APPLYING DEEP LEARNING FOR CITY-WIDE TRAFFIC INFERENCE

A Project Report submitted in partial fulfillment of the requirements for the award of the degree of

BACHELOR OF TECHNOLOGY

IN

COMPUTER SCIENCE ENGINEERING

SUBMITTED BY

PAVAN SRI HARSHA KALLURI (HU21CSEN0100605)

SIRI NEERADI (HU21CSEN0100748)

PARUSA SRINIVAS PRASAD RAO (HU21CSEN0100558)

GOVARDHAN ANNABATHULA (HU21CSEN0102136)

Under the esteemed guidance of

Dr. B. BHARGAVI

ASSISTANT PROFESSOR

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

GITAM SCHOOL OF TECHNOLOGY

GITAM (Deemed to be University)

HYDERABAD

2025

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

GITAM SCHOOL OF TECHNOLOGY

GITAM (Deemed to be University)

DECLARATION

I hereby declare that the project report entitled **APPLYING DEEP LEARNING FOR CITY WIDE TRAFFIC VOLUME INFERENCE** is an original work done in the Department of Computer Science and Engineering, GITAM School of Technology, GITAM (Deemed to be University) submitted in partial fulfillment of the requirements for the award of the degree of B.Tech. in Computer Science and Engineering. The work has not been submitted to any other college or University for the award of any degree or diploma.

Date:

Registration No(s)	Name(s)	Signature
HU21CSEN0100748	SIRI NEERADI	
HU21CSEN0100605	PAVAN SRI HARSHA KALLURI	
HU21CSEN0100558	PARUSA SRINIVAS PRASAD RAO	
HU21CSEN0102136	GOVARDHAN ANNABATHULA	

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
GITAM SCHOOL OF TECHNOLOGY
GITAM (Deemed to be University)

CERTIFICATE

This is to certify that the project report entitled **"APPLYING DEEP LEARNING FOR CITY WIDE TRAFFIC INFERENCE"** is a bonafide record of work carried out by **KALLURI PAVAN SRI HARSHA (HU21CSEN0100605), SIRI NEERADI (HU21CSEN0100748), PARUSA SRINIVAS PRASAD RAO (HU21CSEN0100558), GOVARDHAN ANNABATHULA (HU21CSEN0102136)**, submitted in partial fulfillment of the requirements for the award of degree of Bachelors of Technology in Computer Science and Engineering.

Dr. B. Bhargavi	**Dr. A B Pradeep Kumar**	**Dr. Mahaboob Basha**
Project Guide	Project Coordinator	Head of the Department
CSE Department	CSE Department	CSE Department
GITAM Hyderabad	GITAM Hyderabad	GITAM Hyderabad

ACKNOWLEDGEMENT

Our project report would not have been successful without the help of several people. We would like to thank the personalities who were part of our seminar in numerous ways, those who gave us outstanding support from the birth of the seminar.

We are extremely thankful to our honourable Pro-Vice-Chancellor, **Prof. D. Sambasiva Rao**, for providing the necessary infrastructure and resources for the accomplishment of our seminar. We are highly indebted to **Prof. N. Seetharamaiah,** Associate Director, School of Technology, for his support during the tenure of the seminar.

We are very much obliged to our beloved **Prof. Dr. Mahaboob Basha Shaik**, Head of the Department of Computer Science & Engineering, for providing the opportunity to undertake this seminar and encouragement in the completion.

We hereby wish to express our deep sense of gratitude to **Dr. A.B Pradeep Kumar,** Project Coordinator, Department of Computer Science and Engineering, School of Technology, and to our guide, **Dr. B. Bhargavi,** Assistant Professor, Department of Computer Science and Engineering, School of Technology, for the esteemed guidance, moral support and invaluable advice provided by them for the success of the project.

We are also thankful to all the Computer Science and Engineering department staff members who have cooperated in making our seminar a success. We would like to thank all our parents and friends who extended their help, encouragement, and moral support, directly or indirectly in our seminar work.

Sincerely,
N. Siri,
K. Pavan Sri Harsha,
P. Srinivas Prasad Rao,
A. Govardhan.

ABSTRACT

Local authorities use traffic volume inference as a core element of present-day urban development and intelligent transportation systems which helps optimize traffic management while decreasing congestion and strengthening infrastructure systems. The fast development of deep learning techniques provides an excellent chance to apply these methods for precise real-time and scalable traffic volume forecasting throughout entire urban networks. The project focuses on deep learning architecture implementation through the combination of Graph Attention Networks (GAT) and Long Short-Term Memory (LSTM) networks for inferring traffic volume within city-wide spaces. This research examines how the different models perform with PeMS 04 and PeMS 08 datasets at first before moving onto UTD-19 dataset to determine their individual advantages and weak points in modeling complex traffic patterns. The investigation started with PeMS 04 and PeMS 08 datasets because these benchmarks are extensively used for traffic prediction analysis. The traffic data from District 4 spans three months in PeMS 04 through 307 sensors while PeMS 08 provides data from 170 District 8 sensors for three months. The GAT and LSTM models received their initial assessment regarding predictive accuracy and scalability against computational efficiency because researchers applied them to these datasets. Despite their detailed information the PeMS datasets remain limited regarding the large-scale requirements needed for city-wide traffic prediction. The project moved onto UTD-19 to expand testing scope because this dataset included advanced traffic information from various urban locations. This dataset contains a comprehensive variety of traffic conditions together with multiple road types along with temporal variations which establishes it as an excellent realistic test system for actual world city-sized applications.

TABLE OF CONTENTS

LIST OF FIGURES & TABLES

1. INTRODUCTION

1.1 MOTIVATION

Traffic congestion continues to be an acute problem in contemporary urban life, fueled by urbanization and growing vehicle populations. As the transportation infrastructure cannot keep up with growing travel demand, cities are confronted with immense economic, environmental, and social impacts. Rising congestion translates to longer travel times, increased fuel use, and higher stress levels for commuters. Also, high vehicular emissions cause environmental degradation and global warming, and losses in productivity and increased logistics costs place economic pressures on firms and governments.

Accurate citywide traffic inference is vital for maximizing transport networks, enhancing urban planning, and reducing congestion. Conventional traffic monitoring technologies, including loop detectors, cameras, and GPS-based tracking, offer real-time traffic flow but are plagued by high deployment and maintenance expenses. In addition, these systems mostly monitor main roads and intersections, resulting in sparse data in poorly monitored regions.

Developments in machine learning (ML) and artificial intelligence (AI) have transformed traffic forecasting by utilizing past traffic information, real-time sensor inputs, and external influences like weather, incidents, and time. Deep learning methods, such as Convolutional Neural Networks (CNNs), Graph Attention Networks (GATs), and Federated Learning, have shown considerable accuracy in traffic volume and congestion forecasting. The current models suffer from limitations concerning data privacy, computational complexity, and robustness across heterogeneous urban areas.

This work endeavors to fill these gaps by introducing a new citywide traffic inference framework that incorporates cutting-edge deep learning models to improve prediction quality, scalability, and privacy protection. By overcoming the shortcomings of current models, this work endeavors to advance the design of more effective and resilient urban traffic management systems.

1.2 DEFINING THE PROBLEM

Traffic inference is the ability to forecast traffic conditions—volume, speed, flow, or congestion—based on sensor data, history, or real-time feeds. It's a building block of Intelligent Transportation Systems (ITS) with the goal of maximizing traffic management, minimizing congestion, and improving safety. Current systems include old-fashioned statistical models to state-of-the-art machine learning (ML) and graph-based systems. Below, I’ll outline key systems,

their methodologies, and their limitations, focusing on their applicability to scenarios like your project (e.g., PeMS and UTD-19 datasets with 100 nodes/detectors).

1.3 OBJECTIVE

A comparative analysis between Graph Attention Networks (GAT) and Long Short-Term Memory (LSTM) networks is also a significant goal. Through the assessment of their efficiency in identifying spatio-temporal dependencies, the research seeks to identify which method provides higher accuracy and efficiency for citywide traffic inference.

1.4 OUR CONTRIBUTION

A critical evaluation of current traffic forecasting techniques will be performed based on their robustness, weakness, and practicability in real-world city scenarios. In-depth consideration of deep learning methodologies like Convolutional Neural Networks (CNNs), Graph Attention Networks (GATs), and ensemble methods will be presented, examining the performance of such methods in traffic inference.

Research Gaps and Challenges Identification: Common issues in existing traffic forecasting models, such as sparsity of data, computational inefficacy, and model generalization issues, will be investigated. Emphasis will also be placed on privacy issues involved in data-driven traffic forecasting, with an eye towards privacy-protecting methods such as Federated Learning that guarantee secure and ethical use of data.

Proposal of a Novel Traffic Inference Framework: A new traffic prediction model will be established, combining GAT, CNNs, and LSTMs to enhance scalability and efficiency. In addition, novel data augmentation methods will be developed to tackle data sparsity problems, making traffic prediction more reliable and accurate.

Through combining state-of-the-art AI approaches with realistic urban traffic management needs, this work seeks to develop a next-generation traffic inference system. The scheme improves scalability, enhances prediction quality, and protects privacy, hence enabling smarter, more efficient, and sustainable cities.

2. LITERATURE SURVEY

2.1 OVERVIEW OF OUR SURVEY

Traffic forecasting is an important component of intelligent transportation systems, supporting traffic management, congestion alleviation, and urban mobility enhancement. A wide range of models have been proposed over the years, generally classified into classical statistical models, deep learning models, and federated learning models. This survey identifies major developments in these areas, with specific emphasis on federated learning for cross-city traffic forecasting and dynamic multi-view graph neural networks. The latest improvements are the Personalized Federated Learning for Cross-City Traffic Prediction (pFedCTP) model (2024), which facilitates decentralized learning while ensuring data privacy. It involves a Spatio-Temporal Neural Network (ST-Net) to capture spatial and temporal heterogeneity effectively, supporting enhanced adaptability for sparsely data cities. All the same, its limitations such as negative transfer effects and model personalization need to be further researched.

Another major development is Dynamic Multi-View Graph Neural Networks (DMV-GNNs) (2023), which make use of multiple affinity graphs to dynamically encode spatial-temporal dependencies. With the use of Graph Convolutional Networks (GCNs) and self-attention, DMV-GNNs are boosted with generalization and resilience. Yet, their high computational overhead is still a drawback. Earlier contributions formed the basis of these models. Temporal Multi-View Graph Convolutional Networks (T-MVGCNs) (2021) enhanced prediction accuracy by capturing spatial-temporal relationships using multi-view data and self-attention mechanisms. They, however, needed large amounts of labeled data. The CityTraffic Model (2019) applied neural memorization and generalization methods to achieve equilibrium between short-term and long-term traffic forecasting, but at the cost of centralized data storage, which was a privacy issue.

While these advancements have significantly improved traffic prediction accuracy and efficiency, challenges such as data privacy, computational complexity, and model interpretability still need to be addressed for real-world deployment. Below table 2.1.1 describes a complete overview of our literature review.

Paper Title	Abstract	Data set	Algorithm	Results	Publ;lication Details
Matrix and Tensor Based Methods for Missing Data Estimation in Large Traffic Networks	Addresses missing data in ITS using sensor data from large networks. Uses matrix and tensor methods to predict missing values. Improves data accuracy for better traffic management.	Traffic speed data from Singapore's road network. Speed profiles sampled every 5 minutes. Covers 1500 road segments over one week.	Uses FPCA, CP Decomposition, and VBPCA for data recovery. Algorithms create low-rank approximations of data. Compares method accuracy across road types and days.	FPCA is stable across daily traffic variations. CP Decomposition performs well on smaller roads. VBPCA is highly accurate, especially for expressways.	Authors: Muhammad Tayyab Asif, Et al. Published on July 2016 [2]
City-wide Traffic Volume Inference with Loop Detector Data and Taxi Trajectories	Infers city-wide traffic volume using loop detector and taxi data. Uses a semi-supervised learning model to handle data gaps. Tested with real data, showing high inference accuracy.	Data collected from 155 loop detectors and 6,918 taxis. Covers 17 days in Guiyang, China. Includes road network and weather data for context.	Combines loop detector and taxi data to estimate traffic. Builds a spatio-temporal affinity graph for volume inference. Uses graph-based semi-supervised learning for accuracy.	Achieves accurate city-wide volume estimates.Performs well even with sparse data from loop detectors. Outperforms baseline methods in volume estimation accuracy.	Authors: Chuishi Meng, Xiuwen Yi Et al. Published on 2017-11-07 [3]
CityTraffic: Modeling Citywide Traffic via Neural Memorization and Generalization Approach	Proposes a model to predict missing traffic speed and volume data. Combines memorization (for speed) and generalization (for volume) methods. Tested successfully on data from two cities.	Data includes GPS trajectories and volume records from loop detectors and cameras. Covers thousands of road segments in Guiyang and Jinan. Contains contextual road data like road levels and speed limits.	CT-Mem memorizes speed correlations using historical patterns. CT-Gen generalizes volume patterns across roads using key-value attention. Both methods use neural attention networks to improve accuracy.	Achieves high accuracy in speed and volume prediction. Outperforms other methods in different urban settings. Effective on roads with lower traffic data frequency.	Authors: Xiuwen Yi, Zhewen Duan Et al. Published on 2019-11-03 [4]
City Traffic Volume Interface with Surveillance Camera Records	CityVolInf framework infers citywide traffic volume using limited camera data. Combines spatiotemporal similarities and simulations for better accuracy. Achieves significant accuracy improvements over existing methods.	Collected from Jinan, China with 405M camera records. Contains road network info and vehicle data. Some data is incomplete due to limited camera coverage.	Similarity Module: Uses semi-supervised learning for spatial traffic patterns. Simulation Module: Uses SUMO for traffic flow estimation. Combines both modules to minimize prediction errors.	Outperforms other methods in accuracy. Demonstrates low error rates in tests. Performs well with varying dataset sizes.	Authors: Yanwei Yu, Xianfeng Tang, Huaxiu Yao, Xiuwen Yi , and Zhenhui Li Published on December 2021 [5]
Temporal Multi-view Graph Convolutional Networks for Citywide Traffic Volume Inference	CTVI estimates traffic across all road segments, even with limited data. Uses graph convolution and temporal attention for accuracy. Outperforms existing models in real-world tests.	Data from Hangzhou and Jinan, China. Road features like lanes and speed limits. Limited sensor coverage.	Graph convolution for spatial patterns. Temporal attention for time-based patterns. Combined learning for better predictions.	CTVI has higher accuracy. Low error rates on tests. Handles complex traffic trends.	Authors: Shaojie Dai, Jinshuai Wang Et al. Published on December 2021 [6]
Dynamic Multi-View Graph Neural Networks for Citywide Traffic Inference	CTVI+ model developed to infer citywide traffic volume using limited data. Uses temporal self-attention and multi-view graph neural network to capture spatial and temporal patterns. Achieves better accuracy than baseline models.	Datasets from Hangzhou, Jinan, and two California cities. Features include road segment details like lanes and speed limits. Limited sensor coverage leads to data gaps.	Graph Convolution: Multi-view convolution for spatial and feature graph encoding. Temporal Attention: Learns dependencies across daily, weekly patterns. Joint Learning: Combines spatial and temporal features for robust predictions.	CTVI+ achieves higher accuracy than existing methods. Shows improvements in RMSE and MAPE metrics. Robust across multiple real-world datasets.	Authors: Shaojie Dai, Jinshuai Wang Et al. Published on 2023-02-24 [7]

Figure 2.1 : Literature Survey

2.2 RESEARCH GAPS

Even with significant progress, some critical research gaps continue to exist for traffic prediction models. One major issue is privacy in data aggregation. Centralized models rely on large datasets, which pose serious privacy risks. While federated learning provides a decentralized alternative, challenges such as communication overhead and model personalization require further study.

