Step 1: Setup and Library Imports

This cell sets up the required Python environment and imports all the necessary libraries for data loading, preprocessing, modeling, and visualization.

- `!pip install -q keras` ensures that the Keras library is installed in the Colab environment.
- `numpy` and `pandas` are used for numerical operations and data manipulation.
- `sklearn.model_selection` provides the `train_test_split` method to divide data into training and test sets.
- `sklearn.preprocessing` provides `MinMaxScaler` and `StandardScaler` to normalize the input data.
- `tensorflow.keras` modules are used to build and train an LSTM neural network:
  - `Sequential` defines a linear stack of layers.
  - `LSTM`, `Dense`, `Dropout` are core layers used in the model.
  - `EarlyStopping` and `ReduceLROnPlateau` are callbacks that help prevent overfitting and dynamically adjust the learning rate.
  - `l2` is used for applying L2 regularization to reduce overfitting.
- `matplotlib.pyplot` is used to visualize training performance and model predictions.

Step 2: Load Dataset

```python data = pd.read_csv('EVChargingStationUsage.csv')

```
!pip install -q keras
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping,
ReduceLROnPlateau
from tensorflow.keras.regularizers import l2
import matplotlib.pyplot as plt

%% Load Data

data = pd.read_csv('EVChargingStationUsage.csv')
```

```
<ipython-input-24-442700d5faaa>:17: DtypeWarning: Columns (29,30,32)
have mixed types. Specify dtype option on import or set
low_memory=False.
  data = pd.read_csv('/content/drive/MyDrive/ML
courseproject/EVChargingStationUsage.csv')
```
```

▯ Step 2: Preview the Dataset

```python data.head()

```
data.head()

{"type":"dataframe","variable_name":"data"}
```

## ▯ Step 3: Data Preprocessing and Feature Selection

This cell performs the following tasks:

- Defines a helper function to convert time values from `hh:mm:ss` format into total seconds.
- Selects only the relevant columns from the dataset needed for modeling.
- Converts the two time-related columns (`Total Duration` and `Charging Time`) into numeric seconds using the helper function.
- Removes the original string-based time columns and drops rows with any missing values.
- Defines the feature matrix `X` and the target variable `y` for the model, where `X` contains numerical predictors and `y` contains the energy consumed (`Energy (kWh)`).

```
Preprocess the data
def convert_to_seconds(time_str):
    try:
        if isinstance(time_str, str):
            h, m, s = map(int, time_str.split(':'))
            return h * 3600 + m * 60 + s
    except ValueError as e:
        print(f"Invalid time format: {e}")
    return None

selected_columns = ['Total Duration (hh:mm:ss)', 'Charging Time
(hh:mm:ss)', 'Energy (kWh)',
                    'Fee', 'Gasoline Savings (gallons)', 'GHG Savings
(kg)']
data_selected = data[selected_columns]
data_selected['Total Duration (seconds)'] = data_selected['Total
Duration (hh:mm:ss)'].apply(convert_to_seconds)
data_selected['Charging Time (seconds)'] = data_selected['Charging
Time (hh:mm:ss)'].apply(convert_to_seconds)
data_cleaned = data_selected.drop(columns=['Total Duration
(hh:mm:ss)', 'Charging Time (hh:mm:ss)']).dropna()

%% Feature and Target Definition
X = data_cleaned[['Total Duration (seconds)', 'Charging Time
(seconds)', 'Fee',
                  'Gasoline Savings (gallons)', 'GHG Savings (kg)']]
```
```

```
y = data_cleaned['Energy (kWh)']

X.head()
```

```
<ipython-input-26-f434d202cb38>:11: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation:
https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#
returning-a-view-versus-a-copy
  data_selected['Total Duration (seconds)'] = data_selected['Total
Duration (hh:mm:ss)'].apply(convert_to_seconds)
<ipython-input-26-f434d202cb38>:12: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation:
https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#
returning-a-view-versus-a-copy
  data_selected['Charging Time (seconds)'] = data_selected['Charging
Time (hh:mm:ss)'].apply(convert_to_seconds)
```

{"type":"dataframe","variable_name":"X"}

Step 4: Normalize the Features and Target Variable

This step scales both the feature matrix `X` and the target variable `y` using `StandardScaler`, which standardizes the data to have zero mean and unit variance.

