**TRAFFIC MANAGEMENT USING**

**GNN AND**  **MAB WITH SDN ORCHESTRATION**

811321104014-DHANUSHYA M

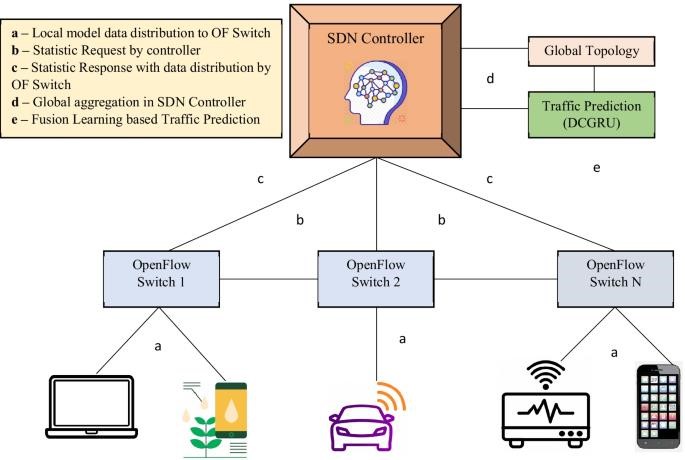
**Phase 5 Submission Document**

**Project Title:** Traffic Management

**Phase 5:** Project Documentation & Submission

**Topic**: In this section we will document the complete project and prepare it for submission.





**TRAFFIC MANAGEMENT**

**Introduction:**

* Traffic management is a critical task in software-defined IoT networks (SDN-IoTs) to efficiently manage network resources and ensure Quality of Service (QoS) for end-users. However, traditional traffic management approaches based on queuing theory or static policies may not be effective due to the dynamic and unpredictable nature of network traffic. In this paper, we propose a novel approach that leverages Graph Neural Networks (GNNs) and multi-arm bandit algorithms to dynamically optimize traffic management policies based on real-time network traffic patterns.

* Specifically, our approach uses a GNN model to learn and predict network traffic patterns and a multi-arm bandit algorithm to optimize traffic management policies based on these predictions. We evaluate the proposed approach on three different datasets, including a simulated corporate network (KDD Cup 1999), a collection of network traffic traces (CAIDA), and a simulated network environment with both normal and malicious traffic (NSL-KDD).

* The results demonstrate that our approach outperforms other state-of-the-art traffic management methods, achieving higher throughput, lower packet loss, and lower delay, while effectively detecting anomalous traffic patterns. The proposed approach offers a promising solution to traffic management in SDNs, enabling efficient resource management and QoS assurance.

**Keywords:**

Traffic management;anomaly detection;Intrusion detection;Network security;Internet of things;Network traffic analysis;Machine learning;SDN(software-defined networking);GNN(graph neural metwork);MAB(multi-armed bandit)

**TECHNIQUES:**

**Traffic Data Collection:**

Gather real-time traffic data using various sources like traffic cameras, sensors, and GPS data from vehicles.

**Data Preprocessing:**

Clean and format the collected data. This may include removing outliers, imputing missing values, and aggregating data. **Traffic Flow Analysis:**

Analyze traffic patterns, congestion points, and identify areas with high traffic volume. Techniques like data visualization and time-series analysis can help with this.

**Machine Learning Models:**

Utilize machine learning algorithms for predictive analysis. Models like regression and time series forecasting can help predict traffic patterns.

**Real-Time Monitoring:**

Implement a system to monitor traffic in real-time. This may involve developing a dashboard that displays live traffic data and alerts for congestion or accidents.

**Optimization Algorithms:**

See optimization techniques to suggest traffic signal timings and lane management for improving traffic flow. **Data Visualization:**

Create interactive maps and charts to visualize traffic data for better decision-making.

**DATASETS:**

**Traffic Flow Dataset:**

Real-time data from IoT sensors, including vehicle count and speed. **Weather Dataset:**

Weather conditions, such as temperature, precipitation, and wind speed.

**Environmental Dataset:**

Environmental data, including air quality, noise levels, and road conditions.

**Traffic Signal Data:**

Real-time data from traffic signal controllers.

**Historical Traffic Data:**

Historical traffic data for training predictive models and understanding long-term traffic trends.

**IMPORTING THE NECESSARY LIBRARIES:**

**1.Pandas:**

For data manipulation and analysis, especially when working with structured traffic data.

**2.NumPy:**

Useful for numerical operations and working with arrays, which can be helpful for mathematical modeling.

**3.Scikit-Learn:**

A machine learning library that provides various algorithms for prediction and classification tasks.

**4.TensorFlow or PyTorch:**

For deep learning and neural network-based traffic prediction models.

**5.Matplotlib and Seaborn:**

For data visualization, which can help in presenting traffic data trends and patterns.

**6.Folium:**

A library for interactive mapping, which can be handy for visualizing traffic conditions on maps.

**7.GeoPandas:**

If your project involves geospatial data, GeoPandas can help with handling and analyzing geospatial data.

**8.Statsmodels:**

Useful for statistical analysis of traffic data, especially when looking at factors affecting traffic.

**9.NetworkX:**

If your project involves road network analysis, NetworkX is a powerful library for working with graphs and networks.

**Proposed model:**

Traffic management is a critical step in optimizing network performance by controlling the rate of data transmission. It can help to manage congestion and reduce packet loss, improving the overall quality of service (QoS) for end-users. In this section, we will discuss the traffic management component of the proposed model.

Traffic management involves regulating the flow of data through the network by introducing delays and buffering packets. This is typically achieved using a token bucket algorithm, where tokens are generated at a fixed rate and consumed by packets as they are transmitted. If the bucket is empty, packets are queued until sufficient tokens are available.

The token bucket algorithm can be represented mathematically as follows: At time *t*, the number of tokens in the bucket can be calculated as:

𝐵(𝑡)=min(𝐵(𝑡−1)+𝑟(𝑡−𝑡𝑙𝑎𝑠𝑡),𝐵𝑚𝑎𝑥)

where 𝐵(𝑡−1)s the number of tokens at the previous time step, *r* is the token generation rate, *t* is the current time, 𝑡𝑙𝑎𝑠𝑡 is the time when the last token was generated, and 𝐵𝑚𝑎𝑥 is the maximum bucket size.