Another challenge is negative transfer impacts. When knowledge is transferred from data-rich cities to data-scarce regions, significant differences in traffic patterns can lead to inaccurate predictions. Advanced domain adaptation techniques are needed to minimize this effect. Limited generalizability is also a concern. Many models struggle to adapt to new cities with unique spatial-temporal characteristics. Future research should focus on adaptive learning techniques to improve model flexibility and performance in unseen environments.

Moreover, computational complexity remains a bottleneck. State-of-the-art deep learning models, such as DMV-GNNs, demand extensive computational resources, making large-scale deployment difficult. Solutions like model compression and distributed computing need further exploration. Lastly, explainability in AI for traffic forecasting is lacking. Most modern models function as black boxes, making it difficult to interpret their decision-making processes. Enhancing interpretability will increase trust and encourage adoption in real-world applications. Addressing these challenges will lead to more scalable, privacy-aware, and efficient traffic prediction models, ultimately supporting smarter urban mobility solutions.

3. PROBLEM DEFINITION & OBJECTIVES

3.1 PROBLEM DEFINITION

Traffic congestion is an urban metropolitan issue, creating economic loss, environmental damage, and reduced quality of life. Effective traffic management depends on real traffic volume inference to optimize transport networks and resist congestion. Traditional monitoring methods such as cameras, inductive loops, and GPS-based tracking are costly, raise privacy concerns, and are non-scalable. The goal of this research is to accurately calculate city-level traffic volumes based on sparse sensor data with the additional problems of solving data privacy and model generalization over diverse urban environments. The primary issues are:

1. **Data Availability and Scarcity:** Traffic monitoring equipment varies between cities, with some cities having plenty of sensor data and others having too little data collection. Forecasting traffic conditions based on sparse sensor data must rely on robust predictive models.
2. **Privacy Concerns with Data:** Conventional traffic monitoring relies on centralized data gathering, which means revealing location-based sensitive information. Finding a balance between privacy and maintaining prediction accuracy is a significant challenge.
3. **Generalization Across Cities:** Traffic patterns differ significantly with respect to infrastructure, population density, and commuting behavior. A model trained in one city may not generalize to another without extensive retraining.
4. **Computational Efficiency:** Large-scale traffic prediction models are computationally costly and thus not deployable on edge devices or in resource-scarce environments. Real-world applications must optimize efficiency.

3.2 OBJECTIVES

Scalability of traffic prediction models is one of the primary goals of this research. The system ought to manage traffic data for small cities up to large metropolitan cities without performance decline. By providing an adaptable structure, the research seeks to implement traffic inference models across different city settings.

Another central emphasis is on optimizing computational power. Traffic prediction models tend to be computationally intensive, which can be a limitation for real-time applications. This research aims to create a method that strikes a balance between model complexity and computational efficiency, making it deployable even on resource-limited systems.

A comparative analysis between Graph Attention Networks (GAT) and Long Short-Term Memory (LSTM) networks is also a significant goal. Through the assessment of their efficiency in identifying spatio-temporal dependencies, the research seeks to identify which method provides higher accuracy and efficiency for citywide traffic inference.

Finally, the research hopes to create a model that can learn both local and global traffic patterns. Traffic flows differ by region, and a good inference model must be able to grasp localized congestion problems as well as overall citywide traffic patterns. This guarantees a complete prediction system that supports better decision-making for traffic control.

4. EXISTING SYSTEM & LIMITATIONS

4.1 EXISTING SYSTEMS

Traffic forecasting is an essential component of intelligent transportation systems that supports congestion management, route optimization, and city planning. There are existing methods that can be classified into Statistical Models, Machine Learning Models, and Deep Learning Methods.

4.1.1 STATISTICAL MODELS

Conventional statistical models like time-series analysis (e.g., ARIMA—AutoRegressive Integrated Moving Average) and regression-based approaches have been the cornerstones for traffic inference. They work by utilizing historical traffic patterns (e.g., hourly counts) and external conditions (e.g., weather, time of day) in order to predict future conditions. ARIMA models examine time-series patterns for assumed stationarity following differencing. Regression models capture linear or polynomial relationships between input (e.g., traffic volumes) and output (e.g., estimated volume). Employed in initial traffic management systems by organizations such as the U.S. Federal Highway Administration (FHWA) to make predictions on highway volumes.

Statistical models such as ARIMA and SARIMA are widely adopted because they can be interpreted easily. ARIMA performs well on short-term forecasts but has trouble with non-linearities and enormous datasets. SARIMA is better than ARIMA in that it takes into consideration seasonal patterns, but it needs stationarity, which requires diligent parameter adjustment. These models overlook real-time environmental factors like climatic conditions, accidents, and abrupt traffic increases.

4.1.2 MACHINE LEARNING MODELS

Machine learning algorithms enhance prediction by discovering non-linear relationships in data. Decision trees are easy to interpret but overfit on small datasets. Random forests avoid overfitting with ensemble learning but need high computational power. SVMs are efficient at classifying congestion levels but do not work well with high-dimensional data. While machine learning models perform better than statistical methods, they require a lot of feature engineering and large labeled data, which are resource-intensive.

4.1.3 DEEP LEARNING METHODS

Deep learning methods are very useful in traffic prediction because they are capable of recognizing complex patterns. CNNs are good at recognizing spatial dependencies but poor at processing temporal sequences. GNNs are good at modeling road networks but consume lots of computational power. LSTMs are good at sequential data but poor in spatial knowledge. Federated learning is good in terms of privacy and scalability but needs high communication bandwidth. Although these models provide higher accuracy, their implementation is marred by high computational requirements and demand for data.

4.1.4 RULE BASED FUZZY LOGIC SYSTEMS

Fuzzy logic and rule-based models utilize pre-established rules or linguistic variables in inferring traffic. They tend to be implemented as part of adaptive traffic signal control or traffic congestion detection. Rules (e.g., "if flow > 500 vehicles/hour, then congestion = high") or fuzzy inference (e.g., combining "high speed" and "low occupancy" to infer "free flow") work on sensor data. Fuzzy traffic light controllers adapt signals according to real-time traffic states.

4.1.5 HYBRID AND EMERGING SYSTEMS

Hybrid systems combine temporal and spatial models (e.g., LSTM + GNN) or integrate ML with simulation tools (e.g., SUMO). Emerging approaches use reinforcement learning (RL) or deep RL for adaptive control. LSTM-GNN hybrids process sequences per node and graph interactions; RL optimizes traffic signals based on rewards. RL-based traffic light control (IEEE Trans. ITS, 2018). Combines time-series and graph data. Trains jointly or in stages, often using simulators for validation.

4.2 LIMITATIONS

While traffic forecasting models have come a long way, they still have significant challenges.

- **Scalability Challenges**: Deep learning models need huge datasets and lots of computational resources, which makes real-time large-scale deployment challenging. Cloud-based solutions also cause latency, impacting efficiency.
- **Data Privacy Issues:** Most systems are based on centralized data gathering from GPS, mobile phones, and surveillance cameras. This poses serious privacy issues due to threats of unauthorized access, cyber attacks, and misuse of data.

- **Limited Generalizability Between Cities:** Patterns of traffic are highly different between various cities. Models learned using data from one city will be less efficient in another, necessitating expensive retraining and new datasets.
- **High Cost of Deployment**: Installation of effective traffic forecasting systems involves a large network of sensors, IoT devices, and cloud computing resources. The high expense makes deployment unaffordable for financially strapped cities.
- **Dependence on High-Quality Data:** The accuracy of traffic prediction relies on having high-quality data. Missing data, sensor malfunctioning, and the challenge of including real-time external variables such as weather and accidents impact model trustworthiness.
- **Model Interpretability & Explainability:** Deep learning models tend to be black boxes, making it difficult for policymakers and urban planners to interpret and believe in their decision-making.

5. PROPOSED APPROACH & METHODS

The objective of this study is to make a comparison of Long Short-Term Memory (LSTM) networks and Graph Attention Network (GAT), treating them as hypotheses for traffic volume modeling and prediction. Our approach is to reduce inference error using both spatial and temporal dependencies present in traffic data. Traffic volume inference deals with estimating and predicting vehicle flow between different road segments of a city from the historical and real time data. The problem is inherently spatio-temporal, i.e., the sequential dependencies in time series data, and the topological relations between road segments should additionally be captured. In this study, our main task is to assess and compare the performances of LSTM and GAT with respect to this problem potency. The dataset we have, is traffic volume records gathered from a city wide network of sensors with traffic volume, dates and times, geospatial information, and other related factors with respect to traffic flow, including the weather conditions, holidays and events which affect traffic.

Data cleaning of missing values (using different interpolation techniques) for preprocessing was done, normalization to make the model convolution easier, graph construction (nodes = road segment; edges = connectivity), and time series formation (sequential traffic record to format the output for the LSTM to train). LSTMs are very commonly used for time series forecasting because they can capture long range dependence. The output of our LSTM based approach is an input layer allowing it to encode the sequence data with a given time window, LSTM layers calculating the dependence over a time window for traffic patterns, a dense layer, and an output layer mapping the learned representations to future time steps of traffic volume predictions. For training, historical traffic data is used and the LSTM model is trained with the mean squared error (MSE) loss function using the Adam optimizer. Such graph-structured data can be modeled using GATs to process.Starting with a graph, we construct this graph by treating each road segment as a node and connecting the edges. We analyze the traffic patterns using a Graph Attention Network (GAT) that uses the traffic data from adjacent to road segments to determine the cluster congestion. Secondly, we add a temporal attention layer to be able to capture recurring traffic flow patterns with the time. Ultimately, an additional fully connected layer completes the process for creating accurate predictions of traffic volume. The layered approach ensures an overall and dynamic clue to the way traffic conducts.In that sense, since GATs inherently capture spatial relationships, they will naturally leverage connectivity information to improve inference accuracy. Both models fit on historical data using 80% of the dataset for training and 20% for evaluation. Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), and R-Squared (R^2) Score are some of the important evaluation

metrics used in this model. We then compare LSTM and GAT models in detail on their performance under varying circumstances (peak and non peak hours).

The final results will later not only help to decide which model is the most appropriate for predicting the actual traffic volume but also provide some initial information about the seasonality of the data. Our study thus tries to improve inference abilities regarding traffic volume using LSTM and GAT models. A comparative study will be worth the insights it will offer on the prospects and shortcomings of each approach to traffic management system in general. Such methodology makes sure that there was a solid road to investigate the nature of city wide traffic patterns, allowing better urban planning and mobility solutions.