- `StandardScaler` is chosen as an alternative to `MinMaxScaler` to ensure features are on a comparable scale for LSTM training.
- The target `y` is reshaped and scaled to match the expected input format for regression.
- The output `X_scaled` and `y_scaled` are now ready for model training.

```
# Normalize using StandardScaler (alternative to MinMaxScaler)
scaler_X = StandardScaler()
scaler_y = StandardScaler()
X_scaled = scaler_X.fit_transform(X)
y_scaled = scaler_y.fit_transform(y.values.reshape(-1, 1))
X_scaled.shape
```

```
(259415, 5)
```

Step 5: Train-Test Split and Reshape for LSTM

- Splits the scaled data into training and testing sets using an 80/20 ratio.
- Reshapes the input features into 3D format as required by LSTM models: `(samples, timesteps, features)`.

- In this case, each sample is treated as a single timestep with multiple features, preparing it for sequential learning.

```
# Split data
X_train, X_test, y_train, y_test = train_test_split(X_scaled,
y_scaled, test_size=0.2, random_state=42)
# Reshape for LSTM
X_train = X_train.reshape((X_train.shape[0], 1, X_train.shape[1]))
X_test = X_test.reshape((X_test.shape[0], 1, X_test.shape[1]))
X_train.shape

(207532, 1, 5)
```

Step 6: Train-Test Split and Reshape for LSTM

- Splits the scaled data into training and testing sets using an 80/20 ratio.
- Reshapes the input features into 3D format as required by LSTM models: `(samples, timesteps, features)`.
- In this case, each sample is treated as a single timestep with multiple features, preparing it for sequential learning.

```
# %% Build the LSTM Model
model = Sequential([
    LSTM(16, activation='tanh', return_sequences=True, input_shape=(1,
X_train.shape[2]),
         kernel_regularizer=l2(0.01)),
    Dropout(0.3),
    LSTM(8, activation='tanh', return_sequences=False,
kernel_regularizer=l2(0.01)),
    Dropout(0.3),
    Dense(15, activation='relu', kernel_regularizer=l2(0.01)),
    Dense(1, activation='linear')
])

# Compile the model
model.compile(optimizer='adam', loss='mse', metrics=['mae'])

# Callbacks: Early Stopping and Learning Rate Scheduler
early_stopping = EarlyStopping(monitor='val_loss', patience=10,
restore_best_weights=True)
lr_scheduler = ReduceLROnPlateau(monitor='val_loss', factor=0.5,
patience=5, min_lr=1e-6)
model.summary()

/usr/local/lib/python3.10/dist-packages/keras/src/layers/rnn/
rnn.py:204: UserWarning: Do not pass an `input_shape`/`input_dim`
argument to a layer. When using Sequential models, prefer using an
`Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)

Model: "sequential_2"
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                         ┃ Output Shape                ┃
Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┩
━━━━━━━━━━━━━━━━━┩
│ lstm_4 (LSTM)                        │ (None, 1, 16)               │
1,408 │
├──────────────────────────────────────┼─────────────────────────────┤
─────────────────┤
│ dropout_4 (Dropout)                  │ (None, 1, 16)               │
0 │
├──────────────────────────────────────┼─────────────────────────────┤
─────────────────┤
│ lstm_5 (LSTM)                        │ (None, 8)                   │
800 │
├──────────────────────────────────────┼─────────────────────────────┤
─────────────────┤
│ dropout_5 (Dropout)                  │ (None, 8)                   │
0 │
├──────────────────────────────────────┼─────────────────────────────┤
─────────────────┤
│ dense_4 (Dense)                      │ (None, 15)                  │
135 │
├──────────────────────────────────────┼─────────────────────────────┤
─────────────────┤
│ dense_5 (Dense)                      │ (None, 1)                   │
16 │
└──────────────────────────────────────┴─────────────────────────────┘
─────────────────┘

 Total params: 2,359 (9.21 KB)

 Trainable params: 2,359 (9.21 KB)

 Non-trainable params: 0 (0.00 B)
```

