Applied Forecasting Methods (IT402)
Instructor Prof. Pritam Anand

Traffic Flow Prediction Using Time Series Analysis

Team 13

Ashish Kar	202418007
Deepanshi Acharya	202418015
Kaustav Mitra	202418023
Krisha Sompura	202418026
Pragnya Dandavate	202418065

Contents

1 Abstract

Urban centers around the globe are increasingly burdened by traffic congestion, a challenge exacerbated by rapid urbanization and the exponential growth of vehicular populations. As transportation infrastructures face mounting strain, the need for accurate and reliable traffic flow forecasting has become paramount. This project aims to address this critical issue by leveraging time series forecasting methodologies to model and predict urban traffic volumes using historical data.

By capturing temporal dynamics—such as daily and weekly seasonality—and uncovering latent trends, the project aspires to inform intelligent traffic management systems capable of proactive intervention. Through the integration of statistical models (e.g., ARIMA, SARIMA) and cutting-edge machine learning and deep learning architectures (e.g., LSTM, GRU), we evaluate the effectiveness of different forecasting techniques in enhancing predictive accuracy.

The outcomes of this project hold significant potential for improving traffic signal control, alleviating congestion, optimizing public transportation scheduling, and reducing environmental impacts through improved fuel efficiency and lower emissions. Ultimately, this work contributes to the development of smart, data-driven urban mobility solutions that align with the broader vision of sustainable and resilient cities.

2 Introduction

One of the most pressing issues in contemporary urban life is traffic congestion. Cities across the world are facing increasing strain on their transportation infrastructure due to rapid urban population growth. The fast pace of urbanization and the rising number of vehicles on the roads have made traffic flow prediction essential.

Accurate traffic prediction enables better infrastructure planning, efficient traffic management, optimized signal timings, and reduced environmental impact through improved fuel efficiency and lower emissions. By applying time series forecasting techniques on historical traffic data, cities can anticipate traffic patterns and make data-driven decisions.

The ability to accurately forecast traffic volumes is critical for intelligent traffic management systems. Predictive analytics, particularly time series forecasting, plays a vital role in identifying patterns in historical traffic data and using them to predict future traffic conditions. With the help of statistical models and machine learning techniques, cities can enhance public transportation planning, optimize infrastructure utilization, and proactively manage traffic flow.

3 Problem Statement

The primary objective of this project is to develop and evaluate predictive models that can accurately forecast traffic volume using historical data. It focuses on analyzing temporal patterns—such as daily and weekly seasonality—in traffic flow data. It aims to uncover trends that help in improving traffic signal control, mitigating congestion, and enabling more efficient route planning. To achieve this, we will explore a range of methods, from classical statistical techniques to more advanced deep learning models.

4 Dataset Overview

Source :Traffic Dataset

Description: The dataset contains **48,120** observations of the number of vehicles each hour in four different junctions. There are **four** columns in the dataset:-

- DateTime - Datetime in hourly frequency
- Junction - Junction Number
- Vehicles - Number of vehicles recorded that hour
- ID - Unique ID

There are no null values in the data.

5 Methodology

5.1 Libraries

To facilitate data preprocessing, visualization, modeling, and evaluation, this project leveraged a comprehensive suite of Python libraries, spanning statistical analysis, machine learning, and deep learning domains:

- **pandas**, **numpy**, **datetime** — Utilized for efficient data ingestion, manipulation, and temporal feature engineering, enabling robust handling of time-indexed datasets.
- **holidays** — Employed to incorporate national and regional holidays into the feature set, enriching the temporal context for improved predictive accuracy.
- **matplotlib.pyplot** — Used extensively for the visualization of time series trends, forecast outputs, residuals, and comparative model performance.
- **statsmodels** — Core to the implementation of classical statistical models such as ARIMA, SARIMA, and SARIMAX, along with diagnostic tools like the Augmented Dickey-Fuller (ADF) test, Autocorrelation Function (ACF), and Partial Autocorrelation Function (PACF).
- **scikit-learn (sklearn)** — Applied for data preprocessing, train-test splitting, and computing key evaluation metrics including RMSE, MAE, and MAPE.
- **torch (PyTorch)** — Leveraged for constructing and training deep learning models, particularly Long Short-Term Memory (LSTM) networks, to capture complex temporal dependencies.
- **itertools**, **tqdm** — Used to streamline hyperparameter grid search procedures and monitor training progress through dynamic progress bars.
- **warnings** — Incorporated to suppress extraneous warnings during model execution and ensure cleaner output logs.

5.2 Data Pre-processing

- We started by loading the dataset and viewing its structure, which includes the following columns: `DateTime`, `Junction`, `vehicles`, and `ID`.
- Filtered the dataset to include only records from **Junction-1**, and dropped the `ID` and `Junction` columns.
- The shape of the filtered dataset is: `(14593, 1)`.
- Checked for null values and found **no missing values**.

5.3 Feature Engineering

To enhance the predictive power of our models, we performed feature engineering by extracting meaningful time-based features from the `DateTime` column. This was essential for capturing temporal trends and variations in traffic volume over different times of the day, week, and year. The following features were created.

The following features were created:

- **Hour of the Day:** Extracted from the timestamp to capture intra-day traffic fluctuations.
- **Day of the Week:** The name of the weekday was derived and encoded as an ordered categorical variable to reflect the natural weekly cycle (Monday to Sunday).
- **Month and Year:** These features were included to help detect monthly or annual trends in traffic behavior.
- **In addition to feature extraction:** The dataset was resampled to daily frequency using the mean value, to smooth out high-frequency noise and better analyse daily traffic trends.

These engineered features, along with the original target variable (`Vehicles`), were combined into a new dataset to be used for model training.

5.4 Explanatory Data Analysis(EDA)

We conducted a comprehensive exploratory data analysis (EDA) as follows :

5.4.1 Traffic Volume Over Time

For better observation, the original hourly dataset was resampled (downsampling from hourly to daily data).

The plot reveals a gradual increase in traffic volume over time, indicating growing vehicle usage. It also displays regular fluctuations, suggesting seasonality and occasional anomalies, likely tied to holidays or special events.

From the visual inspection of the trend, it is evident that the data is **non-stationary**.

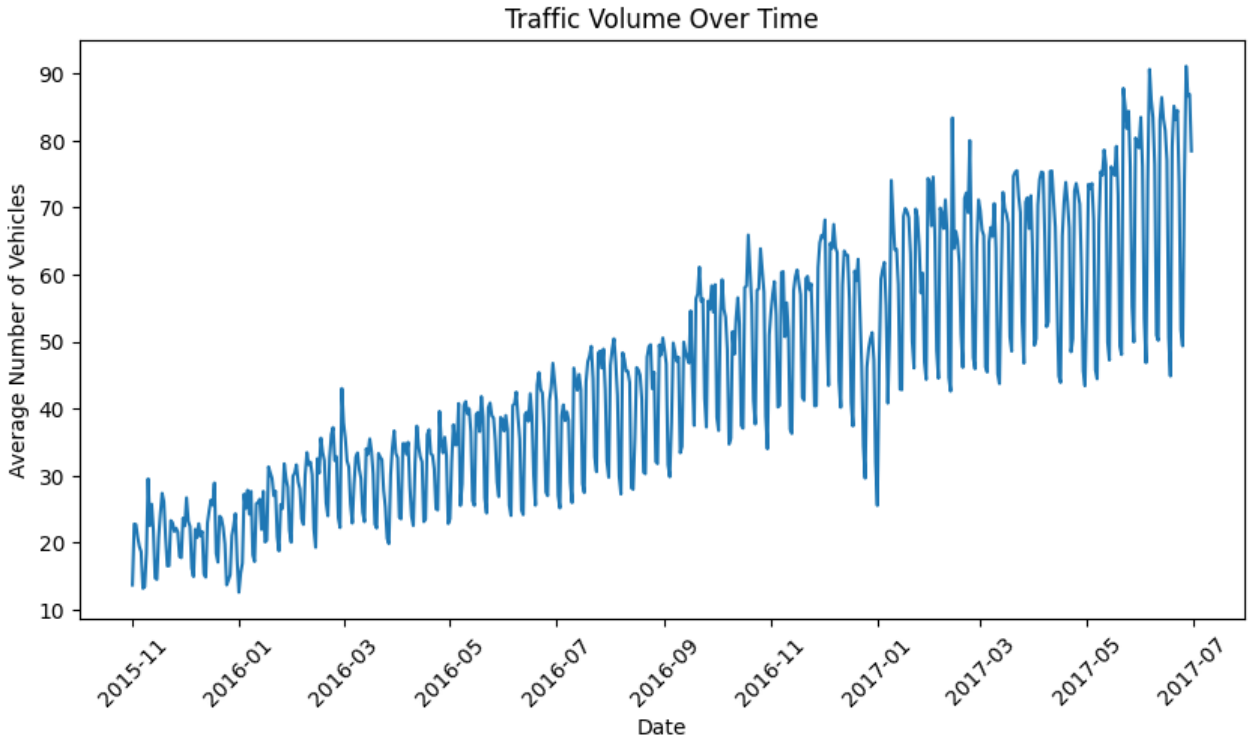


Figure 1: Traffic Volume Over Time

5.4.2 Peak Hour Analysis

The average number of vehicles for each hour of the day was analyzed. It was observed that traffic volume is highest during 10 AM to 1 PM and 6 PM to 9 PM, indicating morning and evening rush hours. Conversely, the lowest volumes occur between midnight and early morning (1 AM to 6 AM). This clear daily pattern reflects typical commuting behavior and confirms the presence of hourly seasonality.