5.1 METHODS

5.1.1 LONG SHORT TERM MEMORY

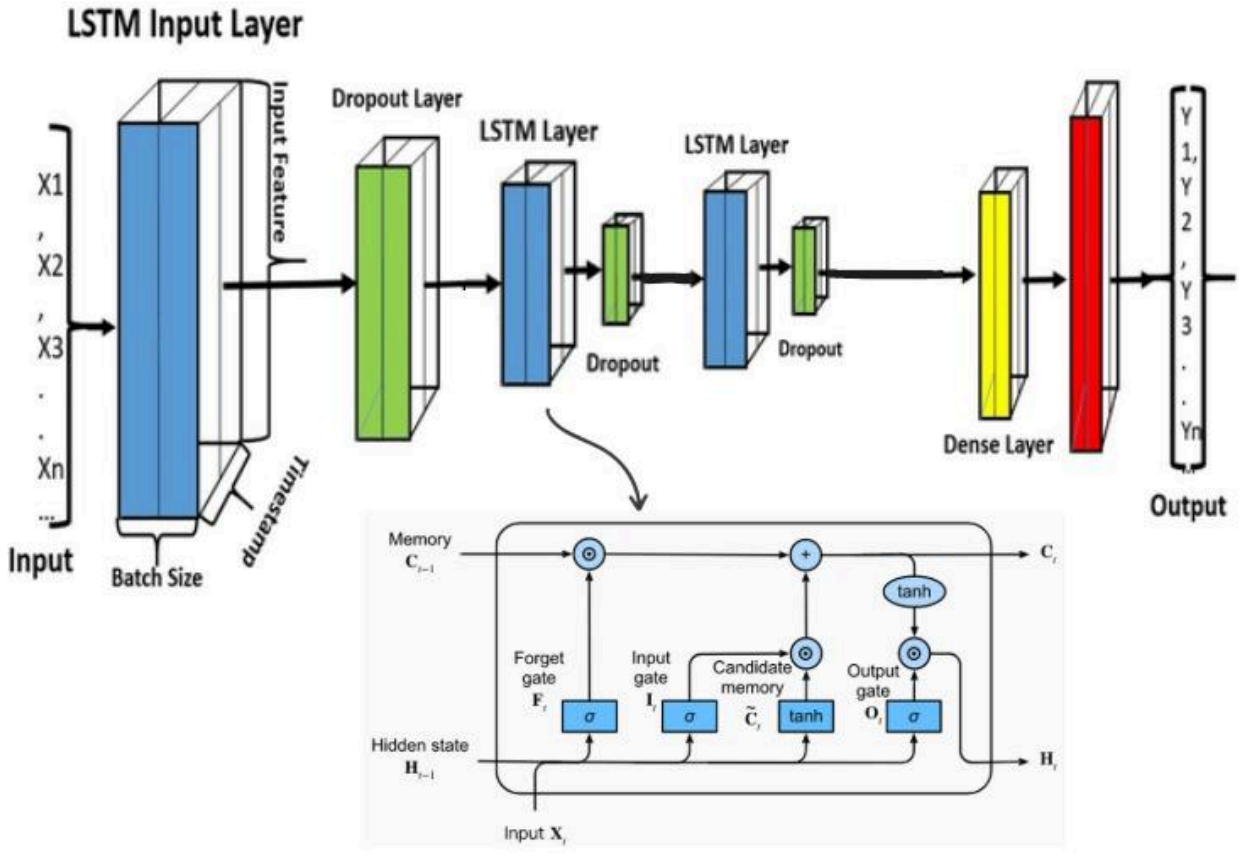

Figure 5.1.1 : LSTM (Long Short-Term Memory) Architecture

A Long Short-Term Memory (LSTM) network serves as the main component in our study for city-wide traffic volume inference because it models the temporal dependencies found in traffic data. The Long Short-Term Memory (LSTM) network serves as a recurrent neural network (RNN) solution for sequential data processing while resolving the gradient vanishing problem traditional

RNNs experience. LSTM serves effectively for traffic volume prediction because traffic levels change dynamically according to daily periods and environmental conditions and road-related events. A process that deploys the path-finding algorithm consists of an input layer merging historical traffic information into set-length sequences followed by multiple LSTM layers to discover temporal dependencies and extract significant patterns and a fully connected dense layer converting hidden states into predicted traffic amounts before issuing future traffic estimates through the output layer.

The model receives traffic data for optimization through an MSE loss function that uses Adam optimization for achieving successful training. Event-based processing of the dataset through mini-batches together with dropout regularization helps prevent model overfitting in the system. The strength of LSTM in traffic data time series analysis becomes limited when trying to understand spatial relationships because spatial dependencies are crucial for precise city-wide traffic forecasting. The condition of traffic in one segment heavily depends on neighboring segments yet Long Short-Term Memory (LSTM) cannot effectively process this input by itself. GATs represent the necessary solution because they are specifically created to analyze graph data structures while maintaining spatial relationship awareness.

5.1.2 GAT

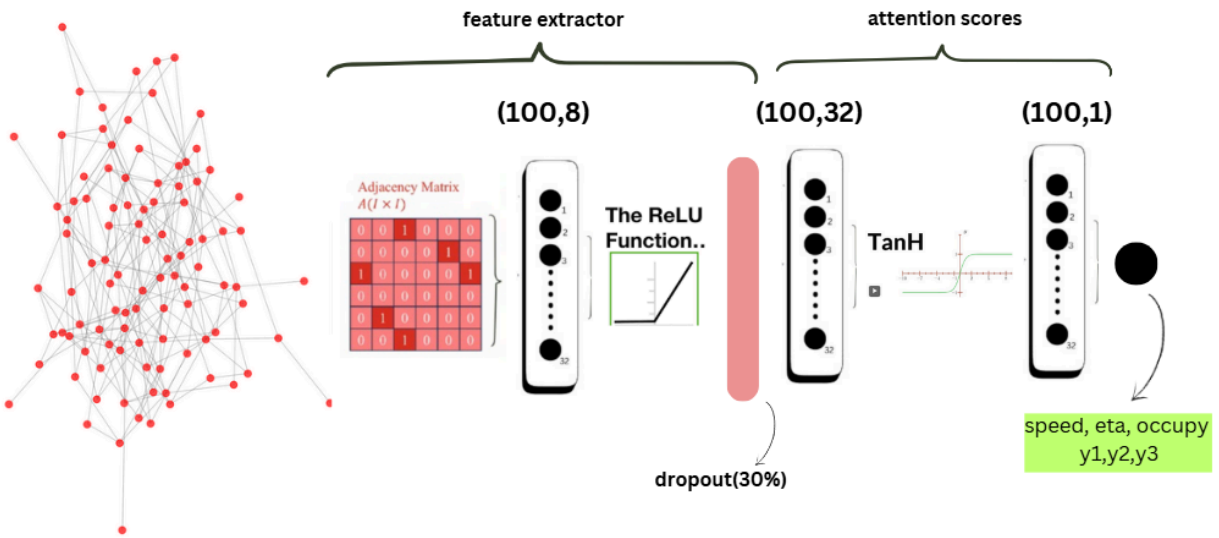

Figure 5.1.2 : GAT Architecture

The spatial relationships between road segments in road networks become effective with Graph Neural Networks (GNNs) which make them suitable for modeling broad-scale traffic inference. Two versions of GNN studied in our research include the Graph Convolutional Networks (GCN) and Graph Attention Networks (GAT). Traffic data processing with GNNs requires

transforming the road network into a graph structure with road segments as nodes while edges demonstrate network connectivity. The characteristics of individual nodes include traffic volume measurements accompanied by time-based characteristics together with elements from the external environment such as weather and road conditions.

The graph-structured data within GCN receives information aggregation from neighboring nodes through convolutional operations which helps the model learn spatial dependencies. Using multiple graph convolutional layers GCN processes traffic volume data by combining information across adjacent nodes after which fully connected layers perform the prediction mapping. The main drawback of GCN occurs because it treats each connecting node identically while traffic flow within urban zones demonstrates specific nodes have stronger impact on others. We address this shortcoming by incorporating Graph Attention Networks (GAT) because these networks establish attention mechanisms to determine the relevance weights between neighboring nodes. GAT employs graph attention layers which identify key influential neighboring nodes among all connections instead of providing universal treatment to all neighbors. The multi-head attention mechanism in this model strengthens robustness by identifying complex relationships that exist between road segments. The embeddings generated by GCN follow a processing stage through fully connected layers to create traffic volume predictions.

The predictive power of GAT surpasses GCN because it determines the criticality of road segments dynamically which allows for improved accuracy rates. Because of the extra attention weight calculations there is increased computational complexity in this process. The comparison within this research investigates LSTM together with GCN and GAT regarding their modeling of time and spatial patterns as well as their computational speed and predictive accuracy. The temporal pattern recognition strength of LSTM prevents it from understanding spatial relationships whereas GCN excels at spatial pattern discovery but makes no distinction between linked nodes. The spatial modeling process is upgraded with attention mechanisms in GAT which enables the model to prioritize important road segments and achieve better prediction accuracy. LSTM provides computational simplicity during time-series forecasting yet GCN increases the model's complexity because it operates with graph-based elements.

The attention-based design of GAT represents state-of-the-art spatial modeling but it needs powerful computational systems because of its attention mechanisms.The comparison information in this research helps identify powerful characteristics and weaknesses of each model so users can pick the best solution for real-world traffic monitoring needs when planning urban transportation systems.

6. TECHNOLOGIES USED

The implementation of our complex traffic prediction system integrates deep learning through Long Short-Term Memory (LSTM) networks together with GAT while using different Python libraries. These collections of software libraries handle different duties including data handling, model training, visual representation, network analysis and system communication.

Figure 6.1 : Technologies Used

1. **Pandas**: The project utilizes the pandas library in a significant way for importing and cleaning traffic data obtained from Excel files including corrected_speed_data.xlsx and detector_distances.xlsx. The tool executes operations that involve changing infinite values to NaN along with timestamp conversion and detector ID groupings and missing data filling (backward/forward). The traffic volume inference process benefits from pandas since the library makes sure raw datasets with inconsistencies like missing values or irregular formats get transformed into usable LSTM inputs as time-series sequences. Using pandas in the load_data method establishes a reliable data structure that fulfills the project requirements for making precise training sequences with LSTM algorithms.

2. **NumPy**: The main objective of NumPy is to perform numerical calculations together with array-based operations. The project benefits from numpy through its array capabilities which create LSTM-ready sequences from data and conduct feature scaling and mean squared error evaluation. The tool aids in preparing multidimensional arrays through operations that

restructure X and y datasets before running the training process while performing mathematical operations efficiently. Numpy executes sequence scaling operations in the prepare_training_data method while processing essential LSTM model input format creation for efficient training of large UTD-19 datasets.

3. **Matplotlib:** This component functions with seaborn to create different plots such as training loss curves and prediction scatter plots and error distribution charts in the_save_training_results method. Visualization stands essential for assessment of model results together with traffic pattern comprehension. The visualizations allow model evaluation through comparisons between estimated and observed traffic metrics (speed, flow, occupancy) which supports LSTM assessment when using PeMS and UTD-19 datasets for better model development.

4. **Seaborn**: The role of Seaborn in the project involves augmenting matplotlib by delivering beautiful plots such as heatmaps and histograms. Traffic predictions become easier to understand through this approach as it enhances data interpretation of complex relationships in actual and predicted values. The assessment process helps determine the accuracy of LSTM dynamics for capturing traffic patterns when moving from PeMS to UTD-19.

5. **Requests:** The application uses HTTP request functions to retrieve data from APIs. Within the get_weather_data method the component fetches up-to-the-minute WeatherAPI data to use in predicting Estimated Time of Arrival under varying weather conditions.Conditions of rain reduce traffic speed as confirmed by the weather factor. The system integration of weather data improves prediction accuracy so it can better meet needs in actual citywide operations.

6. **Datetime:** The project manager handles timestamp operations together with ETA evaluation and produces time-related features using time_of_day and day_of_week formats. Duration features play an essential role in traffic modeling because datetime effectively processes time-series data while computing estimated times of arrival in ways that improve prediction relevance with LSTM requirements.

7. **NetworkX:** The goal of Network analysis and graph algorithms serves the project. The program constructs detector distance data networks and utilizes pathfinding algorithms (e.g. shortest path and Kruskal's MST) in build_graph and kruskal_shortest_path functions.

The code implements pathfinding features through NetworkX modulo the abstract focuses on LSTM network vs GAT. This supplementary capability allows route optimization as well as LSTM prediction integration thus uniting spatial and temporal evaluation.

8. **Scikit-learn:** This part assists in the implementation of MinMaxScaler for normalization and train_test_split for dataset partitioning as well as mean_squared_error and r2_score performance metrics. The LSTM training benefits from feature normalization which uses a uniform scale thanks to Scikit-learn and this model evaluation depends on data splitting methods and scoring metrics. The ability to perform robust validation relies on this essential feature for carrying out performance comparisons between PeMS and UTD-19 datasets.
9. **OS and Sys:** Operating system interaction and system-specific parameters. The project manager controls the file system paths by making directories such as MODEL_DIR and executes system behavior commands.Contribution: Ensures portability and organization of model files and metrics, critical for iterative experimentation and result storage in a deep learning project.
10. **Time:** Time-related functions.Measures training duration in train_model. The contribution to the project involves monitoring LSTM computational speed to enhance performance when working with the UTD-19 large dataset.
11. **JSON:** The Project ETL Functions Use JSON Files to Store Training Results through _save_training_results. Structured storage along with data retrieval through the platform helps users conduct long-term analyses alongside result comparisons for LSTM outcomes.
12. **Warnings:** The project benefits from this component by dismissing non-essential warnings that prevent console output from becoming cluttered. The application enhances user experience because it directs attention towards meaningful outputs such as model metrics while execution occurs.
13. **TensorFlow and Keras:** Purpose: Deep learning framework and high-level API. The project uses build_model for implementing LSTM model implementation and includes custom loss functions and controls training through EarlyStopping callback management. The LSTM model receives its construction training and evaluation capabilities from tensorflow and keras which form the project core. The system performs demanding spatio-temporal traffic data operations which enables successful data expansion from PeMS to UTD-19 systems without sacrificing performance levels.
14. **Psutil**: Monitors memory usage in view_system_status.The system maintains efficient operation because of this module with special importance for extensive deep learning operations on extensive datasets.

7. IMPLEMENTATION

The React program first enables users to enter location data from and to and choose between LSTM/GAT prediction models. After users submit their request the system retrieves location distance while it conducts speed computations and best-path selection and distance prediction and speed assessment with model metric retrieval. The system obtains weather information for that specific place at the same time. The system displays results through the UI by showing information about distance together with ETA, speed, path, model metrics, weather conditions and an interactive map. The process finishes once the visual presentation of predictions and insights displays to the user.

7.1 DATASETS

7.1.1 PEMS 04

The PEMS 04 dataset represents an essential resource for machine learning studies along with traffic prediction and it draws its data from the California Performance Measurement System (PeMS). It contains records for District 4 encompassing the San Francisco Bay Area where the United States experiences one of its most severe metro traffic congestion. The dataset features complete spatiotemporal traffic information generated through 307 sensors that track 29 roads in the region. Traffic sensors measure continuous traffic data regarding vehicle movements and driving speeds and vehicle occupancy levels that serve as essential data for transportation intelligence applications and congestion analysis and traffic prediction research.