Step 7: Train and Evaluate the LSTM Model

- Trains the LSTM model using the training data for up to 100 epochs with a batch size of 36.
- Uses 20% of the training data for validation during training to monitor overfitting.
- Includes two callbacks:
  - **EarlyStopping**: Stops training if validation loss stops improving.
  - **ReduceLROnPlateau**: Lowers the learning rate when the model plateaus.
- After training, evaluates the model on the test set and prints the final loss and mean absolute error (MAE) as performance metrics.

```python
# Train the model
history = model.fit(X_train, y_train, epochs=100, batch_size=36,
validation_split=0.2,
                    callbacks=[early_stopping,lr_scheduler])

# %% Evaluate the Model
loss, mae = model.evaluate(X_test, y_test)
print(f"Test Loss: {loss}")
print(f"Test MAE: {mae}")
```

```
Epoch 1/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 45s 7ms/step - loss: 0.3356 - mae:
0.2441 - val_loss: 0.0575 - val_mae: 0.0559 - learning_rate: 0.0010
Epoch 2/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 23s 5ms/step - loss: 0.1140 - mae:
0.1645 - val_loss: 0.0483 - val_mae: 0.0868 - learning_rate: 0.0010
Epoch 3/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 21s 4ms/step - loss: 0.1037 - mae:
0.1635 - val_loss: 0.0395 - val_mae: 0.0553 - learning_rate: 0.0010
Epoch 4/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 41s 4ms/step - loss: 0.0979 - mae:
0.1641 - val_loss: 0.0335 - val_mae: 0.0568 - learning_rate: 0.0010
Epoch 5/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 20s 4ms/step - loss: 0.0949 - mae:
0.1629 - val_loss: 0.0307 - val_mae: 0.0751 - learning_rate: 0.0010
Epoch 6/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 21s 4ms/step - loss: 0.0919 - mae:
0.1624 - val_loss: 0.0285 - val_mae: 0.0662 - learning_rate: 0.0010
Epoch 7/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 42s 5ms/step - loss: 0.0922 - mae:
0.1635 - val_loss: 0.0277 - val_mae: 0.0569 - learning_rate: 0.0010
Epoch 8/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 38s 4ms/step - loss: 0.0886 - mae:
0.1626 - val_loss: 0.0232 - val_mae: 0.0476 - learning_rate: 0.0010
Epoch 9/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 21s 5ms/step - loss: 0.0894 - mae:
0.1627 - val_loss: 0.0287 - val_mae: 0.0534 - learning_rate: 0.0010
Epoch 10/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 20s 4ms/step - loss: 0.0868 - mae:
0.1609 - val_loss: 0.0273 - val_mae: 0.0550 - learning_rate: 0.0010
Epoch 11/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 20s 4ms/step - loss: 0.0903 - mae:
0.1637 - val_loss: 0.0239 - val_mae: 0.0486 - learning_rate: 0.0010
Epoch 12/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 19s 4ms/step - loss: 0.0844 - mae:
0.1590 - val_loss: 0.0333 - val_mae: 0.0828 - learning_rate: 0.0010
Epoch 13/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 19s 4ms/step - loss: 0.0874 - mae:
0.1595 - val_loss: 0.0292 - val_mae: 0.0895 - learning_rate: 0.0010
Epoch 14/100
```

```
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 19s 4ms/step - loss: 0.0787 - mae:
0.1528 - val_loss: 0.0229 - val_mae: 0.0485 - learning_rate: 5.0000e-
04
Epoch 15/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 21s 4ms/step - loss: 0.0802 - mae:
0.1522 - val_loss: 0.0242 - val_mae: 0.0540 - learning_rate: 5.0000e-
04
Epoch 16/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 20s 4ms/step - loss: 0.0789 - mae:
0.1510 - val_loss: 0.0237 - val_mae: 0.0671 - learning_rate: 5.0000e-
04
Epoch 17/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 20s 4ms/step - loss: 0.0770 - mae:
0.1497 - val_loss: 0.0251 - val_mae: 0.0537 - learning_rate: 5.0000e-
04
Epoch 18/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 22s 4ms/step - loss: 0.0751 - mae:
0.1495 - val_loss: 0.0217 - val_mae: 0.0495 - learning_rate: 5.0000e-
04
Epoch 19/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 19s 4ms/step - loss: 0.0759 - mae:
