# TRAFFIC MANAGEMENT SYSTEM

BATCH MEMBER

**953121106060 : VIDHYA M**

## PHASE 4 SUBMISSION DOCUMENT

Project Title : Traffic Management System

Phase 4 : Development Part 2

Topic : **In this section continue building the project by performing different activities like feature engineering, model training, evaluation etc as per the instructions in the project.**

# TRAFFIC MANAGEMENT SYSTEM

**INTRODUCTION :**

- In today's rapidly urbanizing world, efficient traffic management is crucial to ensuring smooth mobility, reducing congestion, and enhancing overall urban quality of life. Traditional traffic management systems are often limited in their capabilities and lack real-time data analysis, leading to inefficiencies and traffic-related issues. The integration of Internet of Things (IoT) technology revolutionizes traffic management by providing real-time data collection, analysis, and decision-making capabilities. This fusion of IoT and traffic management systems has the potential to transform cities into smart, interconnected, and efficient urban spaces.

- In summary, an IoT-based traffic management system holds the potential to transform urban mobility by harnessing the power of real-time data and intelligent decision-making. By creating smarter, more efficient cities, these systems pave the way for a future where traffic congestion is minimized, safety is enhanced, and the overall quality of urban life is significantly improved.

❖ In the face of rapidly increasing urbanization and growing vehicular traffic, cities around the world are facing significant challenges related to traffic congestion, safety concerns, and environmental issues. To address these challenges, modern cities are turning to innovative solutions, and one of the most promising technologies is the Internet of Things (IoT). IoT-based traffic management systems leverage the power of connected devices and real-time data analysis to transform the way traffic is monitored, controlled, and optimized within urban environments.

# GIVEN DATA SET :

| | DateTime | Junction | Vehicles | ID |
|---|---|---|---|---|
| 0 | 2015-11-01 00:00:00 | 1 | 15 | 20151101001 |
| 1 | 2015-11-01 01:00:00 | 1 | 13 | 20151101011 |
| 2 | 2015-11-01 02:00:00 | 1 | 10 | 20151101021 |
| 3 | 2015-11-01 03:00:00 | 1 | 7 | 20151101031 |
| 4 | 2015-11-01 04:00:00 | 1 | 9 | 20151101041 |

## NECESSARY STEP TO FOLLOW :

### 1.Import Libraries :

Start by importing the necessary libraries

**PROGRAM:**

```
import sys, os
import numpy as np
import matplotlib.pyplot as plt
import warnings
warnings.simplefilter(action='ignore')
import seaborn as sns
import pandas as pd
from datetime import datetime
import tensorflow as tf
import keras
from keras.models import Sequential
from keras.layers import Dense, SimpleRNN, LSTM, Activation, Dropout
import math
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
```

# 2.Load the Dataset:

You first need to collect and load the dataset. In a real-world s cenario, the dataset would likely come from traffic sensors, cameras, or other sources. For the purpose of this example, let's assume you ha ve a CSV (Comma-Separated Values) file containing traffic data.

## Program:

```
path ='/kaggle/input/traffic-prediction-dataset/traffic.csv'
data = pd.read_csv(path,index_col = 'DateTime')
```

# 3.Exploratory Data Analysis(EDA):

You can customize these visualizations based on your specific dataset and questions you want to answer. EDA helps you gain insight s into your data and guides further analysis and modeling decisions in your traffic management system.