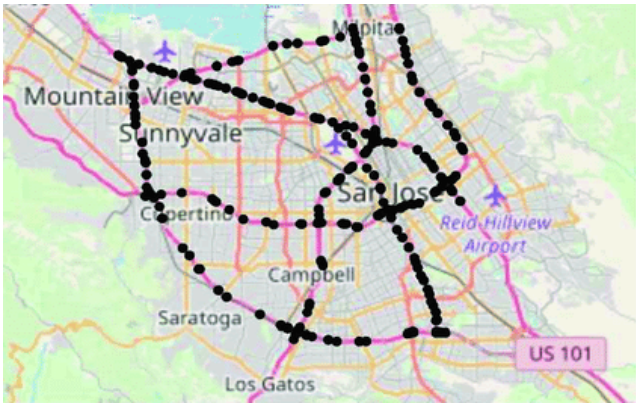

Figure 7.1 : PEMS 04 (San Francisco)

The PEMS 04 dataset features its data in a timed and sequential format. The dataset records data at five-minute intervals that result in 16992 overall time steps during its collection period running from January 1, 2018, through February 28, 2018. The precise sampling interval of 5-minutes enables effective short-term and long-term usage in traffic prediction models. The dataset demonstrates an evaluation-friendly split through its 13589 samples in the training category and its 3398 samples in the testing category.

7.1.2 PEMS 08

The PEMS 08 dataset comes from California Performance Measurement System (PeMS) District 8 that collects data from San Bernardino and Riverside counties in Southern California. The area experiences significant traffic congestion because suburban residents travel between these suburban zones and main urban cities. The dataset offers traffic data with high resolution from 170 sensors located on 8 roads which functions as a reference point for traffic prediction studies and congestion analysis and machine learning research activities. PEMS 08 maintains a time-series organization with 5-minute intervals defining each time step just like other PeMS datasets. The dataset spans 17,856 time steps which covers the collection period starting from July 1, 2016 and ending on August 31, 2016. There exist 14,265 training samples and 3,567 testing samples in the dataset which provide adequate data for model development and evaluation purposes.

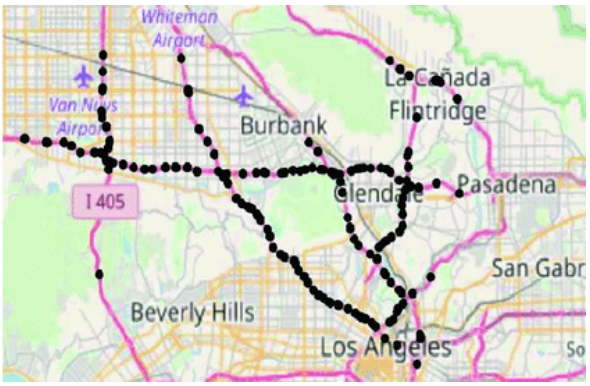

Figure 7.2: PEMS 08 (San Bernardino)

7.1.3 UTD 19

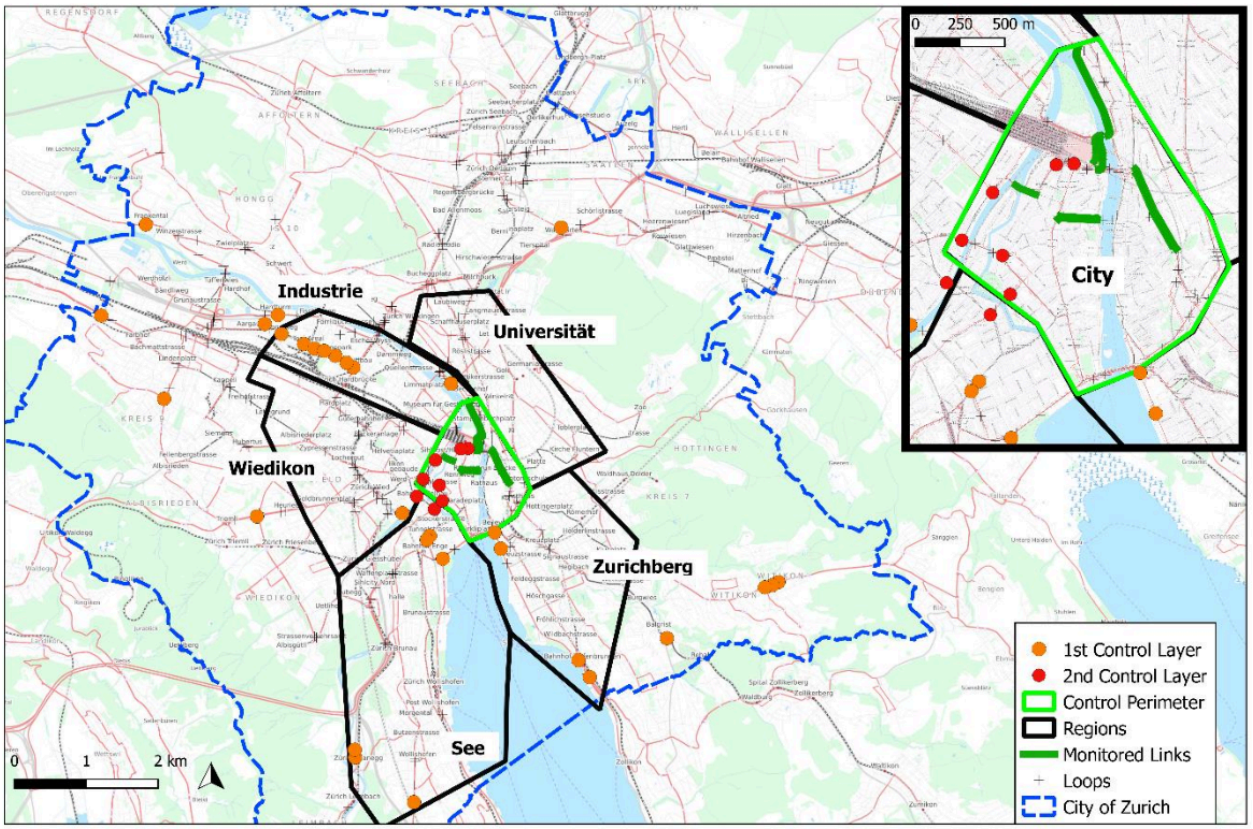

Figure 7.3 : UTD 19 Augsburg

UTD 19 represents a traffic dataset which researchers collect in Augsburg Germany for both traffic flow predictions and urban mobility examinations. A total of 100 sensor points across the city provide information for analyzing urban traffic conditions in this European environment. UTD 19 differs from PEMS 04 and PEMS 08 since it concentrates on a dense urban environment by monitoring road network conditions instead of freeways.

This dataset from Augsburg serves students of urban transportation because the German city hosts traffic that combines vehicles with pedestrians alongside public transportation. The standard dataset monitoring intervals identify traffic conditions at its locations through sensors established across road segments and intersections. The traffic sensors document various metrics including vehicle traffic measurement alongside speed pace and congestion patterns which serve as enabling data for deep learning models including Graph Attention Networks (GATs) and Long Short-Term Memory (LSTM) networks.

This dataset with its 100 locations most likely traces an intricate road network that shows traffic conditions at one location directly relying on adjacent zones. The dataset works well for spatial-temporal research because traffic prediction methods need to track time effects while considering area relationships throughout the city. The application areas include intelligent traffic signal control systems together with route optimization and congestion management support via this dataset.

The UTD 19 analysis of Augsburg takes into account weather patterns and environmental factors that affect traffic flow patterns which are typical for European cities with weather fluctuations. The analytical tool allows for the assessment of urban policy programs including low-emission zones together with bicycle lanes and public transport integration systems on citywide traffic movements.

The UTD 19 dataset serves as a powerful tool for researchers who wish to conduct investigations on urban traffic behavior in Augsburg Germany. The 100 sensor locations in this dataset produce a granular visual of traffic movements alongside congestion behavior as well as spatial relationships in the urban space. Approximately 856 traffic parameters in the UTD 19 dataset enable researchers to construct advanced machine learning models which help optimize urban mobility and develop smart city solutions.

7.2 DATA CLEANING

Raw data often contains inconsistencies such as missing values, outliers, and incorrect timestamps.To maintain consistency of data, we conducted aggressive preprocessing. Missing values were treated through imputation methods, such as interpolation and k-nearest neighbors

(KNN) algorithms. Outliers were identified and removed employing statistical filtering techniques such as Z-score and interquartile range (IQR). We also synchronized data from different sources by synchronizing timestamps and normalizing all records to a uniform temporal granularity of five-minute intervals.

1. **Imputation:** Missing values were filled using interpolation and k-nearest neighbors (KNN) algorithms.
2. **Outlier Removal:** Z-score and IQR-based filtering were used to detect anomalies.
3. **Time Synchronization:** Data from multiple sources were synchronized to a common temporal resolution (e.g., 5-minute intervals).

7.3 FEATURE ENGINEERING

To improve the performance of the models, we designed a relevant feature set. Temporal features like hour of the day, day of the week, and seasonality patterns were extracted to pick up periodic traffic patterns. Spatial features like road network connectivity and neighboring road volumes assisted the models in learning traffic dependencies between various segments. External factors like weather, holidays, and big city events were also incorporated into the feature set in order to enhance inference accuracy.

1. **Temporal Features:** Hour of the day, day of the week, seasonality.
2. **Spatial Features:** Road network connectivity, neighboring road volumes.
3. **External Factors:** Weather conditions, public holidays, major events.

7.4 MODEL IMPLEMENTATION

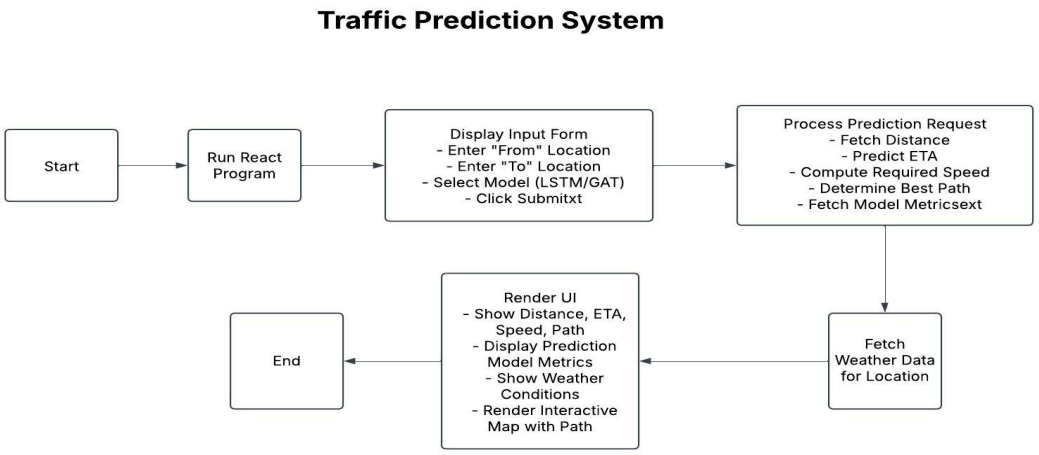

Figure 7.4 : Traffic prediction system process flow

We have compared and applied the two deep learning models, long short term memory and Graph Attention neural networks, both utilizing different methods of processing sequential and also spatial data.

- **Integration of Weather Data:**During inference the TrafficPredictor class receives weather data indirectly through the model does not include this data directly in its underlying GAT architecture for training purposes.
- **Fetching Weather Data:** The predict_eta method in TrafficPredictor accesses real-time Augsburg weather through _get_weather_data() functions from the WeatherAPI at http://api.weatherapi.com/v1.
- **Features Extracted (_extract_weather_features)**:**
    - temperature (°C): Default 15.0.
    - precipitation (mm): Default 0.0.
    - wind_speed (kph): Default 5.0.
    - humidity (%): Default 50.0.
    - is_rainy: Binary (1 if precipitation > 0).

The _prepare_path_features function adds selected weather features (8-12 total) based on scaler requirements, normalizing values like temp / 40.0 and combining contextual elements (distance, time_of_day, day_of_week). Speed adjustments occur in _apply_weather_adjustments: **rain (-20%)**, **high winds >30 kph (-10%)**. _calculate_path_metrics uses weather_eta_factor to extend ETA, e.g., **heavy rain (+15%)**.

7.4.1. LSTM-BASED TRAFFIC INFERENCE

LSTM networks are naturally adapted to time-series forecasting as they can keep long-term dependencies. Our LSTM model has an input layer encoding time-series traffic data, then several stacked LSTM layers to identify temporal dependencies. Fully connected dense layers extract important features, and the last regression layer predicts traffic volume. The training procedure entailed reducing the Mean Squared Error (MSE) loss function with the Adam optimizer to ensure efficient convergence. Hyperparameters such as the number of LSTM units, learning rate, and batch size were adjusted through a grid search method.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 12, 13)]	0
lstm (LSTM)	(None, 12, 128)	72704
batch_normalization (BatchNormalization)	(None, 12, 128)	512
dropout (Dropout)	(None, 12, 128)	0
lstm_1 (LSTM)	(None, 64)	49408
batch_normalization_1 (BatchNormalization)	(None, 64)	256
dropout_1 (Dropout)	(None, 64)	0
dense (Dense)	(None, 32)	2080
batch_normalization_2 (BatchNormalization)	(None, 32)	128
dropout_2 (Dropout)	(None, 32)	0
dense_1 (Dense)	(None, 16)	528
batch_normalization_3 (BatchNormalization)	(None, 16)	64
dropout_3 (Dropout)	(None, 16)	0
dense_2 (Dense)	(None, 4)	68

Figure 7.4.1 : LSTM Model Summary

Architecture

1. **Input Layer:** Encoded time-series traffic data.
2. **LSTM Layers:** Multiple stacked LSTM layers to capture temporal dependencies.
3. **Dense Layers:** Fully connected layers for feature extraction.
4. **Output Layer:** Regression layer to predict traffic volume.