0.1493 - val_loss: 0.0255 - val_mae: 0.0758 - learning_rate: 5.0000e-
04
Epoch 20/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 21s 4ms/step - loss: 0.0753 - mae:
0.1488 - val_loss: 0.0219 - val_mae: 0.0541 - learning_rate: 5.0000e-
04
Epoch 21/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 22s 4ms/step - loss: 0.0784 - mae:
0.1506 - val_loss: 0.0279 - val_mae: 0.0598 - learning_rate: 5.0000e-
04
Epoch 22/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 20s 4ms/step - loss: 0.0780 - mae:
0.1508 - val_loss: 0.0213 - val_mae: 0.0542 - learning_rate: 5.0000e-
04
Epoch 23/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 19s 4ms/step - loss: 0.0756 - mae:
0.1485 - val_loss: 0.0217 - val_mae: 0.0515 - learning_rate: 5.0000e-
04
Epoch 24/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 22s 4ms/step - loss: 0.0759 - mae:
0.1492 - val_loss: 0.0312 - val_mae: 0.0771 - learning_rate: 5.0000e-
04
Epoch 25/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 19s 4ms/step - loss: 0.0735 - mae:
0.1480 - val_loss: 0.0232 - val_mae: 0.0529 - learning_rate: 5.0000e-
04
Epoch 26/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 22s 4ms/step - loss: 0.0743 - mae:
```

```
0.1486 - val_loss: 0.0282 - val_mae: 0.0792 - learning_rate: 5.0000e-
04
Epoch 27/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 39s 4ms/step - loss: 0.0763 - mae:
0.1488 - val_loss: 0.0225 - val_mae: 0.0647 - learning_rate: 5.0000e-
04
Epoch 28/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 22s 4ms/step - loss: 0.0718 - mae:
0.1449 - val_loss: 0.0217 - val_mae: 0.0470 - learning_rate: 2.5000e-
04
Epoch 29/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 39s 4ms/step - loss: 0.0743 - mae:
0.1458 - val_loss: 0.0214 - val_mae: 0.0626 - learning_rate: 2.5000e-
04
Epoch 30/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 20s 4ms/step - loss: 0.0706 - mae:
0.1448 - val_loss: 0.0221 - val_mae: 0.0562 - learning_rate: 2.5000e-
04
Epoch 31/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 19s 4ms/step - loss: 0.0693 - mae:
0.1433 - val_loss: 0.0190 - val_mae: 0.0509 - learning_rate: 2.5000e-
04
Epoch 32/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 19s 4ms/step - loss: 0.0704 - mae:
0.1433 - val_loss: 0.0219 - val_mae: 0.0618 - learning_rate: 2.5000e-
04
Epoch 33/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 20s 4ms/step - loss: 0.0717 - mae:
0.1447 - val_loss: 0.0220 - val_mae: 0.0503 - learning_rate: 2.5000e-
04
Epoch 34/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 41s 4ms/step - loss: 0.0696 - mae:
0.1428 - val_loss: 0.0221 - val_mae: 0.0521 - learning_rate: 2.5000e-
04
Epoch 35/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 21s 4ms/step - loss: 0.0717 - mae:
0.1453 - val_loss: 0.0202 - val_mae: 0.0494 - learning_rate: 2.5000e-
04
Epoch 36/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 41s 5ms/step - loss: 0.0726 - mae:
0.1453 - val_loss: 0.0198 - val_mae: 0.0599 - learning_rate: 2.5000e-
04
Epoch 37/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 20s 4ms/step - loss: 0.0705 - mae:
0.1432 - val_loss: 0.0211 - val_mae: 0.0583 - learning_rate: 1.2500e-
04
Epoch 38/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 21s 5ms/step - loss: 0.0698 - mae:
0.1433 - val_loss: 0.0206 - val_mae: 0.0527 - learning_rate: 1.2500e-
```

```
04
Epoch 39/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 22s 5ms/step - loss: 0.0701 - mae: 
0.1425 - val_loss: 0.0197 - val_mae: 0.0519 - learning_rate: 1.2500e-
04
Epoch 40/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 41s 5ms/step - loss: 0.0713 - mae: 
0.1441 - val_loss: 0.0232 - val_mae: 0.0678 - learning_rate: 1.2500e-
04
Epoch 41/100
4612/4612 ━━━━━━━━━━━━━━━━━━━━ 42s 5ms/step - loss: 0.0723 - mae: 
0.1443 - val_loss: 0.0231 - val_mae: 0.0611 - learning_rate: 1.2500e-
04
1622/1622 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - loss: 0.0194 - mae: 
0.0506
Test Loss: 0.019473360851407051
Test MAE: 0.05087840557098389
```