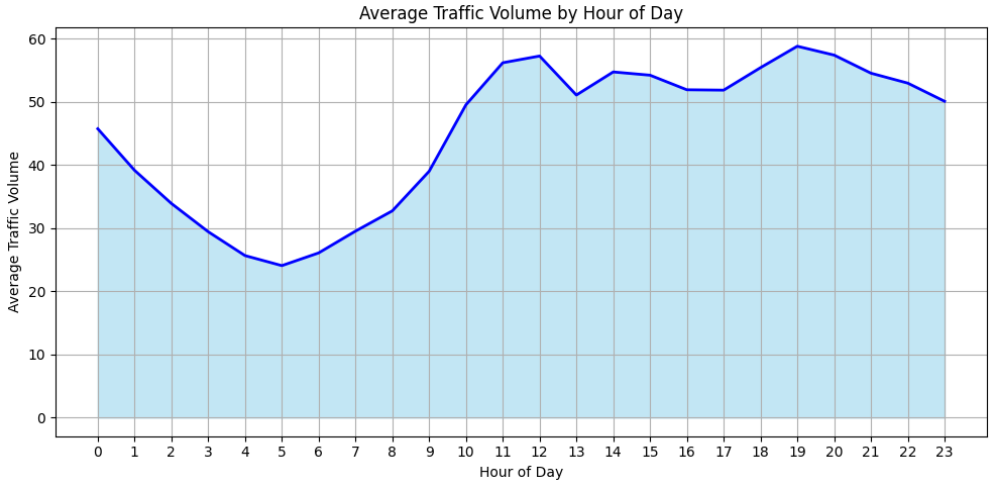


Figure 2: Hourly avg. traffic volume

5.4.3 Weekday Traffic Patterns

The line chart illustrates the average traffic volume for each day of the week. Traffic is highest on weekdays (Monday to Friday), with a noticeable drop on weekends (Saturday and Sunday). This pattern reflects regular workweek commuting behavior, where traffic is heavier during working days and lighter on weekends.

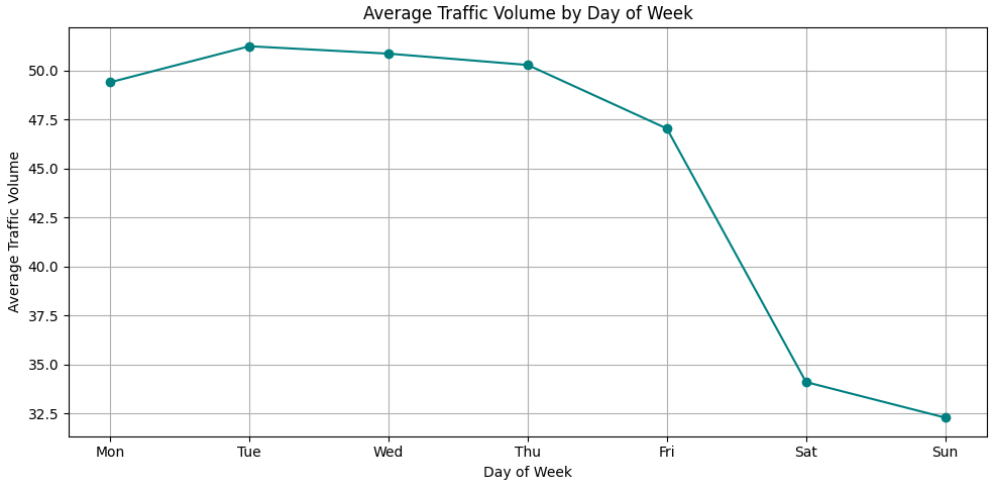


Figure 3: Weekly avg. traffic volume

5.5 Time Series Decomposition

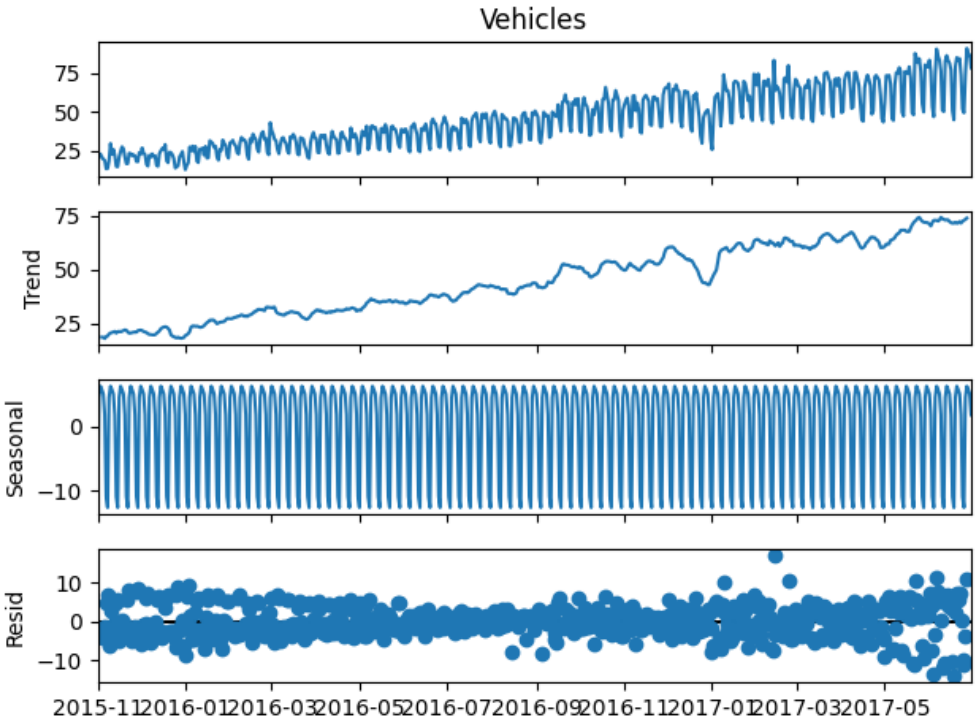


Figure 4: Distribution of Traffic Records by Year

- The traffic volume data shows a repeating pattern, indicating seasonality, likely weekly.
- **The trend component** suggests periods of increase and decrease in traffic.
- **The seasonal component** reveals consistent peaks and troughs, likely tied to workweek patterns.
- **Residuals**, representing irregularities, are mostly random with a few outliers, indicating that the model effectively captures trend and seasonality.

Overall, the analysis highlights a strong seasonal pattern, useful for both long-term planning and short-term traffic management.

Train-Test Split

The time series was split into training and testing subsets. The split point was chosen as **January 1, 2017**, based on temporal relevance and ensuring sufficient observations in both subsets. The training set (`df_train`) includes all data up to and including the split date, while the testing set (`df_test`) comprises data points after this date:

```
train = df_daily[:'2016-12-31']
test = df_daily['2017-01-01':]
```

This split ensures that the model is trained on historical data and tested on future observations, simulating a realistic forecasting scenario.

6 Checking for Stationarity

The Augmented Dickey-Fuller (ADF) test was conducted to check for stationarity in the data. The p-value (> 0.05) suggests that the time-series is not stationary and requires transformations. The table below shows the results obtained.

Metrics	**Value**
ADF Statistic	-0.3413815102045648
P-value	0.9194355530673952

Table 1: ADF Test Results

Differencing allows us to remove trends and seasonality and helps make a time series stationary. We applied first-order differencing and re-conducted the ADF test. The obtained p-value was less than 0.05 and hence stationary.

Metrics	**Value**
ADF Statistic	-5.449863589921822
P-value	0.00000265925043707811 9

Table 2: ADF Test Results

7 Auto-Correlation and Partial Auto-Correlation analysis

To identify the appropriate structure for the time series model, we analyzed the autocorrelation and partial auto-correlation functions of the stationary series.

The Autocorrelation Function (ACF) measures the linear relationship between lagged values of the time series, indicating how observations at previous time points are related to current values. In contrast, the Partial Auto-correlation Function (PACF) shows the correlation between a time series and its lag, controlling for intermediate lags.

Fig 5 shows the autocorrelation function (ACF) of the time series.

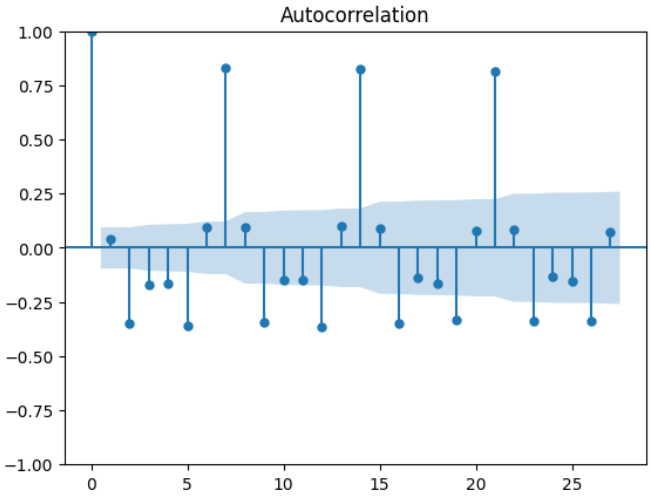


Figure 5: Autocorrelation plot of the time series

The presence of strong autocorrelation suggests that it is not a random walk.

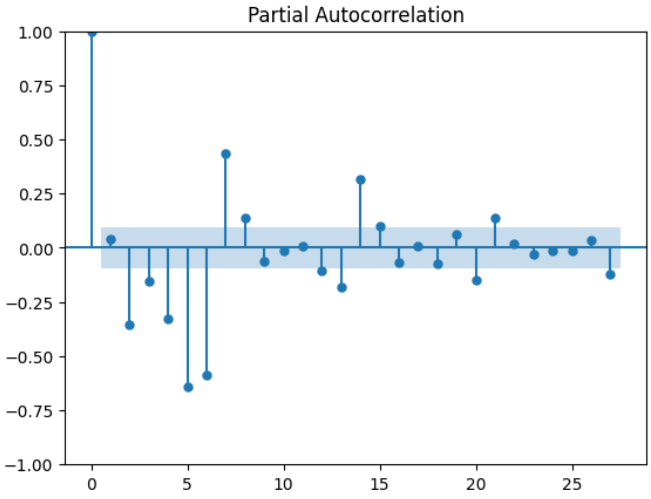


Figure 6: Partial Autocorrelation plot of the time series

Fig 6 shows the partial autocorrelation function (PACF) of the time series.