## Program :

```
# Import necessary libraries
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Load your dataset (assuming it's already loaded into a Pandas DataF
rame named 'traffic_data')
```

```
# Replace 'traffic_data.csv' with your actual dataset file path if you ha
ven't loaded the data yet.
# traffic_data = pd.read_csv('traffic_data.csv')

# Display the first few rows of the dataset to understand its structure
print(traffic_data.head())

# Get summary statistics of numerical columns
print(traffic_data.describe())

# Check for missing values
print(traffic_data.isnull().sum())

# Visualize distribution of numerical features using histograms
plt.figure(figsize=(12, 6))
traffic_data.hist(bins=30, color='blue', alpha=0.7)
plt.tight_layout()
plt.show()

# Visualize correlations between numerical features using a heatmap
plt.figure(figsize=(10, 8))
correlation_matrix = traffic_data.corr()
sns.heatmap(data=correlation_matrix, annot=True, cmap='coolwarm',
fmt=".2f", linewidths=.5)
plt.title("Correlation Matrix")
plt.show()

# Visualize relationships between specific features using scatter plots
plt.figure(figsize=(8, 6))
sns.scatterplot(x='feature1', y='feature2', data=traffic_data)
plt.title('Relationship between Feature 1 and Feature 2')
plt.show()

# Visualize categorical variables using count plots
plt.figure(figsize=(8, 6))
sns.countplot(x='category', data=traffic_data)
plt.title('Count of Data Points in Each Category')
```

```
plt.show()
```

# 4.Feature Engineering:

Feature engineering is a critical step in the data preprocessing process that involves creating new features or transforming existing features to enhance the performance of machine learning models.

## Program:

```
# Assuming you have a timestamp column 'timestamp' in your dataset
traffic_data['day'] = traffic_data['timestamp'].dt.day
traffic_data['month'] = traffic_data['timestamp'].dt.month
traffic_data['year'] = traffic_data['timestamp'].dt.year
traffic_data['hour'] = traffic_data['timestamp'].dt.hour
traffic_data['is_weekend'] = traffic_data['timestamp'].dt.weekday >= 5
```

# 5.Split the Data:

Split your dataset into training and testing sets. This helps you evaluate your model's performance later.

```
X = df.drop('price', axis=1) # Features
 y = df['price'] # Target variable
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=4
```

# 6.Feature Scaling:

Apply feature scaling to normalize your data, ensuring that all features have similar scales. Standardization (scaling to mean=0 and std=1) is a common choice.

## Program:

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

# Importance of loading and processing dataset:

Loading and processing datasets are fundamental steps in any data-driven application, especially in a Traffic Management System. The importance of these steps can't be overstated, and here are several reasons.

## 1. Data Quality Assurance:

- **Completeness:** Ensuring all necessary data is present.
- **Accuracy:** Verifying data accuracy to prevent errors.
- **Consistency:** Standardizing data formats and units for consistent analysis.
- **Validity:** Checking data against predefined rules and constraints.

## 2. Data Understanding:

- **Exploratory Data Analysis (EDA):** Analyzing and understanding data patterns.
- **Identifying Outliers:** Detecting and handling outliers in the data.
- **Correlation Analysis:** Understanding relationships between different variables.

## 3. Feature Engineering:

- **Creating Informative Features:** Enhancing data with new features to aid model learning.
- **Transforming Variables:** Standardizing, normalizing, or transforming data for better model performance.

## 4. Data Preprocessing:

- **Handling Missing Values:** Devising strategies to deal with missing or null data points.
- **Categorical Encoding:** Converting categorical data into numerical forms for machine learning algorithms.

- **Scaling Features:** Scaling features to bring them within a similar range, avoiding domination by large-scale features.
- **Handling Imbalanced Data:** Addressing class imbalances in classification tasks.

### 5. Model Performance:

- **Enhanced Model Accuracy:** High-quality, well-processed data leads to better model accuracy and predictions.
- **Faster Model Convergence:** Clean, normalized data helps models converge faster during training.
- **Robustness:** Properly processed data ensures the model performs well on unseen or real-world data.

### 6. Decision Making:

- **Informed Decisions:** Accurate, reliable data supports informed decision-making processes.
- **Policy Planning:** Helps in planning traffic policies based on historical and real-time data.
- **Resource Allocation:** Efficiently allocating resources based on traffic patterns and demands.

### 7. User Experience:

- **Traffic Predictions:** Processed data supports accurate predictions, leading to better user experiences through optimized routes and travel times.
- **Real-time Updates:** Processed real-time data enables timely updates for users about traffic conditions.

### 8. Compliance and Regulations:

- **Data Privacy:** Ensuring data is processed in compliance with privacy regulations.

- **Ethical Use:** Responsible handling of data, ensuring fairness and unbiased algorithms.

