

Predicting Bike Rentals in Seoul Using Deep Learning

LACHIKET WARULE



Problem Statement



Objective:

- To predict the number of rented bikes in Seoul based on weather and temporal features.
- Help optimize bike-sharing operations through better demand forecasting.

Approach



Data Cleaning, Feature Engineering, Correlation Analysis, Visualization.



Build a deep learning model to predict bike rentals based on input features.

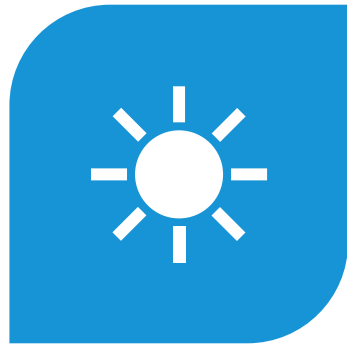


Training the dataset.



Apply preprocessing techniques, including cyclical transformations for time-based data such as hour and month.

Key Features



WEATHER:
TEMPERATURE, HUMIDITY, WIND
SPEED, SOLAR RADIATION.



TIME-BASED FEATURES:
HOUR, MONTH, DAY OF WEEK,
HOLIDAY, FUNCTIONING DAY.



DROPPED FEATURES:
DATE, HOUR, MONTH, RAINFALL,
SNOWFALL.

Steps to Implement



Collect the data



Data Cleaning



Feature Engineering



Correlation Analysis



Model Building



Model Training & Evaluation



Predict the no. of rented bike



Data Visualization

Proposed Model Architecture



Neural Network Structure:

- **Input Layer:** Weather and temporal features.
- **Hidden Layers:** 2 fully connected layers with ReLU activation functions
- **Output Layer:** Single node for predicting the number of rented bikes
- **Loss Function:** Mean Squared Error (MSE) to minimize prediction errors.
- **Optimizer:** Adam optimizer with a learning rate of 0.001.

Libraries



```
# Importing required libraries

import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import torch.nn.functional as F
from sklearn.metrics import mean_absolute_error, r2_score
```

- Tools and Frameworks: We implemented this project using Python, PyTorch for deep learning, and libraries like pandas, scikit-learn, matplotlib, and seaborn for preprocessing and visualization.

Exploratory Data Analysis



THE DATASET INCLUDES WEATHER FEATURES (TEMPERATURE, HUMIDITY, WIND SPEED, VISIBILITY) AND TEMPORAL FEATURES (HOUR, SEASON, HOLIDAY).



THE DATA WAS CLEANED BY HANDLING MISSING VALUES AND ENSURING CONSISTENCY ACROSS FEATURES.



EXAMINED THE DATASET TO UNDERSTAND THE STRUCTURE, TYPES OF FEATURES, AND IDENTIFY MISSING OR INCONSISTENT DATA.



ANALYZE FEATURE RELATIONSHIPS USING A CORRELATION MATRIX AND HEATMAPS TO IDENTIFY RELEVANT VARIABLES.



THE GOAL IS TO ANALYZE THE IMPACT OF WEATHER AND TIME ON BIKE RENTALS TO ENHANCE DEMAND PREDICTION.

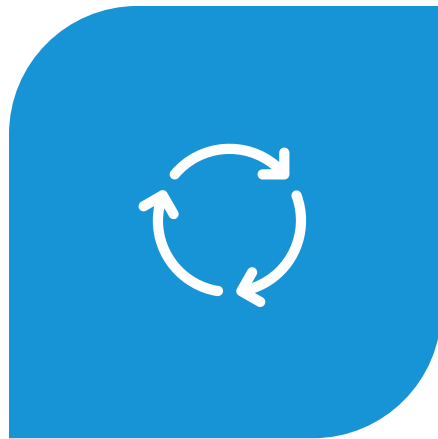
Seoul Bike Rental Dataset

[4] data.head()



	Date	Rented Bike Count	Hour	Temperature(°C)	Humidity(%)	Wind speed (m/s)	Visibility (10m)	Dew point temperature(°C)	Solar Radiation (MJ/m2)	Rainfall(mm)	Snowfall (cm)	Seasons	Holiday	Functioning Day
0	1/12/2017	254	0	-5.2	37	2.2	2000	-17.6	0.0	0.0	0.0	Winter	No Holiday	Yes
1	1/12/2017	204	1	-5.5	38	0.8	2000	-17.6	0.0	0.0	0.0	Winter	No Holiday	Yes
2	1/12/2017	173	2	-6.0	39	1.0	2000	-17.7	0.0	0.0	0.0	Winter	No Holiday	Yes
3	1/12/2017	107	3	-6.2	40	0.9	2000	-17.6	0.0	0.0	0.0	Winter	No Holiday	Yes
4	1/12/2017	78	4	-6.0	36	2.3	2000	-18.6	0.0	0.0	0.0	Winter	No Holiday	Yes