The autocorrelation and partial autocorrelation coefficients do not become non-significant after a few lags. Thus, it cannot be an AR or MA process.

8 Evaluation Metrics

We assessed the model performances using four metrics:

- **Root Mean Squared Error (RMSE):** Measures the standard deviation of prediction errors. Lower values indicate better model accuracy.
- **Mean Absolute Percentage Error (MAPE):** Represents average absolute percentage errors.
- **Prediction Interval Coverage Probability (PICP):** Represents the percentage of true values falling within the model's prediction intervals. Higher PICP indicates better reliability.
- **Mean Prediction Interval Width (MPIW):** Reflects the average width of the prediction intervals. A smaller MPIW is preferred if PICP is sufficiently high, as it indicates sharper predictions.

9 Fundamental Time Series Models

As observed from the ACF and PACF plots, the time series could not be an AR or MA process. Hence, we proceeded directly to ARMA, AR and followed subsequently with SARIMA.

Algorithm 1 Time Series Forecasting using ARIMA / ARMA / SARIMA

Input: Training series $train$, Test series $test$
Output: Forecasts, Confidence Intervals, Evaluation Metrics (RMSE, MAPE, PICP, MPIW)
Perform differencing if needed: $train\_diff = train.diff().dropna()$
Define model type: `ARIMA` / `ARMA` / `SARIMA`
Initialize parameter search space: $p, d, q \in$ defined ranges
Initialize:

$$\text{best\_aic} \leftarrow \infty, \quad \text{best\_order} \leftarrow \text{None}$$

for each (p, d, q) combination **do**
  Fit model on $train\_diff$ using `ARIMA/ARMA/SARIMA`
  Compute AIC
  if current AIC $<$ best_aic **then**
    Update best model and parameters
  end if
end for
Plot model diagnostics
Initialize:

$$\text{history} \leftarrow \text{list(train\_diff)}, \quad \text{last\_obs} \leftarrow \text{last value of train}$$

Initialize empty list: forecasts, confidence_intervals
for each timestep in test **do**
  Fit best model on history
  Forecast next differenced value $\Delta\hat{y}_t$
  Get confidence interval: $[\Delta\hat{y}_t^L, \Delta\hat{y}_t^U]$
  Convert forecast to original scale:

$$\hat{y}_t = last\_obs + \Delta\hat{y}_t$$

  Update confidence interval bounds:

$$y_t^L = last\_obs + \Delta\hat{y}_t^L, \quad y_t^U = last\_obs + \Delta\hat{y}_t^U$$

  Append forecast and interval
  Update $history$ and $last\_obs$
end for
Compute evaluation metrics:

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

$$\text{MAPE} = \frac{1}{n}\sum\left|\frac{y_i - \hat{y}_i}{y_i}\right| \times 100$$

$$\text{PICP} = \frac{1}{n}\sum I(y_i \in [y_i^L, y_i^U])$$

$$\text{MPIW} = \frac{1}{n}\sum(y_i^U - y_i^L)$$

Plot:

- True vs Forecast values
- Confidence Intervals
- Annotate plot with RMSE, MAPE, PICP, MPIW

9.1 Autoregressive Moving Average(ARMA) model

To identify the most suitable ARMA model for the stationary time series, we performed a grid search over various combinations of AR and MA terms. The Akaike Information Criterion (AIC) was used to evaluate model quality, where a lower AIC indicates a better fit while penalizing model complexity.

9.1.1 Model Selection

To select the optimal ARMA model, we evaluated combinations of AR (p) and MA (q) terms using the Akaike Information Criterion (AIC). The terms p and q were evaluated in a range of 0 to 5.

```
for p in p_values:
    for q in q_values:
        try:
            model = ARIMA(train_diff, order=(p, d, q))
            model_fit = model.fit()
            aic = model_fit.aic
            if aic < best_aic:
                best_aic = aic
                best_order = (p, d, q)
                best_model = model_fit
        except:
            continue
```

The ARMA(4,4) model yielded the lowest AIC value of **2339.816**, indicating it is the most suitable among all the tested configurations.

Residual Diagnostics

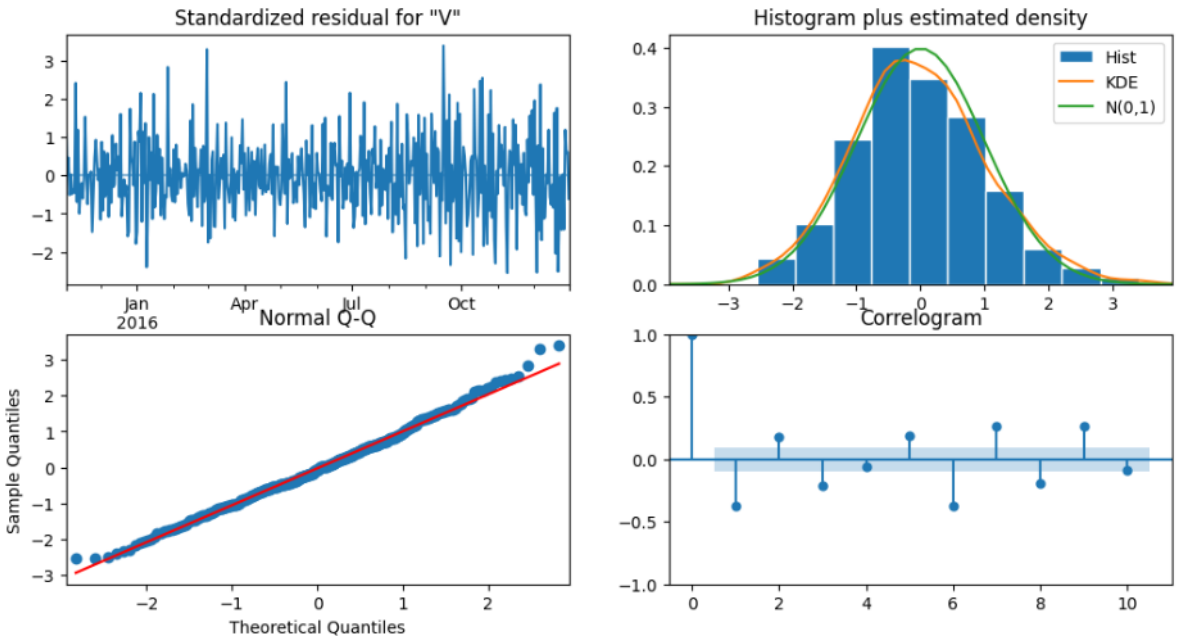


Figure 7: Residual diagnostics of the ARMA model

- **Standardized Residuals:** The residuals fluctuate randomly around zero with consistent variance over time. This suggests that the model has effectively removed temporal structure and that the residuals are stationary.
- **Histogram and KDE:** The histogram of residuals approximates a normal distribution. The kernel density estimate (KDE) and the standard normal curve show close alignment, with only slight tail differences.
- **Normal Q-Q Plot:** Points mostly lie along the diagonal reference line, indicating that the residuals are approximately normally distributed. Slight deviations in the tails are visible but not substantial.
- **Correlogram (ACF):** Most autocorrelation coefficients are within the 95% confidence bands. There is no significant autocorrelation left in the residuals, confirming that the ARMA model captured the time-dependent structure well.

9.1.2 Model training and forecasting

In the rolling forecast process, the model is updated at each step using the most recent observations. At every iteration, the model is retrained to generate a one-step-ahead forecast.

At each time step, the ARMA model was initialized with the order (4, 0, 4). The model was then fitted to the most recent data (history) to generate a forecast for the next step.

```
for _ in range(len(test)):
    model = ARIMA(history, order=(4, 0, 4))
    model_fit = model.fit()
```

Once the model was fitted, we obtained the forecasted difference (predicted mean) for the next step along with the associated confidence intervals.

```
    forecast_result = model_fit.get_forecast(steps=1)
    diff_forecast = forecast_result.predicted_mean[0]
    ci = forecast_result.conf_int(alpha=0.1)
    diff_lower = ci[0][0]
    diff_upper = ci[0][1]
```

The forecasted difference and its confidence intervals were inverted back to the original scale by adding them to the last observed value. The history was updated with the new forecasted difference, and the last observed value was updated for the next iteration.

```
    actual_forecast = last_obs + diff_forecast
    lower_bound = last_obs + diff_lower
    upper_bound = last_obs + diff_upper

    rolling_forecasts.append(actual_forecast)
    conf_ints.append((lower_bound, upper_bound))

    history.append(diff_forecast)
    last_obs = actual_forecast
```

9.1.3 Model Evaluation

The forecasted values and their associated confidence intervals were extracted from the rolling predictions. The lower and upper bounds of the intervals were separated to facilitate interval-based evaluation.

```
forecast_values = rolling_forecasts
lower_bounds = np.array([ci[0] for ci in conf_ints])
upper_bounds = np.array([ci[1] for ci in conf_ints])
```

The performance of the model was evaluated using four metrics: RMSE, MAPE, PICP, and MPIW. These metrics quantify both the accuracy of point forecasts and the reliability of prediction intervals. RMSE and MAPE assess prediction error magnitudes, while PICP and MPIW evaluate how well the prediction intervals capture actual values and how wide those intervals are, respectively.

```
within_interval = (test >= lower_bounds) & (test <= upper_bounds)
picp = np.mean(within_interval)
mpiw = np.mean(np.array(upper_bounds) - np.array(lower_bounds))
mape = np.mean(np.abs((test - forecast_values) / test)) * 100
rmse = np.sqrt(mean_squared_error(test, forecast_values))
```

These were the results obtained:

- **Root Mean Squared Error (RMSE)**: 18.76
- **Mean Absolute Percentage Error (MAPE)**: 24.51%
- **Prediction Interval Coverage Probability (PICP)**: 10.50%
- **Mean Prediction Interval Width (MPIW)**: 10.88

Figure 8 illustrates the comparison between the actual and predicted vehicle counts over the test period.