When a packet of size *P* arrives at time *t*, it is immediately transmitted if there are enough tokens available in the bucket:

if𝐵(𝑡)≥𝑃, thentransmitthepacket

Otherwise, the packet is queued until sufficient tokens become available:

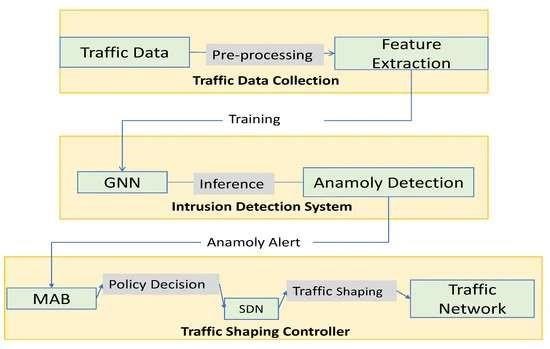
if𝐵(𝑡) <𝑃, thenaddthepackettothequeueif ,thenaddthepackettothequeue. Packets

in the queue are transmitted in order of arrival as soon as sufficient tokens become available.

The token bucket algorithm can be further optimized by adjusting the token generation rate based on network conditions. For example, if congestion is detected, the token generation rate can be reduced to prevent further congestion.

**Data flow:**

The Data flow of the proposed starts with the input data, which is passed through a feature extraction module to extract relevant features that will be used for traffic management. The extracted features are then passed to the GNN module, which is responsible for learning the complex patterns in the traffic data and predicting the optimal traffic-shaping policy.



**GNN for Understanding Traffic Pattern:**

Graph Neural Networks (GNNs) are a type of deep learning method that have recently gained popularity in the field of traffic analysis and prediction. GNNs are particularly effective in modeling and analyzing data that can be represented as graphs, which makes them well-suited for analyzing traffic patterns.

The basic idea behind GNNs is to learn a set of node and edge embeddings that capture the underlying structure of the graph. These embeddings can then be used to perform various downstream tasks, such as node classification, edge prediction, or graph clustering.

In the context of traffic analysis, GNNs can be used to model traffic flow data as a graph, where each node represents a road segment or intersection and each edge represents the flow of traffic between them. The GNN can then learn a set of embeddings that capture the underlying patterns of traffic flow, such as congestion, bottlenecking, and routing preferences.

The mathematical equations used in GNNs are typically based on messagepassing algorithms, which allow nodes in the graph to communicate and update their embeddings based on the embeddings of their neighbours. One commonly used message-passing algorithm is the Graph Convolutional Network (GCN), which is based on the following equation: {ℎ(𝑙+1)𝑖=𝜎(∑𝑗∈𝒩(𝑖)1𝑐𝑖𝑗𝑊(𝑙)ℎ(𝑙)𝑗)

**Necessary Steps To Follow:**

**1.Import Libraries:**

Start by importing the necessary libraries.

**Program:**

|  |
| --- |
| import numpy as np import pandas as pd import matplotlib.pyplot as plt import seaborn as sns import datetime import tensorflow  from statsmodels.tsa.stattools import adfuller from sklearn.preprocessing import MinMaxScaler from tensorflow import keras from keras import callbacks  from tensorflow.keras import Sequential from tensorflow.keras.layers import Conv2D, Flatten, Dense, LSTM, Dropout, GRU  , Bidirectional  from tensorflow.keras.optimizers import SGD from tensorflow.keras.models import Sequential from tensorflow.keras.layers import GRU, Dropout, Dense from tensorflow.keras.optimizers import SGD from tensorflow.keras import callbacks import math from sklearn.metrics import mean\_squared\_error    import warnings  warnings.filterwarnings("ignore") |

**2.load the dataset:**

Load your dataset into a pandas Dataframe.You can typically find traffic prediction datasets in csv format,but you can adapt this code to other format as needed.

**Program:**

data = pd.read\_csv(“../input/traffic-prediction-dataset/traffic.csv”) data.head()

**3.Data Exploration:**

Perform data exploration to understand your data better.This includes checking for missing values,exploring the data’s statistics, and visualizing it to identify patterns.

**Program:**

data["DateTime"]= pd.to\_datetime(data["DateTime"]) data =

data.drop(["ID"], axis=1) *#dropping IDs* data.info()

**4.Feature Engineering:**

Depending on your dataset,you may need to create new features or transform existing ones.This can involve one-hot encoding categorical variables , handling date/time data,or scaling numerical features.

**Program:**

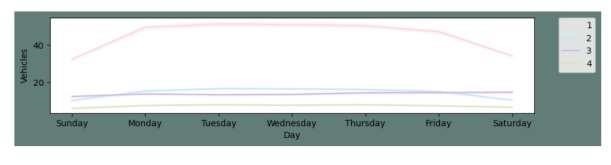
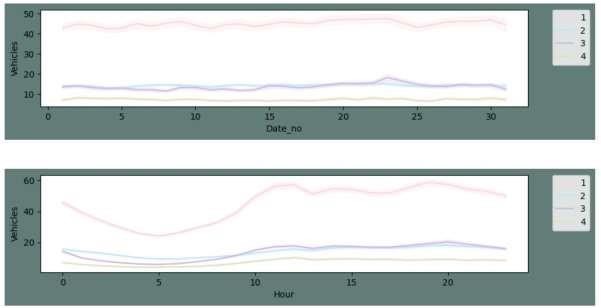
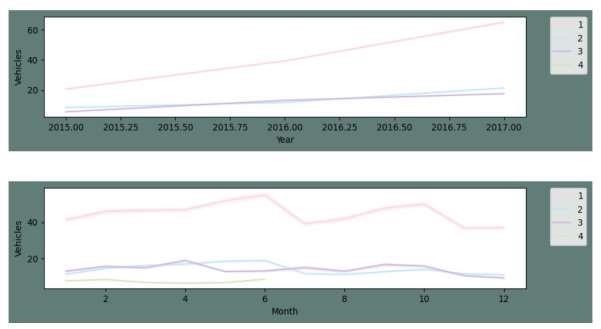
df["Year"]= df['DateTime'].dt.year df["Month"]= df['DateTime'].dt.month df["Date\_no"]= df['DateTime'].dt.day df["Hour"]= df['DateTime'].dt.hour df["Day"]= df.DateTime.dt.strftime("%A") df.head()

**5.Exploratory Data Analysis(EDA):**

Perform data exploration to understand your data better.This includes checking for missing values,exploring the data’s statistics, and visualizing it to identify patterns.

