percentage change
df['Cumulative_Rainfall_Interaction'] =
df["RainStations"].rolling(window=7).sum() * df['Crakehill_Pct_Change']

# Cumulative Skelton flow over 7 days interacting with Westwick percentage change
df['Cumulative_Flow_Interaction'] =
df['Skelton'].rolling(window=7).sum() * df['Westwick_Pct_Change']

return df
```

```
def correlation_hydrology(df):
    """
    Computes the correlation of specified columns with 'Skelton' s
    hifted by 1 and writes the results to a file.

    Parameters:
    df (pd.DataFrame): Input DataFrame with required columns.
    """

    # Call feature function (ensure it's defined elsewhere)
    add_hydrological_features(df)

    print("After adding hydrological features:", df.columns)

    # Shift Skelton by 1 to create a lagged version
    df['Skelton_Lag1'] = df['Skelton'].shift(1)

    # Drop rows with NaN values (this will drop first row)
    df.dropna(inplace=True)

    # Compute correlation matrix
    correlation_matrix = df.corr()

    # Ensure 'Skelton_Lag1' exists in correlation matrix
    if 'Skelton_Lag1' not in correlation_matrix.columns:
        raise ValueError("Column 'Skelton_Lag1' missing in correlation matrix,
        check data processing steps.")

    # Save correlation results to a text file
    correlation_file = "correlation_results_hydrology.txt"
    with open(correlation_file, "w") as f:
        f.write("Correlation with Skelton Flow:\n")
        f.write(correlation_matrix["Skelton_Lag1"].
        sort_values(ascending=True).to_string())

    print(f"Correlation results saved to {correlation_file}")
```

Table 2: Correlation values with Skelton Flow

Variable	**Correlation**
CumulativeFlow	-0.240
Flow_Change	-0.218
Flow_Change_Skelton_Lag1	-0.218
Flow_Ratio	-0.114
Crakehill_Pct_Change * Westwick_SkipBridge_Interaction	0.0157
Crakehill_Westwick_SkipBridge_Interaction	0.0157
Westwick_SkipBridge_Interaction	0.0229
Westwick_Pct_Change * SkipBridge_Pct_Change	0.0229
Crakehill_Lag_Interaction	0.0312
Crakehill_SkipBridge_Interaction	0.0326
Crakehill_Westwick_Interaction	0.0439
...	...
...	...
...	...
Snaizeholme_lag1	0.429
API_7	0.655
Westwick_lag4	0.697
Skip Bridge_lag4	0.703
Crakehill_lag4	0.706
Westwick_lag3	0.717
Skip Bridge_lag3	0.728
Crakehill_lag3	0.734
Skip Bridge_lag2	0.765
Skelton_lag4	0.765
Westwick_lag2	0.774
Crakehill_lag2	0.784
Skelton_lag3	0.820
Skip Bridge_lag1	0.827
Crakehill_lag1	0.856
Westwick_lag1	0.868
Skip Bridge	0.869
Westwick	0.896
Skelton	0.909
Skelton_lag2	0.909
Crakehill	0.915
CWSB	0.920

After generating the hydrological features,I added all the correlations into a file in ascending order, another round of correlation analysis was done to determine which of these newly created features had the strongest relationship with Skelton. The most correlated features were kept to refine the dataset further and ensure only the most informative features were kept as potential inputs.

```
def correlation_analysis_rm_flow_diff(df):
# Create new features
columns = ['Crakehill', 'Skip Bridge', 'Westwick']

# Create all 2-place combinations of the columns
combinations = list(itertools.combinations(columns, 2))

# Define the rolling window sizes (from 3 to 7)
window_sizes = [3, 4, 5, 6, 7]

# Iterate over combinations and window sizes
for (col1, col2) in combinations:
    # Calculate the flow difference for each pair of columns
    diff_column_name = f'Flow_Diff_{col1}_{col2}'
    df[diff_column_name] = df[col1] - df[col2]

    # Apply the rolling mean for each window size
    for window in window_sizes:
        rolling_column_name = f'{diff_column_name}_Rolling_Mean_{window}'
        df[rolling_column_name] =
        df[diff_column_name].rolling(window=window).mean()


df['Skelton_Lag1'] = df['Skelton'].shift(1)

# Compute correlation matrix
correlation_matrix = df.corr()

# Define the file path to save the results
correlation_file = "correlation_results_rolling_mean_flow_diff.txt"

with open(correlation_file, "w") as f:
  # Write correlations with Skelton
  f.write("Correlations with Skelton Flow:\n")
  f.write(correlation_matrix['Skelton_Lag1'].
  sort_values(ascending=True).to_string())

print(f"Correlation results have been saved to {correlation_file}")
```

Table 3: Correlation values with Skelton Flow

Variable	**Correlation**
Flow_Diff_Skip Bridge_Westwick_Rolling_Mean_3	-0.833
Flow_Diff_Skip Bridge_Westwick_Rolling_Mean_4	-0.821
Flow_Diff_Skip Bridge_Westwick_Rolling_Mean_5	-0.814
Flow_Diff_Skip Bridge_Westwick_Rolling_Mean_6	-0.808
Flow_Diff_Skip Bridge_Westwick_Rolling_Mean_7	-0.802
Flow_Diff_Skip Bridge_Westwick	-0.800
Flow_Diff_Crakehill_Westwick_Rolling_Mean_7	-0.402
Flow_Diff_Crakehill_Westwick_Rolling_Mean_6	-0.396
Flow_Diff_Crakehill_Westwick_Rolling_Mean_5	-0.388
Flow_Diff_Crakehill_Westwick_Rolling_Mean_4	-0.381
Flow_Diff_Crakehill_Westwick_Rolling_Mean_3	-0.379
Flow_Diff_Crakehill_Westwick	-0.292
...	...
...	...
Combined_Stations	0.308
RainStations	0.308
Malham Tarn_lag4	0.308
Malham Tarn_lag3	0.339
Snaizeholme	0.340
Snaizeholme_lag4	0.362
Malham Tarn_lag1	0.362
Malham Tarn_lag2	0.364
Snaizeholme_lag3	0.384
Snaizeholme_lag2	0.427
Snaizeholme_lag1	0.431
API_7	0.654
Westwick_lag4	0.697
Skip Bridge_lag4	0.704
Crakehill_lag4	0.707
Westwick_lag3	0.717
Skip Bridge_lag3	0.729
Crakehill_lag3	0.734
Skip Bridge_lag2	0.764
Skelton_lag4	0.765
Westwick_lag2	0.774
Flow_Diff_Crakehill_Skip Bridge_Rolling_Mean_7	0.777
Crakehill_lag2	0.783
Flow_Diff_Crakehill_Skip Bridge_Rolling_Mean_6	0.784
Flow_Diff_Crakehill_Skip Bridge_Rolling_Mean_5	0.791
Flow_Diff_Crakehill_Skip Bridge	0.794
Flow_Diff_Crakehill_Skip Bridge_Rolling_Mean_4	0.797
Flow_Diff_Crakehill_Skip Bridge_Rolling_Mean_3	0.805
Skelton_lag3	0.819
Skip Bridge_lag1	0.827
Crakehill_lag1	0.856
Westwick_lag1	0.868
Skip Bridge	0.869
Westwick	0.896
Skelton	0.908
Skelton_lag2	0.909
Crakehill	0.915
CWSB	0.920

Following the application of rolling means, and adding all the correlations into a file in ascending order,a final correlation analysis was conducted to identify which of these transformed features had the highest correlation with Skelton.The most significant rolling mean features were included in the final feature set to enhance predictive performance.

2.2.1 Why These Features Were Chosen

The selected features were chosen based on their strong correlation with Skelton Flow, hydrological relevance, and predictive power. Here's why each feature was selected:

1. **CWSB_Rolling_Mean_5**
   - Represents a smoothed flow measure across key upstream stations.
   - Helps capture long-term flow trends, reducing short-term noise.
   - Also a rolling mean with window 5 correlates less with the other features but has high enough correlation with our target.
2. **Date_in_days**
   - Captures seasonal and long-term variations in flow.
   - Important for identifying trends influenced by time (e.g., seasonal cycles, climate effects).
3. **Flow_Diff_Crakehill_Skip_Bridge_Rolling_Mean_3**
   - Measures flow difference between key stations, capturing upstream-downstream interactions.
   - Rolling mean smooths fluctuations, focusing on meaningful trends.
4. **Skelton_Lag1**
   - Strongest predictor, as past flow directly impacts future flow.
   - Essential for any time-series forecasting approach.
5. **RainStations**
   - Captures rainfall input from multiple sources, directly influencing flow.
   - Strong hydrological driver for Skelton Flow.
   - The correlation of the station combined against Skelton was higher than each of then individually
6. **Westwick_Rolling_Mean_7**
   - Captures smooth trends from an important upstream location.

- Helps in understanding delayed flow impacts at Skelton.

7. **API_7 (Antecedent Precipitation Index)**
    - Measures accumulated rainfall effects, helping capture saturation levels.
    - Crucial for predicting delayed runoff contributions.
8. **Flow_Propagation_Ratio**
    - Tell us how much flow at upstream stations contributes to Skelton.
    - Helps capture lag and reduced effects.
9. **Cumulative_Flow_Interaction**
    - Represents total flow influence from multiple stations.
    - Helps capture basin-wide hydrological effects.

2.2.2 Why Other Features Were Excluded

Low Correlation with Skelton Flow:

- Features with correlations between 0.05 and 0.5 were removed as they provide limited predictive power.
- Examples: *East Cowton, Recession_K, Crakehill_Pct_Change.*

Redundancy and Multicollinearity:

- Features highly correlated with already-selected ones were removed to avoid redundancy.
- Example: *Multiple lag variables (e.g., Skelton_Lag2, Skelton_Lag3) were removed.*
- Rolling mean variables with similar behavior to selected predictors were also excluded.

Lack of Physical Interpretability:

- Some interaction terms and composite features lacked direct hydrological significance.

Too Much Noise / High Variability:

- Some predictors had extreme fluctuations, making them less reliable.

The selected features balance statistical significance, hydrological relevance, and predictive stability, ensuring a robust model for forecasting Skelton Flow.

2.3 Correlation Heatmap of the Predictors

The heatmap below shows the correlation between the selected predictors:

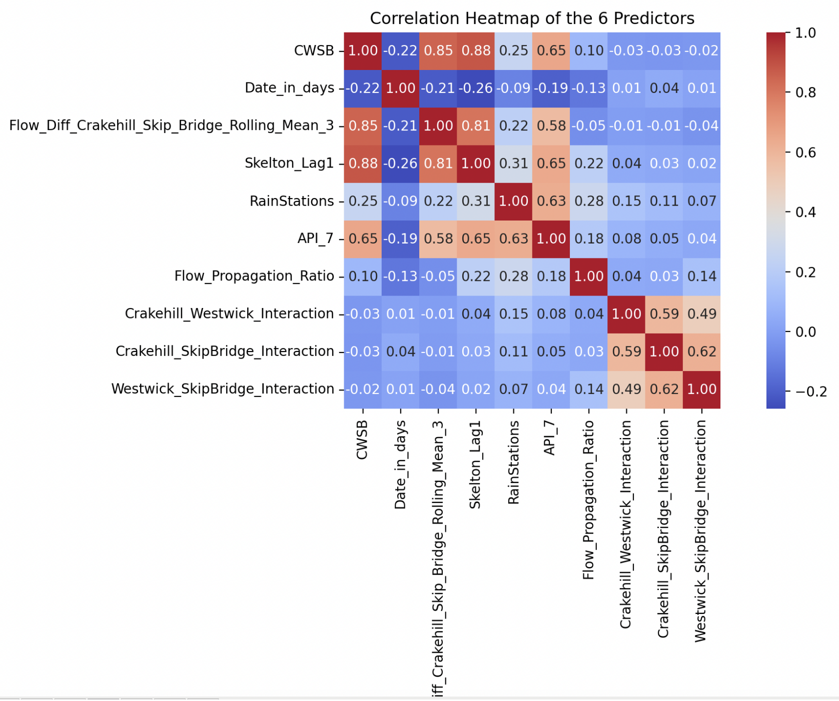


Figure 1: Correlation Heatmap of the Selected Predictors

2.4 Analysis of the Correlation Heatmap

The heatmap provides insight into the relationships between the six selected predictors. Below are the key observations:

2.4.1 Strong Positive Correlations (Red Cells)

- **CWSB & Flow_Diff_Crakehill_Skip_Bridge_Rolling_Mean_3 (0.85)**: This suggests that changes in flow differences at Crakehill and Skip Bridge strongly influence the CWSB variable. This is expected since these locations contribute to overall flow dynamics and are also included in the make up of CWSB which is Crakehill , Westwick and Skip Bridge added together.

- **CWSB & Skelton_Lag1 (0.88)**:

A strong positive correlation indicates that past Skelton flow values are highly predictive of CWSB. This suggests a lagged dependency in the hydrological system.

- **Flow_Diff_Crakehill_Skip_Bridge_Rolling_Mean_3 & Skelton_Lag1 (0.81)**:

  This relationship indicates that past Skelton flow influences flow differences between Crakehill and Skip Bridge.

- **API_7 & RainStations (0.63)**: The strong correlation is expected since the Antecedent Precipitation Index (API_7) reflects accumulated rainfall over seven days, which is directly related to the number of rainfall stations measuring it.

2.4.2 Moderate Positive Correlations

- **API_7 & Flow_Diff_Crakehill_Skip_Bridge_Rolling_Mean_3 (0.58)**: A moderate correlation suggests that prior rainfall (API_7) influences the difference in flow changes between Crakehill and Skip Bridge, likely due to runoff contributions.

- **Flow_Propagation_Ratio & RainStations (0.28)**: A small but notable correlation, indicating that more rainfall stations capturing precipitation may slightly affect how flow propagates downstream.

2.4.3 Negative Correlations (Blue Cells)

- **Date_in_days & CWSB (-0.22)**: A slight negative correlation suggests that, over time, there may be some trend in CWSB values, potentially due to seasonal variations or long-term hydrological shifts.

- **Date_in_days & Skelton_Lag1 (-0.26)**: Similar to the previous point, a weak negative correlation implies potential seasonal dependencies in Skelton flow.

2.4.4 Final Interpretation

- The predictors are well-chosen since they have strong correlations with the target variable (CWSB) while maintaining some diversity in information (e.g., lagged flow values, rainfall effects).

- The negative correlations with `Date_in_days` indicate a possible temporal trend that might need further analysis.

2.5 Why Use Time-Based Splitting?

- Prevents Data Leakage Ensures future data isn't used during training.

- Mimics Real-World Forecasting Helps the model generalize better to future data.
- Captures Trends and Seasonality,preserves annual patterns and cyclical trends.
- More Reliable Model Evaluation,tests the model on realistic future scenarios.
- Better for Sequential Models
- Maintains the natural order of data for time-dependent learning.

2.6 What is StandardScaler?

StandardScaler is a preprocessing tool from *scikit-learn* that standardizes numerical data by transforming it to have:

- Mean = 0
- Standard Deviation = 1

Artificial Neural Networks (ANNs) perform best when input features are on a similar scale. If features have vastly different ranges, the model may struggle to learn effectively, as large values can dominate smaller ones, leading to unstable gradients and slower convergence.

2.7 How Does StandardScaler Standardize Data?

1. Computes the Mean (μ) and Standard Deviation (σ) of Each Feature

   - It calculates these values only from the training data to avoid **data leakage**.

2. Applies the Transformation Using the Formula

$$X_{\text{scaled}} = \frac{X - \mu}{\sigma}$$

where:

- X = original feature value
- μ = mean of that feature in the training set
- σ = standard deviation of that feature in the training set

3. Scales the Training Data (`fit_transform()`)