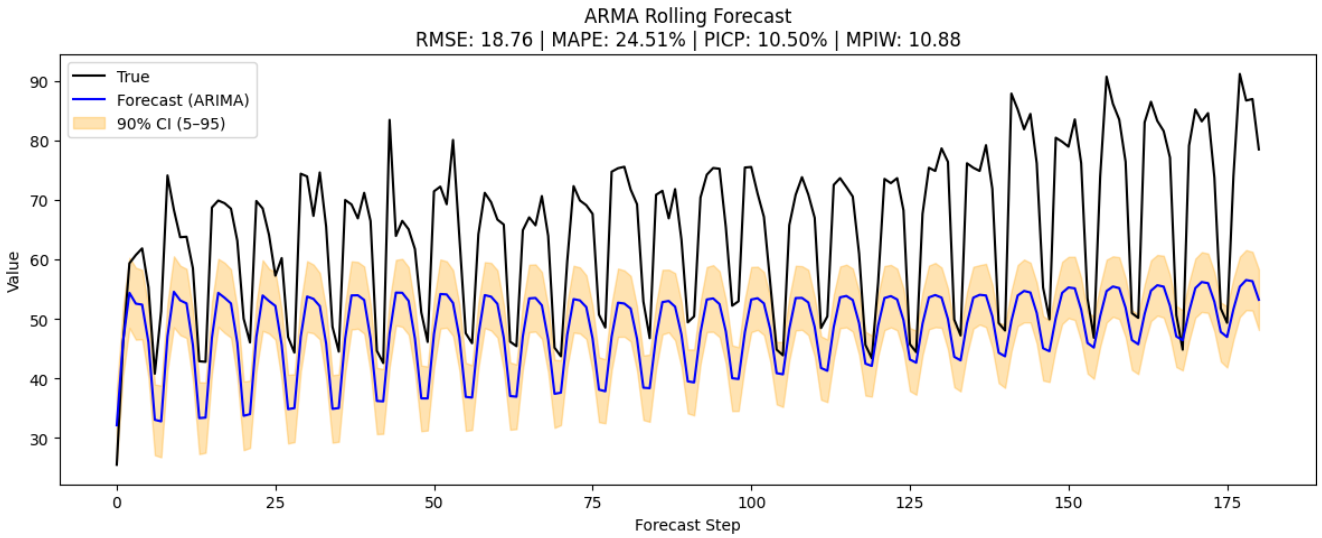


Figure 8: Actual vs Predicted Traffic using ARMA(4, 4) rolling forecast model

- The ARMA model exhibits a strong tendency to underpredict the actual values throughout the forecast horizon.

- The predicted values follow the seasonal pattern well, but with significantly dampened amplitude compared to the true series.
- The 90% confidence interval (shaded region) is consistently narrow (MPIW = 10.88), indicating high certainty in the forecast.
- The low PICP of 10.50% shows that the majority of true values fall outside the prediction interval, highlighting poor uncertainty calibration.
- The RMSE and MAPE indicate high error and deviation from true values, especially during peak periods.

9.2 AutoRegressive Integrated Moving Average(ARIMA) model

Since we saw the drawbacks of ARMA and found that ARMA is not good enough because it does not fully capture the short-term fluctuations or spikes in traffic. So we tried applying ARIMA.

9.2.1 Model Selection

To begin with, the ARIMA (AutoRegressive Integrated Moving Average) model is a widely used statistical approach for time series forecasting. It is denoted as ARIMA(p, d, q), where:

- p denotes the number of autoregressive (AR) terms,
- d represents the number of non-seasonal differences required to make the series stationary,
- q refers to the number of lagged forecast errors in the moving average (MA) component.

Based on preliminary analysis—such as stationarity testing, ACF and PACF plots, and comparison of AIC values from candidate ARMA models—we determined that the optimal parameters for our series are $(4, 1, 4)$.

Hence, an ARIMA$(4, 1, 4)$ model was fit to the training data. This configuration effectively captures both the autoregressive and moving average patterns in the differenced traffic volume series.

9.2.2 Model training and forecasting

The model was first fitted on the training dataset.

```
model = ARIMA(train, order=(4, 1, 4))
model_fit = model.fit()
```

A rolling forecast approach was adopted for out-of-sample predictions. In this method, at each step, the ARIMA model was refitted using all available historical data up to that point, and a one-step-ahead forecast was produced.

1. Initialize the history with the training dataset.
2. For each point in the test set:
    - Fit a new ARIMA$(4, 1, 4)$ model on the current history.
    - Generate a one-step-ahead forecast along with 90% confidence intervals.
    - Store the predicted value and confidence bounds.
    - Append the predicted value to the history.

This dynamic updating ensures that the model continuously adapts to the most recent information, which may improve forecasting performance.

The forecasted values and their respective 90% confidence intervals were recorded and can be visualized for further analysis. These forecasts are expected to capture both the trend and stochastic behavior of the time series within a reasonably bounded uncertainty range.

9.2.3 Model Evaluation

The lower and upper bounds of the intervals were separated to facilitate interval-based evaluation. Then the performance was evaluated using the four chosen metrics in a similar manner as the previous model.

The performance of the ARIMA(4,1,4) model was assessed using standard error metrics on the test dataset:

- **Root Mean Squared Error (RMSE)**: 22.49
- **Mean Absolute Percentage Error (MAPE)**: 30.39%
- **Prediction Interval Coverage Probability (PICP)**: : 1.66
- **Mean Prediction Interval Width (MPIW)** : 10.87

These metrics indicate a moderate forecasting performance. While the model effectively captures the overall level and long-term trend of the traffic volume data, the relatively high MAPE suggests that the predictions deviate by an average of 30.06% from the actual values.

The ARIMA model's performance in forecasting vehicle traffic is visually represented in Figure 9.

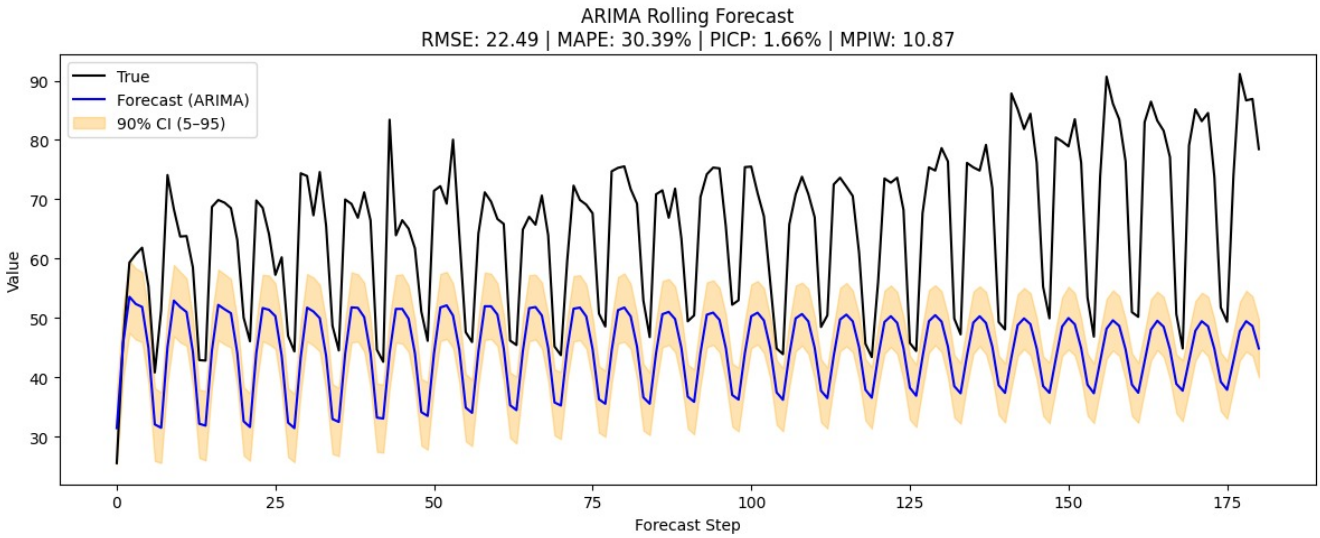


Figure 9: ARIMA : Actual VS Predicted traffic using ARIMA (4,1,4) rolling forecast model

- The ARIMA model successfully captures the overall trend in traffic volume; however, its predictions are relatively smooth and fail to account for clear seasonal patterns and short-term fluctuations.
- The model consistently underestimates peak traffic periods, indicating a limited ability to respond to sudden changes or recurring high-traffic events. This suggests that while ARIMA is effective for modeling general patterns, it may not be well-suited for capturing the dynamic nature of real-world traffic data.

9.3 Seasonal Autoregressive Integrated Moving Average(SARIMA) model

To address the limitations of the ARIMA model in capturing seasonality, we implemented a Seasonal ARIMA (SARIMA) model that extends ARIMA by explicitly modeling seasonal effects in time series data. It's particularly useful when data exhibits repetitive patterns over fixed periods—such as weekly, monthly, or yearly trends. In this case, a weekly seasonality of 7 days is assumed.

9.3.1 Model Selection

To identify the best-performing Seasonal ARIMA (SARIMA) model, a grid search was conducted over a range of candidate values for the non-seasonal and seasonal parameters. The non-seasonal parameters `(p, d, q)` were selected from the range 0 to 2, while the seasonal parameters `(P, D, Q)` were chosen from the range 0 to 1 with a seasonal period $s = 7$. For each combination, a SARIMA model was fitted to the training data, and the Akaike Information Criterion (AIC) was used as the evaluation metric. The model with the lowest AIC was selected as the optimal configuration.

The optimal non-seasonal component identified was $(1, 1, 2)$, which implies:

- $p = 1$: One autoregressive (AR) term,
- $d = 1$: First-order differencing to remove trend and ensure stationarity,
- $q = 2$: Two moving average (MA) terms to capture error dependencies.