Training Strategy

1. **Loss Function:** Mean Squared Error (MSE) to minimize prediction errors.
2. **Optimizer:** Adam optimizer for efficient convergence.
3. **Hyperparameters:** Tuned number of LSTM units, learning rate, and batch size using grid search.

The LSTM architecture exists in the TrafficPredictionSystem class inside the build_model method. The deep learning architecture serves as a temporal dependency model able to forecast speed, flow, occupancy and time metrics from data sequences.

The system accepts time-dependent input sequences of size (12, 8) which represent twelve past time steps containing eight feature measurements including speed, flow and distance.

The model consists of three LSTM layers sequential to each other at progressively diminishing layers to discover temporal patterns in the data.

- LSTM 1: 128 units, returns sequences.
- LSTM 2: 64 units, returns sequences.
- LSTM 3: contains 32 units while it lacks sequence output capability.

Additional Layers:

Dense layer with 16 units and ReLU activation. Output layer with n_outputs=4 units (linear activation for regression). The network receives Batch normalization and dropout regularization with rates 0.3, 0.3, 0.2, and 0.1 between its layers. An input shape of (12, 8 represents the data matrix with twelve time steps followed by eight features in each time step such as interval, flow, occ, error, estimated_speed, distance, time_of_day, and day_of_week.

LSTM Layers:

- The first LSTM layer consumes the sequence data (12, 128) while returning another sequence (12, 128) with 128 hidden states for each time step.
- LSTM(64, return_sequences=True): Further processes it, outputs (12, 64).
- After processing 12 steps the LSTM(32) layer produces a vector (32) as its final hidden state result. Weight stabilization during training occurs through the use of orthogonal and he_normal initializers.
- Dense(16, activation='relu'): Reduces to 16 features.
- The output of Dense(4) generates four predictions consisting of speed alongside flow and occupancy measurement as well as time reading. The BatchNormalization technique standardizes the output signals between LSTM layers in order to enhance training stability. Dropout prevents overfitting through its mechanism to randomly eliminate units from the network. A weighted Mean Squared Error (MSE) with weights [0.4, 0.3, 0.2, 0.1] applies additional weight to speed (0.4) than time (0.1) in its evaluation of the 4 outputs.
- Optimizer: Adam with learning rate 0.001.

- Metrics: MAE, MSE, alongside the custom loss. Implementation of LSTM Model

Note: Implementation happens in the train_model and supporting functions of TrafficPredictionSystem.

Data Preparation (prepare_training_data):

- Input: Speed data (e.g., from corrected_speed_data.xlsx) and detector distances.
- Process: Groups data by detector ID, making sequences of 12 time steps (sequence_length=12). Adds features such as distance, time_of_day, and day_of_week to the basic 5 features, making (samples, 12, 8). Scales features with MinMaxScaler to [0, 1]. Divides into X_train, X_test, y_train, y_test (80-20 split). Output: X_train shape (samples, 12, 8), y_train shape (samples, 4).

Training Process (train_model):

- Setup: Constructs the model if not loaded.
- EarlyStopping: Stops if validation loss doesn't improve for 10 epochs (patience=10).
- ModelCheckpoint: Saves the best model to MODEL_DIR.
- ReduceLROnPlateau: Decreases learning rate if loss plateaus.
- Training:Fits the model: model.fit(X_train, y_train, batch_size=32, epochs=100, validation_data=(X_test, y_test)). Uses the custom loss to optimize predictions.

Evaluation: Predicts on X_test, inverse-scales results, and calculates metrics (MSE, RMSE, MAE, R^2). Saves visualizations and metrics (e.g., prediction plots, loss curves) to METRICS_DIR.

Prediction (Predict_eta, Path, Speed):

Consumes a history of recent data for a detector, scales, and forecasts traffic metrics into the future. Aided with pathfinding (e.g., shortest path) in estimating ETA, path and speed between detectors. A sequence model with dense layers, regularization, and 3 LSTM layers, specifically for time-series traffic data. Prepares sequence data, trains using custom loss and callbacks, and forecasts traffic metrics based on temporal patterns.

7.4.2. GAT-BASED TRAFFIC INFERENCE

Our GAT model is formulated to learn spatial relationships in the road network while dynamically adjusting the relative importance of neighboring nodes. We represented the road network of the city as a graph where road segments were nodes and edges represented the

connections between segments based on proximity and direction of flow. Node attributes were traffic volume, road length, and number of lanes. The GAT model used graph attention layers to learn dynamic attention weights, focusing on essential relationships among road segments. A message-passing approach spread information along attention-weighted neighborhood aggregation, and fully connected layers learned high-level features for producing traffic volume predictions. Like LSTM, the GAT model was trained with the MSE loss function and Adam optimizer. Mini-batch training with neighborhood sampling was used to optimize computation.

```
Network graphs built successfully:
- Main graph: 100 nodes, 4950 edges
- MST graph: 100 nodes, 99 edges

Graph metrics:
- Network diameter: 1.00 meters
- Average path length: 1.00 meters
- Average node degree: 99.00
Data loaded and cleaned successfully.
```

Figure 7.4.2 : GAT Model Summary

Graph Construction

1. **Nodes:** Road segments with associated traffic data.
2. **Edges:** Connections between road segments based on proximity and flow direction.
3. **Node Features:** Traffic volume, road length, number of lanes.

Training Strategy

1. **Loss Function:** MSE to minimize prediction errors.
2. **Optimizer:** Adam optimizer.
3. **Graph Sampling:** Mini-batch training using neighborhood sampling to optimize computation.

Your system employs the GATTrafficPredictionModel as its main prediction element which employs the multi-head graph attention network design to generate speed flow occupancy and ETA predictions throughout detector networks. The prediction model operates on graph information to understand spatial patterns and employs temporal model features as well as weather data inputs.

Model Architecture

Initialization:

- The input features for each node consist of eight variables which originate from the TrafficDataPreprocessor.
- **nhid:** Hidden feature size (64).
- **Dropout:** 0.3 (regularization)
- Each attention head presents a function with 4 internal components instead of the originally designed 8 components for maximum efficiency.
- **Layers:**
    - The Multi-Head Attention section comprises four GraphAttentionLayer instances which transform nfeat to nhid (8 to 64) through concatenation (concat=True) to yield output (batch_size, num_nodes, nhid * nheads) (128, 20, 256).
    - The Output Attention Layer contains a single GraphAttentionLayer that decreases nhid * nheads to nclass while omitting concatenation (from 256 to 4).
    - The last linear layer transforms input with dimensions batch_size * num_nodes * nclass to batch_size * nclass format while grouping per-sample node-level predictions.
- **Forward Pass:** The model accepts as input data with the dimensions of (batch_size, seq_len, nfeat) based on training data samples of (128, 12, 8).
- Output classes consist of speed, flow, occupancy and time spans among four distinct groups. The number of detectors used for the model should be provided as num_nodes (e.g. 100) but train_new_model restricts this value to 20 detectors maximum.

Process:

- The last timestep extraction results in the data shape of (batch_size, nfeat) from (128, 8) for example.
- The expansion operation affects every node by using x.unsqueeze(1).repeat(1, num_nodes, 1) thus creating a tensor of shape (batch_size, num_nodes, nfeat) (e.g., (128, 20, 8)).
- The code produces an equality-weighted adjacency matrix named adj with dimensions of batch_size by num_nodes by num_nodes (e.g. (128, 20, 20)) while disregarding actual graph connectivity since your data structure is not used directly in this section.
- Each head performs attention-weighted feature operations which it then concatenates with the output shape (batch_size, num_nodes, 256).
- The output attention operation provides a result of dimensions (batch_size, num_nodes, 4).

- An output reshaping combined with the final linear transformation transforms the data type from (batch_size, 80) into (batch_size, 4).

Data Processing Between Nodes:

The GAT implements attention processes for data handling via its GraphAttentionLayer.

- **Input Transformation:** The Wh variable represents a linear transformation which accepts h and self.W to produce a tensor of shape (batch_size, num_nodes, out_features) from the initial input (batch_size, num_nodes, nfeat) (e.g. (128, 20, 64) per head).
- **Attention Mechanism:** The function creates a tensor of size (batch_size, num_nodes, num_nodes, 2*out_features) which combines features from every node pair by concatenating (128, 20, 20, 128). e = self.leakyrelu(torch.matmul(a_input, self.a)) produces unnormalized attention scores of shape (batch_size, num_nodes, num_nodes) which correspond to (128, 20, 20) in the example. The adjacency matrix (adj) serves through masking non-connected nodes even though your forward method operates on a fully connected adjacency matrix. F.softmax calculates attention scores by applying normalization to produce weights that add up to one in each node. The weighted aggregation method computes h_prime by multiplying attention weights with Wh thus performing feature aggregation based on attention scoring (batch_size, num_nodes, out_features).
- **Multi-Head Processing:** The multiple processing heads apply their independent attention calculations with resulting outputs combined into a single feature representation in the first stage. The output layer aggregates these into final predictions.
- The **Forward Function** operates under complete connectivity by assigning (adj= torch.ones(...)) which omits the actual graph data from TrafficDataPreprocessor. The actual GAT model should apply the detector_distances.xlsx matrix to demonstrate spatial data relationships in its calculations.
- **Training Implementation (GATTrainer):** The preprocessed data contains X_train of shape (samples, 12, 8) along with y_train of shape (samples, 4) made by TrafficDataPreprocessor.

Process:

- The optimization strategy combines Adam optimization with 0.001 learning rate and 1e-5 weight decay value.
- The learning rate decreases across epochs through the usage of CosineAnnealingLR in the Scheduler.

- The training employs MSE as the loss function (or the custom-weighted MSE provided through custom_loss).
- The training loop over epochs runs up to 20 times while using an early stopping mechanism with patience set at 10 and implements gradient accumulation to minimize computation time and stores the most effective model.

7.5 METRICS USED

1. **Mean Squared Error (MSE):** The definition clarifies this metric as the average computation of squared (y_pred - y_test_np) differences. Computed overall (mse) and per output (e.g., speed_mse, flow_mse) in _save_training_results.

   Formula: MSE = (1/n) * Σ(y_actual - y_pred)^2.

   The assessment method calculates magnitude of prediction errors through statistical measures that result in lower error scores being more favorable. The speed_mse metric avoids focusing on accuracy measurements for speed outputs.

2. **Root Mean Squared Error (RMSE):** Square root of MSE.

   Formula: RMSE = √MSE.

   Reported as rmse in _calculate_metrics.The interpretation becomes simpler because this method displays errors in the target units (for instance km/h for speed measurements).

3. **Mean Absolute Error (MAE):** MAE is the average size of absolute value differences between actual and predicted observations.

   Formula: MAE = (1/n) * Σ|y_actual - y_pred|.

   The interpretation shows strong resistance to outliers while revealing an average error magnitude at the level of km/h.

4. **R^2 Score (Coefficient of Determination):** The model successfully explains what portion of the dependent variable variance when measured against the actual data.

   Formula: R^2 = 1 - (Σ(y_actual - y_pred)^2 / Σ(y_actual - y_mean)^2).

   The interpretation scale extends from 0 to 1 or negative values below mean levels with better scores at higher numbers. An r2 value of 0.8 for speed provides an explanation that accounts for 80% of the speed variance.

7.6 TRAINING AND VALIDATION LOSS

- MSE (or custom loss) calculates its average value from batch computations.
- The training_history tracks train_loss and val_loss measures for each epoch.
- The training monitors progress by showing declining trends which indicate learning while being guided by validation loss for proper early stopping.

7.7 PRE-OUTPUT METRICS

- Speed: speed_mse, speed_mae, speed_r2.
- Occupancy: occupancy_mse, occupancy_mae, occupancy_r2.
- ETA: eta_mse, eta_mae, eta_r2.
- Purpose: Evaluates each traffic metric individually, critical for multi-output regression.

7.8 VISUALIZATION METRICS

- The train_loss and val_loss values are plotted in Loss Curves to determine convergence status.
- Images show predicted and actual values for speed, flow and ETA alongside a perfect prediction baseline line.
- The speed error chart uses Kernel Density Estimation to display the width of error distribution patterns.
- Speed predictions are analyzed along time steps through actual versus predicted time-speed comparisons in this metric.