```
scaler.fit_transform(train_df[features])
```

- Fits the scaler to the training data (computes μ and σ) and transforms it.

4. Applies the Same Scaling to Validation & Test Data (`transform()`)

```
scaler.transform(valid_df[features])
scaler.transform(test_df[features])
```

- Uses the same μ and σ from training data to transform new data.

3 Implementation of MLP

A Multi-Layer Perceptron (MLP) is a type of artificial neural network made up of layers of neurons. It consists of an input layer, one or more hidden layers, and an output layer, with each neuron applying weights and activation functions to learn patterns. MLPs are commonly used for classification and regression tasks in machine learning.

- The choice of programming language and libraries,
- The structure of the implemented neural network,
- The coding process with detailed comments.

3.1 Choice of Programming Langauges

Here I'll explain why I chose Python and the corresponding libraries to build my model.

Python has a easy to read syntax and a set of extensive libraries and frameworks which I can use.

Here are the ones I used:

- **pandas**:
  - Handles data manipulation and preprocessing efficiently.
  - Used to load, clean, and structure datasets before feeding them into an ANN.
  - Provides powerful tools for handling missing values and feature engineering.
- **seaborn**:
  - Used for visualising data distributions, relationships, and correlations.
  - Helps in understanding feature importance and patterns before training an ANN.
- **matplotlib**:
  - Essential for plotting learning curves, loss functions, and model performance metrics.
  - Used to visualize weight updates, activation functions
  - Helps in debugging by analysing trends in ANN training and validation.

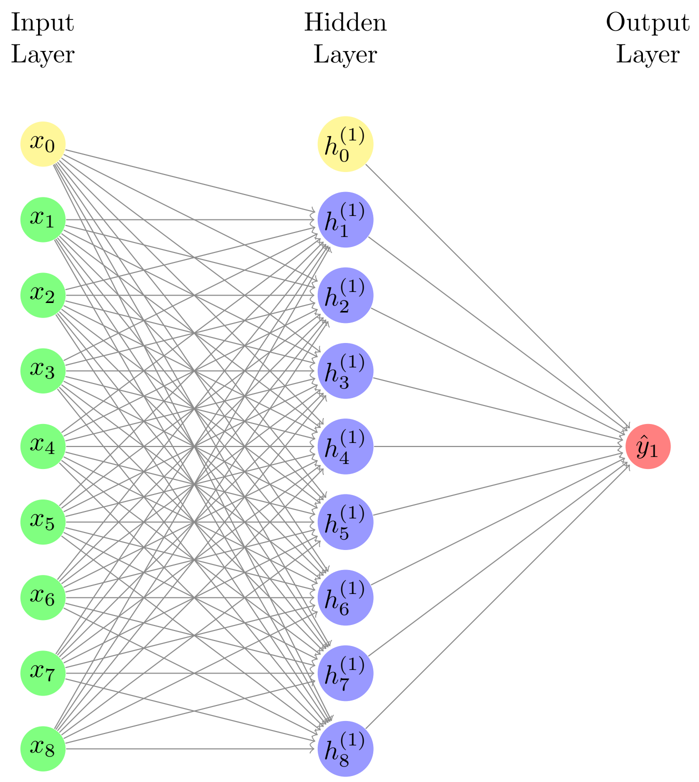
- **itertools**:
  - Useful for generating all possible combinations of hyperparameters in ANN tuning.
  - Helps in efficiently iterating through different activation functions, optimizers, and architectures.
  - Reduces manual work when experimenting with different network configurations.
- **sklearn.preprocessing**:
  - Standardises input features by scaling them to have zero mean and unit variance.
  - Essential for ANNs, as it helps in faster convergence by ensuring uniform feature distribution.
  - Improves model stability and performance by preventing dominance of large-scale features.

3.2 Implementation

3.2.1 Neural Network Architecture

The implemented neural network consists of an input layer with 8 features, a single hidden layer, and an output layer. The decision to use only one hidden layer was made to keep the architecture simple while still allowing for non-linear transformations of the data.

This is just a sample model and not the final. I just wanted to show what a model of the Neural Network might look like.



3.2.2 Activation Function: Leaky ReLU

The Leaky ReLU activation function was chosen for the hidden layer instead of standard ReLU to prevent the issue of dying neurons (where some neurons stop updating their weights because their gradients become zero). Unlike ReLU, which sets negative inputs to zero, Leaky ReLU allows a small negative slope, ensuring that neurons continue learning even for negative inputs. This helps improve the model's ability to learn complex patterns.

```python
def leaky_relu(x, alpha=0.01):
    return np.where(x > 0, x, alpha * x)

def leaky_relu_derivative(x, alpha=0.01):
    return np.where(x > 0, 1, alpha)
```

Leaky ReLU allows small gradients for negative inputs instead of squashing them into near zero values. This is what's done by the sigmoid and tanh functions ([0,1] for Sigmoid, [-1,1] for tanh).

During backpropagation, gradients become extremely small for large or small inputs.This slows down learning because weight updates shrink to nearly zero.

Sigmoid centers activations around 0.5, which can slow learning because weights need constant adjustments.

Tanh centers activations around 0, which helps but still suffers from vanishing gradients for large inputs.

Leaky ReLU allows a wider range of activations and prevents neurons from becoming inactive, leading to better training stability.

Sigmoid and Tanh require expensive exponential calculations:

$$\text{Sigmoid}(x) = \frac{1}{1+e^{-x}}, \quad \text{Tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Leaky ReLU only involves a simple conditional check:

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha x, & \text{otherwise} \end{cases}$$

This makes Leaky ReLU much faster to compute, especially in large networks.

3.2.3 Weight Initialization

He Initialization is used because it is optimized for ReLU-based activations and helps with stable weight updates.

He Initialization sets the initial weights of a neural network using a variance-scaling approach to maintain stable activations. It draws weights from a normal distribution with mean 0 and variance $\frac{2}{n_{in}}$, where n_{in} is the number of input neurons. This ensures that the variance of activations remains consistent across layers, preventing vanishing or exploding gradients. The method is particularly effective for ReLU and its variants, as they discard negative values, making a higher variance necessary for balanced learning.

```python
def he_initialization(input_size, output_size):
    limit = np.sqrt(2 / input_size)  # He initialization formula
    return np.random.randn(input_size, output_size) * limit
```

3.2.4 Mathematics behind MLP

Let the input to the network be represented as a vector:

$$\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$$

where x_i are the features of the input. The network consists of two layers: the hidden layer and the output layer.

The first layer of the MLP is the hidden layer. Let the number of neurons in the hidden layer be h. Each neuron in the hidden layer computes a weighted sum of the inputs, followed by the application of an activation function. The output of the hidden layer is given by:

$$\mathbf{z} = \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$$

- - $\mathbf{W}_1$ is the weight matrix of size $h \times n$,
- - $\mathbf{b}_1$ is the bias vector of size h,
- - σ is the activation function, such as the sigmoid or ReLU.

The second layer is the output layer, where the output is computed by a weighted sum of the hidden layer outputs:

$$\mathbf{y} = \mathbf{W}_2\mathbf{z} + \mathbf{b}_2$$

- $\mathbf{W}_2$ is the weight matrix of size $m \times h$, where m is the number of output neurons,
- $\mathbf{b}_2$ is the bias vector of size m.

The output $\mathbf{y}$ is typically passed through an activation function such as the softmax (for classification tasks) or identity (for regression tasks).

In an MLP with one hidden layer, the mathematical flow is as follows:

$$\mathbf{z} = \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$$

$$\mathbf{y} = \mathbf{W}_2\mathbf{z} + \mathbf{b}_2$$

These computations are learned during training by adjusting the weights $\mathbf{W}_1, \mathbf{W}_2$ and biases $\mathbf{b}_1, \mathbf{b}_2$ using backpropagation and gradient descent.

3.3 Iterative Developments

3.3.1 Baseline MLP

A basic version of the model was implemented. This version included backpropagation with no additional improvements.

This is the Neural Network class I used to build the model, along with a training function to train it.

```
class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        """Initialize the neural network with He-initialized weights and zero biases."""

        self.weights_input_hidden = he_initialization(input_size, hidden_size)
        self.bias_hidden = np.zeros((1, hidden_size))
        self.weights_hidden_output = he_initialization(hidden_size, output_size)
        self.bias_output = np.zeros((1, output_size))

    def forward(self, X):
        """Perform a forward pass through the network."""
        self.input_layer = X

        # Compute hidden layer activation
        self.hidden_layer_sum =
        np.dot(self.input_layer, self.weights_input_hidden) + self.bias_hidden
        self.hidden_layer = leaky_relu(self.hidden_layer_sum)

        # Compute output layer activation
        self.output_layer_sum =
        np.dot(self.hidden_layer, self.weights_hidden_output) + self.bias_output
        self.output_layer = leaky_relu(self.output_layer_sum)

        return self.output_layer

    def backward(self, X, y, learning_rate):
        """Perform backpropagation to compute gradients."""
        # Compute error (difference between predicted and actual values)
        output_error = mse_loss_derivative(y, self.output_layer)
        output_delta = output_error * leaky_relu_derivative(self.output_layer_sum)

        # Compute error propagated back to the hidden layer
        hidden_error = output_delta.dot(self.weights_hidden_output.T)
        hidden_delta = hidden_error * leaky_relu_derivative(self.hidden_layer_sum)

        # Compute gradients for updating weights and biases
        grad_weights_input_hidden = X.T.dot(hidden_delta)
        grad_bias_hidden = np.sum(hidden_delta, axis=0, keepdims=True)
        grad_weights_hidden_output = self.hidden_layer.T.dot(output_delta)
        grad_bias_output = np.sum(output_delta, axis=0, keepdims=True)

        return grad_weights_input_hidden,
        grad_bias_hidden, grad_weights_hidden_output,
        grad_bias_output

    def train(self, X_train, y_train_scaled,
    X_valid, y_valid_scaled, scaler_Y,
    epochs, learning_rate):
        """Train the neural network using gradient descent."""
        for epoch in range(epochs):
            # Forward pass: predict output
            y_pred_scaled = self.forward(X_train)

            # Compute gradients for weights and biases using backpropagation
            grad_weights_input_hidden,
            grad_bias_hidden,
            grad_weights_hidden_output,
            grad_bias_output = self.backward(X_train, y_train_scaled, learning_rate)

            # Update weights and biases using the computed gradients
            self.weights_input_hidden -=
            grad_weights_input_hidden * learning_rate  # Adjusting weights
            self.bias_hidden -=
            grad_bias_hidden * learning_rate  # Adjusting biases
            self.weights_hidden_output -=
            grad_weights_hidden_output * learning_rate
            self.bias_output -=
            grad_bias_output * learning_rate

            # Print training error every 100 epochs to track progress
            if epoch % 100 == 0:
                y_pred_real = scaler_Y.inverse_transform(y_pred_scaled)
                # Convert predictions back to original scale
                y_train_real = scaler_Y.inverse_transform(y_train_scaled)
                train_error = mse_loss(y_train_real, y_pred_real)
                # Compute loss

                print(f"Epoch {epoch}, Train Error: {train_error:.6f}")

        return y_pred_real, y_train_real
```

Training Function

```
 def train_and_predict(file_path):

    # Step 1: Load and preprocess the dataset
    # The function `split_and_scale_data` handles data loading,
    # splitting, and scaling
    X_train, X_valid, X_test, Y_train, Y_valid, Y_test = split_and_scale_data(file_path)

    # Step 2: Convert the data to NumPy arrays (required for matrix operations)
    X_train, X_valid, X_test = X_train.to_numpy(), X_valid.to_numpy(), X_test.to_numpy()

    # Reshape Y values to ensure they have the correct format
    Y_train = Y_train.to_numpy().reshape(-1, 1)
    Y_valid = Y_valid.to_numpy().reshape(-1, 1)
    Y_test = Y_test.to_numpy().reshape(-1, 1)