Time Feature Engineering:



CYCLICAL ENCODING: CONVERTED HOUR AND MONTH INTO CYCLICAL FEATURES. TO BETTER CAPTURE DAILY AND SEASONAL PATTERNS IN RENTALS.



DATE COMPONENTS: EXTRACTED YEAR, MONTH, DAY, AND DAYOFWEEK TO EXAMINE TRENDS OVER DIFFERENT PERIODS.

- Convert the 'Date' column into a datetime object and extract year, month, and day as separate columns
Apply cyclical encoding to 'Hour' and 'Month' for better representation in the model



Preprocess the date column

```
data['Date'] = pd.to_datetime(data['Date'], format='%d/%m/%Y')
data['Year'] = data['Date'].dt.year
data['Month'] = data['Date'].dt.month
data['Day'] = data['Date'].dt.day
data.drop(columns=['Date'], inplace=True)
```

[6] # Cyclical encoding for Hour and Month

```
data['Hour_sin'] = np.sin(2 * np.pi * data['Hour'] / 24)
data['Hour_cos'] = np.cos(2 * np.pi * data['Hour'] / 24)
data['Month_sin'] = np.sin(2 * np.pi * data['Month'] / 12)
data['Month_cos'] = np.cos(2 * np.pi * data['Month'] / 12)
data.drop(['Hour', 'Month'], axis=1, inplace=True)
```

Categorical Encoding:

- One-Hot Encoding: Used for Seasons, Holiday, and Functioning Day columns to convert categorical data into numerical format.

```
▶ # One-hot encoding for Seasons
```

```
data = pd.get_dummies(data, columns=['Seasons'], drop_first=False)
```

```
[10] # One-hot encoding for Holiday and Functioning Day
```

```
data = pd.get_dummies(data, columns=['Holiday', 'Functioning Day'], drop_first=False)
```

data.head()

	Rented Bike Count	Temperature(°C)	Humidity(%)	Wind speed (m/s)	Visibility (10m)	Dew point temperature(°C)	Solar Radiation (MJ/m2)	Rainfall(mm)	Snowfall (cm)	Year	...	Month_sin	Month_cos	Seasons_Autumn	Seasons_Spring	Seasons_Summer
0	254	-5.2	37	2.2	2000	-17.6	0.0	0.0	0.0	2017	...	-2.449294e-16	1.0	False	False	True
1	204	-5.5	38	0.8	2000	-17.6	0.0	0.0	0.0	2017	...	-2.449294e-16	1.0	False	False	True
2	173	-6.0	39	1.0	2000	-17.7	0.0	0.0	0.0	2017	...	-2.449294e-16	1.0	False	False	True
3	107	-6.2	40	0.9	2000	-17.6	0.0	0.0	0.0	2017	...	-2.449294e-16	1.0	False	False	True
4	78	-6.0	36	2.3	2000	-18.6	0.0	0.0	0.0	2017	...	-2.449294e-16	1.0	False	False	True

5 rows × 23 columns

After One~Hot Encoding

Data Type

data.dtypes # Check data types of columns	
	0
Rented Bike Count	int64
Temperature(°C)	float64
Humidity(%)	int64
Wind speed (m/s)	float64
Visibility (10m)	int64
Dew point temperature(°C)	float64
Solar Radiation (MJ/m2)	float64
Rainfall(mm)	float64
Snowfall (cm)	float64
Year	int32
Day	int32
Hour_sin	float64
Hour_cos	float64
Month_sin	float64
Month_cos	float64
Seasons_Autumn	bool
Seasons_Spring	bool

Correlation Analysis:

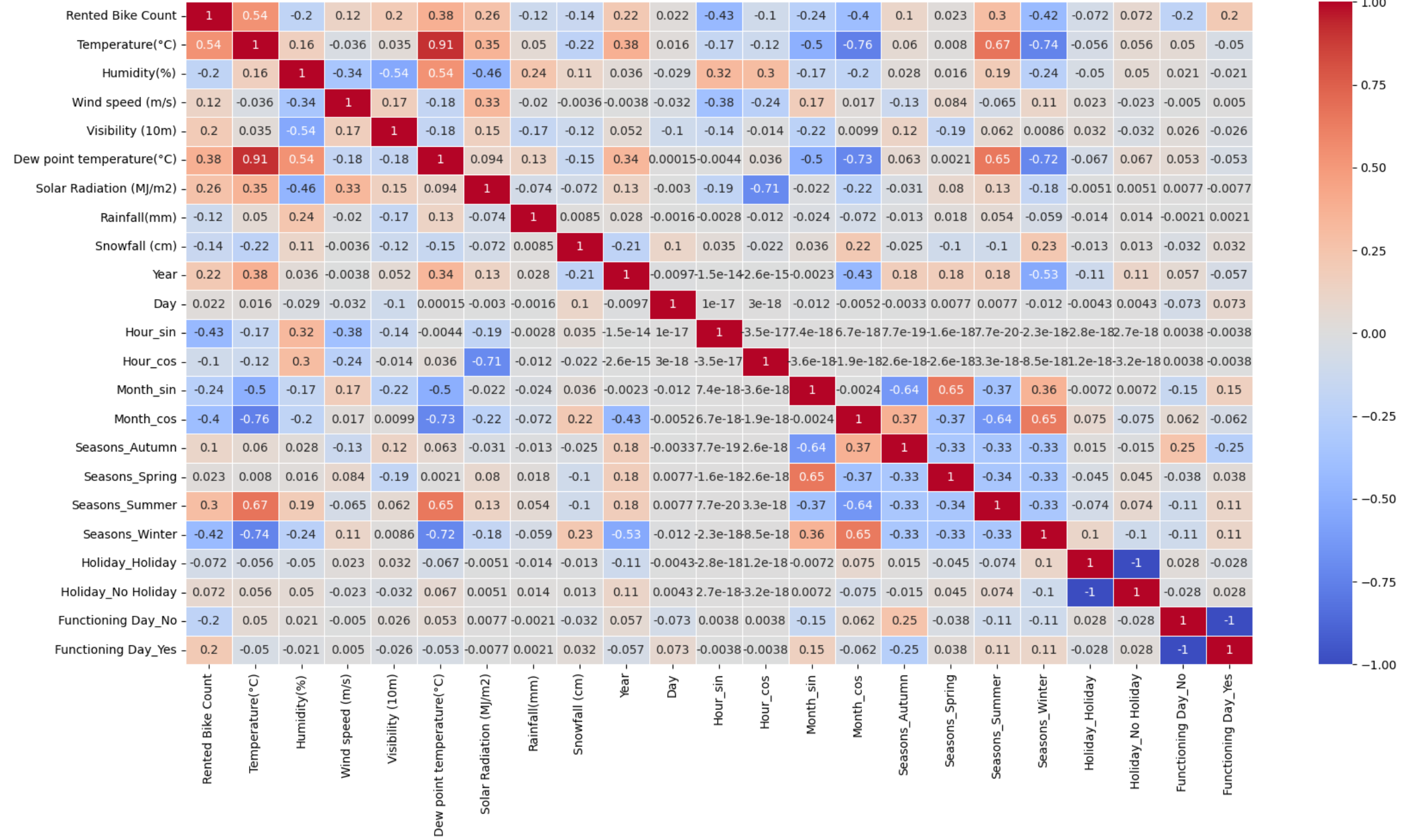
We computed the correlation matrix to understand relationships between features and the target variable. Strongly correlated features are prioritized.

To visualize these relationships, we created a heatmap, which highlights both positive and negative correlations.

```
[13] correlation_matrix = data.corr()  
correlation_matrix
```