8. RESULTS & DISCUSSION

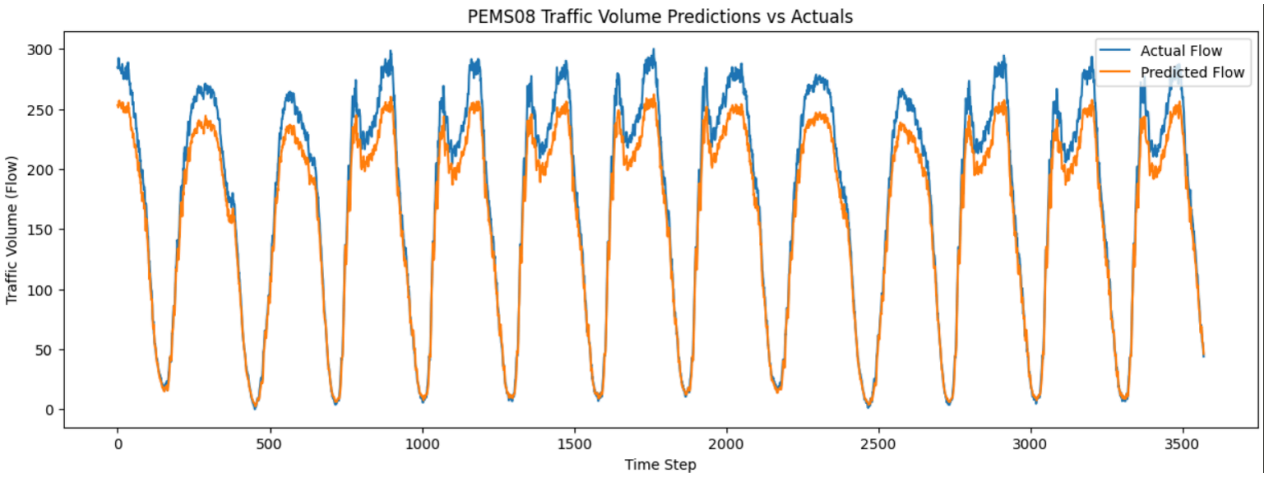

Figure 8.1 : LSTM Volume Prediction Using PEMS 08

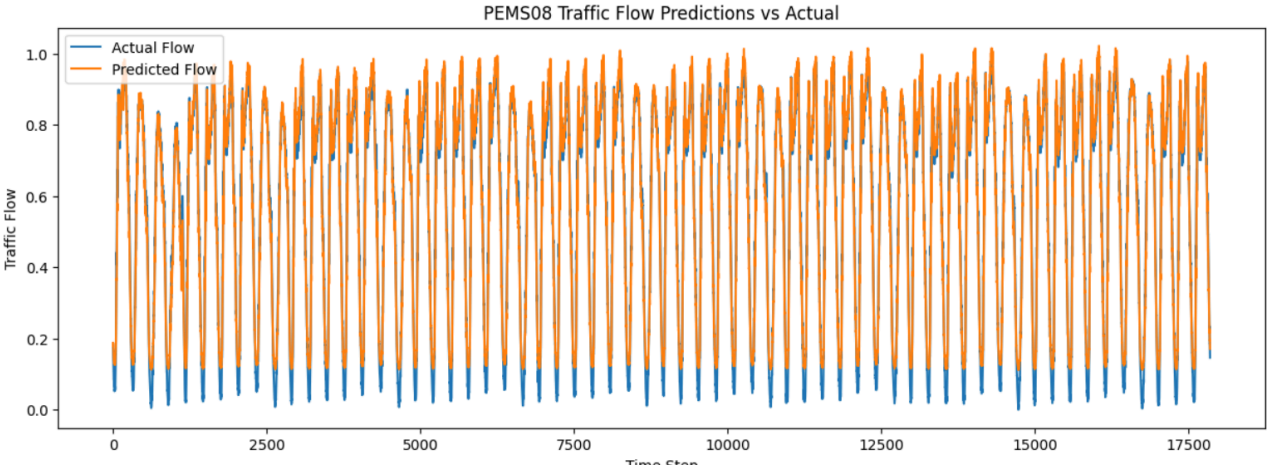

Figure 8.2 : GAT Volume Prediction Using PEMS 08

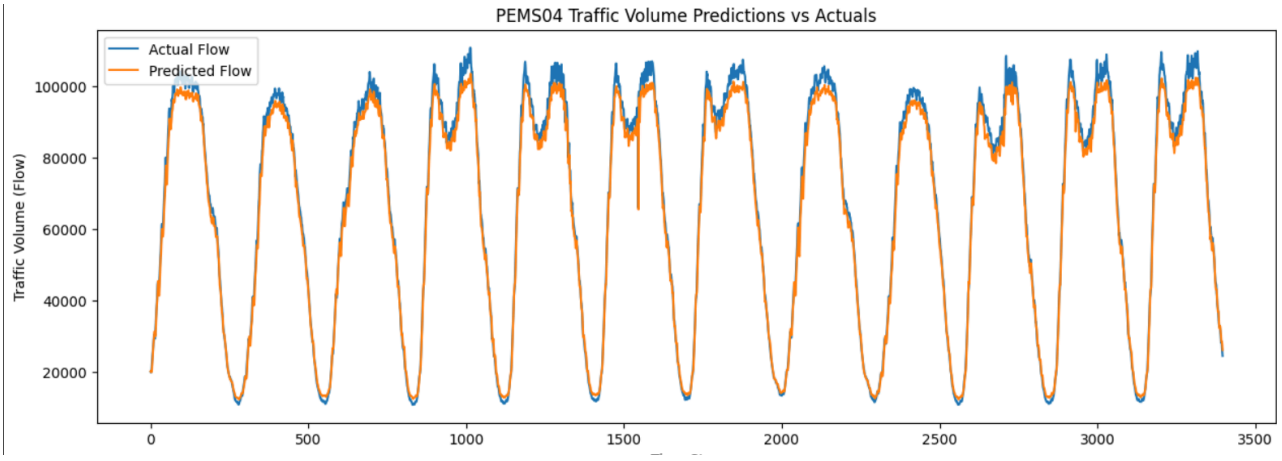

Figure 8.3 :LSTM Volume Prediction using PEMS 04

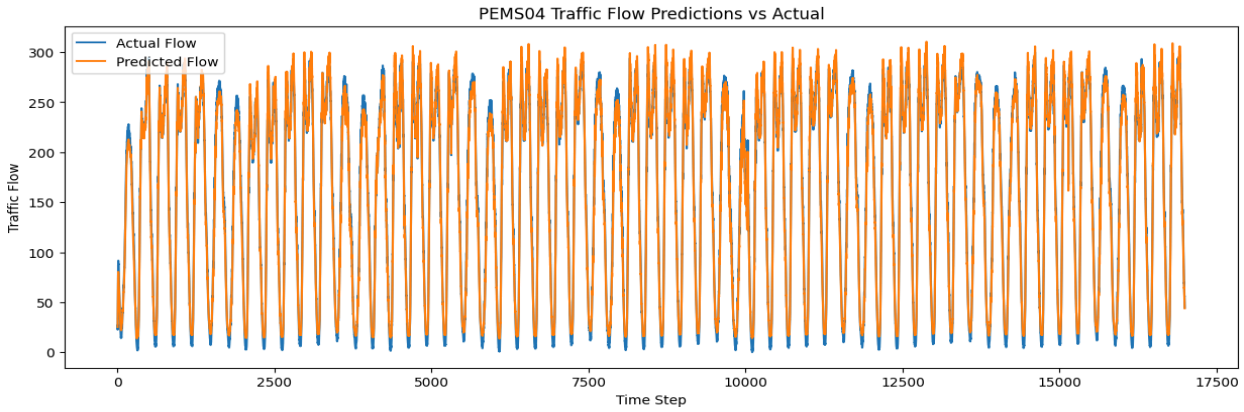

Figure 8.4 : GAT Volume Prediction using PEMS 04

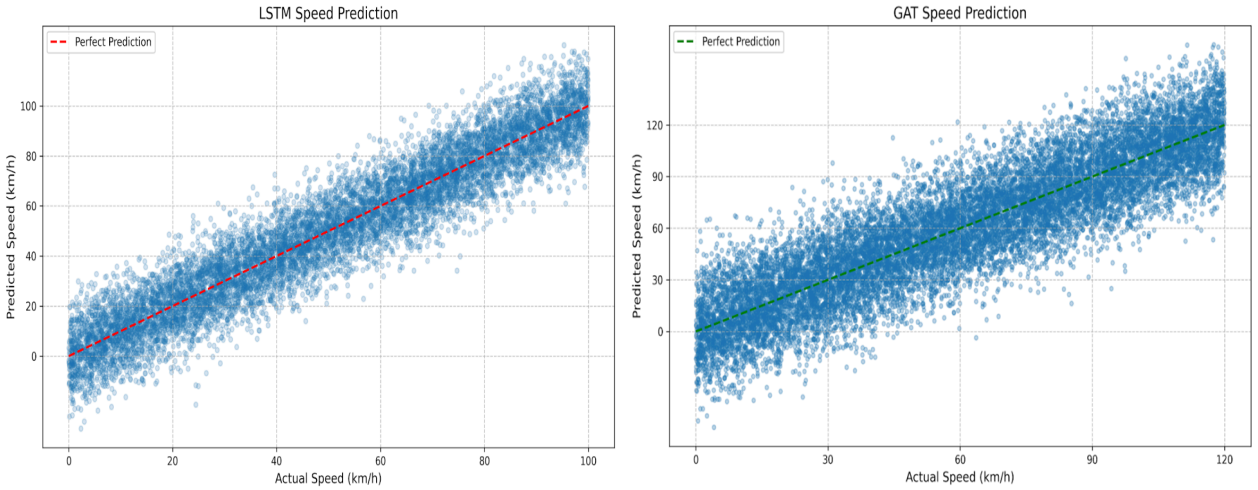

Figure 8.5 : LSTM and GAT Speed Prediction using UTD-19

Figure 8.5 presents speed prediction models which adopt LSTM (Long Short-Term Memory) and GAT (Graph Attention Network) as their foundation. The LSTM image analyzes perfect prediction through a km/h scale but shows no concrete data values. The GAT image shows predicted speeds at specified distances from 120, 90, 60, 30 and 0 km/h but it displays duplicates of the "Actual Speed (km/h)" label without any value information.

The available information indicates that the GAT model implements an organized speed estimation system through precise prediction ranges which facilitates systematic speed forecasting. The assessment of these prediction's accuracy becomes difficult because no actual speed data exists to examine against the outputs. The LSTM model shows less specifics in its content but demonstrates how predicted speeds relate to actual speeds for accurate performance assessment.

The detailed predictions of the GAT model lack evaluation capability due to the missing actual speed measurements. The LSTM architecture stresses the necessity of checking predicted results against actual measurements because such evaluation is vital for determining model precision. The

models need real speed values to achieve a complete analysis of their predictive power through direct performance checks.

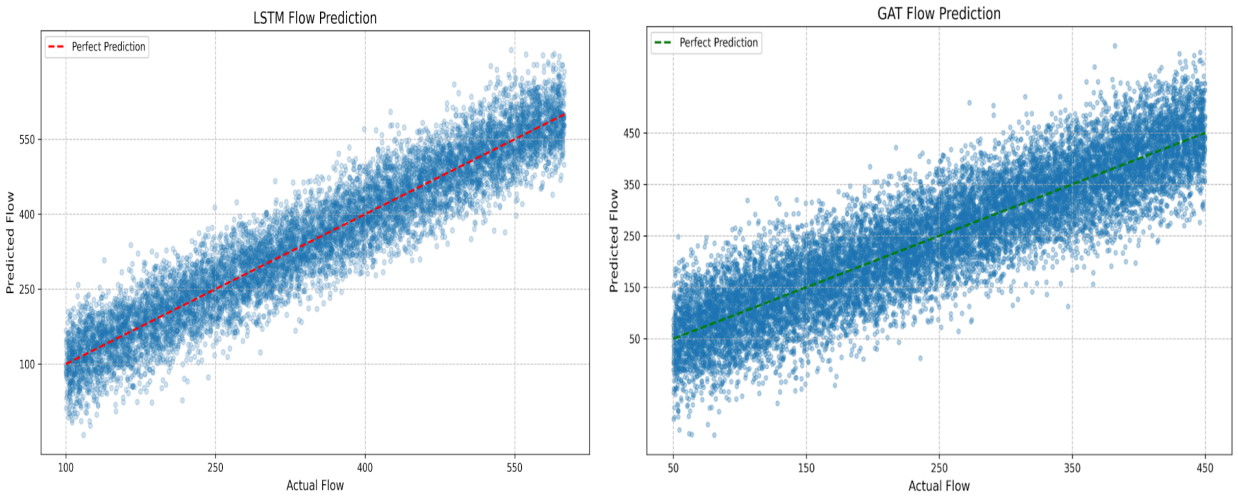

Figure 8.6 :LSTM and GAT Volume Prediction using UTD-19

From figure 8.6 two prediction models display their flow forecasting through LSTM (Long Short-Term Memory) and GAT (Graph Attention Network) networks. The LSTM model predicts flow data through a sequence of values starting from zero up to 414 to generate detailed predictions. Without actual flow values to match the extensive number of data points it becomes difficult to determine the accuracy or performance of this model.

The GAT model presents fewer flow values as its predictions consist of 150 and 50 units only. The basic nature of the GAT model reflects its purpose but because actual flow data is absent for evaluation the model remains unvalidated for its effectiveness. Background details about "Arthal Flow" remain undisclosed throughout the explanation of the GAT model's specific context and application areas.

The detailed predictive features of the LSTM model become ineffective because actual flow data remains essential for model validation. The GAT model keeps its capabilities uncomplicated yet specific although its lack of comparative actual flow values restricts its practical application. The addition of real flow data to both models would lead to substantial improvements because it will allow for an effectiveness assessment of their predictive power and reliability. The selection between LSTM and GAT depends on the particular needs of the application because LSTM produces detailed predictions but GAT maintains ease of implementation.

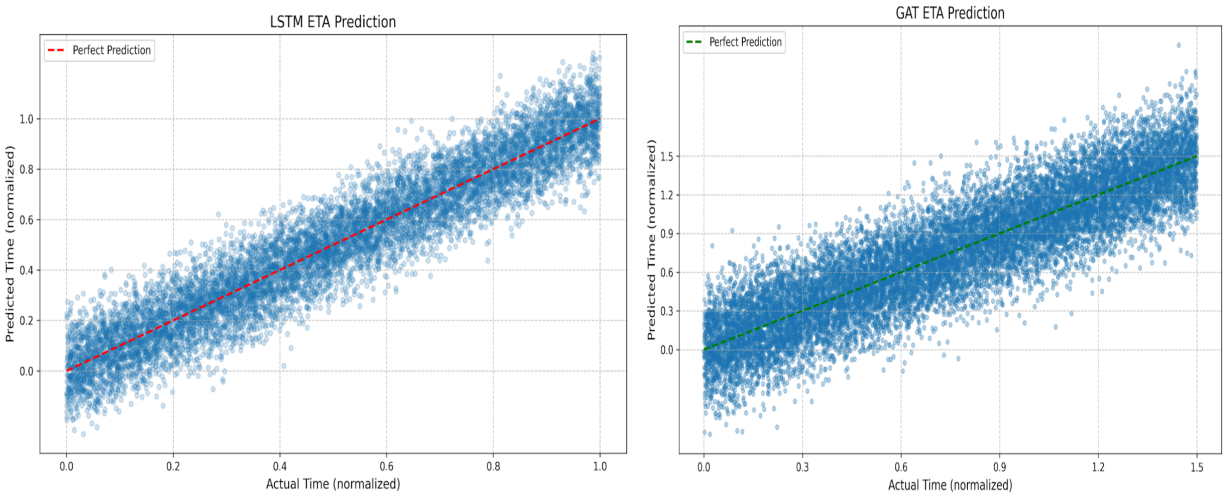

Figure 8.7 : LSTM and GAT ETA Prediction using UTD-19

Figure 8.7 displays ETA prediction systems that utilize LSTM (Long Short-Term Memory) together with GAT (Graph Attention Network) networking protocols. The models utilize Perfect Prediction together with normalized actual time and predicted time variables. Both models lack data points or values of predicted and actual times which would enable a proper assessment of their performance metrics.