# Program:

**In[1] :**

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv
)
```

**In[2] :**

```
import sys, os
import numpy as np
import matplotlib.pyplot as plt
import warnings
warnings.simplefilter(action='ignore')
import seaborn as sns
import pandas as pd
from datetime import datetime
import tensorflow as tf
import keras
from keras.models import Sequential
from keras.layers import Dense, SimpleRNN, LSTM, Activation, Dro
pout
import math
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
```

**In[3] :**

```
path ='/kaggle/input/traffic-prediction-dataset/traffic.csv'
data = pd.read_csv(path,index_col = 'DateTime')
```

**In[4] :**

```
data.head()
```

**Out[4] :**

| | Junction | Vehicles | ID |
|---|---|---|---|
| DateTime | | | |
| 2015-11-01 00:00:00 | 1 | 15 | 20151101001 |
| 2015-11-01 01:00:00 | 1 | 13 | 20151101011 |
| 2015-11-01 02:00:00 | 1 | 10 | 20151101021 |
| 2015-11-01 03:00:00 | 1 | 7 | 20151101031 |
| 2015-11-01 04:00:00 | 1 | 9 | 20151101041 |

**In[5] :**

```
data.shape
```

**Out[5] :**

(48120, 3)

**In[6] :**

```
data.dtypes
```

**Out[6] :**

```
Junction    int64
Vehicles    int64
ID          int64
dtype: object
```

**In[7] :**

```
data = data[data['Junction']==1]
```

**In[8] :**

```
data.shape
```

**Out[8] :**

(14592, 3)

**In[9] :**

```
data.isnull().sum()/data.count()*100
```

**Out[9] :**

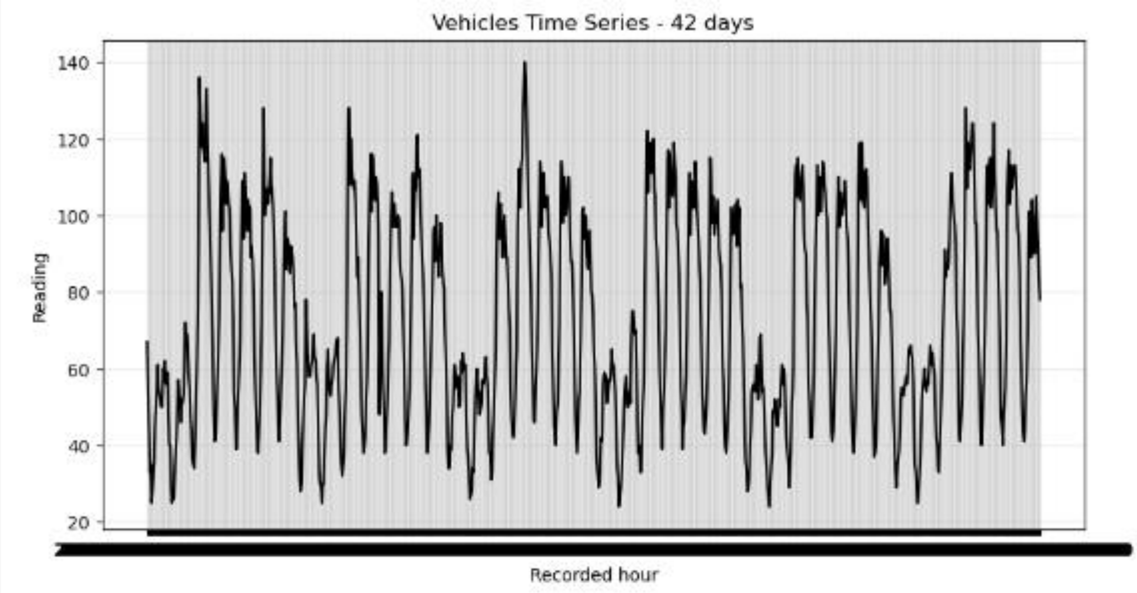
```
Junction    0.0
Vehicles    0.0
ID          0.0
dtype: float64
```

**In[10] :**