**Program:**

|  |
| --- |
| new\_features = [ "Year","Month", "Date\_no", "Hour", "Day"] for i **in** new\_features: plt.figure(figsize=(10,2),facecolor="#627D78")  ax=sns.lineplot(x=df[i],y="Vehicles",data=df, hue="Junction", palette=colo rs )  plt.legend(bbox\_to\_anchor=(1.05, 1), loc=2, borderaxespad=0.) |



**From The Above Plot Following Things Can Be Concluded:**

* Yearly, there has been an upward trend for all junctions except for the fourth junction. As we already established above that the fourth junction has limited data and that don't span over a year.
* We can see that there is an influx in the first and second junctions around June. I presume this may be due to summer break and activities around the same.
* Monthly, throughout all the dates there is a good consistency in data.
* For a day, we can see that are peaks during morning and evening times and a decline during night hours. This is as per expectation.
* For weekly patterns, Sundays enjoy smoother traffic as there are lesser vehicles on roads. Whereas Monday to Friday the traffic is steady.

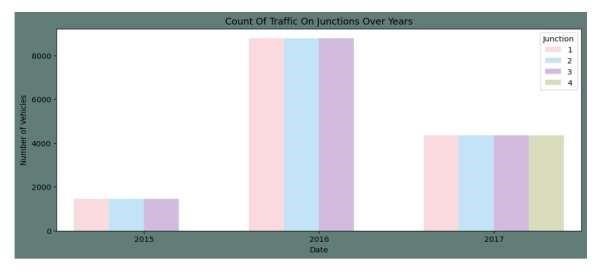
**Program:**

plt.figure(figsize=(12,5),facecolor="#627D78") count = sns.countplot(data=df,

x =df["Year"], hue="Junction", palette=colors)

count.set\_title("Count Of Traffic On Junctions Over Years")

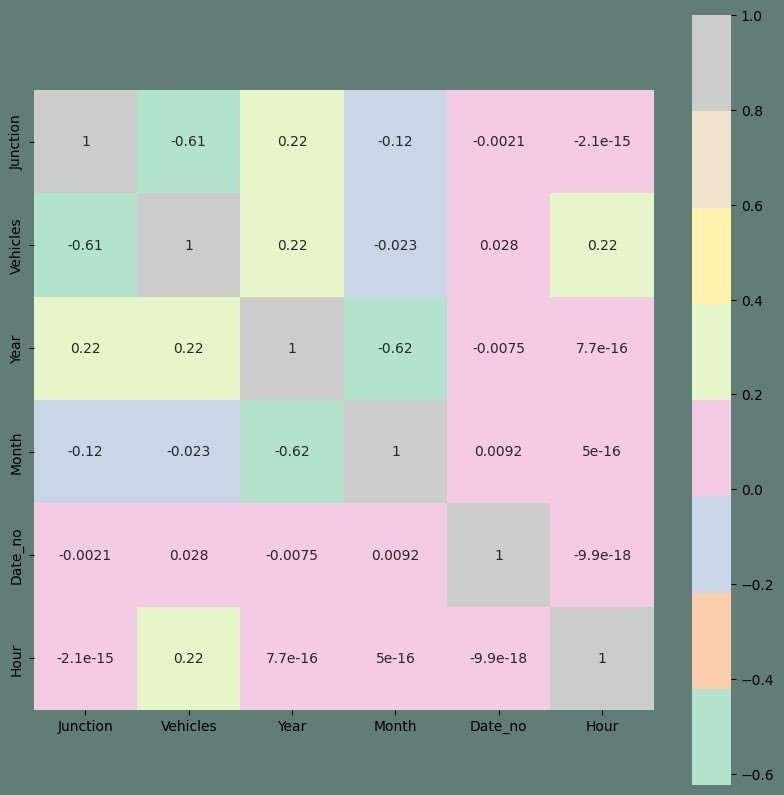
count.set\_ylabel("Number of Vehicles") count.set\_xlabel("Date")



numeric\_df = df.select\_dtypes(include=[np.number]) *# Select only numeric colu*  *mns* corrmat = numeric\_df.corr()

plt.subplots(figsize=(10,10),facecolor="#627D78") sns.heatmap(corrmat,cmap= "Pastel2",annot=True,square=True, )

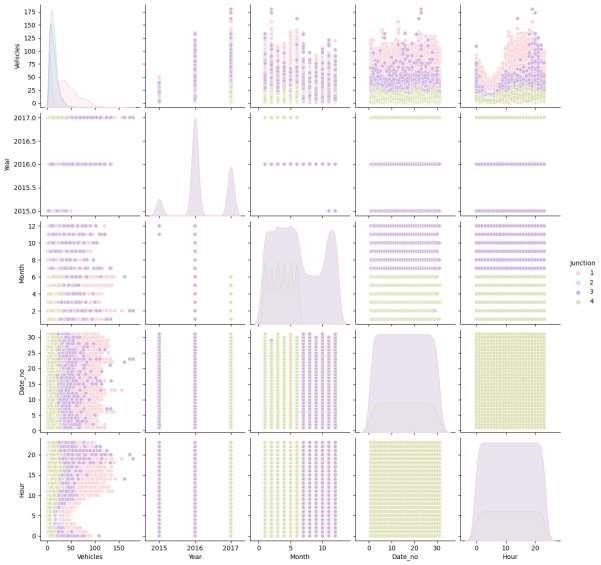
<Axes:>



The highest correlation is certainly with the preexisting feature. I will conclude my EDA with a pair plot. It's an interesting overall representation of any data

sns.pairplot(data=df, hue= "Junction",palette=colors)

<seaborn.axisgrid.PairGrid at 0x7adb85952830>



**Data Transformation And Preprocessing:**

* Creating different frames for each Junction and plotting them
* Transforming the series and plotting them
* Performing the Augmented Dickey-Fuller test to check the seasonality of transformed series
* Creating test and train sets

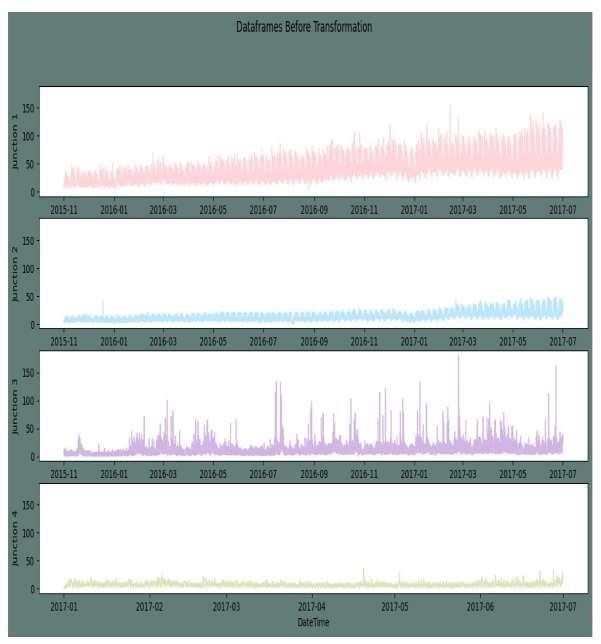
**Program:**

*#Pivoting data fron junction*  df\_J = data.pivot(columns="Junction", index="DateTime") df\_J.describe()