    # Step 3: Standardize the target variable (Y) using StandardScaler
    # This ensures that all target values
    # have a mean of 0 and a standard deviation of 1
    scaler_Y = StandardScaler()
    Y_train_scaled = scaler_Y.fit_transform(Y_train)
    Y_valid_scaled = scaler_Y.transform(Y_valid)
    Y_test_scaled = scaler_Y.transform(Y_test)

    # Step 4: Initialize the neural network
    # The input size is determined by the number of features in X_train
    input_size = X_train.shape[1]  # Number of features in the dataset
    nn = NeuralNetwork(input_size=input_size, hidden_size=12, output_size=1)

    # Step 5: Train the neural network
    # The training function updates the model's weights using gradient descent
    nn.train(X_train, Y_train_scaled, X_valid, Y_valid_scaled,
    scaler_Y, epochs=40000, learning_rate=0.1)

    # Step 6: Evaluate the trained model
    # This function plots the predicted values vs. actual values to assess performance
    plot_predictions_and_correlation(nn, X_test, Y_test_scaled, scaler_Y)

# Specify the file path to your dataset
file_path = "Ouse93-96-Student.xlsx"

# Call the function to train the model and plot predictions
train_and_predict(file_path)
```

This code defines a simple neural network class with forward and backward propagation methods. Here's a brief breakdown of each part:

- Initialization (__init__): The network is initialized with random weights using He initialization and zero biases for both the hidden and output layers.
- Forward Pass (forward): The method computes the activations of the hidden and output layers using matrix multiplication, biases, and the leaky ReLU activation function.
- Backward Pass (backward): It computes gradients for weights and biases through backpropagation using the chain rule. This involves computing the error at the output, propagating it back to the hidden layer, and calculating the gradients.
- Training (train): The network is trained using gradient descent. It performs forward and backward passes for a specified number of epochs and updates the weights and biases based on the computed gradients.
- Error Tracking: Every 100 epochs, the training error is printed to monitor the model's progress, converting predictions back to the original scale using the scaler_Y.

This structure is typical for a neural network model with a hidden layer, using basic gradient descent for training and leaky ReLU for activation.

3.3.2 Momentum

$$v_t = \beta v_{t-1} + (1-\beta)\nabla_\theta J(\theta)$$
$$\theta_t = \theta_{t-1} - \alpha v_t$$

where:

- v_t is the velocity (momentum term),
- β is the momentum factor (typically close to 1, e.g., 0.9),
- $\nabla_\theta J(\theta)$ is the gradient of the loss function with respect to the weights,
- α is the learning rate,
- θ represents the model parameters.

Adding a momentum term to gradient descent leads to more rapid descent along the direction you're going in compared with unmodified gradient descent. Allows for faster convergence meaning less epochs required to train model. It also helps smooth out weight updates

Here are the changes I made into the original MLP to add momentum.

```
def __init__(self, input_size, hidden_size, output_size, momentum=0.9):
    self.weights_input_hidden =
    he_initialization(input_size, hidden_size)
    self.bias_hidden = np.zeros((1, hidden_size))
    self.weights_hidden_output =
    he_initialization(hidden_size, output_size)
    self.bias_output = np.zeros((1, output_size))

    # Initialise momentum terms
    self.m_wih = np.zeros_like(self.weights_input_hidden)
    self.m_bh = np.zeros_like(self.bias_hidden)
    self.m_who = np.zeros_like(self.weights_hidden_output)
    self.m_bo = np.zeros_like(self.bias_output)

    self.momentum = momentum


def backward(self, X, y, learning_rate):
    output_error = rmse_loss_derivative(y, self.output_layer)
    output_delta =
    output_error * leaky_relu_derivative(self.output_layer_sum)
    hidden_error = output_delta.dot(self.weights_hidden_output.T)
    hidden_delta =
    hidden_error * leaky_relu_derivative(self.hidden_layer_sum)

    grad_wih = X.T.dot(hidden_delta)
    grad_bh = np.sum(hidden_delta, axis=0, keepdims=True)
    grad_who = self.hidden_layer.T.dot(output_delta)
    grad_bo = np.sum(output_delta, axis=0, keepdims=True)

    # Apply momentum
    self.m_wih =
    self.momentum * self.m_wih - learning_rate * grad_wih
    self.m_bh =
    self.momentum * self.m_bh - learning_rate * grad_bh
    self.m_who =
    self.momentum * self.m_who - learning_rate * grad_who
    self.m_bo =
    self.momentum * self.m_bo - learning_rate * grad_bo

    # Update weights and biases
    self.weights_input_hidden += self.m_wih
    self.bias_hidden += self.m_bh
    self.weights_hidden_output += self.m_who
    self.bias_output += self.m_bo
```

$$J_{\text{regularized}}(\theta) = J(\theta) + \frac{\lambda}{2}\|\theta\|^2$$

where:

- $J(\theta)$ is the original loss function,
- λ is the weight decay coefficient (regularization parameter),
- $\|\theta\|^2$ is the L2 norm of the weights (i.e., the sum of squared weights).

3.3.3 Weight Decay

$$J_{\text{regularized}}(\theta) = J(\theta) + \frac{\lambda}{2}\|\theta\|^2$$

where:

- $J(\theta)$ is the original loss function,
- λ is the weight decay coefficient (regularization parameter),
- $\|\theta\|^2$ is the L2 norm of the weights (i.e., the sum of squared weights).

Weight decay is implemented to prevent overfitting by penalizing large weights. This is achieved by adding a regularization term, typically L2 regularization, to the loss function. By discouraging large weight magnitudes, weight decay helps improve generalization, making the model perform better on unseen data.
Here are the changes I made into the original MLP to add weight decay.

```
def __init__(self, input_size, hidden_size, output_size,weight_decay=0.01):
    """Initialize the neural network with He-initialized weights and zero biases."""
    self.weights_input_hidden =
    he_initialization(input_size, hidden_size)
    self.bias_hidden = np.zeros((1, hidden_size))
    self.weights_hidden_output =
    he_initialization(hidden_size, output_size)
    self.bias_output = np.zeros((1, output_size))

    self.weight_decay = weight_decay  # L2 Regularization Strength

def backward(self, X, y, learning_rate):
    """Perform backpropagation to compute gradients."""
    # Compute error (difference between predicted and actual values)
    output_error =
    mse_loss_derivative(y, self.output_layer)
    output_delta =
    output_error * leaky_relu_derivative(self.output_layer_sum)
```

```
        # Compute error propagated back to the hidden layer
        hidden_error =
        output_delta.dot(self.weights_hidden_output.T)
        hidden_delta =
        hidden_error * leaky_relu_derivative(self.hidden_layer_sum)

        # Compute gradients for updating weights and biases
        grad_weights_input_hidden = X.T.dot(hidden_delta)
        + self.weight_decay * self.weights_input_hidden
        # L2 Regularization
        grad_bias_hidden = np.sum(hidden_delta, axis=0, keepdims=True)
        grad_weights_hidden_output =
        self.hidden_layer.T.dot(output_delta)
        + self.weight_decay * self.weights_hidden_output
        # L2 Regularization
        grad_bias_output = np.sum(output_delta, axis=0, keepdims=True)

        return grad_weights_input_hidden,
        grad_bias_hidden,
        grad_weights_hidden_output,
        grad_bias_output
```

3.3.4 Bold Driver

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta J(\theta)$$

where:

- η is the learning rate, adjusted dynamically depending on whether the loss decreases or increases.

The bold driver technique dynamically adjusts the learning rate based on loss changes. If the loss decreases, the learning rate is increased to speed up convergence; if the loss increases, the learning rate is reduced to avoid divergence.
Here are the changes I made into the original MLP to add bold driver.

```
    def train(self, X_train, y_train_scaled,
    X_valid, y_valid_scaled, scaler_Y, epochs, learning_rate):
        prev_loss = float('inf')
        loss_history = []  # Store loss values

        for epoch in range(epochs):
            self.forward(X_train)
            self.backward(X_train, y_train_scaled, learning_rate)
            if epoch % 100 == 0:
                y_pred_real = scaler_Y.inverse_transform(self.output_layer)
                y_train_real = scaler_Y.inverse_transform(y_train_scaled)
                train_error = rmse_loss(y_train_real, y_pred_real)
                loss_history.append(train_error)  # Save loss
                print(f"Epoch {epoch},
                Train Error: {train_error:.6f},
                LR: {learning_rate:.6f}")

                if train_error < prev_loss:
                    learning_rate *= 1.05
                else:
                    learning_rate *= 0.7

                prev_loss = train_error

        return y_pred_real, y_train_real
```

3.3.5 Annealing

$$\text{learning\_rate}(x) = p + (q - p)\left(1 - \frac{1}{1 + \exp\left(\frac{10-20x}{r}\right)}\right)$$

where:

- p is the initial learning rate,
- q is the final learning rate,
- r is the maximum number of epochs (often the total number of training iterations),
- x is the current epoch or iteration.

Annealing is applied to gradually decrease the learning rate over time. By starting with a higher learning rate and progressively lowering it, the model can make significant initial progress while refining its parameters in later stages. This approach helps in fine-tuning the model and achieving a better final solution.
Here are the changes I made into the original MLP to add simulated annealing.

```
    def annealing_function(x, p=0.01, q=0.1, r=40000):  # r = max epochs
        """Learning rate annealing function"""
        return p + (q - p) * (1 - 1 / (1 + np.exp((10 - 20 * x) / r)))

    def train(self, X_train, y_train_scaled, X_valid, y_valid_scaled, scaler_Y, epochs):
        for epoch in range(epochs):
            learning_rate = annealing_function(epoch)
            # Get adaptive learning rate
            self.forward(X_train)
            self.backward(X_train, y_train_scaled, learning_rate)
            if epoch % 100 == 0:
                y_pred_real = scaler_Y.inverse_transform(self.output_layer)
                y_train_real = scaler_Y.inverse_transform(y_train_scaled)
                train_error = rmse_loss(y_train_real, y_pred_real)
                print(f"Epoch {epoch},
                Learning Rate: {learning_rate:.6f},
                Train Error: {train_error:.6f}")

        return y_pred_real, y_train_real
```

3.3.6 Mini-Batch Training

Mini-batch training is introduced to improve generalization and training stability. Instead of updating weights after every sample or after the entire dataset , mini-batch training updates weights using small subsets of data. This allows greater computational efficiency and convergence speed while reducing variance in gradient updates.
Here are the changes I made into the original MLP to add mini-batch training.

```
    def train(self, X_train,
    y_train_scaled, X_valid, y_valid_scaled,
    scaler_Y, epochs, learning_rate, batch_size=16):
        num_samples = X_train.shape[0]

        for epoch in range(epochs):
            indices = np.random.permutation(num_samples)
            X_train_shuffled = X_train[indices]
            y_train_shuffled = y_train_scaled[indices]

            for i in range(0, num_samples, batch_size):
                X_batch = X_train_shuffled[i:i+batch_size]
                y_batch = y_train_shuffled[i:i+batch_size]

                self.forward(X_batch)
                self.backward(X_batch, y_batch, learning_rate)

            if epoch % 100 == 0:
                y_pred_real = scaler_Y.inverse_transform(self.forward(X_train))
                y_train_real = scaler_Y.inverse_transform(y_train_scaled)
                train_error = rmse_loss(y_train_real, y_pred_real)
```

```
loss_history.append(train_error)

print(f"Epoch {epoch},
Train Error: {train_error:.6f},
LR: {learning_rate:.6f}")
```

4 Training and Network Selection

The training process involves tuning the ANN model to improve performance. This section discusses:

- Selecting an appropriate loss function,
- Backpropagation improvements and optimisation.
- Configuring hyperparameters such as learning rate and batch size,

4.1 Loss Function

RMSE (Root Mean Squared Error) is used because it penalizes large errors more and is suitable for regression problems.

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

RMSE Derivative:

$$RMSE' = \frac{-(y_i - \hat{y}_i)}{n * RMSE}$$

4.2 Mini-Batch Optimization with Momentum

All the models have 8 hidden nodes in the hidden layer as a standard way to compare them all.

4.2.1 Mini-Batch + Momentum

Momentum helps accelerate gradient descent in relevant directions, leading to faster convergence and reduced oscillations.