󰊤 Step 8: Make Predictions and Evaluate Model Performance

- Uses the trained LSTM model to predict energy consumption on the test set.
- Applies inverse transformation to convert both predicted and actual values back to their original scale.
- Calculates key performance metrics:
  - **Mean Absolute Error (MAE)**: Average absolute difference between predicted and actual values.
  - **Root Mean Square Error (RMSE)**: Penalizes larger errors more than MAE.
  - **R-squared (R^2)**: Measures how well the model explains variance in the target variable.
- Prints the evaluation results to assess model accuracy.

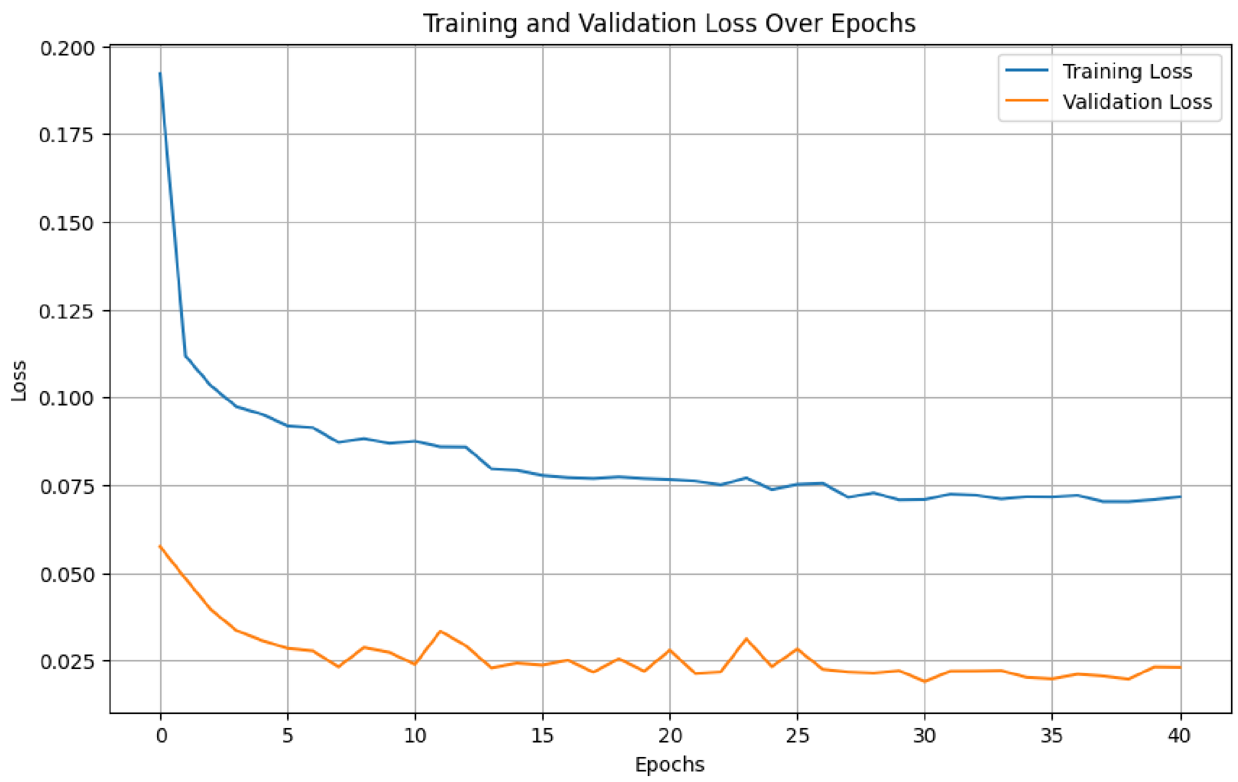
```python
# Predictions and inverse scaling
y_pred = model.predict(X_test)
y_pred_actual = scaler_y.inverse_transform(y_pred)
y_test_actual = scaler_y.inverse_transform(y_test)

# %% Performance Metrics
from sklearn.metrics import mean_squared_error, mean_absolute_error, 
r2_score

mae = mean_absolute_error(y_test_actual, y_pred_actual)
rmse = np.sqrt(mean_squared_error(y_test_actual, y_pred_actual))
r2 = r2_score(y_test_actual, y_pred_actual)

print(f"Mean Absolute Error (MAE): {mae:.2f}")
print(f"Root Mean Square Error (RMSE): {rmse:.2f}")
print(f"R-squared (R2): {r2:.2f}")
```

```
1622/1622 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step
Mean Absolute Error (MAE): 0.37
Root Mean Square Error (RMSE): 0.57
R-squared (R2): 0.99
```