	Rented Bike Count	Temperature(°C)	Humidity(%)	Wind speed (m/s)	Visibility (10m)	Dew point temperature(°C)	Solar Radiation (MJ/m2)	Rainfall(mm)	Snowfall (cm)	Year	...	Month_sin	Month_cos	Seasons_
Rented Bike Count	1.000000	0.538558	-0.199780	0.121108	0.199280	0.379788	0.261837	-0.123074	-0.141804	2.151618e-01	...	-2.442890e-01	-3.994031e-01	1.0275
Temperature(°C)	0.538558	1.000000	0.159371	-0.036252	0.034794	0.912798	0.353505	0.050282	-0.218405	3.777958e-01	...	-4.955616e-01	-7.569603e-01	5.9728
Humidity(%)	-0.199780	0.159371	1.000000	-0.336683	-0.543090	0.536894	-0.461919	0.236397	0.108183	3.592468e-02	...	-1.703203e-01	-2.026380e-01	2.8366
Wind speed (m/s)	0.121108	-0.036252	-0.336683	1.000000	0.171507	-0.176486	0.332274	-0.019674	-0.003554	-3.780878e-03	...	1.729095e-01	1.671976e-02	-1.2800
Visibility (10m)	0.199280	0.034794	-0.543090	0.171507	1.000000	-0.176630	0.149738	-0.167629	-0.121695	5.238110e-02	...	-2.206436e-01	9.912367e-03	1.1741
Dew point temperature(°C)	0.379788	0.912798	0.536894	-0.176486	-0.176630	1.000000	0.094381	0.125597	-0.150887	3.363497e-01	...	-5.004186e-01	-7.261556e-01	6.2878
Solar Radiation (MJ/m2)	0.261837	0.353505	-0.461919	0.332274	0.149738	0.094381	1.000000	-0.074290	-0.072301	1.280860e-01	...	-2.205032e-02	-2.201085e-01	-3.1374
Rainfall(mm)	-0.123074	0.050282	0.236397	-0.019674	-0.167629	0.125597	-0.074290	1.000000	0.008500	2.752192e-02	...	-2.440129e-02	-7.185402e-02	-1.3246
Snowfall (cm)	-0.141804	-0.218405	0.108183	-0.003554	-0.121695	-0.150887	-0.072301	0.008500	1.000000	-2.064178e-01	...	3.572781e-02	2.150463e-01	-2.4742
Year	0.215162	0.377796	0.035925	-0.003781	0.052381	0.336350	0.128086	0.027522	-0.206418	1.000000e+00	...	-2.323275e-03	-4.309865e-01	1.7557
Day	0.022291	0.015645	-0.029044	-0.031977	-0.101759	0.000153	-0.002982	-0.001623	0.102077	-9.678711e-03	...	-1.209978e-02	-5.157310e-03	-3.2901
Hour_sin	-0.431773	-0.170147	0.321202	-0.380380	-0.139144	-0.004398	-0.194779	-0.002751	0.035239	-1.517549e-14	...	7.429658e-18	6.741324e-18	7.6646
Hour_cos	-0.099507	-0.117906	0.299885	-0.239333	-0.014424	0.035611	-0.706984	-0.012444	-0.021674	-2.596145e-15	...	-3.556076e-18	-1.922479e-18	2.5686

Heatmap



```
▶ # Drop less relevant features based on the correlation analysis  
  
columns_to_drop = ['Rainfall(mm)', 'Snowfall (cm)']  
data.drop(columns=columns_to_drop, inplace=True)
```

- Features like Rainfall(mm) and Snowfall(cm) were dropped as their correlation with the target variable was negligible.

Data Split

- We split the dataset into training (80%) and testing (20%) subsets to evaluate the model's performance on unseen data.

```
[17] # Split the data into features (X) and target (y), then into training and testing sets

X = data.drop('Rented Bike Count', axis=1)
y = data['Rented Bike Count']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Standardization

- To ensure all features are on a similar scale, we applied standardization using scikit-learn's StandardScaler. This improves the convergence speed during training.
- Standardization of features in both train and test sets to maintain consistency in data distribution for the model.

```
[18] # Standardize the features for better performance during training

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Neural Network Model



- Our neural network consists of two hidden layers with 64 and 32 neurons, and one output layer. ReLU activation is applied in the hidden layers for non-linearity.
- This architecture was implemented using PyTorch. Each layer was defined using `torch.nn.Linear`, and the forward method calculates predictions.

```
# Defining the Neural Network Model

class LinearRegressionModel(nn.Module):
    def __init__(self, input_size):
        super(LinearRegressionModel, self).__init__()
        self.hidden1 = nn.Linear(input_size, 64)
        self.hidden2 = nn.Linear(64, 32)
        self.output = nn.Linear(32, 1)

    def forward(self, x):
        x = F.relu(self.hidden1(x))
        x = F.relu(self.hidden2(x))
        return self.output(x)
```