```
class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size, momentum=0.9):
        # Initialize weights and biases
        self.weights_input_hidden =
        he_initialisation(input_size, hidden_size)
        self.bias_hidden = np.zeros((1, hidden_size))
        self.weights_hidden_output =
        he_initialisation(hidden_size, output_size)
        self.bias_output = np.zeros((1, output_size))
        # Initialize velocity for momentum
        self.velocity_wih = np.zeros_like(self.weights_input_hidden)
        self.velocity_bh = np.zeros_like(self.bias_hidden)
        self.velocity_who = np.zeros_like(self.weights_hidden_output)
        self.velocity_bo = np.zeros_like(self.bias_output)

        self.momentum = momentum

    def forward(self, X):
        self.input_layer = X
        self.hidden_layer_sum =
        np.dot(self.input_layer, self.weights_input_hidden) + self.bias_hidden
        self.hidden_layer =
        leaky_relu(self.hidden_layer_sum)
        self.output_layer_sum =
        np.dot(self.hidden_layer, self.weights_hidden_output) + self.bias_output
        self.output_layer =
        leaky_relu(self.output_layer_sum)
        return self.output_layer

    def backward(self, X, y, learning_rate):
        output_error =
        rmse_loss_derivative(y, self.output_layer)
        output_delta =
        output_error * leaky_relu_derivative(self.output_layer_sum)
        hidden_error =
        output_delta.dot(self.weights_hidden_output.T)
        hidden_delta =
        hidden_error * leaky_relu_derivative(self.hidden_layer_sum)

        grad_wih = X.T.dot(hidden_delta)
        grad_bh = np.sum(hidden_delta, axis=0, keepdims=True)
        grad_who = self.hidden_layer.T.dot(output_delta)
        grad_bo = np.sum(output_delta, axis=0, keepdims=True)

        # Update velocities with momentum
        self.velocity_wih =
        self.momentum * self.velocity_wih + (1 - self.momentum) * grad_wih
        self.velocity_bh =
        self.momentum * self.velocity_bh + (1 - self.momentum) * grad_bh
        self.velocity_who =
        self.momentum * self.velocity_who + (1 - self.momentum) * grad_who
        self.velocity_bo =
        self.momentum * self.velocity_bo + (1 - self.momentum) * grad_bo
        # Update weights and biases using velocity
        self.weights_input_hidden -= learning_rate * self.velocity_wih
        self.bias_hidden -= learning_rate * self.velocity_bh
        self.weights_hidden_output -= learning_rate * self.velocity_who
        self.bias_output -= learning_rate * self.velocity_bo

    def train(self, X_train, y_train_scaled,
    X_valid, y_valid_scaled, scaler_Y, epochs,
    learning_rate, batch_size=16):
        prev_loss = float('inf')
        loss_history = []
        val_loss_history = []  # History of validation losses
        num_samples = X_train.shape[0]

        for epoch in range(epochs):
            indices = np.random.permutation(num_samples)
            X_train_shuffled = X_train[indices]
            y_train_shuffled = y_train_scaled[indices]

            for i in range(0, num_samples, batch_size):
                X_batch = X_train_shuffled[i:i+batch_size]
                y_batch = y_train_shuffled[i:i+batch_size]

                self.forward(X_batch)
                self.backward(X_batch, y_batch, learning_rate)

            if epoch % 100 == 0:
                y_pred_real =
                scaler_Y.inverse_transform(self.forward(X_train))
                y_train_real =
                scaler_Y.inverse_transform(y_train_scaled)
                train_error =
                rmse_loss(y_train_real, y_pred_real)

                # Validation loss calculation
                y_valid_pred_real =
                scaler_Y.inverse_transform(self.forward(X_valid))
                y_valid_real =
                scaler_Y.inverse_transform(y_valid_scaled)
                val_error = rmse_loss(y_valid_real, y_valid_pred_real)

                loss_history.append(train_error)
                val_loss_history.append(val_error)

                print(f"Epoch {epoch},
                Train Error: {train_error:.6f},
                Validation Error: {val_error:.6f},
                LR: {learning_rate:.6f}")

        return y_pred_real, y_train_real
```

4.2.2 Mini-Batch + Momentum + Bold Driver

The bold driver dynamically adjusts the learning rate based on progress, allowing faster convergence while avoiding stagnation or divergence.

The only change is adding bold driver to the train method of the Neural Network Class.

```
def train(self, X_train, y_train_scaled,
X_valid, y_valid_scaled,
scaler_Y, epochs, learning_rate, batch_size=16):
    prev_loss = float('inf')
    loss_history = []
    val_loss_history = []  # History of validation losses
    num_samples = X_train.shape[0]

    for epoch in range(epochs):
        indices = np.random.permutation(num_samples)
        X_train_shuffled = X_train[indices]
        y_train_shuffled = y_train_scaled[indices]

        for i in range(0, num_samples, batch_size):
            X_batch = X_train_shuffled[i:i+batch_size]
            y_batch = y_train_shuffled[i:i+batch_size]

            self.forward(X_batch)
            self.backward(X_batch, y_batch, learning_rate)

        if epoch % 100 == 0:
            y_pred_real = scaler_Y.inverse_transform(self.forward(X_train))
            y_train_real = scaler_Y.inverse_transform(y_train_scaled)
            train_error = rmse_loss(y_train_real, y_pred_real)

            # Validation loss calculation
            y_valid_pred_real = scaler_Y.inverse_transform(self.forward(X_valid))
            y_valid_real = scaler_Y.inverse_transform(y_valid_scaled)
            val_error = rmse_loss(y_valid_real, y_valid_pred_real)

            loss_history.append(train_error)
            val_loss_history.append(val_error)
```

```
                print(f"Epoch {epoch},
                Train Error: {train_error:.6f},
                Validation Error: {val_error:.6f},
                LR: {learning_rate:.6f}")

                if train_error < prev_loss:
                    learning_rate *= 1.05
                else:
                    learning_rate *= 0.85

                prev_loss = train_error

        return y_pred_real, y_train_real
```

4.2.3 Mini-Batch + Momentum + Bold Driver + Weight Decay

Weight decay helps prevent overfitting by penalising large weights, enhancing generalisation in addition to the benefits of momentum and adaptive learning rates.
The only change is adding weight decay to the backward method of the Neural Network Class.

```
    def backward(self, X, y, learning_rate):

        lambda_wd = 0.0001  # Weight decay coefficient

        output_error = rmse_loss_derivative(y, self.output_layer)
        output_delta = output_error * leaky_relu_derivative(self.output_layer_sum)
        hidden_error = output_delta.dot(self.weights_hidden_output.T)
        hidden_delta = hidden_error * leaky_relu_derivative(self.hidden_layer_sum)

        grad_wih = X.T.dot(hidden_delta)
        grad_bh = np.sum(hidden_delta, axis=0, keepdims=True)
        grad_who = self.hidden_layer.T.dot(output_delta)
        grad_bo = np.sum(output_delta, axis=0, keepdims=True)

        # Update velocities with momentum
        # Update with both momentum and weight decay
        self.velocity_wih =
        self.momentum * self.velocity_wih + (1 - self.momentum) *
        (grad_wih + lambda_wd * self.weights_input_hidden)
        self.velocity_who =
        self.momentum * self.velocity_who + (1 - self.momentum) *
        (grad_who + lambda_wd * self.weights_hidden_output)
        self.velocity_bo =
        self.momentum * self.velocity_bo + (1 - self.momentum) * grad_bo
        self.velocity_bh =
        self.momentum * self.velocity_bh + (1 - self.momentum) * grad_bh

        # Update weights and biases using velocity
        self.weights_input_hidden -= learning_rate * self.velocity_wih
        self.weights_hidden_output -= learning_rate * self.velocity_who
        self.bias_output -= learning_rate * self.velocity_bo
        self.bias_hidden -= learning_rate * self.velocity_bh
```

4.3 Bold Driver and Weight Decay Optimization

4.3.1 Bold Driver + Weight Decay + Momentum

Combining weight decay and momentum ensures both generalisation and faster convergence by balancing weight regularisation with acceleration.

```
     def __init__(self, input_size, hidden_size, output_size,
     momentum=0.9, weight_decay=0.01):
        self.weights_input_hidden = he_initialization(input_size, hidden_size)
        self.bias_hidden = np.zeros((1, hidden_size))
        self.weights_hidden_output = he_initialization(hidden_size, output_size)
        self.bias_output = np.zeros((1, output_size))

        # Initialize velocity terms for momentum
        self.v_wih = np.zeros_like(self.weights_input_hidden)
        self.v_bh = np.zeros_like(self.bias_hidden)
        self.v_who = np.zeros_like(self.weights_hidden_output)
        self.v_bo = np.zeros_like(self.bias_output)

        self.momentum = momentum
        self.weight_decay = weight_decay  # L2 Regularization Strength

    def backward(self, X, y, learning_rate):
        output_error = rmse_loss_derivative(y, self.output_layer)
        output_delta = output_error * leaky_relu_derivative(self.output_layer_sum)
        hidden_error = output_delta.dot(self.weights_hidden_output.T)
        hidden_delta = hidden_error * leaky_relu_derivative(self.hidden_layer_sum)

        grad_wih = X.T.dot(hidden_delta) + self.weight_decay *
        self.weights_input_hidden
        # L2 Regularization
        grad_bh = np.sum(hidden_delta, axis=0, keepdims=True)
        grad_who = self.hidden_layer.T.dot(output_delta)
        + self.weight_decay * self.weights_hidden_output
        grad_bo = np.sum(output_delta, axis=0, keepdims=True)
        # Apply momentum
        self.v_wih = self.momentum * self.v_wih - learning_rate * grad_wih
        self.v_bh = self.momentum * self.v_bh - learning_rate * grad_bh
        self.v_who = self.momentum * self.v_who - learning_rate * grad_who
        self.v_bo = self.momentum * self.v_bo - learning_rate * grad_bo

        # Update weights and biases
        self.weights_input_hidden += self.v_wih
        self.bias_hidden += self.v_bh
        self.weights_hidden_output += self.v_who
        self.bias_output += self.v_bo

    def train(self, X_train, y_train_scaled,
    X_valid, y_valid_scaled, scaler_Y, epochs,
    learning_rate):
        prev_loss = float('inf')  # Initialize prev_loss outside of the loop

        # List to track training errors for plotting
        train_errors = []

        for epoch in range(epochs):
            # Perform a forward pass
            self.forward(X_train)
            # Perform a backward pass and weight updates
            self.backward(X_train, y_train_scaled, learning_rate)

            # Print and adjust learning rate every 100 epochs
            if epoch % 100 == 0:
                # Get predicted values on the validation set (scaled)
                y_pred_real = scaler_Y.inverse_transform(self.output_layer)
                y_train_real = scaler_Y.inverse_transform(y_train_scaled)

                # Calculate RMSE loss
                train_error = rmse_loss(y_train_real, y_pred_real)
                print(f"Epoch {epoch},
                Train Error: {train_error:.6f},
                LR: {learning_rate:.6f}")
                train_errors.append(train_error)

                # Adjust the learning rate based on previous loss
                if train_error < prev_loss:
                    learning_rate *= 1.05
                else:
                    learning_rate *= 0.7
                # Update prev_loss to the current train_error for the next iteration
                prev_loss = train_error

        return y_pred_real, y_train_real, train_errors
```

4.3.2 Bold Driver + Weight Decay + No Momentum

Using bold driver and weight decay without momentum ensures adaptive learning rates and regularisation without potential over-reliance on past gradients.

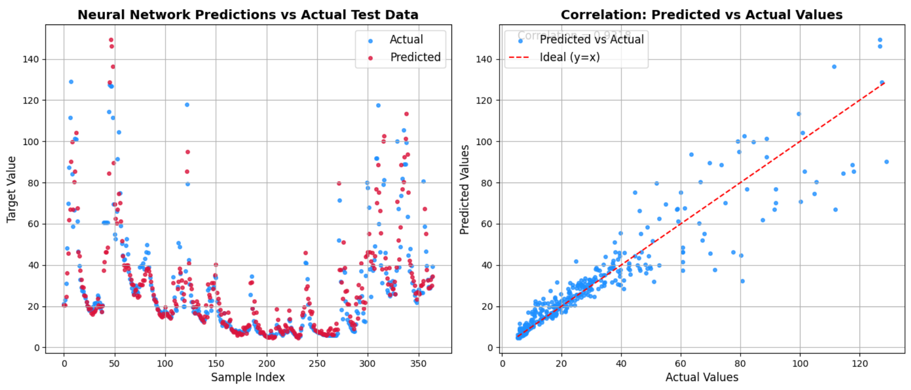
```
def __init__(self, input_size, hidden_size, output_size, weight_decay=0.01):
    self.weights_input_hidden = he_initialization(input_size, hidden_size)
    self.bias_hidden = np.zeros((1, hidden_size))
    self.weights_hidden_output = he_initialization(hidden_size, output_size)
    self.bias_output = np.zeros((1, output_size))
    self.weight_decay = weight_decay  # L2 Regularization Strength

def backward(self, X, y, learning_rate):
    output_error = rmse_loss_derivative(y, self.output_layer)
    output_delta = output_error * leaky_relu_derivative(self.output_layer_sum)
    hidden_error = output_delta.dot(self.weights_hidden_output.T)
    hidden_delta = hidden_error * leaky_relu_derivative(self.hidden_layer_sum)

    grad_wih = X.T.dot(hidden_delta) +
    self.weight_decay * self.weights_input_hidden
    # L2 Regularization
    grad_bh = np.sum(hidden_delta, axis=0, keepdims=True)
    grad_who = self.hidden_layer.T.dot(output_delta)
    + self.weight_decay * self.weights_hidden_output
    grad_bo = np.sum(output_delta, axis=0, keepdims=True)