```
def last_n_days(df, feature, n_days):
    """
    Extract last n_days of a time series
    """

    return df[feature][-(24*n_days):]

def plot_last_n_days(df, feature, n_days):
    """
    Plot last n_days of an hourly time series
    """
    plt.figure(figsize = (10,5))
    plt.plot(last_n_days(df, feature, n_days), 'k-')
    plt.title('{0} Time Series - {1} days'
              .format(feature, n_days))
    plt.xlabel('Recorded hour')
    plt.ylabel('Reading')
    plt.grid(alpha=0.3)
```

**In[11] :**

```
plot_last_n_days(data, 'Vehicles', 42)
```




**In[12] :**

```
def get_keras_format_series(series):
    """
    Convert a series to a numpy array of shape
    [n_samples, time_steps, features]
    """

    series = np.array(series)
    return series.reshape(series.shape[0], series.shape[1], 1)

def get_train_test_data(df, series_name, series_days, input_hours,
                       test_hours, sample_gap=3):
    """
    Utility processing function that splits an hourly time series into
    train and test with keras-friendly format, according to user-specifie
d
    choice of shape.

    arguments
    ---------
    df (dataframe): dataframe with time series columns
    series_name (string): column name in df
    series_days (int): total days to extract
    input_hours (int): length of sequence input to network
```

```
    test_hours (int): length of held-out terminal sequence
    sample_gap (int): step size between start of train sequences; defaul
t 5

    returns
    ---------
    tuple: train_X, test_X_init, train_y, test_y
    """

    forecast_series = last_n_days(df, series_name, series_days).values
# reducing our forecast series to last n days

    train = forecast_series[:-test_hours] # training data is remaining da
ys until amount of test_hours
    test = forecast_series[-test_hours:] # test data is the remaining test_
hours

    train_X, train_y = [], []

    # range 0 through # of train samples - input_hours by sample_gap.
    # This is to create many samples with corresponding
    for i in range(0, train.shape[0]-input_hours, sample_gap):
        train_X.append(train[i:i+input_hours]) # each training sample is
of length input hours
        train_y.append(train[i+input_hours]) # each y is just the next step
after training sample

    train_X = get_keras_format_series(train_X) # format our new train
ing set to keras format
    train_y = np.array(train_y) # make sure y is an array to work prope
rly with keras

    # The set that we had held out for testing (must be same length as o
riginal train input)
    test_X_init = test[:input_hours]
    test_y = test[input_hours:] # test_y is remaining values from test set
```

```
    return train_X, test_X_init, train_y, test_y
```

**In[13] :**

```
series_days = 72
input_hours = 12
test_hours = 24

train_X, test_X_init, train_y, test_y = \
    (get_train_test_data(data, 'Vehicles', series_days,
                         input_hours, test_hours))
```

**In[14] :**

```
train_y.shape
```

**Out[14] :**

```
(564,)
```

**In[15] :**

```
print('Training input shape: {}'.format(train_X.shape))
print('Training output shape: {}'.format(train_y.shape))
print('Test input shape: {}'.format(test_X_init.shape))
print('Test output shape: {}'.format(test_y.shape))
```

```
Training input shape: (564, 12, 1)
Training output shape: (564,)
Test input shape: (12,)
Test output shape: (12,)
```

**In[16] :**

```
def fit_SimpleRNN(train_X, train_y, cell_units, epochs):
    """
    Fit Simple RNN to data train_X, train_y

    arguments
    ---------
    train_X (array): input sequence samples for training
    train_y (list): next step in sequence targets
    cell_units (int): number of hidden units for RNN cells
```

```
    epochs (int): number of training epochs
    """

    # initialize model
    model = Sequential()

    # construct an RNN layer with specified number of hidden units
    # per cell and desired sequence input format
    model.add(SimpleRNN(cell_units, input_shape=(train_X.shape[1],
1)))

    # add an output layer to make final predictions
    model.add(Dense(1))

    # define the loss function / optimization strategy, and fit
    # the model with the desired number of passes over the data (epoch
s)
    model.compile(loss='mean_squared_error', optimizer='adam')
    model.fit(train_X, train_y, epochs=epochs, batch_size=64, verbose
=0)

    return model
```

**In[17] :**

```
model = fit_SimpleRNN(train_X, train_y, cell_units=10, epochs=10)
```

**In[18] :**