Data Conversion for PyTorch

- Before training the neural network, the standardized data was converted into PyTorch tensors. This format is required for PyTorch models to process the data efficiently.
- We validated the shapes of the training and testing tensors to ensure compatibility with the neural network's input and output layers.

```
# Convert the standardized data into PyTorch tensors

X_train_tensor = torch.tensor(X_train, dtype = torch.float32).to(device)
y_train_tensor = torch.tensor(y_train.values, dtype = torch.float32).to(device).view(-1, 1)
X_test_tensor = torch.tensor(X_test, dtype = torch.float32).to(device)
y_test_tensor = torch.tensor(y_test.values, dtype = torch.float32).to(device).view(-1, 1)

# Check the shapes

X_train_tensor.shape, y_train_tensor.shape, X_test_tensor.shape, y_test_tensor.shape

(torch.Size([7008, 20]),
 torch.Size([7008, 1]),
 torch.Size([1752, 20]),
 torch.Size([1752, 1]))
```

Initializing the neural network model

- We initialized the neural network model by specifying the input size, which corresponds to the number of features in the training data.

```
[ ] # Initialize the neural network model

input_size = X_train.shape[1]
model = LinearRegressionModel(input_size).to(device)

[ ] model

⇄ LinearRegressionModel(
  (hidden1): Linear(in_features=20, out_features=64, bias=True)
  (hidden2): Linear(in_features=64, out_features=32, bias=True)
  (output): Linear(in_features=32, out_features=1, bias=True)
)
```

Model Training

- **Loss Function (MSE):** Chosen to minimize squared prediction errors, effectively penalizing larger errors and refining prediction accuracy.
- **Optimizer (Adam):** Utilized for efficient parameter updates, leading to faster and more stable convergence during training with learning rate or 0.001.

```
# Define the loss function (Mean Squared Error) and optimizer (Adam)

criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```


Training

- The model is trained for **3000 epochs**, allowing the model to learn the data and optimize the weights gradually.
- The input tensor (`X_train_tensor`) is passed through the model to get predictions (outputs).
- The loss is calculated using the predicted output and the actual target tensor (`y_train_tensor`) using the defined criterion (MSE loss).
- The training loss is stored in `train_losses` at each epoch to track the model's progress during training.
- The validation loss is stored in `val_losses` at each epoch for monitoring overfitting or underfitting.
- Every 100 epochs, the training loss and validation loss are printed to monitor the model's performance. This allows tracking how the model is learning and adjusting to the data over time.