For the seasonal component, the configuration $(0, 1, 1, 7)$ was found to be most suitable:

- $P = 0$: No seasonal AR terms,
- $D = 1$: First-order seasonal differencing to capture weekly cycles,
- $Q = 1$: One seasonal MA term,
- $s = 7$: Seasonal cycle length set to 7 (days), reflecting weekly periodicity.

This configuration, **SARIMA(1,1,2) × (0,1,1,7)** was selected based on the lowest Akaike Information Criterion (AIC), with an AIC score of **2013.03**. The chosen model effectively incorporates both trend and weekly seasonality components in the traffic volume data.

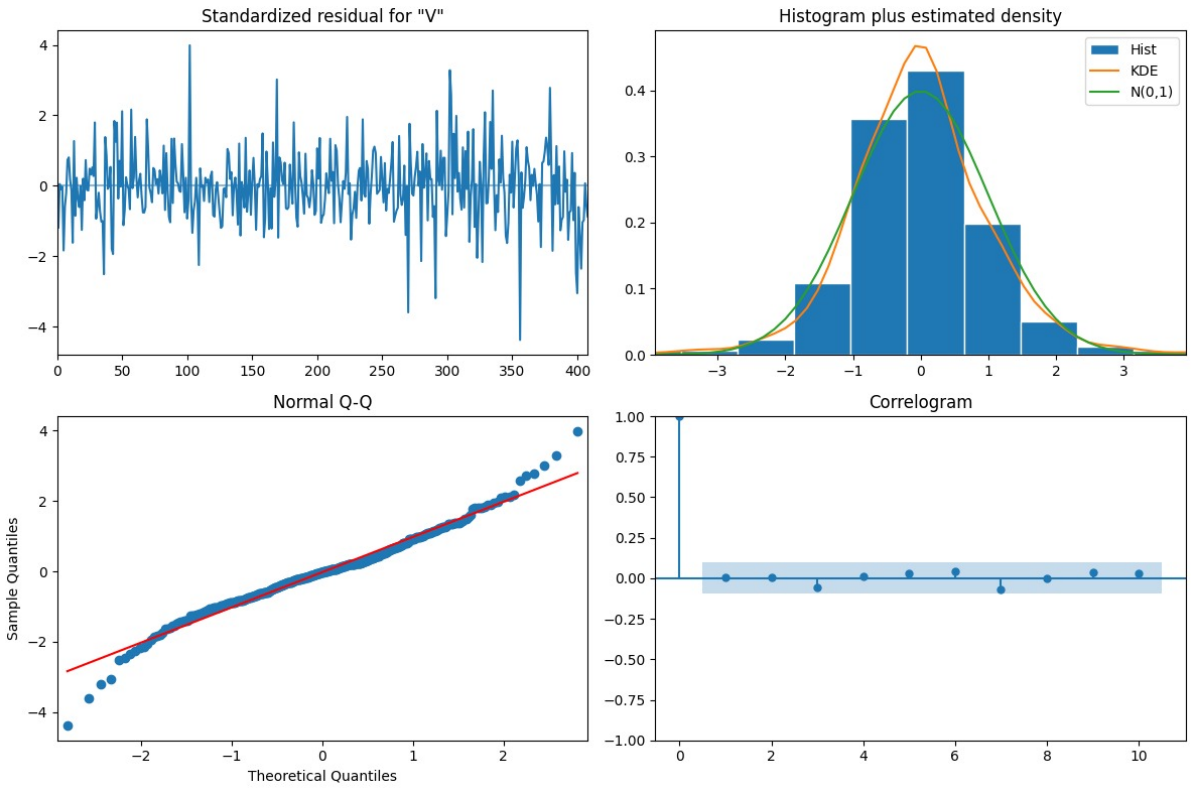


Figure 10: Residual diagnostics of the SARIMA model

- **Standardized Residuals:** The residuals fluctuate randomly around zero with consistent variance over time. This suggests that the model has effectively removed temporal structure and that the residuals are stationary.
- **Histogram and KDE:** The histogram of residuals approximates a normal distribution. The kernel density estimate (KDE) and the standard normal curve show close alignment, with only slight tail differences.
- **Normal Q-Q Plot:** Points mostly lie along the diagonal reference line, indicating that the residuals are approximately normally distributed. Slight deviations in the tails are visible but not substantial.
- **Correlogram (ACF):** Most autocorrelation coefficients are within the 95% confidence bands. There is no significant autocorrelation left in the residuals, confirming that the SARIMA model captured the time-dependent structure well.

9.3.2 Model training and forecasting

At each step, the model is re-fitted on an expanding window of past data (`history`), which helps capture both evolving patterns and recent dynamics in the time series. The model configuration includes both non-seasonal and seasonal components to better reflect the recurring weekly cycles identified in the data.

Specifically, the seasonal order `(P,D,Q,s)` is retained throughout the loop to preserve the weekly periodicity, while the non-seasonal order `(p,d,q)` enables the model to capture local trends and short-term dependencies.

```
for _ in range(len(test)):
    model = SARIMAX(history,
                    order=best_order,
                    seasonal_order=seasonal_order_full,
```

```
                    enforce_stationarity=False,
                    enforce_invertibility=False)

    model_fit = model.fit(disp=False)
```

In each step of the rolling forecast procedure, the model generates a one-step-ahead prediction along-with a 90% confidence interval. The forecasted value is appended to the historical data, enabling the model to update dynamically and account for evolving trends and seasonal behavior.

9.3.3 Model Evaluation

The lower and upper bounds of the intervals were separated to facilitate interval-based evaluation like the previous models. The performance of the model was assessed using our chosen metrics: RMSE, MAPE, PICP and MPIW.

- **Root Mean Squared Error (RMSE)**: 7.93
- **Mean Absolute Percentage Error (MAPE)**: 9.67%
- **Prediction Interval Coverage Probability (PICP)**: 36.46%
- **Mean Prediction Interval Width (MPIW)**: 8.28

This figure illustrates the comparison between the actual and predicted vehicle count over the test set.

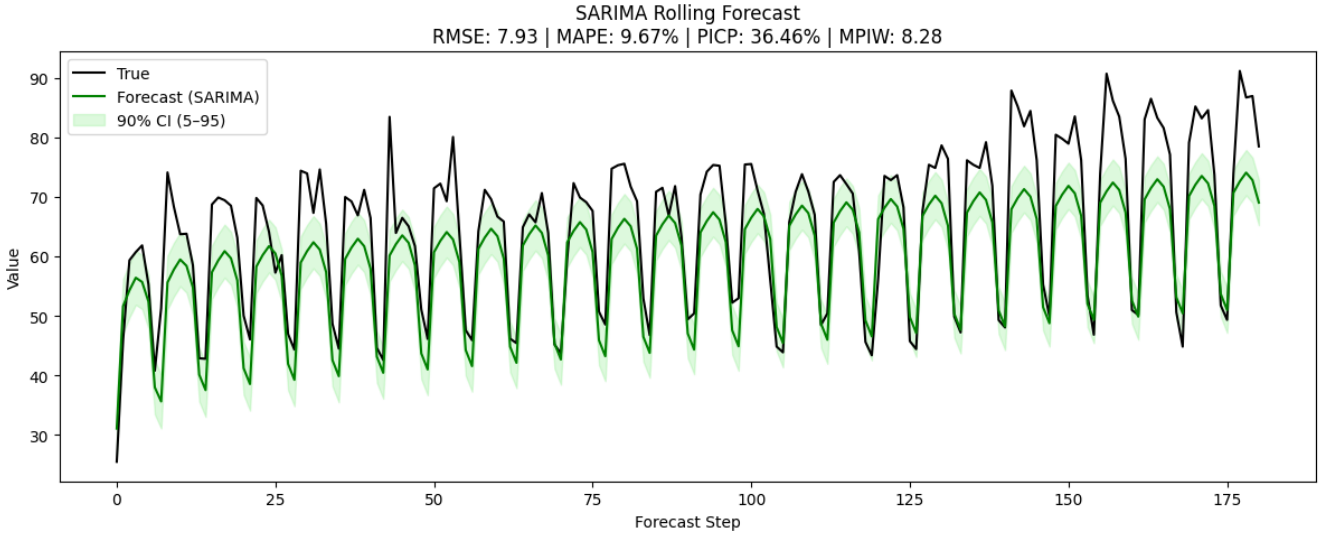


Figure 11: Actual vs Predicted Traffic using SARIMA rolling forecast model

- The SARIMA model captures the seasonality of the time series effectively, with clear periodic patterns reflected in the predicted trend.
- Despite reasonable accuracy in point forecasts (RMSE: 7.93, MAPE: 9.67%), the model exhibits a low prediction interval coverage (PICP: 36.46%), indicating underestimation of uncertainty.
- The 90% confidence intervals remain narrow (MPIW: 8.28), reflecting high confidence but at the cost of missing several true values outside the prediction band.
- The forecasted series maintains alignment with the true signal's structure, but deviations become noticeable during high peaks, suggesting room for improvement in capturing volatility.

10 Deep-Learning models

After performing forecasts using the fundamental models, we moved on to more advanced deep learning models to capture complex nonlinear temporal dependencies and improve predictive performance. These architectures allowed for more flexible uncertainty estimation and better adaptation to sequential patterns in the data.

We adopted a quantile regression approach to model the predictive uncertainty inherent in time series data. Unlike traditional point forecasting methods, which only provide a single estimate, quantile-based models are capable of generating prediction intervals by estimating multiple conditional quantiles . This is particularly valuable for real-world applications such as traffic forecasting, where understanding the range of possible outcomes is critical for risk-sensitive decision-making. By learning the distributional characteristics of future values, quantile regression enhances the model's robustness and provides a more informative and interpretable forecast.