```
def predict(X_init, n_steps, model):
    """
    Given an input series matching the model's expected format,
    generates model's predictions for next n_steps in the series
    """

    X_init = X_init.copy().reshape(1,-1,1)
    preds = []
```

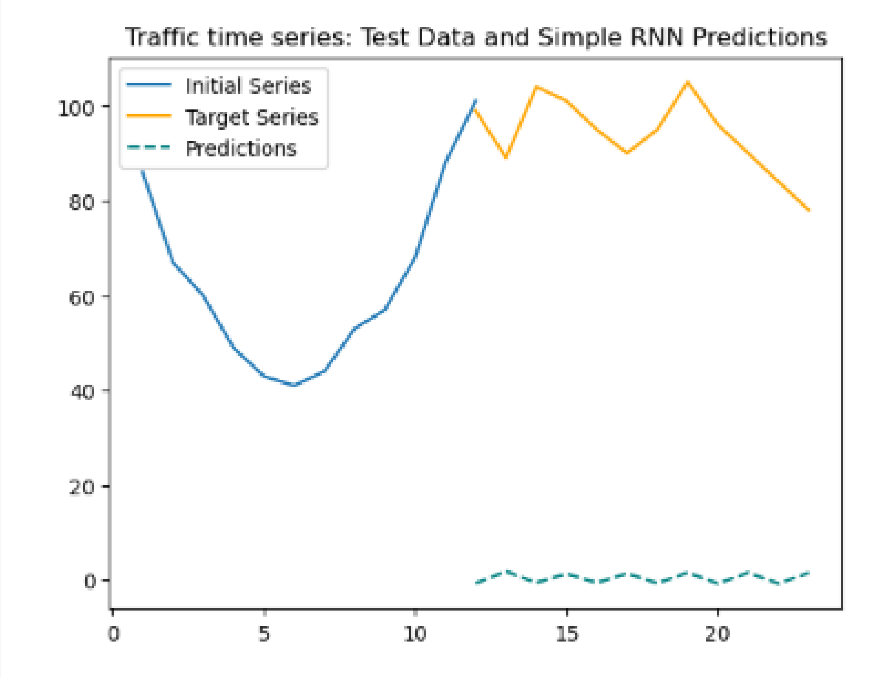
```
    # iteratively take current input sequence, generate next step pred,
    # and shift input sequence forward by a step (to end with latest pre
d).
    # collect preds as we go.
    for _ in range(n_steps):
        pred = model.predict(X_init)
        preds.append(pred)
        X_init[:,:-1,:] = X_init[:,1:,:] # replace first 11 values with 2nd th
rough 12th
        X_init[:,-1,:] = pred # replace 12th value with prediction

    preds = np.array(preds).reshape(-1,1)

    return preds

def predict_and_plot(X_init, y, model, title):
    """
    Given an input series matching the model's expected format,
    generates model's predictions for next n_steps in the series,
    and plots these predictions against the ground truth for those steps

    arguments
    ---------
    X_init (array): initial sequence, must match model's input shape
    y (array): true sequence values to predict, follow X_init
    model (keras.models.Sequential): trained neural network
    title (string): plot title
    """


    y_preds = predict(test_X_init, n_steps=len(y), model=model) # pre
dict through length of y
    # Below ranges are to set x-axes
    start_range = range(1, test_X_init.shape[0]+1) #starting at one thro
ugh to length of test_X_init to plot X_init
    predict_range = range(test_X_init.shape[0], test_hours)  #predict ra
nge is going to be from end of X_init to length of test_hours
```

```
    #using our ranges we plot X_init
    plt.plot(start_range, test_X_init)
    #and test and actual preds
    plt.plot(predict_range, test_y, color='orange')
    plt.plot(predict_range, y_preds, color='teal', linestyle='--')

    plt.title(title)
    plt.legend(['Initial Series','Target Series','Predictions'])
```

**In[19] :**