**Program:**

|  |
| --- |
| *#Creating new sets* df\_1 = df\_J[[('Vehicles', 1)]] df\_2 = df\_J[[('Vehicles', 2)]] df\_3 = df\_J[[('Vehicles', 3)]] df\_4 = df\_J[[('Vehicles', 4)]]  df\_4 = df\_4.dropna() *#Junction 4 has limited data only for a few months*  *#Dropping level one in dfs's index as it is a multi index data frame* list\_dfs = [df\_1, df\_2, df\_3, df\_4] for i **in** list\_dfs: i.columns= i.columns.droplevel(level=1)    *#Function to plot comparitive plots of dataframes* def Sub\_Plots4(df\_1, df\_2,df\_3,df\_4,title): fig, axes = plt.subplots(4, 1, figsize=(15, 8),facecolor="#627D78", sharey  =True)  fig.suptitle(title)  *#J1* pl\_1=sns.lineplot(ax=axes[0],data=df\_1,color=colors[0])  *#pl\_1=plt.ylabel()* axes[0].set(ylabel ="Junction 1")  *#J2*  pl\_2=sns.lineplot(ax=axes[1],data=df\_2,color=colors[1]) axes[1].set(ylabel ="Junction 2")  *#J3* pl\_3=sns.lineplot(ax=axes[2],data=df\_3,color=colors[2]) axes[2].set(ylabel ="Junction 3")  *#J4* pl\_4=sns.lineplot(ax=axes[3],data=df\_4,color=colors[3]) axes[3].set(ylabel ="Junction 4") |



**Steps For Transforming:**

* Normalizing
* Differencing

**Program:**

*# Normalize Function* def

Normalize(df,col): average = df[col].mean() stdev = df[col].std() df\_normalized = (df[col] - average) / stdev df\_normalized = df\_normalized.to\_frame() return df\_normalized, average, stdev

*# Differencing Function* def Difference(df,col, interval): diff = [] for i **in** range(interval, len(df)): value = df[col][i] - df[col][i - interval] diff.append(value) return diff

**In accordance with the above observations, Differencing to eliminate the seasonality should be performed as follows:**

* For Junction one, I will be taking a difference of weekly values.
* For junction two, The difference of consecutive days is a better choice  For Junctions three and four, the difference of the hourly values will serve the purpose.

|  |
| --- |
| *#Normalizing and Differencing to make the series stationary* |
| * df\_N1, av\_J1, std\_J1 = Normalize(df\_1, "Vehicles") * Diff\_1 = Difference(df\_N1, col="Vehicles", interval=(24\*7)) *#taking a w eek's diffrence*  df\_N1 = df\_N1[24\*7:] * df\_N1.columns = ["Norm"]   df\_N1["Diff"]= Diff\_1  df\_N2, av\_J2, std\_J2 = Normalize(df\_2, "Vehicles") |
| Diff\_2 = Difference(df\_N2, col="Vehicles", interval=(24)) *#taking a day 's diffrence* |
| df\_N2 = df\_N2[24:] |
| df\_N2.columns = ["Norm"] |
| df\_N2["Diff"]= Diff\_2 df\_N3, av\_J3, std\_J3 = Normalize(df\_3, "Vehicles") |
| Diff\_3 = Difference(df\_N3, col="Vehicles", interval=1) *#taking an hour' s diffrence* |
| df\_N3 = df\_N3[1:] |
| df\_N3.columns = ["Norm"] |
| df\_N3["Diff"]= Diff\_3 df\_N4, av\_J4, std\_J4 = Normalize(df\_4, "Vehicles") |
| * Diff\_4 = Difference(df\_N4, col="Vehicles", interval=1) *#taking an hour' s diffrence* * df\_N4 = df\_N4[1:]  df\_N4.columns = ["Norm"] |
| df\_N4["Diff"]= Diff\_4 |

The plots above seem linear. To ensure they are Stationary I will be performing an Augmented Dickey-Fuller test.

|  |
| --- |
| *#Stationary Check for the time series Augmented Dickey Fuller test* def Stationary\_check(df): check = adfuller(df.dropna()) print(f"ADF Statistic: **{**check[0]**}**") print(f"p-value: **{**check[1]**}**") print("Critical Values:") for key, value **in** check[4].items(): print('**\t%s**: **%.3f**' % (key, value)) if check[0] > check[4]["1%"]: print("Time Series is Non-Stationary")  else: print("Time Series is  Stationary")      *#Checking if the series is stationary*    List\_df\_ND = [ df\_N1["Diff"], df\_N2["Diff"], df\_N3["Diff"], df\_N4["Diff"]] print("Checking the transformed series for stationarity:") for i **in**  List\_df\_ND: print("**\n**")  Stationary\_check(i) |

**Checking the transformed series for stationarity:**

ADF Statistic: -15.265303390415504 pvalue:

4.798539876395756e-28 Critical Values:

1%: -3.431

5%: -2.862

10%: -2.567

Time Series is Stationary

ADF Statistic: -21.795891026940108 p-value: 0.0 Critical Values:

1%: -3.431

5%: -2.862

10%: -2.567

Time Series is Stationary

ADF Statistic: -28.001759908832508 p-value: 0.0 Critical Values:

1%: -3.431

5%: -2.862

10%: -2.567

Time Series is Stationary

ADF Statistic: -17.97909256305238 pvalue:

2.7787875325952613e-30 Critical Values:

1%: -3.432

5%: -2.862

10%: -2.567

Time Series is Stationary

**Program:**

|  |
| --- |
| *#Differencing created some NA values as we took a weeks data into consideratio n while difrencing* df\_J1 = df\_N1["Diff"].dropna() df\_J1  = df\_J1.to\_frame() df\_J2 =  df\_N2["Diff"].dropna() df\_J2  = df\_J2.to\_frame() df\_J3 =  df\_N3["Diff"].dropna() df\_J3  = df\_J3.to\_frame() df\_J4 =  df\_N4["Diff"].dropna() df\_J4  = df\_J4.to\_frame()    *#Splitting the dataset* def Split\_data(df):  training\_size = int(len(df)\*0.90) data\_len = len(df) train, test = df[0:training\_size],df[training\_size:data\_len] train, test = train.values.reshape(-1, 1), test.values.reshape(-1, 1) return train, test  *#Splitting the training and test datasets*  J1\_train, J1\_test = Split\_data(df\_J1)  J2\_train, J2\_test = Split\_data(df\_J2)  J3\_train, J3\_test = Split\_data(df\_J3)  J4\_train, J4\_test = Split\_data(df\_J4)    *#Target and Feature* def TnF(df): end\_len = len(df) X = [] y = [] steps = 32 for i **in** range(steps, end\_len):  X.append(df[i - steps:i, 0])  y.append(df[i, 0]) |
| X, y = np.array(X), np.array(y) return X ,y    *#fixing the shape of X\_test and X\_train* def FeatureFixShape(train, test): train = np.reshape(train, (train.shape[0], train.shape[1], 1)) test = np.reshape(test,  (test.shape[0],test.shape[1],1)) return train, test    *#Assigning features and target*  X\_trainJ1, y\_trainJ1 = TnF(J1\_train)  X\_testJ1, y\_testJ1 = TnF(J1\_test)  X\_trainJ1, X\_testJ1 = FeatureFixShape(X\_trainJ1, X\_testJ1)  X\_trainJ2, y\_trainJ2 = TnF(J2\_train)  X\_testJ2, y\_testJ2 = TnF(J2\_test)  X\_trainJ2, X\_testJ2 = FeatureFixShape(X\_trainJ2, X\_testJ2)  X\_trainJ3, y\_trainJ3 = TnF(J3\_train)  X\_testJ3, y\_testJ3 = TnF(J3\_test)  X\_trainJ3, X\_testJ3 = FeatureFixShape(X\_trainJ3, X\_testJ3)  X\_trainJ4, y\_trainJ4 = TnF(J4\_train)  X\_testJ4, y\_testJ4 = TnF(J4\_test)  X\_trainJ4, X\_testJ4 = FeatureFixShape(X\_trainJ4, X\_testJ4) |

**Define a function for linear regression training:**

|  |
| --- |
| def train\_linear\_regression(X\_train, y\_train, learning\_rate, epochs):  *# Initialize weights and bias* num\_features = X\_train.shape[1] weights = np.random.randn(num\_features) bias = np.random.randn() for epoch **in** range(epochs): *# Compute predictions* predictions = np.dot(X\_train, weights) + bias  *# Compute the mean squared error*  mse = np.mean((predictions - y\_train) \*\* 2)  *# Compute gradients* gradient\_weights = -2 \* np.dot(X\_train.T,  (y\_train - predictions)) / l en(X\_train)  gradient\_bias = -2 \* np.sum(y\_train - predictions) / len(X\_train)  *# Update weights and bias* weights -= learning\_rate \*  gradient\_weights bias -= learning\_rate \* gradient\_bias if (epoch + 1) % 100 == 0:  print(f'Epoch **{**epoch + 1**}**/**{**epochs**}**, MSE: **{**mse**:**.4f**}**') |

return weights, bias *# Function to make predictions* def predict(X, weights, bias): return np.dot(X, weights) + bias

|  |
| --- |
| *# Ensure that y\_train has the shape (number\_of\_samples,)* y\_trainJ1 = y\_trainJ1.reshape(-1)    *# Reshape X\_trainJ1 to have shape (number\_of\_samples,*  *number\_of\_features)* X\_trainJ1 = X\_trainJ1.reshape(X\_trainJ1.shape[0], 1)  *# Train the linear regression model for Junction*  *1* learning\_rate = 0.001 epochs = 1000    weights\_J1, bias\_J1 = train\_linear\_regression(X\_trainJ1, y\_trainJ1, learning\_r ate, epochs)    *# Ensure that y\_train has the shape (number\_of\_samples,)* y\_trainJ2 = y\_trainJ2.reshape(-1)    *# Reshape X\_trainJ2 to have shape (number\_of\_samples,*  *number\_of\_features)* X\_trainJ2 = X\_trainJ2.reshape(X\_trainJ2.shape[0], 1)  *# Train the linear regression model for Junction 2*  weights\_J2, bias\_J2 = train\_linear\_regression(X\_trainJ2, y\_trainJ2, learning\_r ate, epochs)    *# Ensure that y\_train has the shape (number\_of\_samples,)* y\_trainJ3 = y\_trainJ3.reshape(-1)    *# Reshape X\_trainJ3 to have shape (number\_of\_samples,*  *number\_of\_features)* X\_trainJ3 = X\_trainJ3.reshape(X\_trainJ3.shape[0], 1)  *# Train the linear regression model for Junction 3* weights\_J3, bias\_J3 = train\_linear\_regression(X\_trainJ3, y\_trainJ3, learning\_r ate, epochs)    *# Ensure that y\_train has the shape (number\_of\_samples,)* y\_trainJ4 = y\_trainJ4.reshape(-1)    *# Reshape X\_trainJ4 to have shape (number\_of\_samples,*  *number\_of\_features)* X\_trainJ4 = X\_trainJ4.reshape(X\_trainJ4.shape[0], 1)  *# Train the linear regression model for Junction 4* weights\_J4, bias\_J4 = train\_linear\_regression(X\_trainJ4, y\_trainJ4, learning\_r ate, epochs) |