10.1 Data Preparation

The time index was converted to datetime format, and the target feature was normalized to the $[0, 1]$ range using `MinMaxScaler`:

```
df1.index = pd.to_datetime(df1.index)
data = df1['Vehicles'].values.reshape(-1, 1)

scaler = MinMaxScaler()
data = scaler.fit_transform(data)
```

Sequences of length 24 were generated to capture temporal dependencies:

```
def create_sequences(data, seq_len):
    X, y = [], []
    for i in range(len(data) - seq_len):
        X.append(data[i:i + seq_len])
        y.append(data[i + seq_len])
    return np.array(X), np.array(y)

SEQ_LEN = 24
X, y = create_sequences(data, SEQ_LEN)
```

The resulting tensors were split into training and validation sets and wrapped into PyTorch `DataLoader`s for efficient batch processing:

```
X_tensor = torch.tensor(X, dtype=torch.float32)
y_tensor = torch.tensor(y, dtype=torch.float32)

X_train, X_val, y_train, y_val = train_test_split(
    X_tensor, y_tensor, test_size=0.2, random_state=42)

train_dataloader = DataLoader(TensorDataset(X_train, y_train), batch_size=64, shuffle=True)
val_dataloader = DataLoader(TensorDataset(X_val, y_val), batch_size=64, shuffle=False)
```

Loss function

To train our quantile-based deep learning models, we utilized the *quantile loss function*, also known as the *pinball loss*. This loss is defined as:

$$\mathcal{L}_\tau(y, \hat{y}) = \begin{cases} \tau(y - \hat{y}) & \text{if } y \geq \hat{y}, \\ (1 - \tau)(\hat{y} - y) & \text{otherwise,} \end{cases} \tag{1}$$

where y is the true value, $\hat{y}$ is the predicted value, and $\tau \in (0, 1)$ is the desired quantile level (e.g., 0.05, 0.5, 0.95).

This asymmetric loss penalizes underestimation and overestimation differently depending on the quantile, making it particularly suitable for modeling the conditional distribution of the target variable. It enables the model to predict not just a central estimate but also lower and upper bounds, thus capturing predictive uncertainty. This is crucial in time series forecasting tasks involving high variability or risk-sensitive applications.

The quantile loss used in our deep learning models is implemented using a custom PyTorch module, shown below:

```
class QuantileLoss(nn.Module):
    def __init__(self, quantiles):
        super().__init__()
        self.quantiles = quantiles

    def forward(self, preds, target):
        losses = []
        for i, q in enumerate(self.quantiles):
            errors = target - preds[:, i:i+1]
            loss = torch.max((q - 1) * errors, q * errors)
            losses.append(loss)
        return torch.mean(torch.sum(torch.cat(losses, dim=1), dim=1))
```

This class computes the pinball loss for multiple quantiles in a single forward pass. For each quantile q, the asymmetric loss penalizes underpredictions and overpredictions differently. The final loss is the average total quantile loss across all specified quantiles and all samples in the batch. This allows the model to learn the conditional distribution of the target and produce prediction intervals.

Pre-defined function

The function `train_and_evaluate_quantile_model` trains and evaluates a quantile regression model that predicts multiple quantiles (Q05, Q50, and Q90). It uses a quantile loss function and computes key evaluation metrics, including RMSE, MAPE, PICP, and MPIW, with a rolling forecast evaluation.

During training, the model is optimized using the Adam optimizer with a quantile loss function. The loss is computed for each batch and the optimizer updates the model parameters.

```
criterion = QuantileLoss(quantiles)
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
```

After training, the model is evaluated on the validation set by computing the validation loss.

```
model.eval()
with torch.no_grad():
    for xb, yb in val_dataloader:
        preds = model(xb)
        loss = criterion(preds, yb)
```

Once the model is trained, a rolling forecast window is used to predict the quantiles (Q05, Q50, Q90). The true and predicted values are then compared, and the results are scaled back to the original domain.

```
for i in range(rolling_steps):
    input_x = X_roll[i:i+1]
    pred = model(input_x).numpy()
    q05 = scaler.inverse_transform(pred[:, 0:1])[0, 0]
    q50 = scaler.inverse_transform(pred[:, 1:2])[0, 0]
    q90 = scaler.inverse_transform(pred[:, 2:3])[0, 0]
```

The function also evaluates and returns our four chosen metrics along-with the prediction and train-val plots. This function has been called later each time during our model run to return the required outputs.

Algorithm 2 Training and Evaluation of Quantile-based RNN, GRU, and LSTM Models

Input: Training data $X_{\text{train}}, y_{\text{train}}$, Validation data $X_{\text{val}}, y_{\text{val}}$
Output: Predicted quantiles and evaluation metrics
Initialize model parameters θ
Initialize optimizer (e.g., Adam)
Initialize loss function (Quantile Loss function for quantiles q_1, q_2, q_3)
for epoch = 1 to $E_{\max}$ **do**
  Shuffle training data
  for batch = 1 to number of batches in training set **do**
    Get batch data $X_{\text{batch}}, y_{\text{batch}}$
    Pass X_{batch} through the model to get predictions $\hat{y}_{\text{batch}}$
    Compute Quantile Loss:

$$\mathcal{L}(\hat{y}_{\text{batch}}, y_{\text{batch}}) = \sum_{q \in \{q_1, q_2, q_3\}} \max\left(q \cdot (y_{\text{batch}} - \hat{y}_{\text{batch}}), (q-1) \cdot (\hat{y}_{\text{batch}} - y_{\text{batch}})\right)$$

    Compute gradients and update model parameters θ
  end for
end for
Evaluate the model on validation set
Compute Quantile Predictions: $\hat{y}_{\text{val}} = f(X_{\text{val}}; \theta)$
Compute inverse transformation of predictions using scaler (e.g., MinMaxScaler)
Metrics:
Compute Root Mean Squared Error (RMSE) for the predicted median quantile

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

Compute Prediction Interval Coverage Probability (PICP)

$$\text{PICP} = \frac{1}{n}\sum_{i=1}^{n} I(y_i \in [\hat{y}_{q_1}, \hat{y}_{q_3}])$$

Compute Mean Prediction Interval Width (MPIW)

$$\text{MPIW} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_{q_3} - \hat{y}_{q_1})$$

Compute Mean Absolute Percentage Error (MAPE)

$$\text{MAPE} = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \times 100$$

Output: RMSE, PICP, MPIW, MAPE

10.2 Recurrent Neural Network(RNN) model

Our first deep learning model was RNN, which demonstrated the ability to track overall trends and turning points in the data. However, due to its limited memory and susceptibility to vanishing gradients, the RNN occasionally underperformed during highly volatile phases.

10.2.1 Model Architecure

The RNNQuantile model uses a basic Recurrent Neural Network (RNN) for time series quantile regression. The model processes sequential input data using an RNN layer, followed by a fully connected layer that outputs the predicted quantiles. The model is flexible with respect to the number of RNN layers and dropout rate.

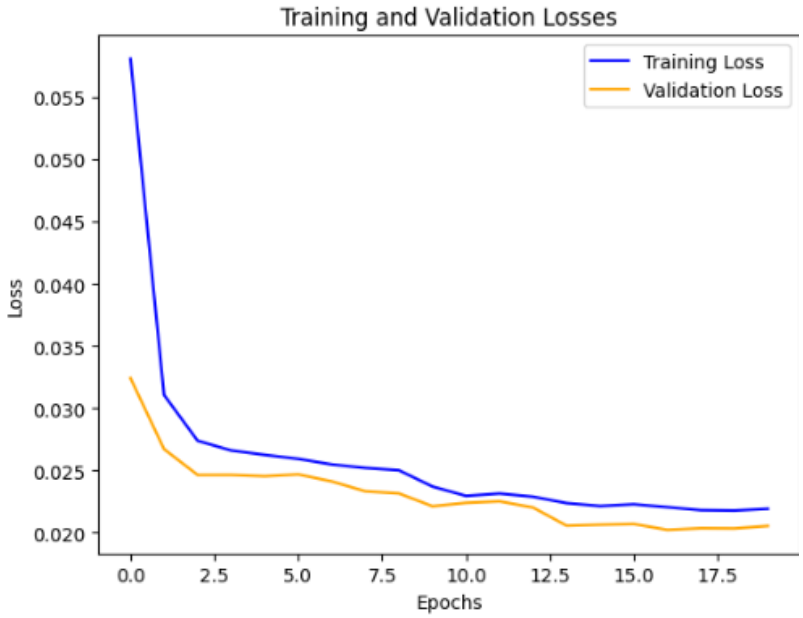
```
    def __init__(self, input_size, hidden_size, quantiles=[0.05, 0.5, 0.95], num_layers=2,
                 dropout=0.2):
        super().__init__()
        self.rnn = nn.RNN(input_size, hidden_size, num_layers=num_layers,dropout=dropout if
                    num_layers > 1 else 0.0,batch_first=True)
        self.fc = nn.Linear(hidden_size,len(quantiles))
```

10.2.2 Training and Evaluation

An RNN-based quantile regression model was trained to predict multiple quantiles (0.05, 0.5, 0.9) for time series data. The model is trained using a custom training function that optimizes the parameters and evaluates performance on the validation set.

```
quantiles = [0.05, 0.5, 0.9]
rnn = RNNQuantile(input_size=1, hidden_size=64, quantiles=quantiles)
metrics = train_and_evaluate_quantile_model(
    model=rnn,
    dataloader=train_dataloader,
    X_tensor=X_tensor,
    y_tensor=y_tensor,
    val_dataloader=val_dataloader,
    scaler=scaler,
    quantiles=quantiles,
    epochs=20,
    lr=0.001
)
```

The training and validation losses at each epoch was plotted and the plot below displays it.



- Both training and validation losses decrease consistently across epochs, indicating effective learning without early stagnation.
- The validation loss closely follows the training loss throughout the training process. There is no divergence between the two, suggesting that the model is not overfitting.
- The model shows a significant reduction in both training and validation losses, and they plateau at low values. This implies the model is adequately complex and has learned the underlying patterns.
- The small gap between training and validation losses towards the final epochs indicates good generalization to unseen data.
- No abrupt spikes or irregularities are present in the curves, pointing to a stable training process without significant learning rate or optimization issues.