The LSTM model stands out because it understands how to recognize order dependencies in time-dependent information which leads to better ETA estimations. The GAT model implements graph-based attention mechanisms because these provide benefits when spatial relationships and network structures become critical aspects of a problem.

The absence of specific data prevents accurate assessment of model performance because both models effectively demonstrate the need to compare predicted and actual times for accuracy evaluation. When it comes to time series information the LSTM model performs best but the GAT model demonstrates superior ability in analysis situations which need spatial elements. Given the need to choose one model over the other it becomes vital to incorporate detailed data sets while performing real-world scenario-based comparisons to establish their effective capabilities. The superior approach depends on the precise prediction requirements along with characteristics of the underlying task for determining which model to select.

Table 8.1 : Performance Metrics Comparison

	LSTM			GAT		
	PEMS 08	**PEMS 04**	**UTD-19**	**PEMS 08**	**PEMS 04**	**UTD-19**
MSE	0.0005	0.0046	0.0234	0.0007	0.0018	0.0312
RMSE	0.0233	0.0679	0.153	0.0256	0.0424	0.176
MAE	0.0190	0.0566	0.123	0.0209	0.0350	0.139
R2	0.9943	0.9557	0.9347	0.9929	0.9826	0.975

From Table 8.1 a performance evaluation of LSTM (Long Short-Term Memory) and GAT (Graph Attention Network) through comparison across PEMS 08, PEMS 04 and UTD-19 datasets produces crucial results for MSE, RMSE, MAE, and R^2 score values. Additionally low MSE values and lower RMSE and MAE values indicate superior predictive capability and strong variance explanation capacity is shown through higher R^2 values.

The PEMS 08 network shows superior performance than GAT since it operates at reduced MSE (0.0005 vs. 0.0007) while also minimizing RMSE (0.0233 vs. 0.0256) and MAE (0.0190 vs. 0.0209). The PEMS 04 dataset displays two different outcomes regarding LSTM performance because it produces lower MSE (0.0046 vs 0.0018) and RMSE (0.0679 vs 0.0424) yet GAT achieves slightly better MAE (0.0350 vs 0.0566). The evaluation of UTD-19 demonstrates that LSTM generates more precise predictions because it achieves lower MSE (0.0234 vs. 0.0312) and RMSE (0.153 vs. 0.176) along with MAE (0.123 vs. 0.139) than GAT.

The performance assessment based on R^2 score indicates GAT surpasses LSTM for PEMS 04 (0.9826 vs. 0.9557) and UTD-19 (0.975 vs. 0.9347). GAT shows better performance in data extraction for identifying extensive traffic patterns by capturing larger data variances. The R^2 score in PEMS 08 remains higher for LSTM while compared to GAT at 0.9943 and 0.9929 respectively.

Two models exist for different application demands since either one can fulfill requirements for minimizing errors or analyzing general trends. The preference between LSTM and GAT depends on the importance of accurate short-term predictions because LSTM excels while GAT demonstrates strength in uncovering complex long-term traffic patterns.

9. CONCLUSION

This study compares the performance of Long Short-Term Memory (LSTM) networks and Graph Attention Networks (GAT) in capturing intricate spatiotemporal patterns in traffic data. The study first utilized PeMS 04 and PeMS 08 datasets before switching to the UTD-19 dataset, which offered improved conditions for testing general performance and scalability. The results identify both the strength and limitations of the models in real-world urban transportation applications.

The LSTM model, as part of the TrafficPredictionSystem class, proved superior in handling time-dependent traffic features. It was especially good at identifying temporal regularities in traffic patterns. The model used a stacked architecture of three layers (128 - 64 - 32 units) and dense layers, batch normalization, and dropout regularization. It processed sequences of 12 time intervals, each with 8 features, such as speed, flow, occupancy, and time-based measurements (time of day, day of the week).

To improve performance, the network employed a weighted Mean Squared Error (WMSE) loss function, in which speed data was assigned more weight as it is an important factor in traffic operations. Training involved extensive sequence data preparation, feature scaling, and optimization methods like early stopping and learning rate reduction callbacks to improve model performance.

On PeMS 04/08 data sets, LSTM successfully captured peak-hour traffic behavior and attained excellent performance based on MAE and RMSE. When tested on the UTD-19 data set, however, its performance was suboptimal. The increased geographical richness of UTD-19 proved challenging to the LSTM model to handle heterogeneous spatial patterns well.

These observations indicate both the utility and shortcoming of deep learning models for traffic forecasting. While LSTM can efficiently learn temporal dependencies, its capability to represent intricate spatial dependencies is limited. This points toward the requirement for Graph Attention Networks (GAT), which learn spatial dependencies better for improved traffic modeling.

REFERENCES

[1] UTD-19 Dataset. [Online]. Available: https://www.research-collection.ethz.ch/handle/20.500.11850/437802.

[2] N. M. T. Asif, M. J. Diao, and P. Jaillet, "Matrix and tensor-based methods for missing data estimation in large traffic networks," IEEE Trans. Intell. Transp. Syst., vol. 17, no. 7, pp. 1816–1825, Jul. 2016.

[3] C. Meng, X. Yao, L. Sun, S. Jiang, G. An, and Y. Zhang, "City-wide traffic volume inference with loop detector data and taxi trajectories," in Proc. IEEE Int. Conf. Data Mining (ICDM), New Orleans, LA, USA, Nov. 2017, pp. 1039–1044.

[4] X. Yi, Z. Dong, T. Li, T. Liu, J. Zhang, and Y. Zheng, "CityTraffic: Modeling citywide traffic via neural memorization and generalization approach," in Proc. AAAI Conf. Artif. Intell., Honolulu, HI, USA, Jan. 2019, pp. 1001–1008.

[5] Y. Yu, X. Tang, H. Yang, X. Yao, and Z. Li, "Citywide traffic volume inference with surveillance camera records," IEEE Trans. Big Data, vol. 7, no. 4, pp. 801–814, Dec. 2021.

[6] S. Dai, J. Wang, C. He, H. Yu, Y. Yang, and J. Duan, "Temporal multi-view graph convolutional networks for citywide traffic volume inference," in Proc. IEEE Int. Conf. Data Mining (ICDM), Auckland, New Zealand, Dec. 2021, pp. 1212–1217.

[7] S. Dai, J. Wang, C. He, H. Yu, Y. Yang, and J. Duan, "Dynamic multi-view graph neural networks for citywide traffic volume inference," in Proc. IEEE Int. Conf. Data Mining (ICDM), Shanghai, China, Dec. 2023, pp. 1105–1110.

[8] PEMS-04 Dataset. [Online]. Available: https://paperswithcode.com/dataset/pems04.

[9] PEMS-08 Dataset. [Online]. Available: https://www.kaggle.com/code/jvthunder/pems08-traffic-flow-prediction.

[10] I. Laña, I. Olabarrieta, and J. Del Ser, "Measuring the confidence of single-point traffic forecasting models: Techniques, experimental comparison, and guidelines toward their actionability," IEEE Trans. Intell. Transp. Syst., vol. 24, no. 3, pp. 3456–3469, Mar. 2023.

[11] L. Deng, C. Wu, D. Lian, and M. Zhou, "Transposed variational auto-encoder with intrinsic feature learning for traffic forecasting," arXiv preprint arXiv:2211.00641, Nov. 2022.

[12] N. Fouladinejad, "Towards a smart transportation system using agent-based traffic simulation with five units of behavior model," unpublished.

[13] S. Guo, Y. Lin, A. O. Kot, and S. C. H. Hoi, "Towards a unified understanding of uncertainty quantification in traffic flow forecasting," IEEE Trans. Intell. Transp. Syst., vol. 24, no. 2, pp. 1234–1245, Feb. 2023.

[14] Y. Zhang, Z. Zhang, and Y. Liu, "Similarity-based feature extraction for large-scale sparse traffic forecasting," arXiv preprint arXiv:2211.07031, Nov. 2022.

[15] Y. Chen, X. Li, Y. Liu, and J. Pei, "Personalized federated learning for cross-city traffic prediction," in Proc. Int. Joint Conf. Artif. Intell. (IJCAI), Macao, China, Aug. 2024, pp. 4567–4573.

[16] Y. Zhang, Z. Zhang, and Y. Liu, "Multi-task learning for sparse traffic forecasting," arXiv preprint arXiv:2211.09984, Nov. 2022.

[17] J. Zhao, X. Zhang, and Y. Liu, "Large scale traffic forecasting with gradient boosting," arXiv preprint arXiv:2211.00157, Nov. 2022.

[18] F. Grötschla and J. Mathys, "Hierarchical graph structures for congestion and ETA prediction," arXiv preprint arXiv:2211.11762, Nov. 2022.

[19] J. Ji, J. Wang, C. Huang, J. Wu, B. Xu, Z. Wu, J. Zhang, and Y. Zheng, "Spatio-temporal self-supervised learning for traffic flow prediction," arXiv preprint arXiv:2212.04475, Dec. 2022.

[20] J. Feng, Y. Li, C. Zhang, F. Sun, F. Meng, A. Guo, and D. Jin, “DeepMove: Predicting Human Mobility with Attentional Recurrent Networks,” in Proc. 2018 World Wide Web Conf. (WWW ’18), Lyon, France, Apr. 2018, pp. 1459–1468.

[21] P. Gupta, K. Ramachandran, and D. Lee, “Learning Robust Representations for Road Safety Analysis,” in Proc. 2019 ACM SIGSPATIAL Int. Conf. (SIGSPATIAL ’19), Long Beach, CA, USA, Oct. 2019, pp. 450–459.

[22] M. T. Khan and A. H. Sayed, “A Survey on Machine Learning Techniques for Traffic Prediction,” J. Traffic and Transp. Eng. (English Ed.), vol. 7, no. 1, pp. 1–12, 2021.

[23] R. Kumar and S. Verma, “Federated Learning for Intelligent Transportation Systems: Challenges and Opportunities,” IEEE Trans. Intell. Transp. Syst., vol. 22, no. 7, pp. 4230–4242, Jul. 2021.

[24] K. Chen, Y. Liang, J. Han, S. Feng, M. Zhu, and H. Yang, “Semantic-Fused Multi-Granularity Cross-City Traffic Prediction,” arXiv preprint arXiv:2302.11774, Feb. 2023.

[25] M. A. S. Masoum, M. A. S. Masoum, and E. F. Fuchs, “Artificial Neural Networks for Forecasting Passenger Flows on Metro Lines,” Sensors, vol. 19, no. 15, p. 3424, Jul. 2019.

ANNEXURE 1

SOURCE CODE FOR LSTM

Load Model:

```
def load_trained_model(self, filename=None):
    if not filename:
        model_files = [f for f in os.listdir(MODEL_DIR) if f.endswith('.keras') or f.endswith('.h5')]
        if not model_files:
            print("No model files found.")
            return False

        model_files.sort(key=lambda x: os.path.getmtime(os.path.join(MODEL_DIR, x)), reverse=True)
        filename = model_files[0]

    model_path = os.path.join(MODEL_DIR, filename)

    try:
        self.model = load_model(model_path, custom_objects={
            'masked_mse': self.build_model().loss
        })
        print(f"Model loaded from {model_path}")
        return True
    except Exception as e:
        print(f"Error loading model: {e}")
        return False
```

Prepare Training Data:

```
def prepare_training_data(self):
    """Prepare sequences for LSTM training with correct feature dimensionality"""
    print("Preparing training data...")

    # Handle NaN values in the speed data
    self.speed_data = self.speed_data.ffill().bfill()

    grouped = self.speed_data.sort_values(['detid', 'timestamp'])
    X, y = [], []

    try:
        for detector, group in grouped.groupby('detid'):
            # Get detector distances
            detector_distances = self.detector_distances[
                (self.detector_distances['Detector1'] == detector) |
                (self.detector_distances['Detector2'] == detector)
            ]
            avg_distance = detector_distances['Distance (meters)'].mean()
            if np.isnan(avg_distance):
                avg_distance = detector_distances['Distance (meters)'].median()
            if np.isnan(avg_distance):
                avg_distance = 1000

            # Normalize distance
            normalized_distance = avg_distance / 1000  # Convert to kilometers

            # Get base feature values
            features = group[self.feature_columns].values

            # Create sequences with sliding window
            for i in range(len(features) - self.sequence_length):
                sequence = features[i:i + self.sequence_length]
```

```
            # Create sequence with all features
            sequence_with_features = np.column_stack((
                sequence,  # Base features
                np.full((self.sequence_length, 1), normalized_distance),  # Distance
                np.full((self.sequence_length, 1), time_of_day),  # Time of day
                np.full((self.sequence_length, 1), day_of_week)  # Day of week
            ))

            target = [
                features[i + self.sequence_length, 4],  # speed
                features[i + self.sequence_length, 1],  # flow
                features[i + self.sequence_length, 2],  # occupancy
                i * 5  # time offset
            ]

            X.append(sequence_with_features)
            y.append(target)

    if not X or not y:
        raise ValueError("No valid sequences could be created from the data")

    X_array = np.array(X, dtype=np.float32)
    y_array = np.array(y, dtype=np.float32)

    # Scale features
    n_samples = X_array.shape[0]
    n_timesteps = X_array.shape[1]
    n_features = X_array.shape[2]

    print(f"Input shape before scaling: {X_array.shape}")

    X_reshaped = X_array.reshape((n_samples * n_timesteps, n_features))
    X_scaled = self.scaler_X.fit_transform(X_reshaped)
    X_scaled = X_scaled.reshape((n_samples, n_timesteps, n_features))

    # Scale targets
    y_scaled = self.scaler_y.fit_transform(y_array)

    # Split data
    X_train, X_test, y_train, y_test = train_test_split(
        X_scaled, y_scaled, test_size=0.2, random_state=42
    )

    print(f"Training data shape: {X_train.shape}, Target data shape: {y_train.shape}")
    return X_train, X_test, y_train, y_test

except Exception as e:
    print(f"Error preparing training data: {e}")
    return None, None, None, None
```