Step 9: Visualize Training and Validation Loss

- Plots the training and validation loss over each epoch to visualize how the model learned over time.
- Helps identify signs of overfitting or underfitting:
  - A large gap between training and validation loss may indicate overfitting.
  - Parallel or converging lines suggest good generalization.
- This diagnostic plot is essential for understanding the training dynamics and fine-tuning the model if needed.

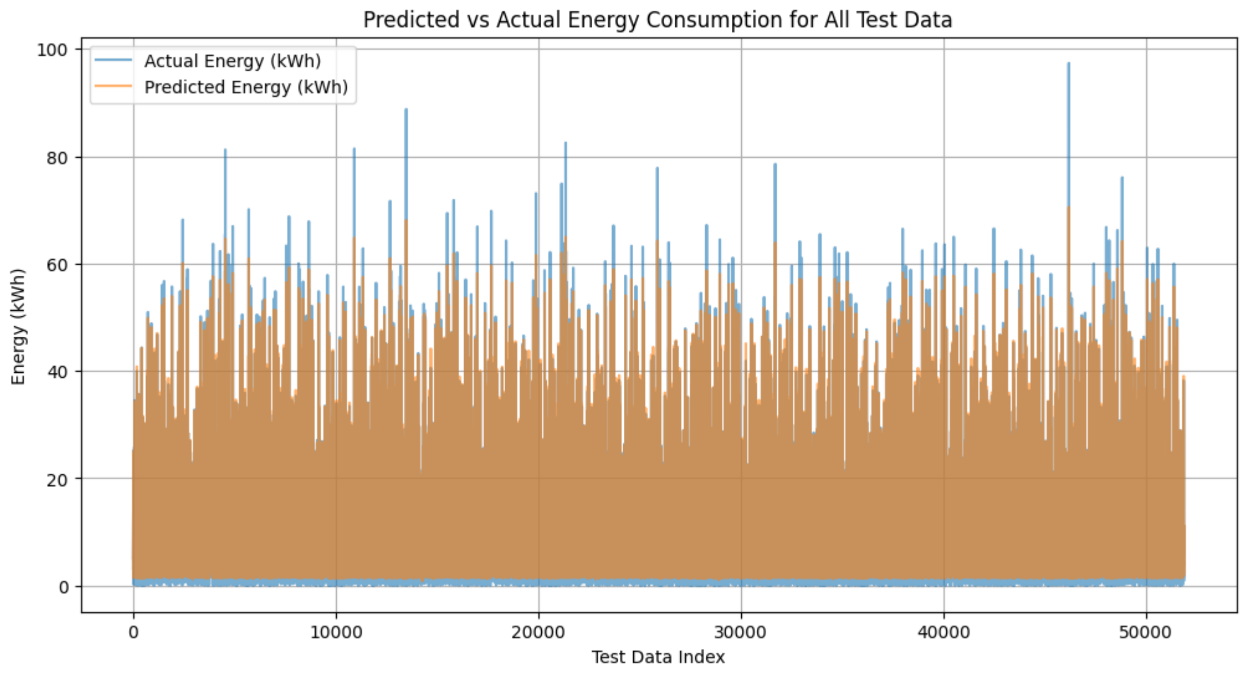
```
# %% Visualizations
# Training vs Validation Loss
plt.figure(figsize=(10, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss Over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid()
plt.show()
```



□ Step 10: Plot Actual vs. Predicted Energy for All Test Data

- Plots the predicted and actual energy consumption values across the entire test dataset.
- Visually compares the model's predictions against ground truth to assess alignment.
- A close overlap between the two lines indicates high prediction accuracy and model reliability.
- Helps detect patterns or outliers the model may have missed.