```
# Train the model for a specified number of epochs
```

```
num_epochs = 3000
```

```
train_losses = []
```

```
val_losses = []
```

```
for epoch in range(num_epochs):
```

```
    model.train()
```

```
    outputs = model(X_train_tensor)
```

```
    loss = criterion(outputs, y_train_tensor)
```

```
    optimizer.zero_grad()
```

```
    loss.backward()
```

```
    optimizer.step()
```

```
    train_losses.append(loss.item())
```

```
    model.eval()
```

```
    with torch.no_grad():
```

```
        y_pred_tensor = model(X_test_tensor)
```

```
        val_loss = criterion(y_pred_tensor, y_test_tensor)
```

```
        val_losses.append(val_loss.item())
```

```
if (epoch + 1) % 100 == 0:
```

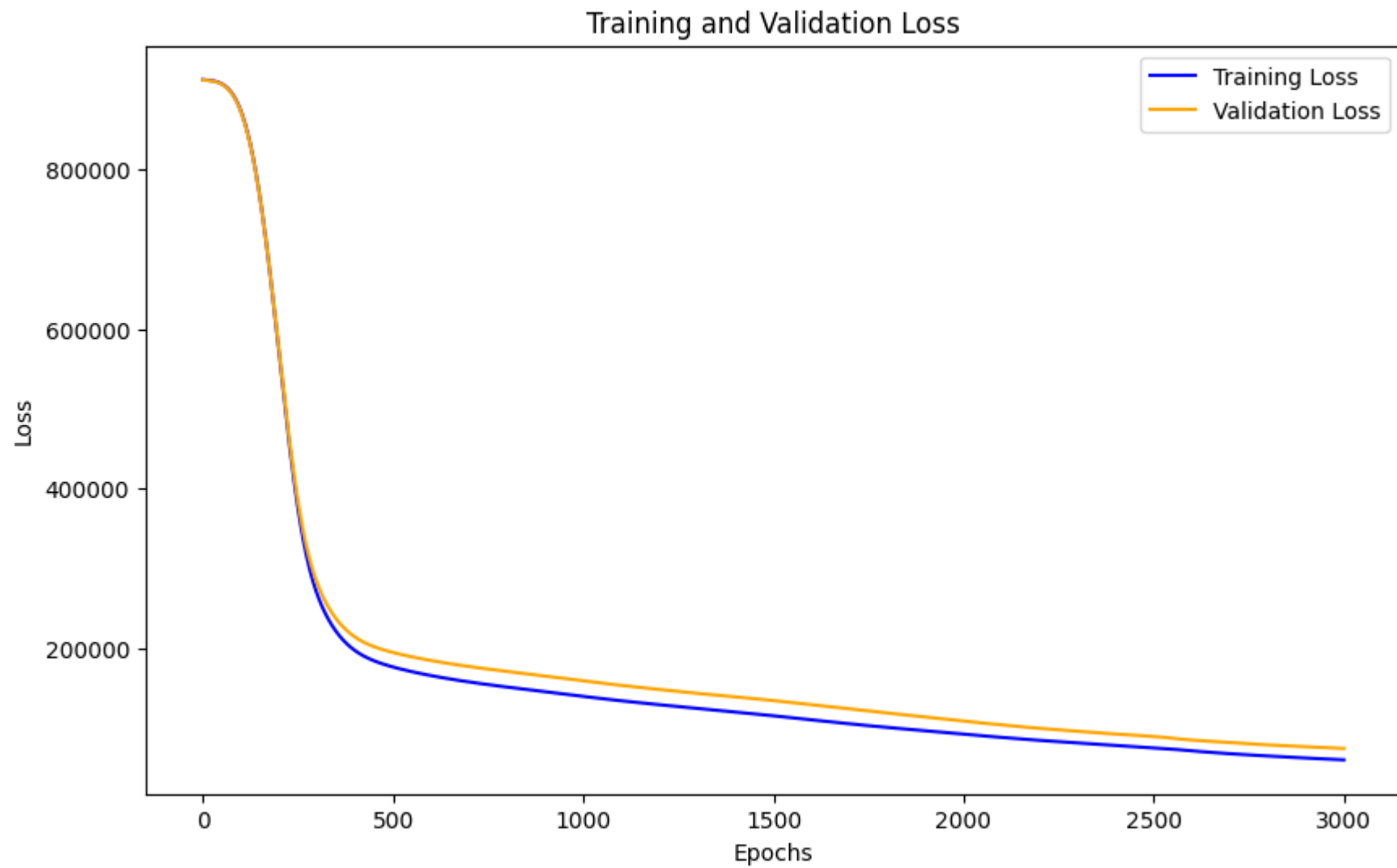
```
    print(f'Epoch [{epoch + 1}/{num_epochs}], Training Loss: {loss.item():.4f}, Validation Loss: {val_loss.item():.4f}')
```