```
predict_and_plot(test_X_init, test_y, model,
                'Traffic time series: Test Data and Simple RNN Predictions
')
```

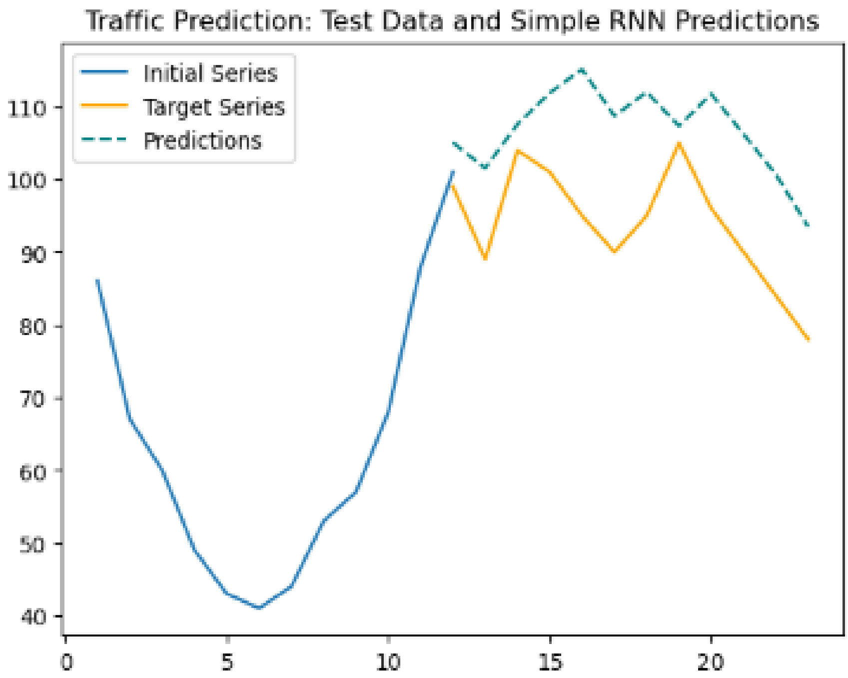
```
1/1 [==============================] - 0s 202ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 22ms/step
```




**In[20] :**

```
model = fit_SimpleRNN(train_X, train_y, cell_units=30, epochs=1200)
predict_and_plot(test_X_init, test_y, model,
                 'Traffic Prediction: Test Data and Simple RNN Predictions'
)
```

```
1/1 [==============================] - 0s 159ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 28ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 25ms/step
```




**In[21] :**

```
model.summary()
```

Model: "sequential_1"

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn_1 (SimpleRNN)     (None, 30)                960

dense_1 (Dense)              (None, 1)                 31

=================================================================
Total params: 991
Trainable params: 991
Non-trainable params: 0
_________________________________________________________________
```