Epoch 100/1000, MSE: 5.7588

Epoch 200/1000, MSE: 4.1379

Epoch 300/1000, MSE: 3.1330

Epoch 400/1000, MSE: 2.4985

Epoch 500/1000, MSE: 2.0886

Epoch 600/1000, MSE: 1.8163

Epoch 700/1000, MSE: 1.6293

Epoch 800/1000, MSE: 1.4961

Epoch 900/1000, MSE: 1.3973

Epoch 1000/1000, MSE: 1.3211

Epoch 100/1000, MSE: 9.4276

Epoch 200/1000, MSE: 6.9069

Epoch 300/1000, MSE: 5.4402

Epoch 400/1000, MSE: 4.5275

Epoch 500/1000, MSE: 3.9165

Epoch 600/1000, MSE: 3.4775

Epoch 700/1000, MSE: 3.1415

Epoch 800/1000, MSE: 2.8713

Epoch 900/1000, MSE: 2.6456

Epoch 1000/1000, MSE: 2.4520

Epoch 100/1000, MSE: 7.4961

Epoch 200/1000, MSE: 6.5373

Epoch 300/1000, MSE: 5.7255

Epoch 400/1000, MSE: 5.0329

Epoch 500/1000, MSE: 4.4383

Epoch 600/1000, MSE: 3.9252

Epoch 700/1000, MSE: 3.4808

Epoch 800/1000, MSE: 3.0945

Epoch 900/1000, MSE: 2.7579

Epoch 1000/1000, MSE: 2.4639

Epoch 100/1000, MSE: 14.2825

Epoch 200/1000, MSE: 10.1263

Epoch 300/1000, MSE: 7.3097

Epoch 400/1000, MSE: 5.3842

Epoch 500/1000, MSE: 4.0551

Epoch 600/1000, MSE: 3.1277

Epoch 700/1000, MSE: 2.4728

Epoch 800/1000, MSE: 2.0044

Epoch 900/1000, MSE: 1.6648 Epoch 1000/1000, MSE: 1.4151

**Program:**

|  |
| --- |
| *# Ensure that weights\_J1 has the correct shape* weights\_J1 = weights\_J1.reshape(-1)    *# Ensure that X\_testJ1 has the correct shape* X\_testJ1 =  X\_testJ1.reshape(X\_testJ1.shape[0], X\_testJ1.shape[1]) *#*  *Make predictions on the test data for Junction 1* y\_pred\_J1 = predict(X\_testJ1, weights\_J1, bias\_J1)  *# Calculate and print the Mean Squared Error (MSE) for Junction 1* mse\_J1 = np.mean((y\_pred\_J1 - y\_testJ1) \*\* 2) print(f'MSE for  Junction 1: **{**mse\_J1**:**.4f**}**')    *# Repeat the process for Junction 2* weights\_J2  = weights\_J2.reshape(-1)  X\_testJ2 = X\_testJ2.reshape(X\_testJ2.shape[0], X\_testJ2.shape[1]) |
| y\_pred\_J2 = predict(X\_testJ2, weights\_J2, bias\_J2) mse\_J2 = np.mean((y\_pred\_J2 - y\_testJ2) \*\* 2) print(f'MSE for Junction 2: **{**mse\_J2**:**.4f**}**')    *# Repeat the process for Junction 3* weights\_J3 = weights\_J3.reshape(-1)  X\_testJ3 = X\_testJ3.reshape(X\_testJ3.shape[0],  X\_testJ3.shape[1]) y\_pred\_J3 = predict(X\_testJ3, weights\_J3, bias\_J3) mse\_J3 = np.mean((y\_pred\_J3 - y\_testJ3) \*\* 2) print(f'MSE for Junction 3: **{**mse\_J3**:**.4f**}**')    *# Repeat the process for Junction 4* weights\_J4 = weights\_J4.reshape(-1)  X\_testJ4 = X\_testJ4.reshape(X\_testJ4.shape[0],  X\_testJ4.shape[1]) y\_pred\_J4 = predict(X\_testJ4, weights\_J4, bias\_J4) mse\_J4 = np.mean((y\_pred\_J4 - y\_testJ4) \*\* 2) print(f'MSE for Junction 4: **{**mse\_J4**:**.4f**}**') |

MSE for Junction 1: 1.9239

MSE for Junction 2: 4.0010

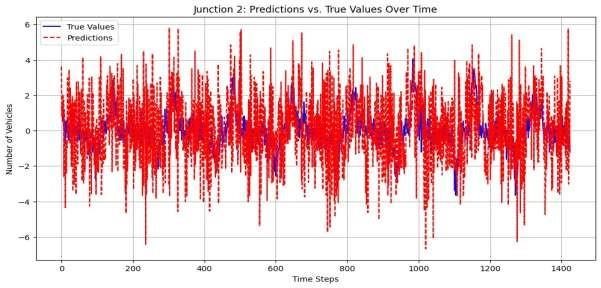
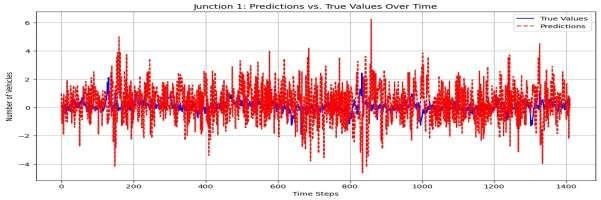
MSE for Junction 3: 3.5524