```
Epoch [100/3000], Training Loss: 875218.5000, Validation Loss: 874099.8125
Epoch [200/3000], Training Loss: 577129.7500, Validation Loss: 578787.0000
Epoch [300/3000], Training Loss: 269833.8750, Validation Loss: 283287.2500
Epoch [400/3000], Training Loss: 197058.2812, Validation Loss: 213995.2031
Epoch [500/3000], Training Loss: 176155.4375, Validation Loss: 194334.7188
Epoch [600/3000], Training Loss: 165312.8750, Validation Loss: 184195.3125
Epoch [700/3000], Training Loss: 157346.8438, Validation Loss: 176670.7812
Epoch [800/3000], Training Loss: 150906.1250, Validation Loss: 170476.4844
Epoch [900/3000], Training Loss: 145102.7500, Validation Loss: 164711.0781
Epoch [1000/3000], Training Loss: 139227.7344, Validation Loss: 158760.0625
Epoch [1100/3000], Training Loss: 133685.8594, Validation Loss: 152987.4531
Epoch [1200/3000], Training Loss: 128755.4844, Validation Loss: 147861.2969
Epoch [1300/3000], Training Loss: 124073.6719, Validation Loss: 143002.0156
Epoch [1400/3000], Training Loss: 119580.7812, Validation Loss: 138509.9844
Epoch [1500/3000], Training Loss: 114860.3359, Validation Loss: 133948.0625
Epoch [1600/3000], Training Loss: 109816.3047, Validation Loss: 128823.5000
Epoch [1700/3000], Training Loss: 104834.8516, Validation Loss: 123519.7734
Epoch [1800/3000], Training Loss: 100239.3281, Validation Loss: 118326.6641
Epoch [1900/3000], Training Loss: 96007.8516, Validation Loss: 113288.4531
Epoch [2000/3000], Training Loss: 91793.2344, Validation Loss: 108237.1172
Epoch [2100/3000], Training Loss: 87734.2031, Validation Loss: 103429.3125
Epoch [2200/3000], Training Loss: 84028.0156, Validation Loss: 99101.1641
Epoch [2300/3000], Training Loss: 80798.2969, Validation Loss: 95379.7891
Epoch [2400/3000], Training Loss: 77749.1875, Validation Loss: 91953.1250
Epoch [2500/3000], Training Loss: 74774.7891, Validation Loss: 88906.6484
Epoch [2600/3000], Training Loss: 70862.9609, Validation Loss: 84367.0078
Epoch [2700/3000], Training Loss: 67270.8438, Validation Loss: 81131.8438
Epoch [2800/3000], Training Loss: 64406.9531, Validation Loss: 78283.0781
Epoch [2900/3000], Training Loss: 61800.7656, Validation Loss: 75992.5000
Epoch [3000/3000], Training Loss: 59571.9141, Validation Loss: 73805.0156
```

Visualizing Training & Validation Loss

- **Loss Reduction:** Training and validation losses decreased progressively, reflecting the model's capacity to fit and generalize.
- **Validation Stability:** Consistent validation loss shows effective learning without overfitting, a common issue in neural networks.



Evaluate the model performance

- Converts the model's predictions (`y_pred_tensor`), which are in the form of a PyTorch tensor, into a NumPy array.
- Converts the ground truth values (`y_test_tensor`) from a PyTorch tensor into a NumPy array.

```
# Evaluate the model performance on test data

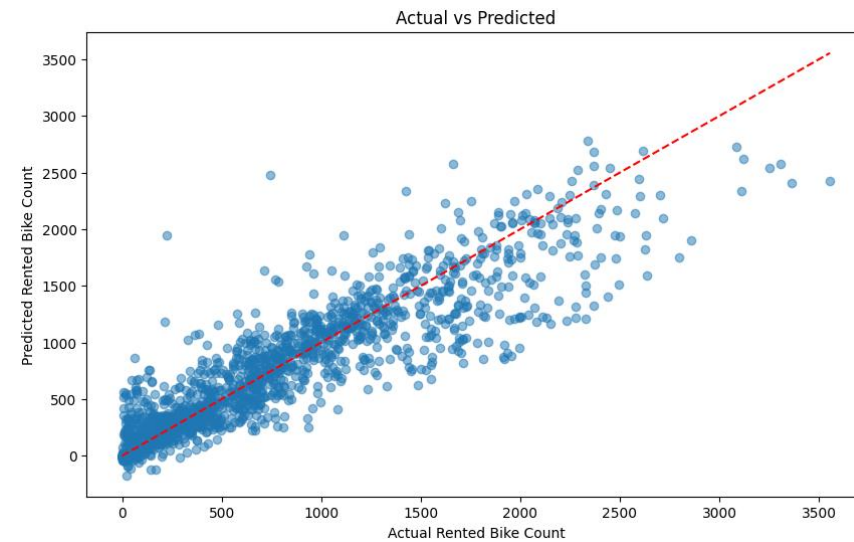
y_pred = y_pred_tensor.cpu().numpy() |
y_test_np = y_test_tensor.cpu().numpy()

[ ] # Visualize actual vs. predicted values using a scatter plot

plt.figure(figsize=(10, 6))
plt.scatter(y_test_np, y_pred, alpha=0.5)
plt.plot([y_test_np.min(), y_test_np.max()], [y_test_np.min(), y_test_np.max()], 'r--')
plt.xlabel('Actual Rented Bike Count')
plt.ylabel('Predicted Rented Bike Count')
plt.title('Actual vs Predicted')
plt.show()
```

Actual vs. Predicted Scatter Plot:

- Demonstrated alignment between true and predicted bike rental counts, validating the model's predictive accuracy.




Evaluation Matrix and Visualization

- We evaluated the model using metrics like Mean Absolute Error (MAE) and R-squared. These provide insights into prediction accuracy and model fit.
- The MAE value of 176.00 means that, on average, the model's predictions deviate by 176 bike rentals from the actual values in the test data.
- The R^2 score of 0.8229 means that approximately 82.29% of the variance in the actual rented bike counts can be explained by the model.