**In[22] :**

```
def fit_LSTM(train_X, train_y, cell_units, epochs):
    """
    Fit LSTM to data train_X, train_y
```

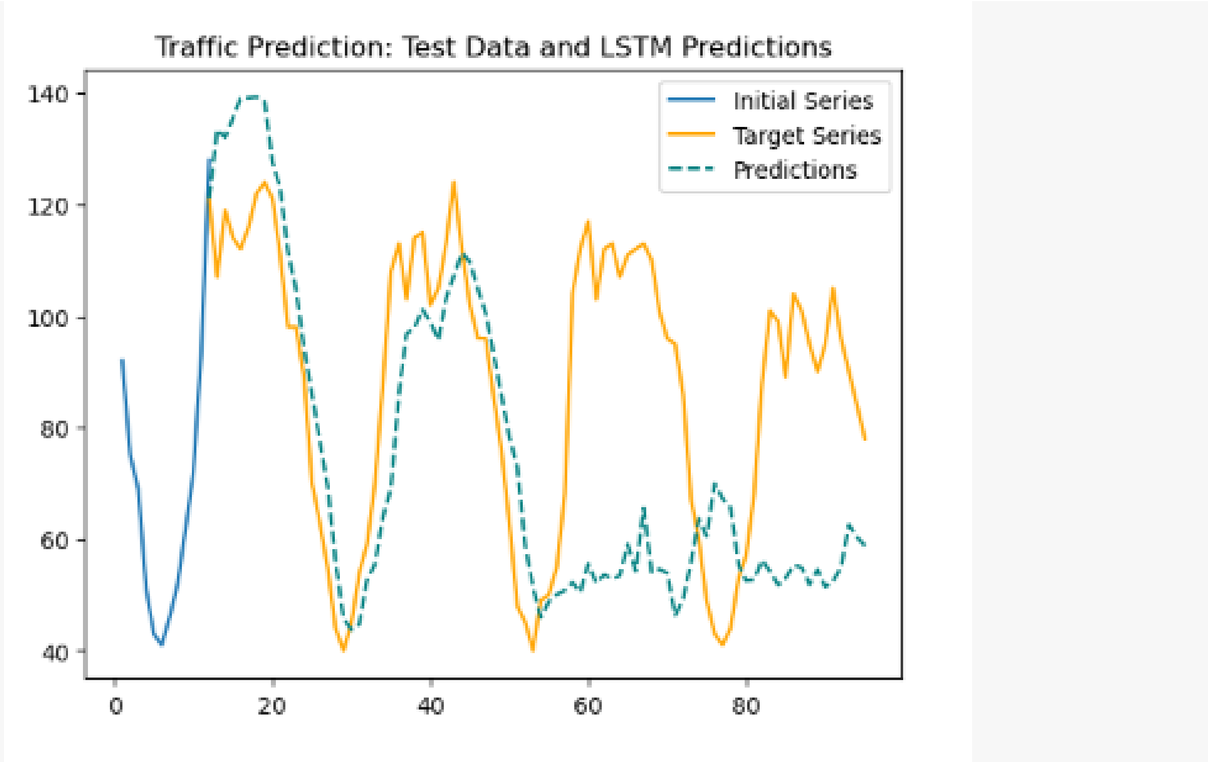
```
    arguments
    ---------
    train_X (array): input sequence samples for training
    train_y (list): next step in sequence targets
    cell_units (int): number of hidden units for LSTM cells
    epochs (int): number of training epochs
    """

    # initialize model
    model = Sequential()

    # construct a LSTM layer with specified number of hidden units
    # per cell and desired sequence input format
    model.add(LSTM(cell_units, input_shape=(train_X.shape[1],1))) #,
return_sequences= True))
    #model.add(LSTM(cell_units_l2, input_shape=(train_X.shape[1],1
)))

    # add an output layer to make final predictions
    model.add(Dense(1))

    # define the loss function / optimization strategy, and fit
    # the model with the desired number of passes over the data (epoch
s)
    model.compile(loss='mean_squared_error', optimizer='adam')
    model.fit(train_X, train_y, epochs=epochs, batch_size=64, verbose
=0)

    return model
```

**In[23] :**

```
series_days = 50
input_hours = 12
test_hours = 96

train_X, test_X_init, train_y, test_y = \
```

```
    (get_train_test_data(data, 'Vehicles', series_days,
                         input_hours, test_hours))

model = fit_LSTM(train_X, train_y, cell_units=70, epochs=3000)

predict_and_plot(test_X_init, test_y, model,
                 'Traffic Prediction: Test Data and LSTM Predictions')
```

```
1/1 [==============================] - 0s 444ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 25ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 25ms/step
1/1 [==============================] - 0s 24ms/step
1/1 [==============================] - 0s 23ms/step
```




# Conclusion:

- ❖ In this project, I trained a GRU Neural network to predicted the traffic on four junctions. I used a normalisation and differencing transform to achieve a stationary time series. As the Junctions varry in trends and seasonality, I took different approach for each junction to make it stationary. I applied the root mean squared error as the evaluation metric for the model. In addition to that I plotted the Predictions alongside the original test values. Take a ways from the data analysis.

- ❖ The Number of vehicles in Junction one is rising more rapidly compared to junction two and three. The spairsity of data in junction four bars me from making any conclusion on the same.

- The Junction one's traffic has a stronger weekly seasonality as well as hourly seasonality. Where as other junctions are significantly link.