The model's predictive performance was measured using four key metrics:

- **Root Mean Squared Error (RMSE) (Median):** 3.92
- **Mean Absolute Percentage Error (MAPE) (Median):** 11.38%
- **Prediction Interval Coverage Probability (PICP) (Coverage):** 84.00%
- **Mean Prediction Interval Width (MPIW) (Interval Width):** 11.03

The plot below illustrates the comparison between actual and predicted vehicle count.

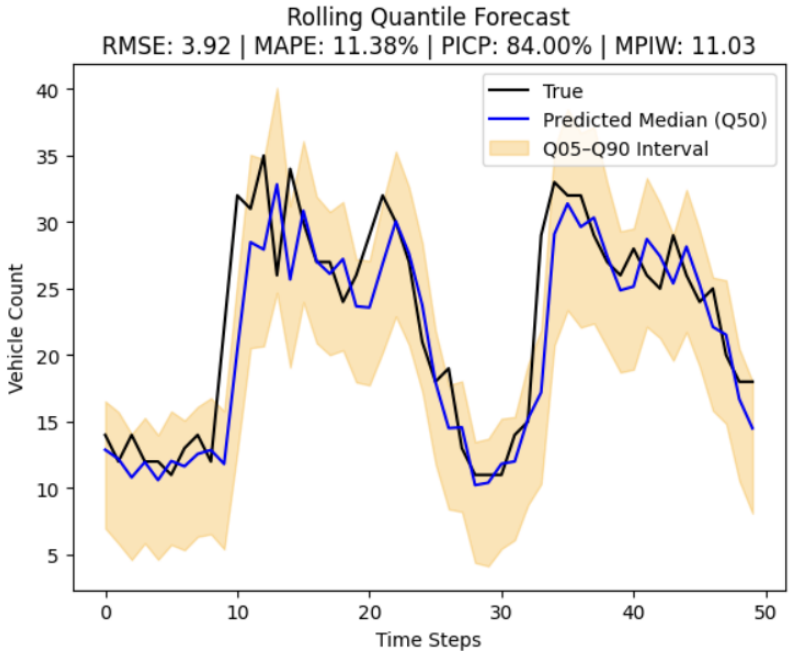


Figure 12: Actual vs Predicted Traffic using RNN rolling quantile forecast model

- The predicted median (Q50) captures the overall trend of true vehicle counts, showing good alignment across fluctuating patterns.
- The 90% prediction interval (Q05–Q90 shaded region) adapts well to uncertainty, especially around sharp peaks and troughs.
- A relatively low RMSE of 3.92 and MAPE of 11.38% indicate strong point forecasting accuracy.
- The coverage probability (PICP) of 84.00% shows the interval captures most true values, balancing reliability and precision.
- The mean prediction interval width (MPIW) of 11.03 reflects a moderate level of forecast confidence.

10.3 Long Short-Term Memory(LSTM) model

To improve the model's ability to capture long-term temporal dependencies in the traffic data, a Long Short-Term Memory (LSTM) network was implemented. LSTM networks are well-suited for time series forecasting as they can learn patterns over longer sequences compared to simple RNNs.

10.3.1 Model Architecture

The LSTM Quantile model was designed for quantile regression using Long Short-Term Memory (LSTM) layers. The model processes sequential input data, capturing temporal dependencies through stacked LSTM layers followed by dropout layers to prevent overfitting. The final output is generated through a fully connected (linear) layer, predicting the specified quantiles (e.g., 0.05, 0.5, 0.95).

```
def __init__(self, input_size, hidden_size, quantiles=[0.05, 0.5, 0.95]):
    super().__init__()
    self.lstm1 = nn.LSTM(input_size, hidden_size, batch_first=True)
```

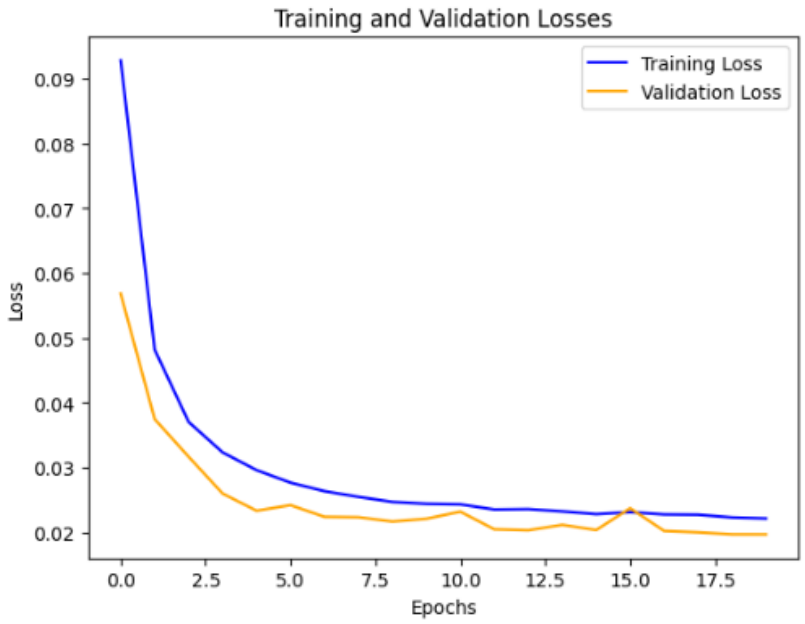
```
        self.dropout1 = nn.Dropout(0.2)
        self.lstm2 = nn.LSTM(hidden_size, hidden_size, batch_first=True)
        self.dropout2 = nn.Dropout(0.2)
        self.fc = nn.Linear(hidden_size, len(quantiles))
```

10.3.2 Training and evaluation

An LSTM-based quantile regression model was trained to predict multiple quantiles for time series data. The model consists of LSTM layers followed by dropout and a fully connected output layer. The training was carried out using a custom function that handles the optimization and validation process.

```
lstm = LSTMQuantile(input_size=1, hidden_size=64)
metrics = train_and_evaluate_quantile_model(
    model=lstm,
    dataloader=train_dataloader,
    X_tensor=X_tensor,
    y_tensor=y_tensor,
    val_dataloader=val_dataloader,
    scaler=scaler,
    quantiles=quantiles,
    epochs=20,
    lr=0.001
)
```

The training and validation losses at each epoch was plotted and the plot below displays it.



- Both training and validation losses show a clear downward trend, indicating that the model is learning meaningful patterns from the data.

- The narrow gap between training and validation losses throughout training suggests the model achieves a balanced fit — neither overfitting nor underfitting.
- Loss curves begin to plateau around epoch 10–12, indicating that the model is approaching its optimal learning capacity for this dataset without signs of overfitting.
- While the overall trend is downward, the validation loss exhibits minor spikes. This could indicate momentary instability in generalization or slight overfitting to mini-batches.

The model's predictive performance was measured using four key metrics:

- **Root Mean Squared Error (RMSE) (Median):** 3.58
- **Mean Absolute Percentage Error (MAPE) (Median):** 11.49%
- **Prediction Interval Coverage Probability (PICP) (Coverage):** 88.00%
- **Mean Prediction Interval Width (MPIW) (Interval Width):** 10.52

Figure 13 illustrates the comparison between the actual and predicted vehicle counts over the test period.

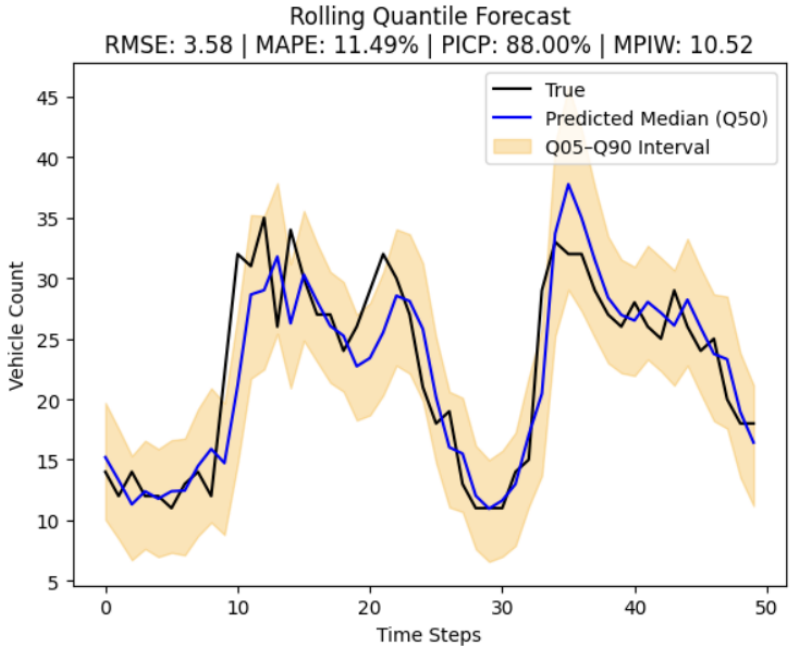


Figure 13: Actual vs Predicted Traffic using LSTM rolling quantile forecast model

- The LSTM-based model outperforms the RNN model with a lower RMSE of 3.58 compared to 3.92, reflecting improved prediction accuracy.
- MAPE is slightly higher in LSTM (11.49%) than RNN (11.38%), indicating a marginal increase in relative error.
- PICP improves in LSTM (88.00%) compared to RNN (84.00%), suggesting better coverage of the true values within the predicted intervals.
- MPIW is lower in LSTM (10.52), indicating that it achieves this better coverage with tighter (narrower) prediction intervals.

10.4 Gated Recurrent Unit (GRU) model

We moved on to GRU as our final model which has similar gating mechanisms to retain important information over long time steps, but with a simpler structure as well as fewer parameters as compared to LSTMs.

10.4.1 Model Architecture

The model used is a two-layer Gated Recurrent Unit (GRU) neural network tailored for quantile regression. It processes sequential time series input and outputs predictions for specified quantiles (e.g., 0.05, 0.5, 0.95). The architecture includes stacked GRU layers, dropout regularization to reduce overfitting, and a fully connected output layer.