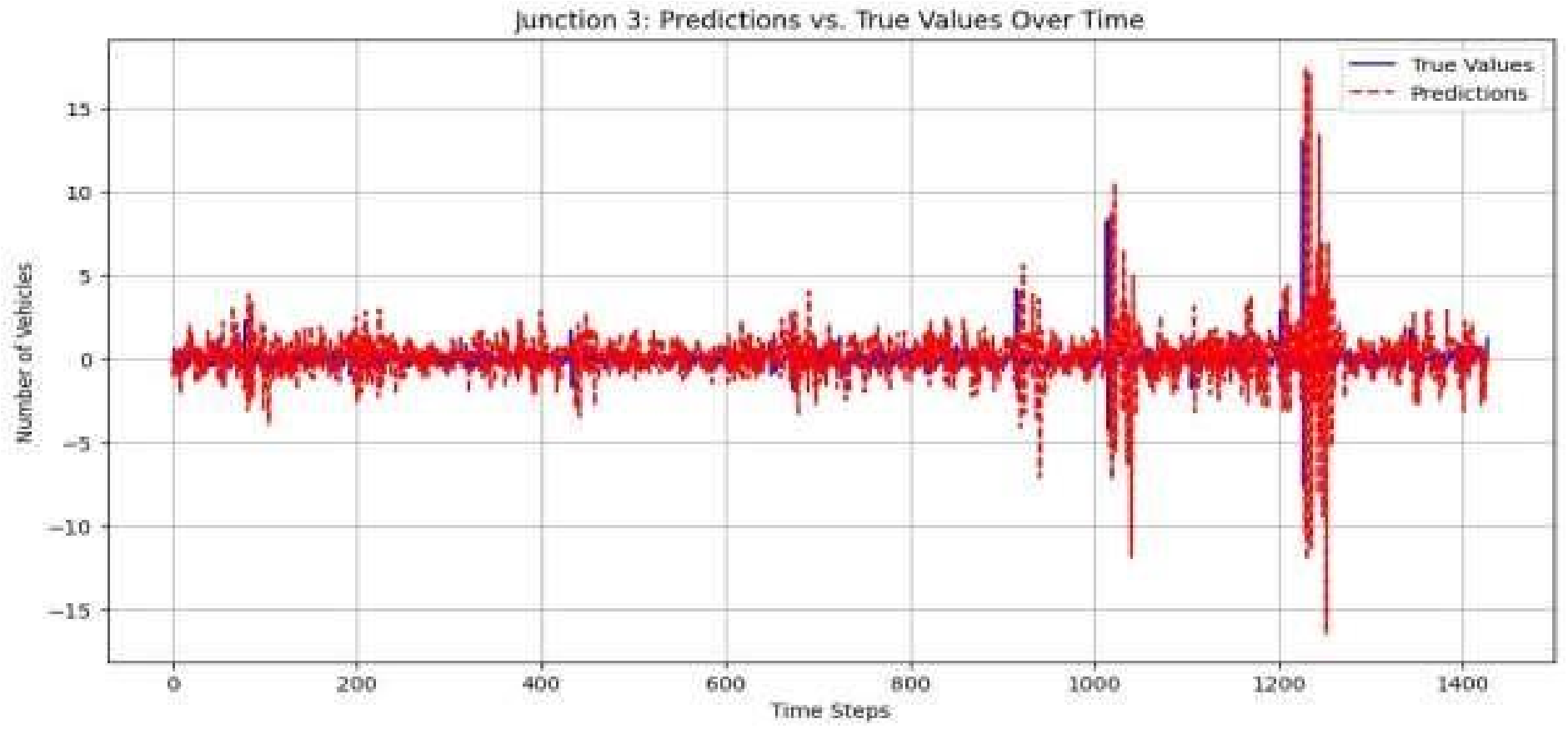
MSE for Junction 4: 2.5888

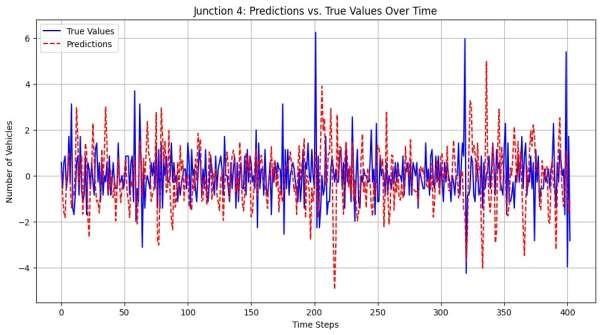
**Program:**

|  |
| --- |
| def plot\_predictions\_vs\_true(junction\_name, y\_true, y\_pred): plt.figure(figsize=(12, 6))  plt.plot(y\_true, label='True Values', color='blue') plt.plot(y\_pred, label='Predictions', color='red', linestyle='dashed') plt.title(f'**{**junction\_name**}**: Predictions vs. True Values Over Time')  plt.xlabel('Time Steps') plt.ylabel('Number of Vehicles') plt.legend() plt.grid(True) plt.show() |

|  |
| --- |
| *# Plot for Junction 1* plot\_predictions\_vs\_true("Junction 1", y\_testJ1, y\_pred\_J1)    *# Plot for Junction 2* plot\_predictions\_vs\_true("Junction 2", y\_testJ2, y\_pred\_J2)    *# Plot for Junction 3* plot\_predictions\_vs\_true("Junction 3", y\_testJ3, y\_pred\_J3)    *# Plot for Junction 4* plot\_predictions\_vs\_true("Junction 4", y\_testJ4, y\_pred\_J4) |

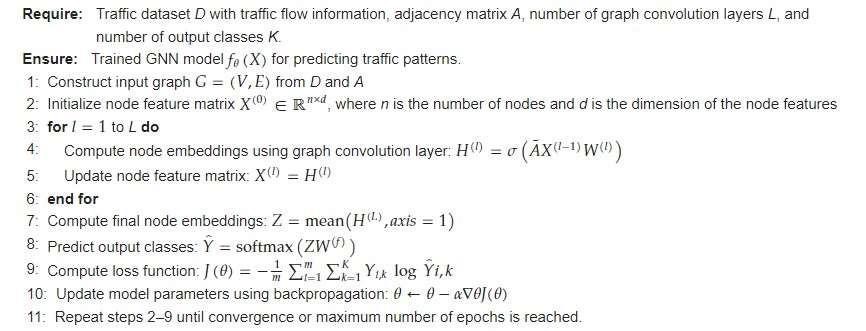




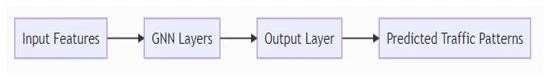


**GNN For Understanding Traffic Patterns:**

Algorithm:

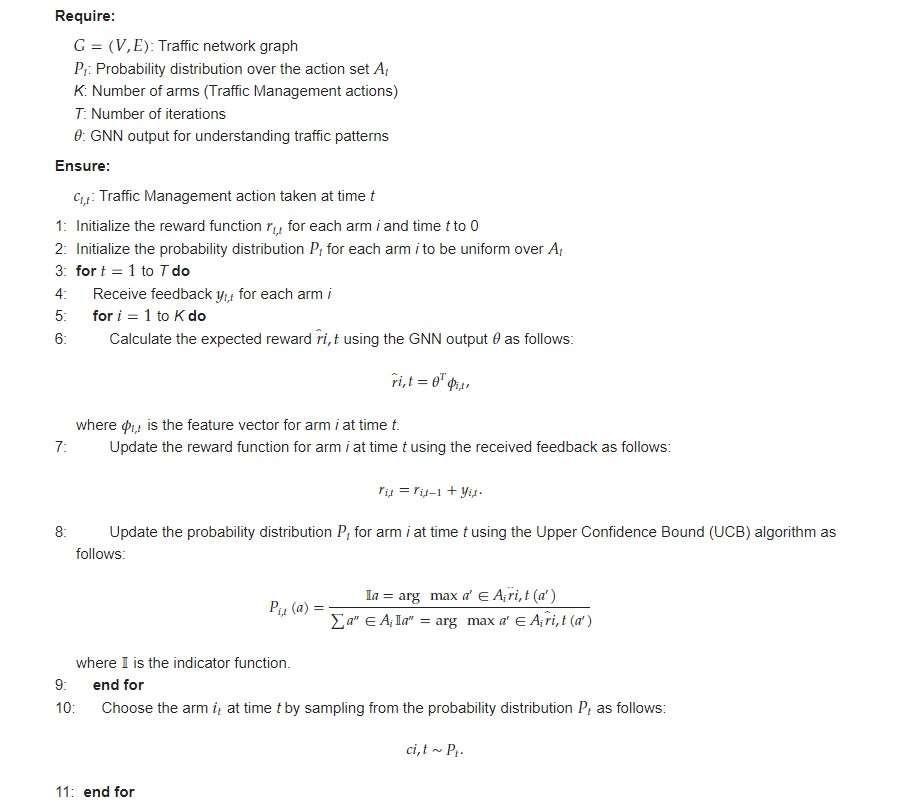


**GNN Architecture:**



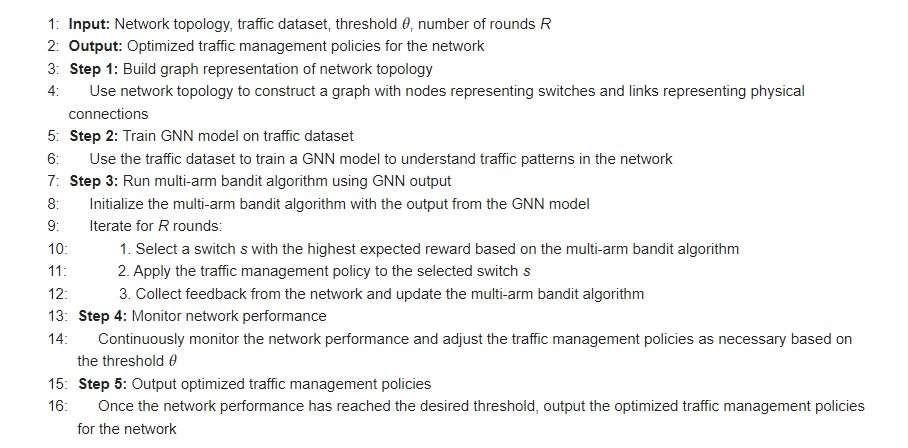
**Multi-Arm Bandit Algorithm For Traffic Management Using GNN Output:**

Algorithm:



**SDN orar Machestration algorithm for traffic management using GNN and multi-arm bandit:**

Algorithm:



**Hardware and software:**

The experiments are conducted on a server with 64GB RAM, Intel Xeon CPU, and Ubuntu 18.04 operating system. We use Python 3.7 and PyTorch 1.8.1 for developing the GNN model and multi-arm bandit algorithm. The SDN controller is implemented using Ryu v4.34.

**Preprocessing:**

Before training the GNN model, we preprocess the network traffic dataset by extracting features such as packet sizes, flow duration, and number of packets per flow. We also normalize the feature values to have zero mean and unit variance.

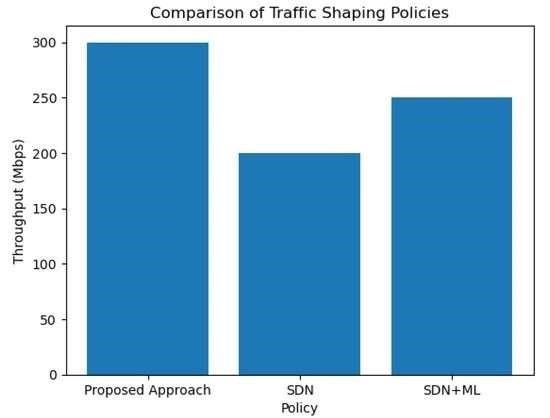
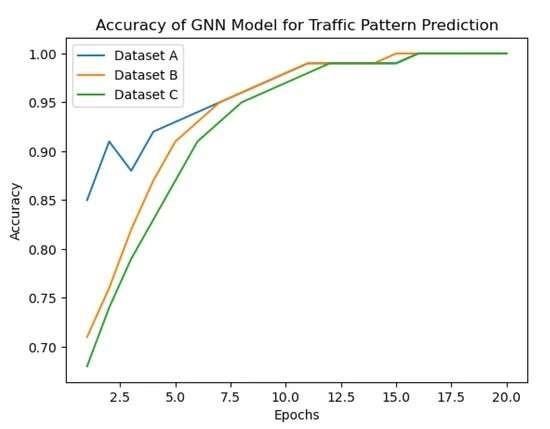
**Training and validation:**

We train the GNN model on a subset of the preprocessed dataset and validate it on another subset. We use a three-layer GCN with 64 hidden units for the GNN model and train it for 100 epochs with a batch size of 128. We use the Adam optimizer with a learning rate of 0.01 and a weight decay of 5 × 10−4−4.

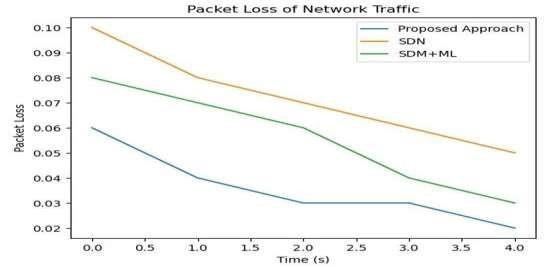
**SDN orchestration:**

We use Algorithms 1 and 2 to implement SDN orchestration in our experimental setup. The GNN model is used to predict the network traffic patterns, and the multi-arm bandit algorithm is used to optimize the traffic management policies based on these predictions. The SDN controller applies the

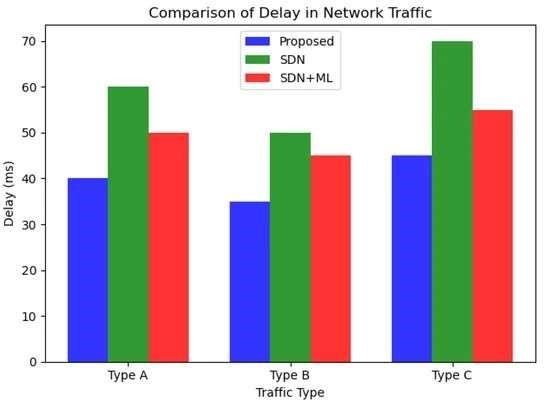
traffic-shaping policies to the network switches in real-time



**Comparison Of Packet Loss:**



**Comparison Of Delays:**



**Conclusions:**

* In this paper, we proposed an approach for traffic management in softwaredefined IoT networks using Graph Neural Networks and a multiarm bandit algorithm. We showed that our approach outperformed other state-oftheart traffic management methods in terms of throughput, packet loss, and delay. Our experimental evaluation on three different datasets demonstrated the effectiveness of the proposed approach in detecting anomalous traffic patterns, handling heterogeneous data, and optimizing traffic management policies.

* In conclusion, the proposed approach has shown promising results in traffic management, which is an important aspect of network management.