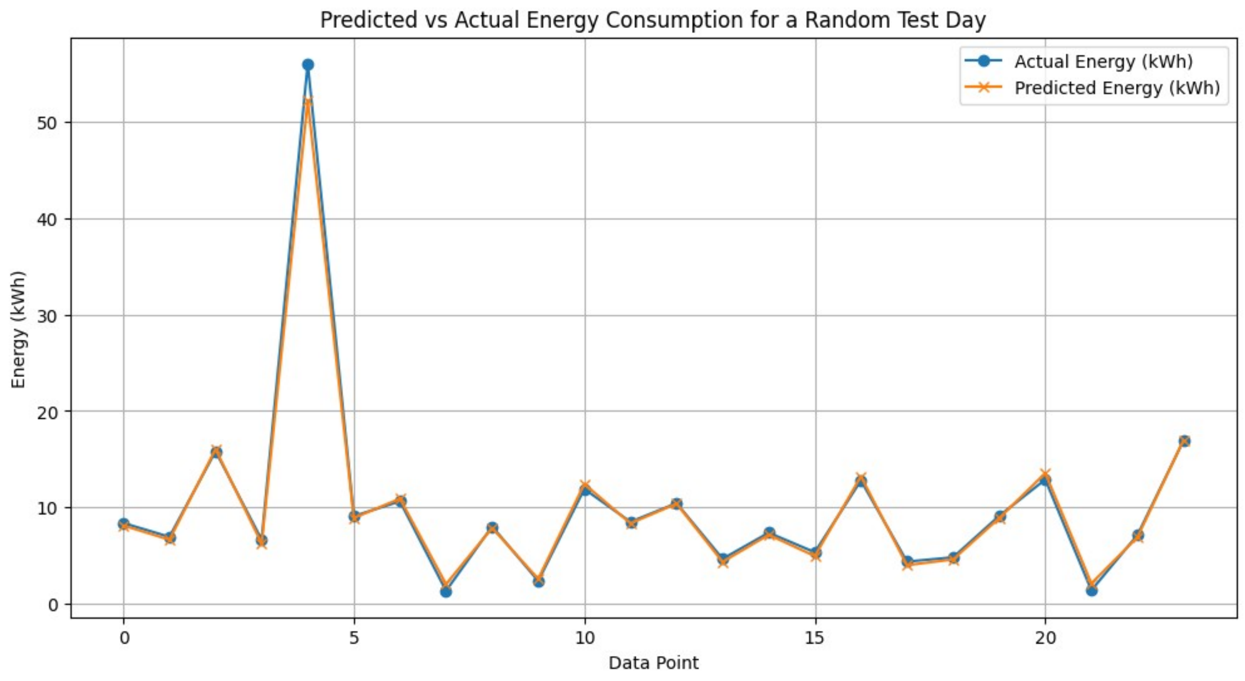
```
# Actual vs Predicted for All Test Data
plt.figure(figsize=(12, 6))
plt.plot(y_test_actual, label="Actual Energy (kWh)", alpha=0.6)
plt.plot(y_pred_actual, label="Predicted Energy (kWh)", alpha=0.6)
plt.title("Predicted vs Actual Energy Consumption for All Test Data")
plt.xlabel("Test Data Index")
plt.ylabel("Energy (kWh)")
plt.legend()
plt.grid()
plt.show()
```



Step 11: Predicted vs Actual Energy for a Random Test Day

- Selects a random 24-hour segment (assumed to represent one day) from the test set.
- Plots actual and predicted energy consumption values for that day.
- Allows for detailed inspection of model performance on a short time window.
- Useful for assessing how well the model captures daily usage patterns and fluctuations.