    # Update weights and biases
    self.weights_input_hidden -= learning_rate * grad_wih
    self.bias_hidden -= learning_rate * grad_bh
    self.weights_hidden_output -= learning_rate * grad_who
    self.bias_output -= learning_rate * grad_bo
```

4.4 Comparing the Models

Model	**Train Error**	**Validation Error**	**Correlation**
Mini-Batch + Momentum	5.58	7.51	0.936
Mini-Batch + Bold Driver + Weight Decay + Momentum	**5.97**	**7.73**	**0.938**
Mini-Batch + Bold Driver + Momentum	5.78	7.47	0.936
Bold Driver with Momentum	5.94	7.43	0.932
Bold Driver with No Momentum	5.38	9.79	0.892

Table 4: Comparison of different training configurations

The best model is "Mini-Batch + Bold Driver + Weight Decay + Momentum". This model has the highest correlation (0.938), which is a strong indicator of predictive performance. It also includes Weight Decay to reduce overfitting and ensure better generalization. The validation error is slightly higher than others, but it is a balanced model with good regularization and high predictive accuracy.



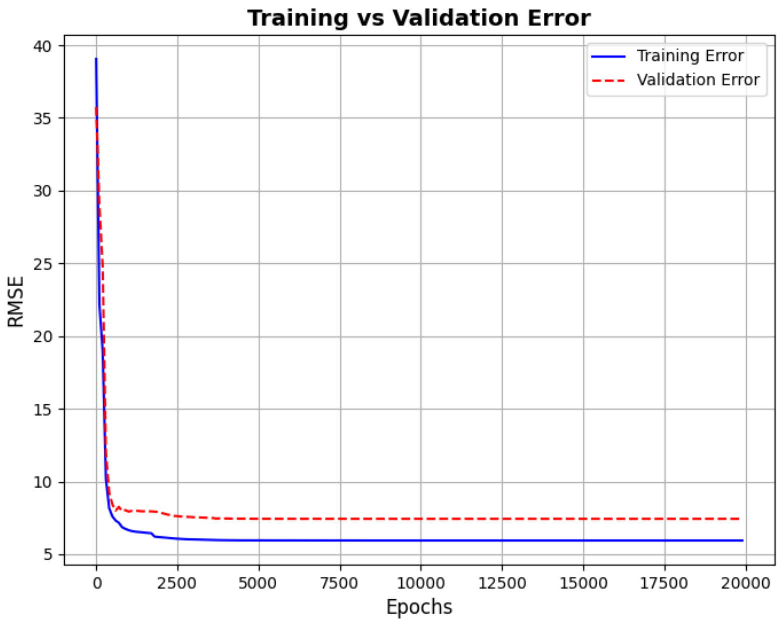
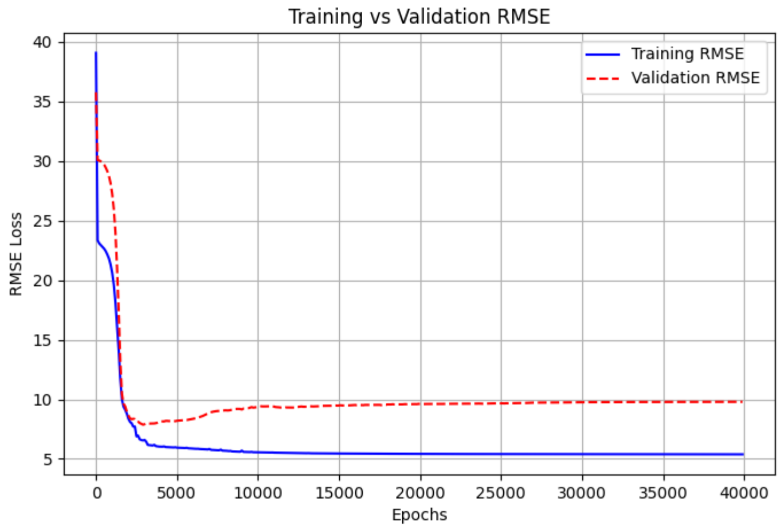


Figure 2: Bold Driver with momentum



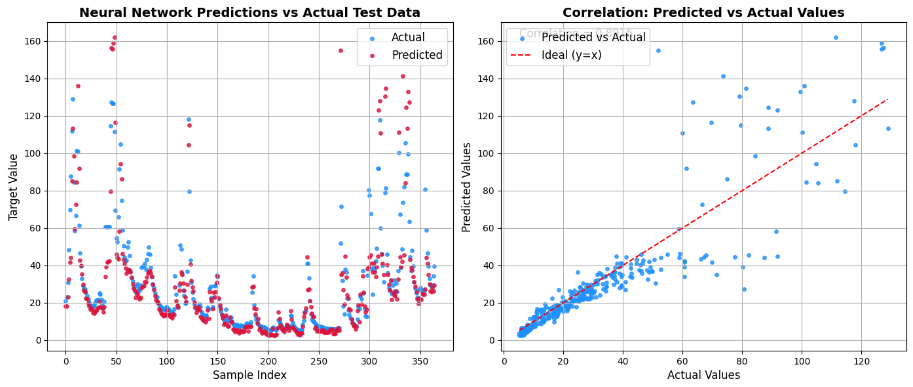


Figure 3: Bold Driver with no momentum

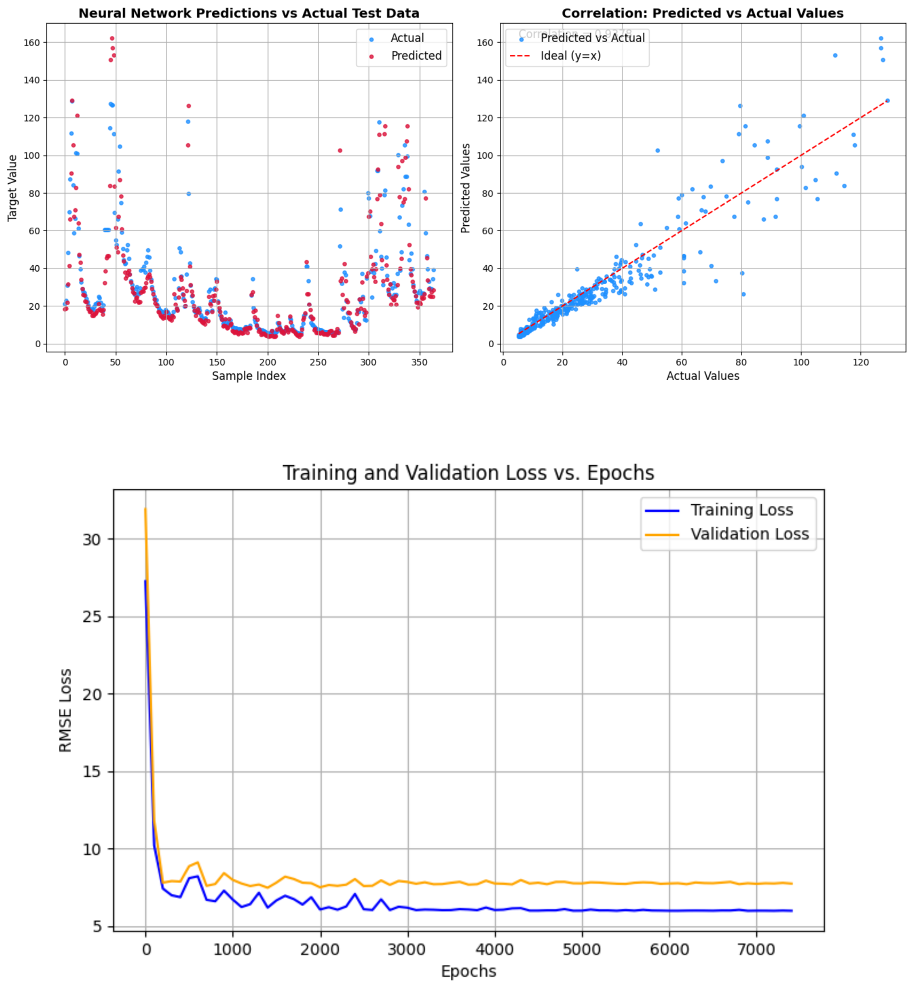
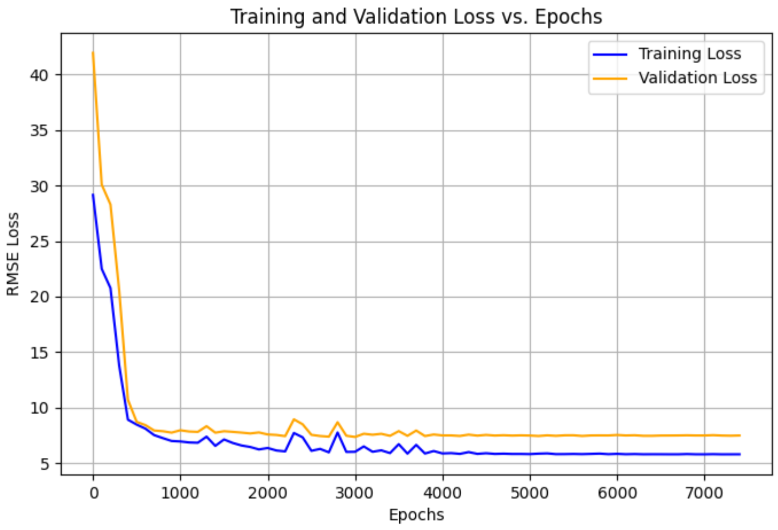


Figure 4: Mini-Batch + Bold driver + Weight decay + Momentum



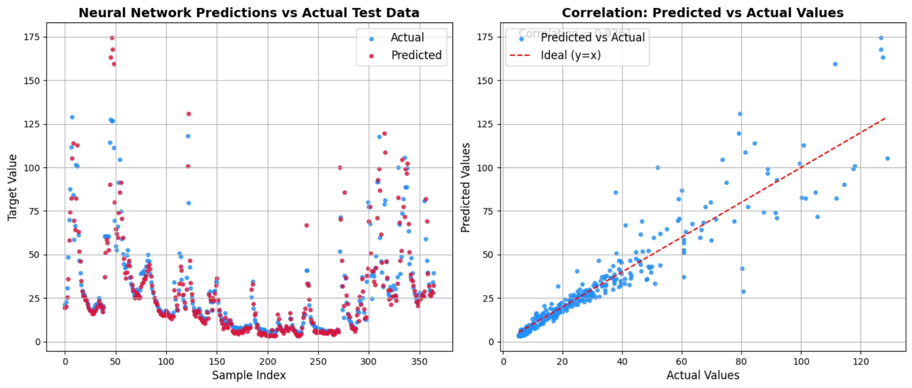


Figure 5: Mini-Batch + Bold driver + Momentum

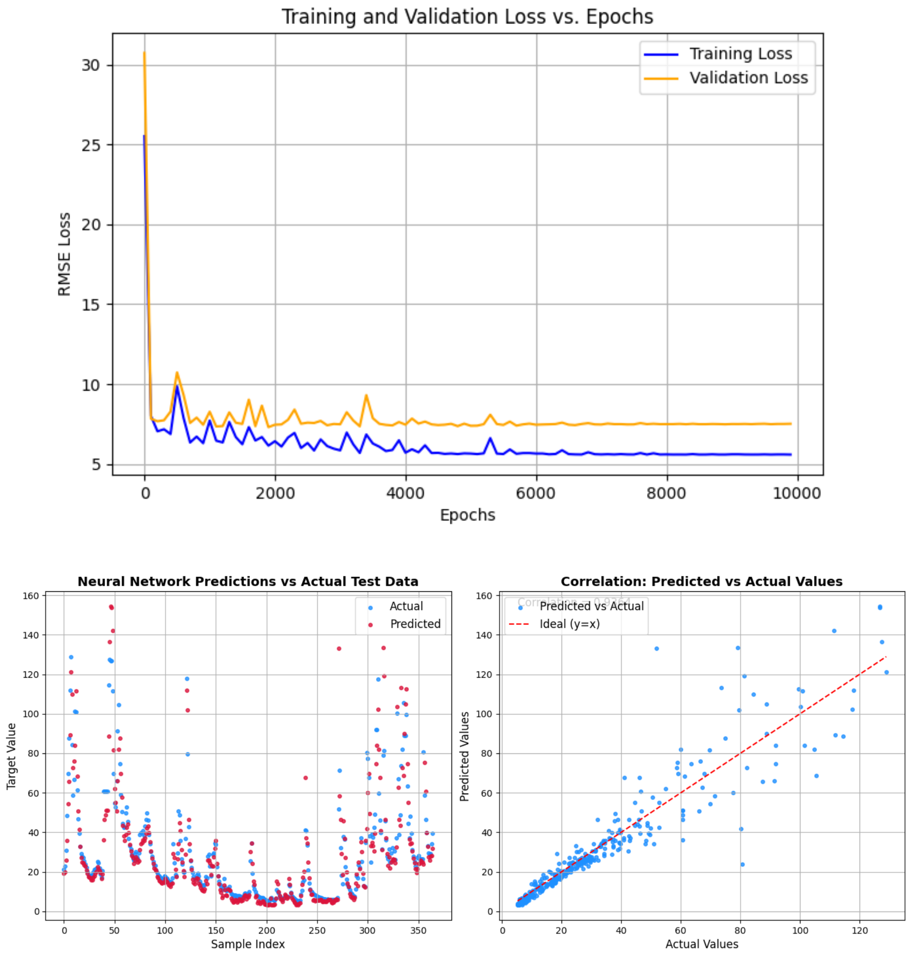


Figure 6: Mini-Batch + Momentum

4.5 Hidden Node Variations

In this subsection I experiment with different numbers of hidden nodes to identify what can give me the best model. I experiement with nodes from the range of $\frac{n}{2}$ to 2n in steps of 2 and compare them identifying what's best.

Nodes	**Train Error**	**Validation Error**	**Correlation**
16	4.93	7.68	0.942
14	5.01	8.14	0.922
12	5.29	8.57	0.936
10	5.71	8.40	0.933
8	5.97	7.73	0.938
6	7.41	8.26	0.934
4	7.25	7.56	0.941

Table 5: Training and Validation Errors with Correlation for Different Node Configurations

The 16-node model appears to be the best, as it has the lowest validation error (7.68) and the highest correlation (0.942).

However, the 4-node model has a slightly lower validation error (7.56) but a slightly lower correlation (0.941).

Given the small difference, if generalisation (validation error) is the priority, the 4-node model might be preferable. But if overall consistency (low error + high correlation) is preferred, the 16-node model remains the best choice.

I've gone for the 16 due to it's lower training error and slightly higher correlation.

A lower training error means the model learns patterns better, capturing important relationships in the data.

A higher correlation means the model's predictions are more aligned with the actual outcomes, leading to better reliability.

Having more hidden nodes also allows me to capture more complex patterns.

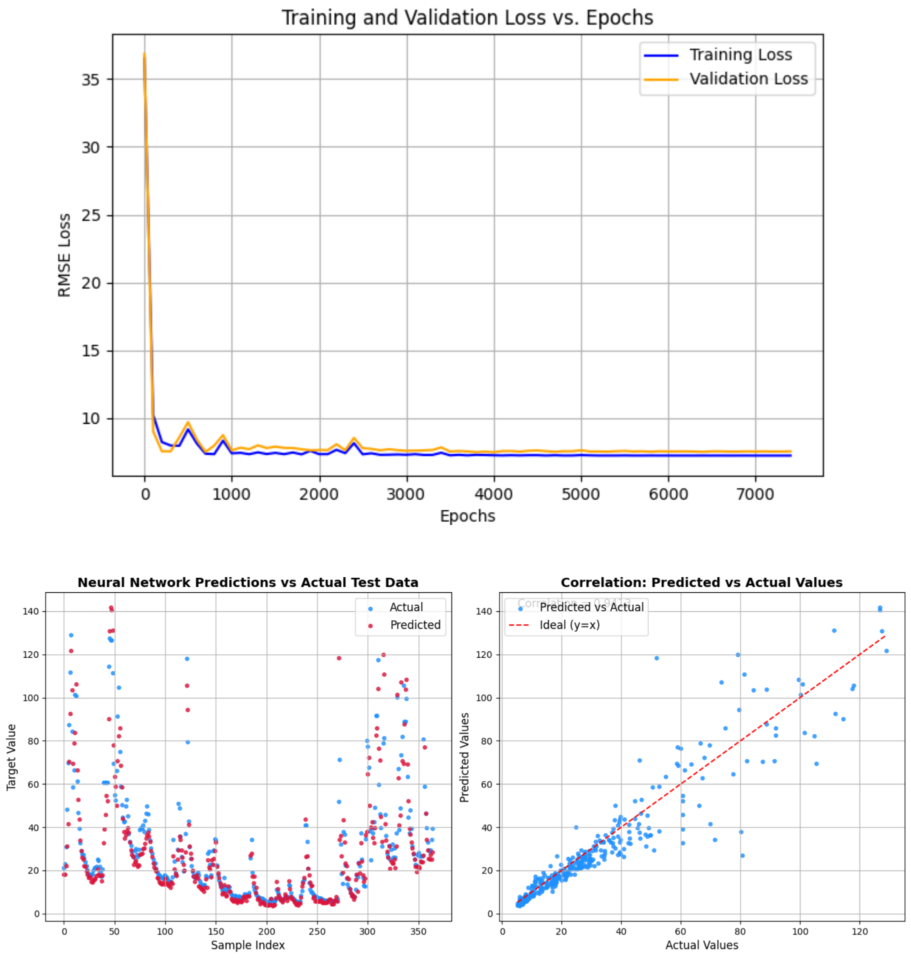


Figure 7: 4 nodes

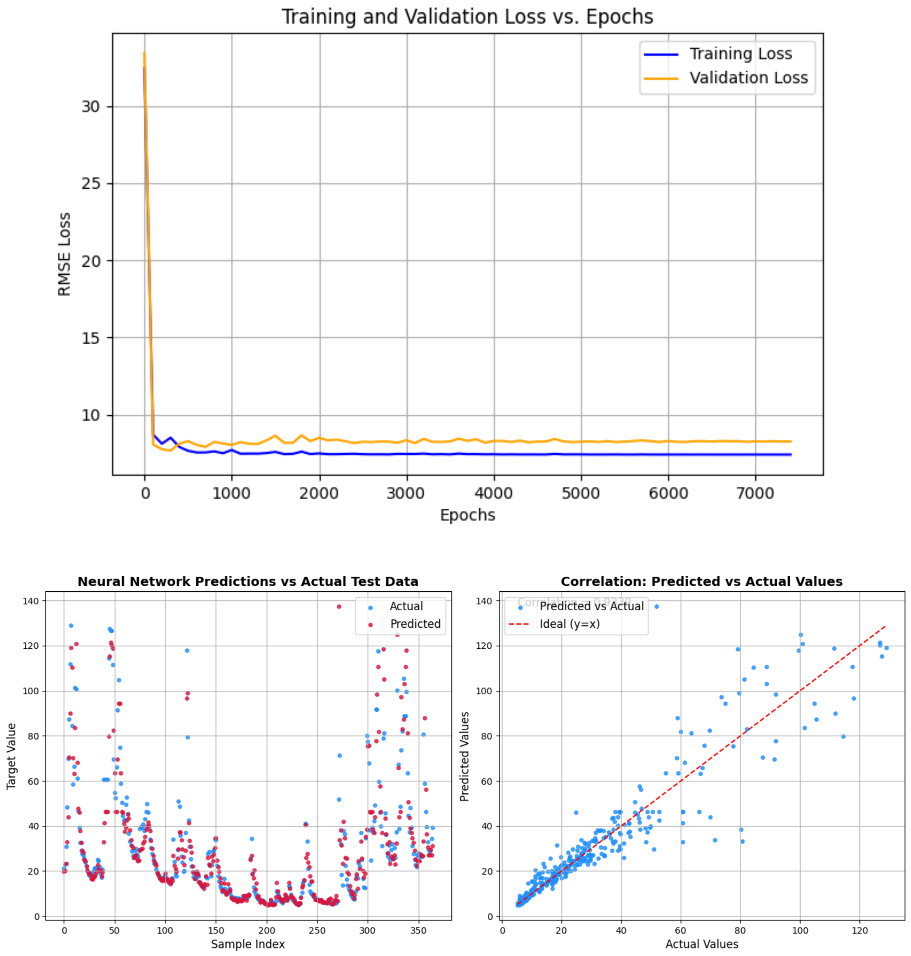


Figure 8: 6 nodes

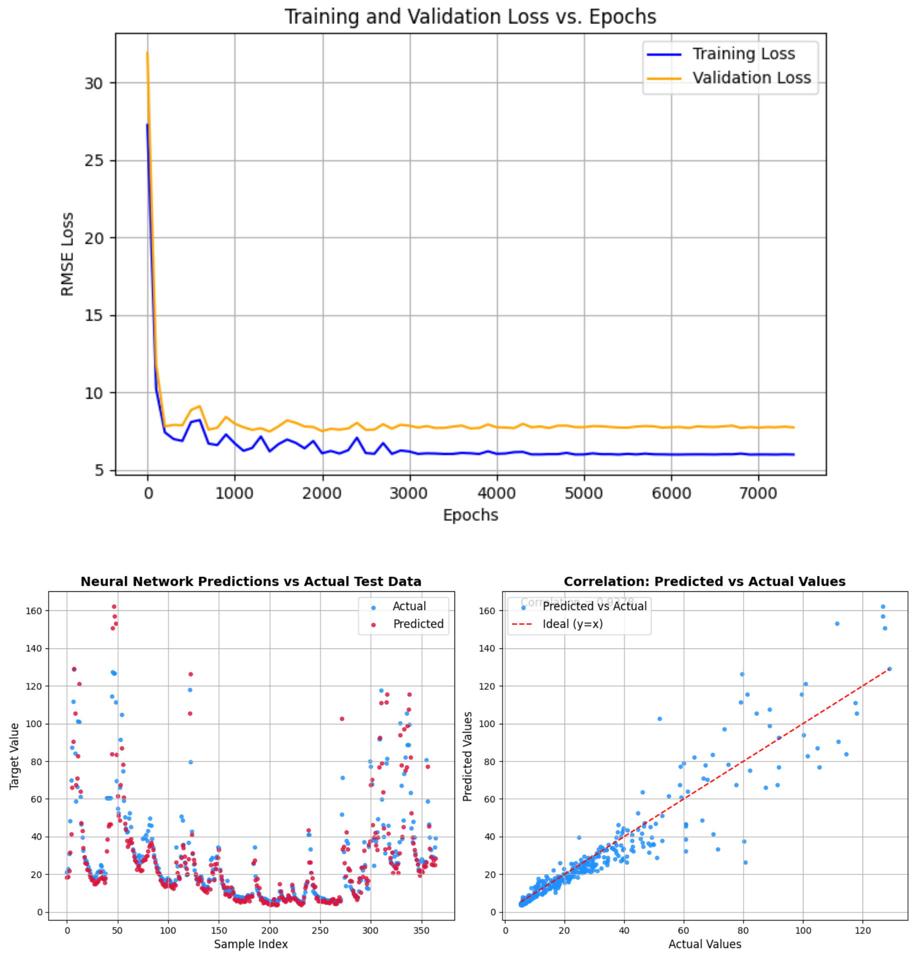


Figure 9: 8 nodes

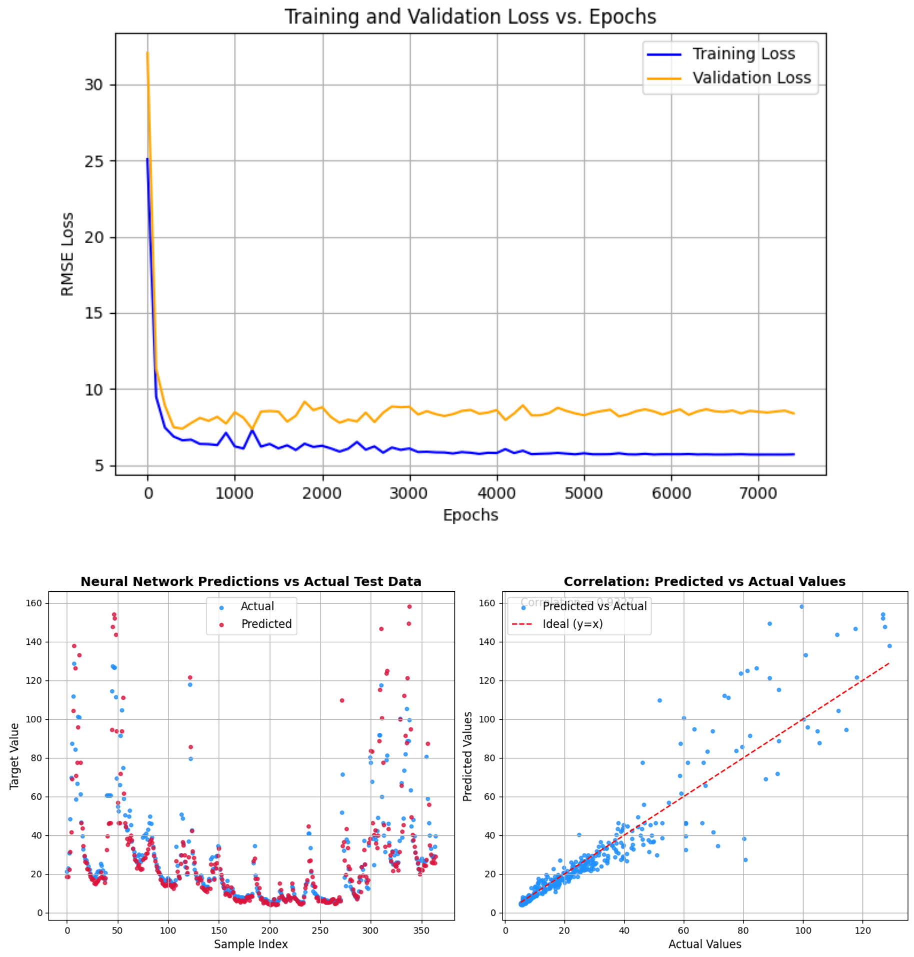


Figure 10: 10 nodes

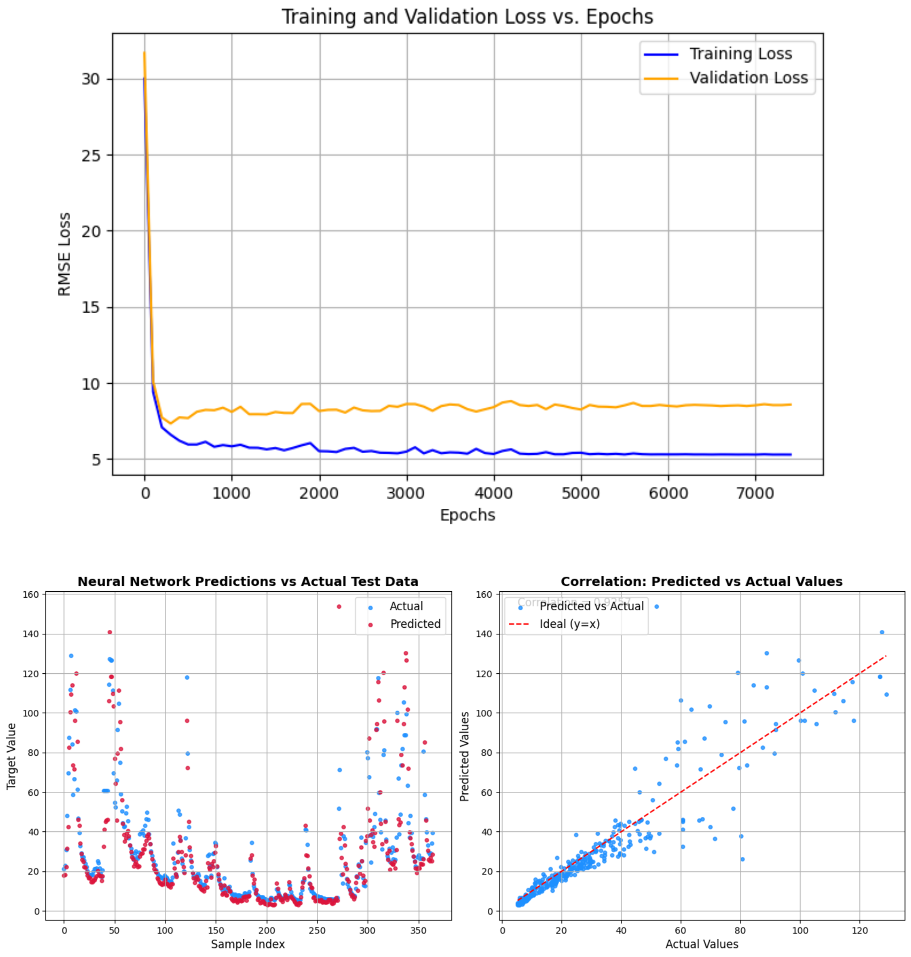


Figure 11: 12 nodes

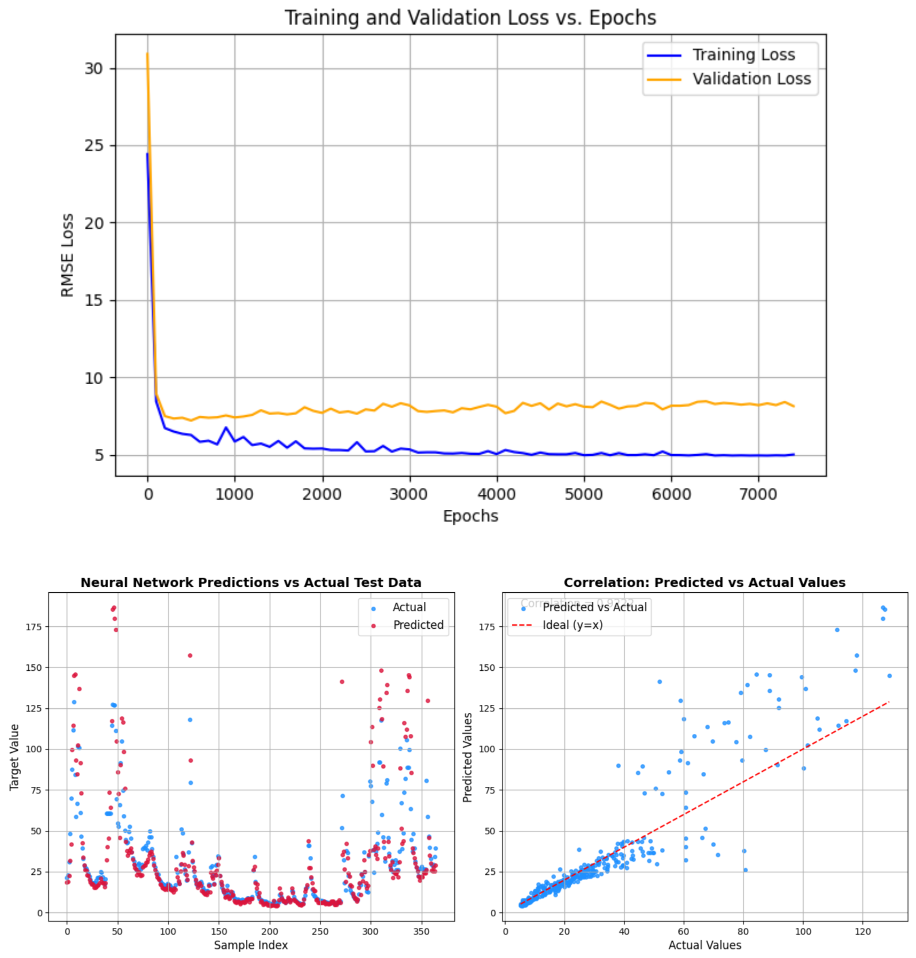
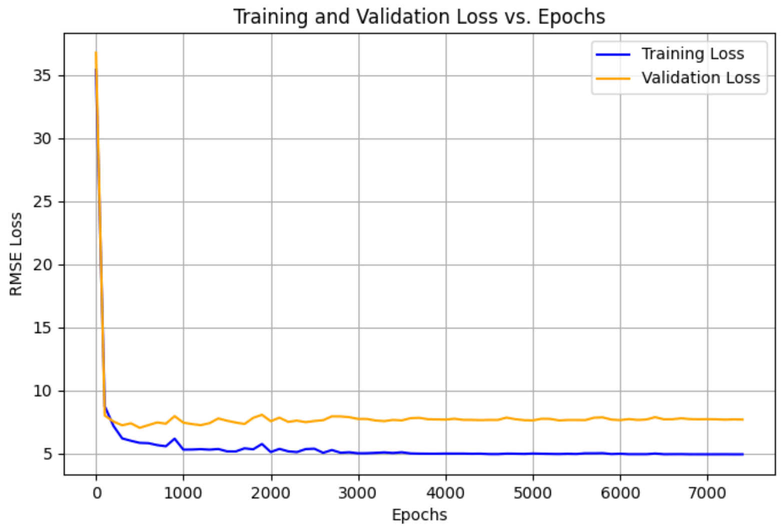


Figure 12: 14 nodes



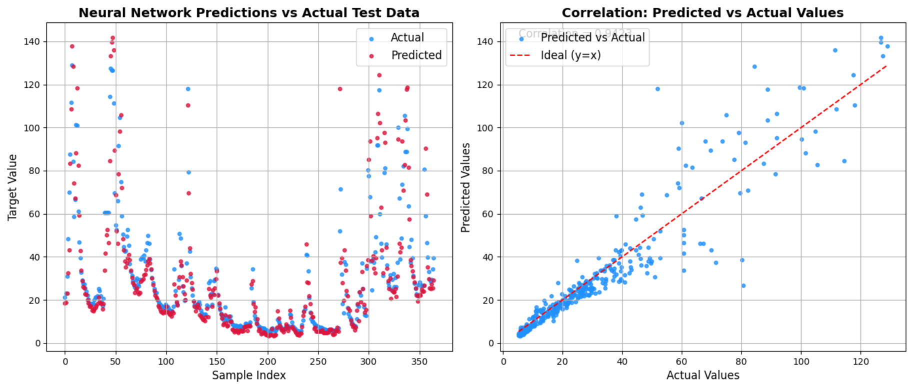


Figure 13: 16 nodes