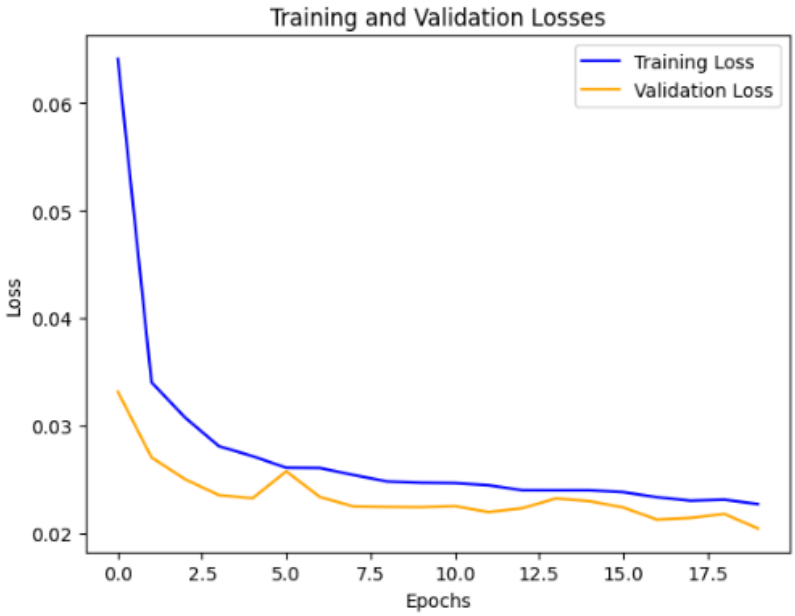
```
    def __init__(self, input_size, hidden_size, quantiles=[0.05, 0.5, 0.95]):
        super().__init__()
        self.gru1 = nn.GRU(input_size, hidden_size, batch_first=True)
        self.dropout1 = nn.Dropout(0.2)
        self.gru2 = nn.GRU(hidden_size, hidden_size, batch_first=True)
        self.dropout2 = nn.Dropout(0.2)
        self.fc = nn.Linear(hidden_size, len(quantiles))
```

10.4.2 Training and evaluation

The GRU-based quantile regression model was trained using a custom training function. The model takes time series input with a single feature and outputs predictions for multiple quantiles. The training process involves feeding data through a GRU with 64 hidden units and optimizing it using quantile-specific loss functions. The code snippet below shows the setup and execution of the training:

```
gru = GRUQuantile(input_size=1, hidden_size=64)
metrics = train_and_evaluate_quantile_model(
    model=gru,
    dataloader=train_dataloader,
    X_tensor=X_tensor,
    y_tensor=y_tensor,
    val_dataloader=val_dataloader,
    scaler=scaler,
    quantiles=quantiles,
    epochs=20,
    lr=0.001)
```

The training and validation losses at each epoch was plotted and the plot below displays it.



- The validation loss shows noticeable spikes while the training loss continues to decrease smoothly, which may indicate the model is starting to fit too closely to the training data during those periods.
- Despite the spikes, the validation loss remains close to the training loss and eventually stabilizes. This suggests that while some overfitting occurs intermittently, it is not severe or persistent.
- Both training and validation losses start high and decrease significantly, showing that the model is learning relevant patterns from the data.

The model's predictive performance was measured using four key metrics:

- **Root Mean Squared Error (RMSE) (Median):** 3.57
- **Mean Absolute Percentage Error (MAPE) (Median):** 11.82%
- **Prediction Interval Coverage Probability (PICP) (Coverage):** 90.00%
- **Mean Prediction Interval Width (MPIW) (Interval Width):** 10.24

Figure 14 illustrates the comparison between the actual and predicted vehicle counts over the test period.

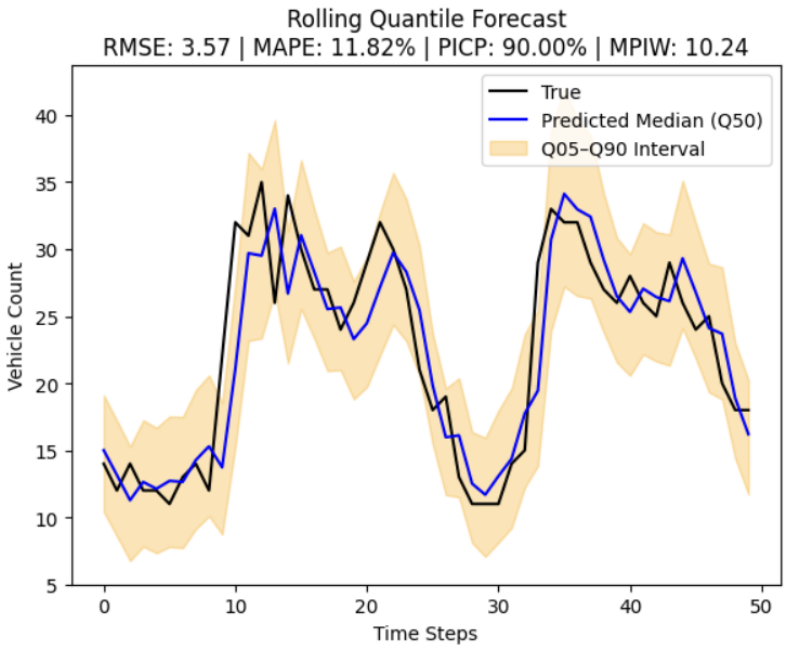


Figure 14: Actual vs Predicted Traffic using GRU rolling quantile forecast model

- The model shows the narrowest prediction interval (MPIW of 10.24) among the evaluated configurations, while still maintaining high coverage (PICP of 90.00%).
- The reduced interval width indicates improved confidence and efficiency in uncertainty quantification, without compromising reliability.
- The predicted median closely follows the actual trend, even in regions with sharp fluctuations, indicating effective temporal pattern learning.
- The balance of low RMSE (3.57), moderate MAPE (11.82%), and high PICP highlights the model's capability to provide accurate and trustworthy predictions in a compact interval.

11 Model Comparison

The models were compared together on the basis of our chosen error metrics and the results have been displayed below.

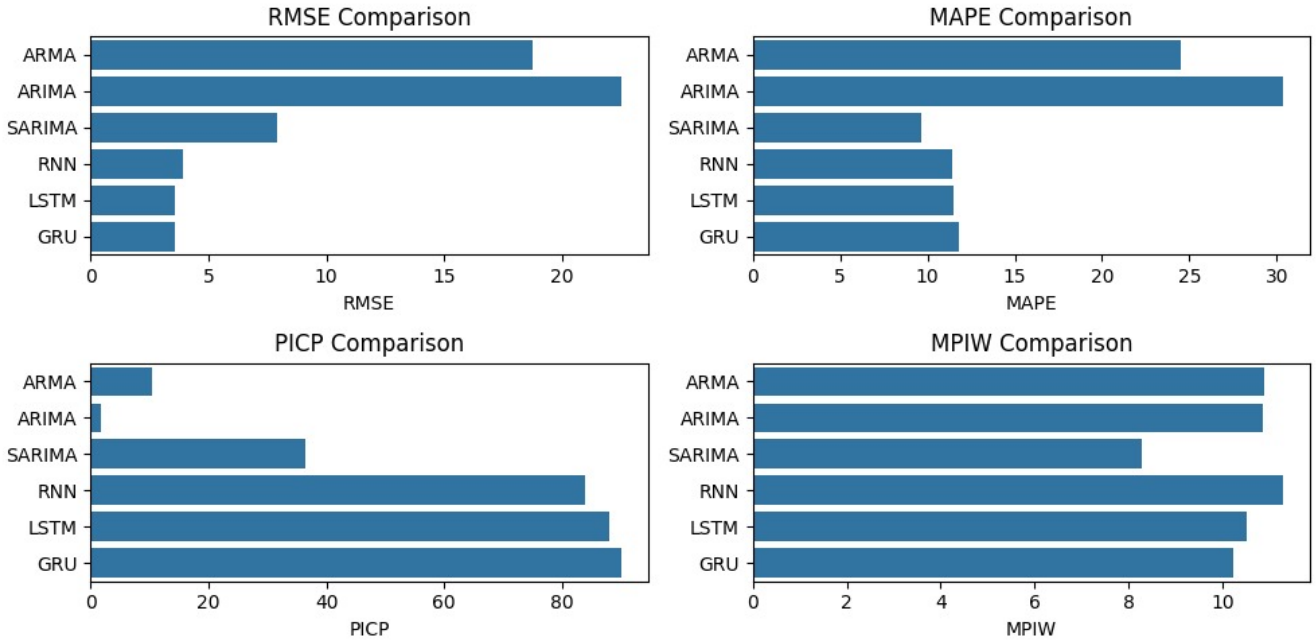


Figure 15: Comparison of models across evaluation metrics

- **RMSE (Root Mean Squared Error)**: ARIMA exhibited the highest RMSE, reflecting substantial prediction errors. In contrast, GRU, LSTM, and RNN yielded significantly lower RMSE values, with GRU performing best, indicating superior predictive accuracy.
- **MAPE (Mean Absolute Percentage Error)**: Consistent with RMSE results, ARIMA had the highest MAPE. Deep learning models—GRU, LSTM, and RNN—achieved notably lower MAPE values, with GRU slightly outperforming the others. SARIMA also performed moderately well.
- **PICP (Prediction Interval Coverage Probability)**: GRU achieved the highest PICP, closely followed by LSTM and RNN, indicating that these models captured the uncertainty in forecasts effectively. Traditional models like ARMA , ARIMA and SARIMA lagged behind, with ARIMA showing especially poor coverage.
- **MPIW (Mean Prediction Interval Width)**: ARMA, ARIMA and RNN had the widest prediction intervals (high MPIW), suggesting less precise uncertainty estimates. GRU and LSTM maintained a more favorable balance, providing reliable coverage (high PICP) while keeping the intervals reasonably narrow.

Overall, the deep learning models especially GRU and LSTM outperformed traditional time series approaches across all metrics. They delivered higher forecast accuracy and better-calibrated prediction intervals, making them more robust and reliable choices for time series forecasting tasks.