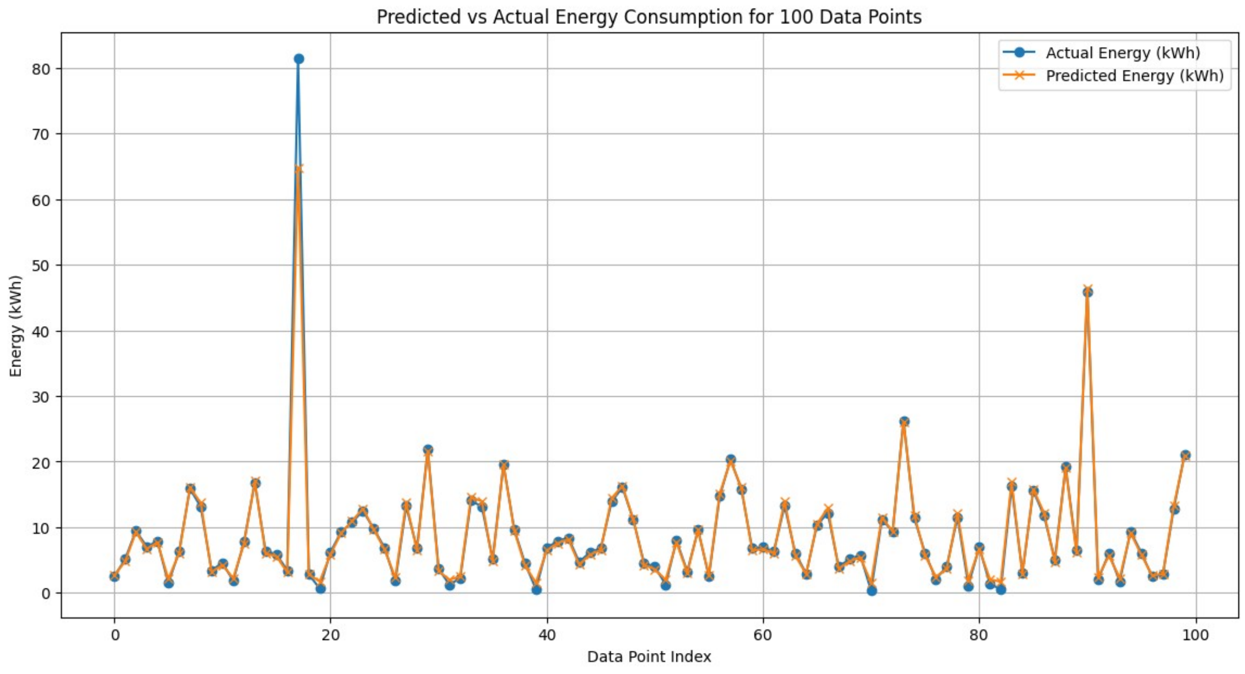
```
# Actual vs Predicted for a Random Day
random_day_index = np.random.choice(len(y_test_actual) // 24) * 24
y_test_day = y_test_actual[random_day_index:random_day_index + 24]
y_pred_day = y_pred_actual[random_day_index:random_day_index + 24]

plt.figure(figsize=(12, 6))
plt.plot(range(24), y_test_day, label="Actual Energy (kWh)",
marker='o')
plt.plot(range(24), y_pred_day, label="Predicted Energy (kWh)",
marker='x')
plt.title("Predicted vs Actual Energy Consumption for a Random Test
Day")
plt.xlabel("Data Point")
plt.ylabel("Energy (kWh)")
plt.legend()
plt.grid()
plt.show()
```



Step 12: Predicted vs Actual Energy for a Random 100-Point Segment

- Selects a random continuous segment of 100 data points from the test set.
- Plots both actual and predicted energy consumption values for detailed comparison.
- Helps evaluate local prediction accuracy and identify any short-term deviations.
- Useful for visualizing how the model handles real-world fluctuations over smaller time windows.

```
# Actual vs Predicted for a Random Segment of 100 Data Points
random_segment_index = np.random.choice(len(y_test_actual) - 100)  #
Ensure a valid range for 50 points
y_test_segment =
y_test_actual[random_segment_index:random_segment_index + 100]
y_pred_segment =
y_pred_actual[random_segment_index:random_segment_index + 100]

plt.figure(figsize=(14, 7))
plt.plot(range(100), y_test_segment, label="Actual Energy (kWh)",
marker='o')
plt.plot(range(100), y_pred_segment, label="Predicted Energy (kWh)",
marker='x')
plt.title("Predicted vs Actual Energy Consumption for 100 Data
Points")
plt.xlabel("Data Point Index")
plt.ylabel("Energy (kWh)")
plt.legend()
plt.grid()
plt.show()
```


Predicted vs Actual Energy Consumption for 100 Data Points
Actual Energy (kWh)
Predicted Energy (kWh)
Energy (kWh)
Data Point Index