4.6 Hyperparameter Tuning & Changes

- **Batch Size**: 32
- **Learning Rate**: 0.1 (with Bold Driver factor of 0.85 and 1.05)
- **Momentum**: 0.85
- **Epochs**: 7500

These values were selected based on a trade-off between stability, speed, and generalisation.
Below, I compare them against alternative values.
Learning Rate

- **Learning Rate = 0.01**: Too slow, delays convergence. **(Not optimal)**
- **Learning Rate = 0.05**: More stable than 0.1 but slightly slower.It could also result in overfitting of the model. **(Suboptimal)**
- **Learning Rate = 0.1**: Allows faster convergence, especially with momentum and adaptive scaling (Bold Driver). **(Optimal)**
- **Learning Rate = 0.001**: Extremely slow, leading to long training times. **(Not optimal)**

Chosen: 0.1 - Fast learning, stabilized by the Bold Driver.
Momentum Factor

- **Momentum 0.8 - 0.95**: Smoother training but risks overshooting. **(Not optimal)**
- **Momentum 0.85**: Balances stability and responsiveness, avoiding excessive oscillations. **(Optimal)**

Chosen: 0.85 - Prevents overshooting and works well with adaptive learning rates.
Epochs

- **1000 epochs**: Underfits; model does not reach optimal performance. **(Not optimal)**
- **5000 epochs**: Might work, but stopping too soon could leave potential gains. **(Suboptimal)**
- **7500 epochs**: Best balance of convergence vs. computation time. **(Optimal)**
- **10,000 epochs**: Likely unnecessary, as diminishing returns occur after 7500 epochs. **(Not optimal)**

Chosen: 7500 - Ensures full optimization without unnecessary computation.
Bold Driver factors

- **Decrease LR by 0.85 when loss increases, increase by 1.05 when loss decreases.**
- This dynamic adjustment prevents the learning rate from being stuck at a suboptimal value.
- Prevents overshooting while accelerating training when improving.

Chosen:(0.85, 1.05) This configuration provides the best balance between convergence speed, stability, and generalization.

A weight decay value of $\lambda = 0.0005$ is commonly used because it offers a balance between regularizing the model and allowing it to fit the data.

This is my final model with 16 hidden nodes , 7500 epochs , 0.85 momentum , bold driver factors of 1.05 and 0.85 , weight decay value of 0.0005 and a learning rate of 0.1.

5 Evaluation of the ANN Model

To assess the performance of the trained ANN model, evaluation metrics are used. This section presents:

- Performance metrics such as RMSE,MSRE etc
- Cross-validation
- Comparison with initial MLP

5.1 MSRE , RMSE , R Squared

Metric	Train	Validation
RMSE	4.93	7.68
MSRE	0.0168	0.0429
R^2	0.982	0.950

Table 6: Model Performance Metrics

- **RMSE (Root Mean Squared Error):** Measures the average magnitude of prediction errors. A lower RMSE indicates better model performance.
- **MSRE (Mean Squared Relative Error):** Represents the mean squared difference between predicted and actual values, normalized by actual values.

- R^2 **(Coefficient of Determination):** Indicates how well the model explains the variance in the data. A value closer to 1 means a better fit.

Root Mean Squared Error (RMSE):

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

Mean Squared Relative Error (MSRE):

$$MSRE = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{y_i - \hat{y}_i}{y_i}\right)^2 \tag{2}$$

Coefficient of Determination (R²):

$$R^2 = 1 - \frac{\sum(y_i - \hat{y}_i)^2}{\sum(y_i - \bar{y})^2} \tag{3}$$

where $\bar{y}$ is the mean of actual values.