```
[ ] # Compute evaluation metrics

mae = mean_absolute_error(y_test_np, y_pred)
r2 = r2_score(y_test_np, y_pred)
print(f"Mean Absolute Error: {mae:.2f}")
print(f"R-squared: {r2:.4f}")
```



```
Mean Absolute Error: 176.00
R-squared: 0.8229
```

Predictions for Manual Inputs



The model allows us to predict bike rentals based on specific inputs such as temperature, humidity, and season.



We simulated predictions for different seasons using hypothetical conditions like a temperature of 22°C, moderate humidity, and a specific hour of the day.



We created a new input dictionary with all necessary features, scaled it using the same scaler used for training, and fed it into the trained model for predictions.


```
▶ # Predicting bike rentals for each season based on a input
seasons = ['Spring', 'Summer', 'Autumn', 'Winter']
```

```
▶ season_predictions = []

for season in seasons:
    new_input = {col: 0 for col in X.columns}
    new_input.update({
        'Temperature(°C)': 22,
        'Humidity(%)': 40,
        'Wind speed (m/s)': 3.5,
        'Visibility (10m)': 1500,
        'Solar Radiation (MJ/m2)': 0.5,
        'Hour_sin': np.sin(2 * np.pi * 14 / 24),
        'Hour_cos': np.cos(2 * np.pi * 14 / 24),
        'Month_sin': np.sin(2 * np.pi * 8 / 12),
        'Month_cos': np.cos(2 * np.pi * 8 / 12),
        f'Seasons_{season}': 1,
        'Holiday_No Holiday': 1,
        'Functioning Day_Yes': 0,
        'Year': 2024,
        'Day': 98
    })
```

```
input_df = pd.DataFrame([new_input])[X.columns]
input_scaled = scaler.transform(input_df)
input_tensor = torch.tensor(input_scaled, dtype=torch.float32).to(device)
```

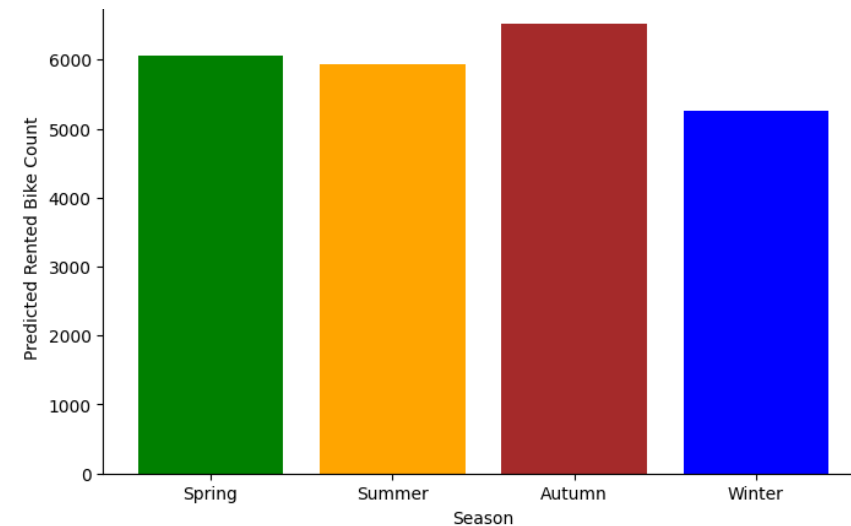
```
model.eval()
with torch.no_grad():
    prediction = model(input_tensor).cpu().numpy()[0][0]
    season_predictions.append((season, prediction))
```

```
# Display predictions
for season, pred in season_predictions:
    print(f"Predicted Rented Bike Count for {season}: {pred:.2f}")
```

```
Predicted Rented Bike Count for Spring: 6053.55
Predicted Rented Bike Count for Summer: 5938.88
Predicted Rented Bike Count for Autumn: 6521.32
Predicted Rented Bike Count for Winter: 5252.48
```

Seasonal Predictions Visualize

- The predicted bike rental counts for each season provide valuable insights into demand patterns.
- A bar chart summarizes the predictions, making it easier to compare across seasons.





Thank you