```
# RMSE loss and its derivative
def rmse_loss(y_true, y_pred):
    return np.sqrt(((y_true - y_pred) ** 2).mean())

def rmse_loss_derivative(y_true, y_pred):
    return
    (y_pred - y_true) / (y_true.size * np.sqrt(((y_true - y_pred) ** 2).mean()))

# MSRE (Mean Squared Relative Error)
def msre_loss(y_true, y_pred):
    return np.mean(((y_true - y_pred) / y_true) ** 2)

# R² (Coefficient of Determination)
def r_squared_loss(y_true, y_pred):
    ss_total = np.sum((y_true - np.mean(y_true)) ** 2)
    ss_residual = np.sum((y_true - y_pred) ** 2)
    return 1 - (ss_residual / ss_total)

def train(self, X_train, y_train_scaled,
X_valid, y_valid_scaled, scaler_Y, epochs,
learning_rate, batch_size=32):
        prev_loss = float('inf')
        loss_history = []
        val_loss_history = []  # History of validation losses
        train_msre_history = []  # History of training MSRE
```

```
val_msre_history = []  # History of validation MSRE
train_r2_history = []  # History of training R²
val_r2_history = []  # History of validation R²
num_samples = X_train.shape[0]

for epoch in range(epochs):
    indices = np.random.permutation(num_samples)
    X_train_shuffled = X_train[indices]
    y_train_shuffled = y_train_scaled[indices]

    for i in range(0, num_samples, batch_size):
        X_batch = X_train_shuffled[i:i+batch_size]
        y_batch = y_train_shuffled[i:i+batch_size]

        self.forward(X_batch)
        self.backward(X_batch, y_batch, learning_rate)

    if epoch % 100 == 0:
        # Train predictions and error
        y_pred_real = scaler_Y.inverse_transform(self.forward(X_train))
        y_train_real = scaler_Y.inverse_transform(y_train_scaled)
        train_error = rmse_loss(y_train_real, y_pred_real)
        train_msre = msre_loss(y_train_real, y_pred_real)
        train_r2 = r_squared_loss(y_train_real, y_pred_real)

        # Validation predictions and error
        y_valid_pred_real = scaler_Y.inverse_transform(self.forward(X_valid))
        y_valid_real = scaler_Y.inverse_transform(y_valid_scaled)
        val_error = rmse_loss(y_valid_real, y_valid_pred_real)
        val_msre = msre_loss(y_valid_real, y_valid_pred_real)
        val_r2 = r_squared_loss(y_valid_real, y_valid_pred_real)

        loss_history.append(train_error)
        val_loss_history.append(val_error)
        train_msre_history.append(train_msre)
        val_msre_history.append(val_msre)
        train_r2_history.append(train_r2)
        val_r2_history.append(val_r2)

        print(f"Epoch {epoch},
        Train RMSE: {train_error:.6f},
        Validation RMSE: {val_error:.6f},
        Train MSRE: {train_msre:.6f},
        Validation MSRE: {val_msre:.6f},
        Train R²: {train_r2:.6f},
        Validation R²: {val_r2:.6f},
        LR: {learning_rate:.6f}")

        if train_error < prev_loss:
            learning_rate *= 1.05
        else:
            learning_rate *= 0.85

        prev_loss = train_error

return y_pred_real, y_train_real
```

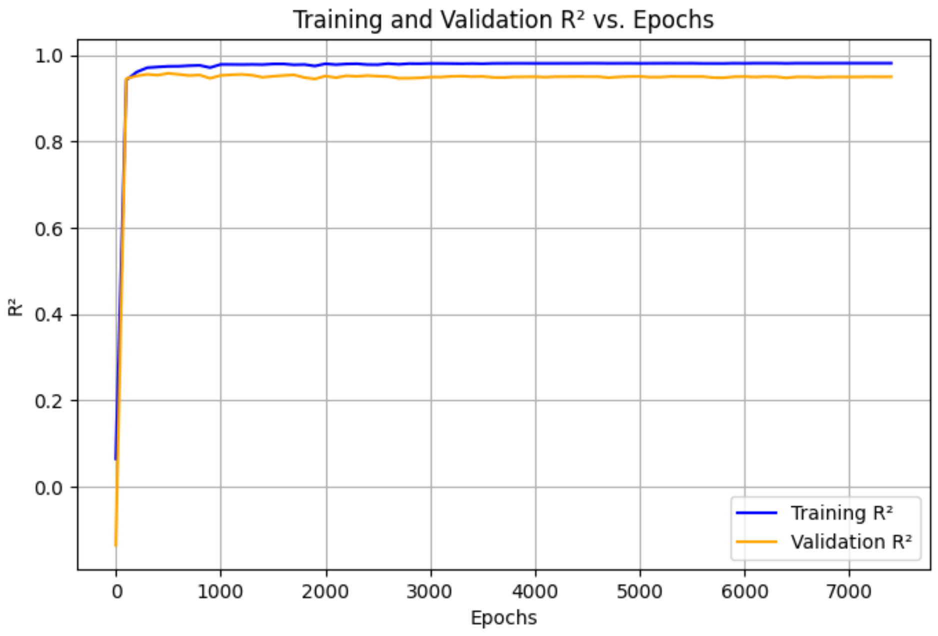


Figure 14: R^2 error vs epochs

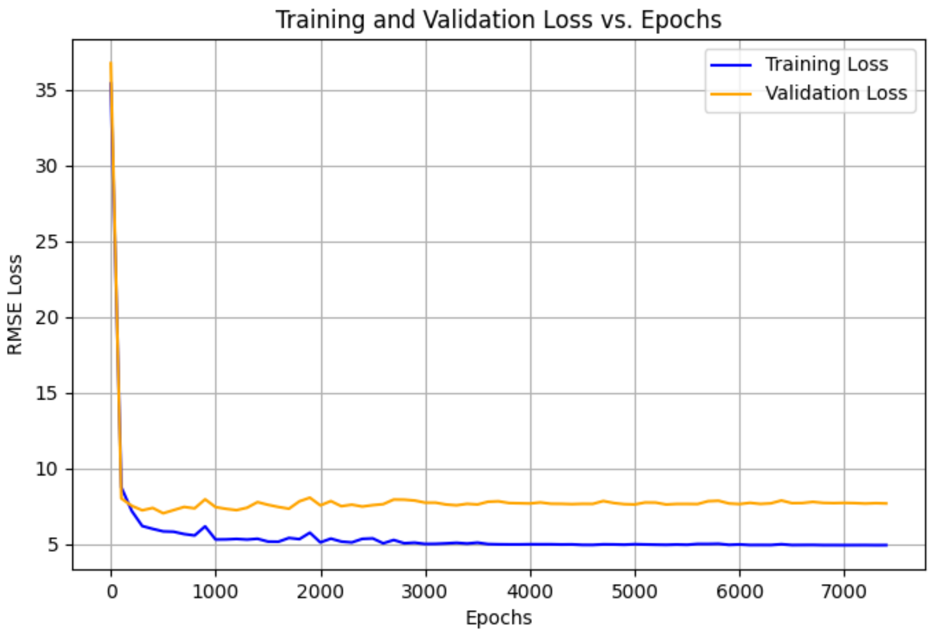


Figure 15: RMSE error vs epochs

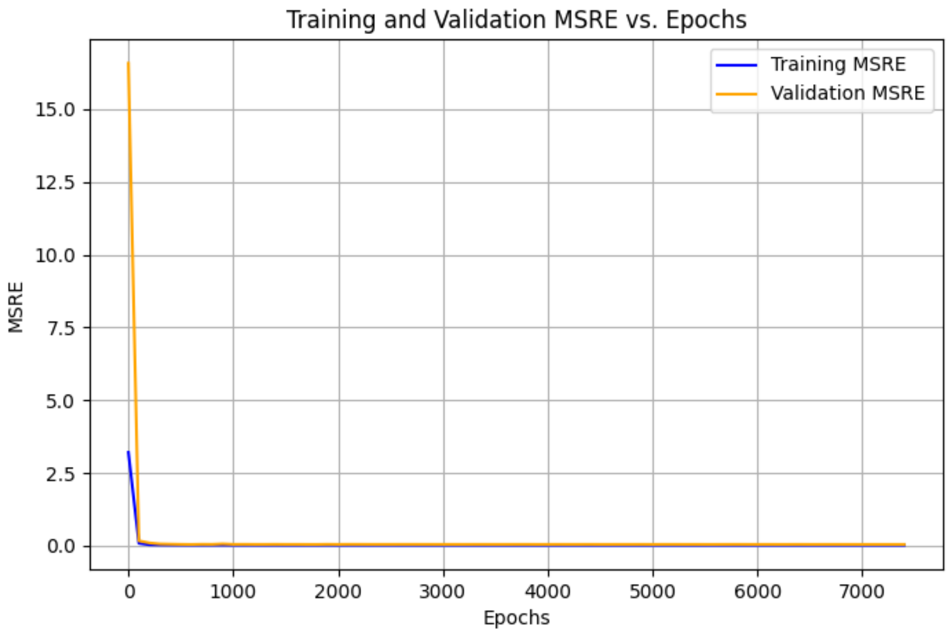


Figure 16: MSRE error vs epochs

5.2 K-Fold Cross-Validation

Cross-validation R^2 Scores:

[0.940 0.955 0.935 0.933 0.940] to 3 decimal points

Average R^2: 0.940 ± 0.008
Cross-validation is a technique used to evaluate a model's performance by splitting the dataset into multiple subsets. The model is trained on some subsets and tested on the remaining ones, rotating the test set in each iteration. This helps assess generalization and prevent overfitting.

1. Stability
    - The standard deviation (± 0.008) is small, meaning the model performs consistently across different validation folds.
2. Good generalization
    - The model generalises well to unseen data

- There is no significant overfitting since the performance remains stable across folds

3. Consistently high R^2 scores

   - The scores range from 0.933 to 0.955, all above 0.930, which means the model explains a large portion of the variance in the data.
   - A high R^2 suggests your model is accurate.

```
class NeuralNetwork(BaseEstimator, RegressorMixin):  # Extend scikit-learn classes for compa
    def __init__(self, input_size,
    hidden_size=16, output_size=1, momentum=0.85, epochs=500, learning_rate=0.1):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.momentum = momentum
        self.epochs = epochs
        self.learning_rate = learning_rate
        self.scaler_Y = None  # Placeholder for scaler

        # Initialize weights and biases
        self.weights_input_hidden = he_initialisation(input_size, hidden_size)
        self.bias_hidden = np.zeros((1, hidden_size))
        self.weights_hidden_output = he_initialisation(hidden_size, output_size)
        self.bias_output = np.zeros((1, output_size))

        # Initialize velocity for momentum
        self.velocity_wih = np.zeros_like(self.weights_input_hidden)
        self.velocity_bh = np.zeros_like(self.bias_hidden)
        self.velocity_who = np.zeros_like(self.weights_hidden_output)
        self.velocity_bo = np.zeros_like(self.bias_output)

    def forward(self, X):
        self.input_layer = X
        self.hidden_layer_sum = np.dot(self.input_layer, self.weights_input_hidden)
        + self.bias_hidden
        self.hidden_layer = leaky_relu(self.hidden_layer_sum)
        self.output_layer_sum = np.dot(self.hidden_layer, self.weights_hidden_output)
        + self.bias_output
        self.output_layer = leaky_relu(self.output_layer_sum)
        return self.output_layer

    def backward(self, X, y, learning_rate):
        lambda_wd = 0.0005  # Weight decay coefficient

        output_error = rmse_loss_derivative(y, self.output_layer)
        output_delta = output_error * leaky_relu_derivative(self.output_layer_sum)
        hidden_error = output_delta.dot(self.weights_hidden_output.T)
        hidden_delta = hidden_error * leaky_relu_derivative(self.hidden_layer_sum)

        grad_wih = X.T.dot(hidden_delta)
        grad_bh = np.sum(hidden_delta, axis=0, keepdims=True)
        grad_who = self.hidden_layer.T.dot(output_delta)
        grad_bo = np.sum(output_delta, axis=0, keepdims=True)

        # Update with momentum and weight decay
        self.velocity_wih =
        self.momentum * self.velocity_wih + (1 - self.momentum) *
        (grad_wih + lambda_wd * self.weights_input_hidden)
        self.velocity_who =
        self.momentum * self.velocity_who + (1 - self.momentum) *
        (grad_who + lambda_wd * self.weights_hidden_output)
        self.velocity_bo =
        self.momentum * self.velocity_bo + (1 - self.momentum) * grad_bo
        self.velocity_bh =
        self.momentum * self.velocity_bh + (1 - self.momentum) * grad_bh

        self.weights_input_hidden -= learning_rate * self.velocity_wih
        self.weights_hidden_output -= learning_rate * self.velocity_who
        self.bias_output -= learning_rate * self.velocity_bo
        self.bias_hidden -= learning_rate * self.velocity_bh

    def fit(self, X_train, y_train):
        self.scaler_Y = StandardScaler()
        y_train_scaled = self.scaler_Y.fit_transform(y_train.reshape(-1, 1))

        num_samples = X_train.shape[0]
        for epoch in range(self.epochs):
            indices = np.random.permutation(num_samples)
            X_train_shuffled = X_train[indices]
            y_train_shuffled = y_train_scaled[indices]

            for i in range(0, num_samples, 32):
                X_batch = X_train_shuffled[i:i+32]
                y_batch = y_train_shuffled[i:i+32]

                self.forward(X_batch)
                self.backward(X_batch, y_batch, self.learning_rate)

        return self

    def predict(self, X):
        y_pred_scaled = self.forward(X)
        return self.scaler_Y.inverse_transform(y_pred_scaled)


def cross_validate_nn(X, Y, k=5):
    """
    Performs k-fold cross-validation on the NeuralNetwork model.

    Parameters:
    X (numpy.ndarray): Features dataset.
    Y (numpy.ndarray): Target dataset.
    k (int): Number of folds for cross-validation.

    Returns:
    None: Prints the cross-validation scores.
    """
    nn = NeuralNetwork(input_size=X.shape[1], hidden_size=16,
    output_size=1, epochs=500, learning_rate=0.1)
    kf = KFold(n_splits=k, shuffle=True, random_state=42)

    scores = cross_val_score(nn, X, Y, cv=kf, scoring='r2')
    print(f"Cross-validation R² Scores: {scores}")
    print(f"Average R²: {np.mean(scores):.6f} ± {np.std(scores):.6f}")


def train_and_cross_validate(file_path):
    # Get the train, validation, and test sets
    X_train, X_valid, X_test, Y_train, Y_valid, Y_test = split_and_scale_data(file_path)

    # Convert to NumPy arrays
    X_train, X_valid, X_test = X_train.to_numpy(), X_valid.to_numpy(), X_test.to_numpy()
    Y_train, Y_valid, Y_test = Y_train.to_numpy(), Y_valid.to_numpy(), Y_test.to_numpy()

    # Perform cross-validation
    cross_validate_nn(X_train, Y_train, k=5)
```

5.3 Linear Regression

Model	Train RMSE	Validation RMSE	Test RMSE
Neural Network (NN)	4.93	7.68	9.68
Linear Regression (LR)	10.6	8.73	9.86

Table 7: Comparison of Neural Network and Linear Regression Performance

5.3.1 About Linear Regression

Linear regression is a simple statistical model that finds the best-fit line through data by minimizing the squared differences between predicted and actual values. It assumes a linear relationship between input features and the target variable.

5.3.2 Insights from Results

The neural network significantly outperforms linear regression, particularly in training, suggesting it captures complex patterns better. However, the validation error remains relatively high, indicating possible slight overfitting.

Linear regression struggles more overall, highlighting the need for a more complex model.

```python
def train_baseline_linear_regression(X_train, Y_train, X_valid, Y_valid, X_test, Y_test):
    """
    Trains a baseline Linear Regression model
    and evaluates it on the validation and test sets.
    """
    model = LinearRegression()
    model.fit(X_train, Y_train)

    # Predictions
    Y_train_pred = model.predict(X_train)
    Y_valid_pred = model.predict(X_valid)
    Y_test_pred = model.predict(X_test)

    # Compute RMSE for comparison
    train_rmse = rmse_loss(Y_train, Y_train_pred)
    valid_rmse = rmse_loss(Y_valid, Y_valid_pred)
    test_rmse = rmse_loss(Y_test, Y_test_pred)

    print("Baseline Linear Regression:")
    print(f"Train RMSE: {train_rmse:.6f},
    Validation RMSE: {valid_rmse:.6f},
    Test RMSE: {test_rmse:.6f}")

    return model, Y_test_pred
```

5.4 Evaluation of Model

5.4.1 Positives

- **Model Performance Comparison:** The Neural Network (NN) outperforms Linear Regression (LR) in terms of training, validation, and test RMSE values. NN has a significantly lower training RMSE (4.93) compared to LR (10.56), indicating better fitting on the training data.
- **Validation and Test RMSE:** NN's validation RMSE (7.68) and test RMSE (9.68) are also lower than LR's (8.73 and 9.86), showcasing better generalization performance. The smaller discrepancy between training and validation/test RMSE for NN suggests that it is less prone to overfitting than LR.
- **Model Accuracy on Validation:** The R^2 value on validation for NN (0.950) is substantially better than LR (0.931), indicating that the NN is explaining more of the variance in the validation data.
- **Cross-validation R^2:** The cross-validation R^2 scores (mean = 0.940 ± 0.008) suggest consistent model performance across different data splits, with little variance in performance, which is ideal for a reliable model.
- **Good Fit on Train Data:** The NN's R^2 on training (0.982) indicates a strong fit, suggesting the model captures the underlying patterns in the training data well.
- **Good Generalization:** The R^2 on validation (0.950) and the average cross-validation score (0.940) suggest that the model generalizes well, not just memorizing the training data but also performing well on unseen data.
- **Model Stability:** The small standard deviation in cross-validation R^2 (± 0.008) indicates the model's stability across different splits of the data, implying robustness in its performance.
- **Error Analysis:** While the NN outperforms LR, the test RMSE (9.68) could be further improved, but the model seems to already handle unseen data reasonably well.
- **Model Selection Justification:** Given the NN's better performance metrics across all datasets (training, validation, and test) compared to LR, it is a stronger choice for this problem, providing higher predictive accuracy and lower error.

5.4.2 Improvements

- **Hyperparameter Tuning:** The performance can be improved by experimenting with different hyperparameters, such as the number of layers, number of neurons per layer, learning rate, or activation functions. We could do endless experimentation and notice slight improvements.
- **Feature Engineering:** The model could benefit from additional feature engineering. Exploring new features, scaling features would help.
- **More Data:** Having access to more data would allow me to be able to engineer more features like for example condensation or latitude,longitude or even the altitude and elevation of the rivers would allow me to grasp an even greater understanding.
- **Inability to Generalize to Extreme Events Outside Training Data:** While the model performs well on the training, validation, and test data, it may struggle to generalise to extreme events or rare occurrences that were not well-represented in the training data. The model's ability to predict these types of events is constrained by the distribution of the data, and thus, its predictions may be unreliable in outlier scenarios or unexpected situations.Like a flood for example.