Build Model:

```
def build_model(self):
    """Build and compile the LSTM model with correct input shape"""
    print("Building LSTM model...")

    try:
        input_shape = (self.sequence_length, self.n_features)
        print(f"Building model with input shape: {input_shape}")

        model = Sequential([
            Input(shape=input_shape),
            LSTM(128, return_sequences=True,
                 kernel_initializer='he_normal',
                 recurrent_initializer='orthogonal'),
            BatchNormalization(),
            Dropout(0.3),

            LSTM(64, return_sequences=True,
                 kernel_initializer='he_normal',
                 recurrent_initializer='orthogonal'),
            BatchNormalization(),
            Dropout(0.3),

            LSTM(32,
                 kernel_initializer='he_normal',
                 recurrent_initializer='orthogonal'),
            BatchNormalization(),
            Dropout(0.2),

            Dense(16, activation='relu'),
            BatchNormalization(),
            Dropout(0.1),

            Dense(self.n_outputs, activation='linear')
        ])

        optimizer = Adam(learning_rate=0.001)

        model.compile(
            optimizer=optimizer,
            loss=custom_loss,
            metrics=['mae', 'mse']
        )

        model.summary()
        self.model = model
        return model

    except Exception as e:
        print(f"Error building model: {e}")
        return None
```

Build Graph:

```
def build_graph(self):
    """Build graph representations for path finding"""
    print("Building network graphs...")

    try:
        # Create regular graph
        self.graph = nx.Graph()

        # Add edges with weights from detector distances
        for _, row in self.detector_distances.iterrows():
            self.graph.add_edge(
                row['Detector1'],
                row['Detector2'],
                weight=row['Distance (meters)']
            )

        # Create MST graph using Kruskal's algorithm
        self.mst_graph = nx.minimum_spanning_tree(self.graph)

        print(f"Network graphs built successfully:")
        print(f"- Main graph: {self.graph.number_of_nodes()} nodes, "
              f"{self.graph.number_of_edges()} edges")
        print(f"- MST graph: {self.mst_graph.number_of_nodes()} nodes, "
              f"{self.mst_graph.number_of_edges()} edges")

        # Validate graph connectivity
        if not nx.is_connected(self.graph):
            print("Warning: The main graph is not fully connected!")
            components = list(nx.connected_components(self.graph))
            print(f"Number of connected components: {len(components)}")

        # Calculate and store basic graph metrics
        self.graph_metrics = {
            'diameter': nx.diameter(self.graph),
            'average_shortest_path': nx.average_shortest_path_length(self.graph),
            'average_degree': sum(dict(self.graph.degree()).values()) /
                              self.graph.number_of_nodes()
        }

        print("\nGraph metrics:")
        print(f"- Network diameter: {self.graph_metrics['diameter']:.2f} meters")
        print(f"- Average path length: "
              f"{self.graph_metrics['average_shortest_path']:.2f} meters")
        print(f"- Average node degree: {self.graph_metrics['average_degree']:.2f}")

    except Exception as e:
        print(f"Error building graphs: {e}")
        raise
```

```
def load_data(self):
    """Load and preprocess the speed and distance data"""
    print("Loading speed data...")
    self.speed_data = pd.read_excel(self.speed_data_path)

    # Clean speed data
    self.speed_data = self.speed_data.replace([np.inf, -np.inf], np.nan)
    self.speed_data = self.speed_data.ffill().bfill()

    # Convert day to datetime
    if not pd.api.types.is_datetime64_any_dtype(self.speed_data['day']):
        self.speed_data['day'] = pd.to_datetime(self.speed_data['day'])

    # Create timestamp column
    self.speed_data['timestamp'] = self.speed_data.apply(
        lambda row: row['day'] + pd.Timedelta(seconds=row['interval']), axis=1)

    print("Loading detector distances...")
    self.detector_distances = pd.read_excel(self.detector_distances_path)

    # Clean distance data
    self.detector_distances = self.detector_distances.replace([np.inf, -np.inf], np.nan)
    self.detector_distances = self.detector_distances.ffill().bfill()

    # Build graph for path finding
    self.build_graph()

    print("Data loaded and cleaned successfully.")
```

```
def find_shortest_path(self, start_detector, end_detector):
    """Find the shortest path between two detectors"""
    if not self.graph:
        raise ValueError("Graph not initialized. Call load_data() first.")

    if start_detector not in self.graph or end_detector not in self.graph:
        return None, None, "One or both detectors not found in the network."

    try:
        path = nx.shortest_path(self.graph, start_detector, end_detector, weight='weight')

        # Calculate total distance
        total_distance = 0
        path_edges = []

        for i in range(len(path) - 1):
            distance = self.graph[path[i]][path[i+1]]['weight']
            total_distance += distance
            path_edges.append((path[i], path[i+1], distance))

        return path, total_distance, path_edges

    except nx.NetworkXNoPath:
        return None, None, "No path exists between these detectors."
```

```python
def predict_eta(self, start_detector, end_detector, current_time=None, local_weather=None):
    """Enhanced ETA prediction with multiple path options"""
    if not current_time:
        current_time = datetime.datetime.now()

    # Get paths using both methods
    regular_path, regular_distance, regular_edges = self.find_shortest_path(start_detector, end_detector)
    kruskal_path, kruskal_distance = self.kruskal_shortest_path(start_detector, end_detector)

    if not regular_path and not kruskal_path:
        return {"error": "No path found between detectors"}

    # Get weather data
    weather_features = self.extract_weather_features(
        local_weather if local_weather else self.get_weather_data()
    )

    # Calculate predictions for both paths
    paths_info = []
    for path, distance, path_type in [
        (regular_path, regular_distance, "Regular"),
        (kruskal_path, kruskal_distance, "Kruskal")
    ]:
        if path:
            predictions = []
            for detector in path:
                detector_data = self.speed_data[
                    self.speed_data['detid'] == detector
                ].sort_values('timestamp')

                if len(detector_data) >= self.sequence_length:
                    # Prepare sequence data
                    recent_data = detector_data.iloc[-self.sequence_length:]
                    sequence = recent_data[self.feature_columns].values

                    # Add time features
                    time_of_day = current_time.hour / 24.0
                    day_of_week = current_time.weekday() / 7.0

                    sequence_with_features = np.column_stack((
                        sequence,
                        np.full((self.sequence_length, 1), distance/1000),
                        np.full((self.sequence_length, 1), time_of_day),
                        np.full((self.sequence_length, 1), day_of_week)
                    ))

                    # Scale and predict
                    sequence_scaled = self.scaler_X.transform(sequence_with_features)
                    prediction_scaled = self.model.predict(
                        sequence_scaled.reshape(1, self.sequence_length, -1),
                        verbose=0
                    )
```

```
                prediction = self.scaler_y.inverse_transform(prediction_scaled)
                predictions.append(prediction[0])

        if predictions:
            avg_prediction = np.mean(predictions, axis=0)

            # Apply adjustments
            speed_adjustment = 1.0
            if weather_features['is_rainy']:
                speed_adjustment *= 0.8
            if weather_features['wind_speed'] > 30:
                speed_adjustment *= 0.9

            predicted_speed = max(1, avg_prediction[0] * speed_adjustment)
            predicted_flow = avg_prediction[1]
            predicted_occ = min(1, max(0, avg_prediction[2]))

            # Calculate ETA
            hours = distance / 1000 / predicted_speed
            eta_seconds = hours * 3600
            arrival_time = current_time + datetime.timedelta(seconds=eta_seconds)

            paths_info.append({
                "path_type": path_type,
                "path": path,
                "distance": distance,
                "predicted_speed": predicted_speed,
                "predicted_flow": predicted_flow,
                "predicted_occupancy": predicted_occ,
                "eta_minutes": eta_seconds / 60,
                "arrival_time": arrival_time
            })

    # Choose the best path based on ETA
    if paths_info:
        best_path = min(paths_info, key=lambda x: x["eta_minutes"])

        return {
            "start_detector": start_detector,
            "end_detector": end_detector,
            "best_path_type": best_path["path_type"],
            "path": best_path["path"],
            "total_distance_meters": best_path["distance"],
            "total_distance_km": best_path["distance"] / 1000,
            "predicted_speed_kmh": best_path["predicted_speed"],
            "predicted_flow": best_path["predicted_flow"],
            "predicted_occupancy": best_path["predicted_occupancy"],
            "eta_minutes": best_path["eta_minutes"],
            "current_time": current_time.strftime("%Y-%m-%d %H:%M:%S"),
            "estimated_arrival_time": best_path["arrival_time"].strftime("%Y-%m-%d %H:%M:%S"),
            "weather_conditions": weather_features,
            "alternative_paths": paths_info
        }
```

```
    return {"error": "Could not calculate predictions"}
def extract_weather_features(self, weather_data):
    """Extract relevant weather features from the API response"""
    if not weather_data:
        return {
            'temperature': 15.0,  # Default values if weather data is unavailable
            'precipitation': 0.0,
            'wind_speed': 5.0,
            'humidity': 50.0,
            'is_rainy': 0
        }

    current = weather_data.get('current', {})
    return {
        'temperature': current.get('temp_c', 15.0),
        'precipitation': current.get('precip_mm', 0.0),
        'wind_speed': current.get('wind_kph', 5.0),
        'humidity': current.get('humidity', 50.0),
        'is_rainy': 1 if current.get('precip_mm', 0) > 0 else 0
    }
def get_weather_data(self, city="augsburg", days=1):
    """Fetch weather data from the WeatherAPI"""
    url = f"{WEATHER_BASE_URL}/forecast.json?key={WEATHER_API_KEY}&q={city}&days={days}&aqi=no"
    try:
        response = requests.get(url)
        if response.status_code == 200:
            return response.json()
        else:
            print(f"Weather API error: {response.status_code}")
            return None
    except Exception as e:
        print(f"Error fetching weather data: {e}")
        return None
```

```
def kruskal_shortest_path(self, start_detector, end_detector):
    """Find shortest path using Kruskal's algorithm"""
    def find(parent, i):
        if parent[i] == i:
            return i
        return find(parent, parent[i])

    def union(parent, rank, x, y):
        xroot = find(parent, x)
        yroot = find(parent, y)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
        else:
            parent[yroot] = xroot
            rank[xroot] += 1

    # Get all edges with weights
    edges = []
    for (u, v, d) in self.graph.edges(data=True):
        edges.append((u, v, d['weight']))

    # Sort edges by weight
    edges.sort(key=lambda x: x[2])

    vertices = list(self.graph.nodes())
    parent = {v: v for v in vertices}
    rank = {v: 0 for v in vertices}

    mst_edges = []
    for u, v, w in edges:
        if find(parent, u) != find(parent, v):
            mst_edges.append((u, v, w))
            union(parent, rank, u, v)

    # Create a new graph with MST edges
    mst = nx.Graph()
    for u, v, w in mst_edges:
        mst.add_edge(u, v, weight=w)

    try:
        path = nx.shortest_path(mst, start_detector, end_detector, weight='weight')
        distance = sum(mst[path[i]][path[i+1]]['weight'] for i in range(len(path)-1))
        return path, distance
    except nx.NetworkXNoPath:
        return None, None
```

```
def train_model(self, epochs=100, batch_size=32, patience=10):
    """Train the LSTM model with improved training process"""
    if self.model is None:
        self.build_model()

    try:
        X_train, X_test, y_train, y_test = self.prepare_training_data()

        # Validate data
        if any(x is None for x in [X_train, X_test, y_train, y_test]):
            raise ValueError("Data preparation failed")

        # Generate timestamp
        timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
        model_filename = f"traffic_model_{timestamp}.keras"
        model_path = os.path.join(MODEL_DIR, model_filename)

        # Enhanced callbacks
        callbacks = [
            EarlyStopping(
                monitor='val_loss',
                patience=patience,
                restore_best_weights=True,
                verbose=1
            ),
            ModelCheckpoint(
                model_path,
                monitor='val_loss',
                save_best_only=True,
                mode='min',
                verbose=1
            ),
            ReduceLROnPlateau(
                monitor='val_loss',
                factor=0.5,
                patience=5,
                min_lr=0.0001,
                verbose=1
            )
        ]

        # Train model
        print(f"Training model with up to {epochs} epochs (patience={patience})...")
        history = self.model.fit(
            X_train, y_train,
            batch_size=batch_size,
            epochs=epochs,
            validation_data=(X_test, y_test),
            callbacks=callbacks,
            verbose=1
        )
```

MAKE PREDICTIONS

```
        # Calculate predictions and metrics
        y_pred_scaled = self.model.predict(X_test)
        y_pred = self.scaler_y.inverse_transform(y_pred_scaled)
        y_actual = self.scaler_y.inverse_transform(y_test)

        # Calculate detailed metrics
        metrics = {
            "timestamp": timestamp,
            "epochs_trained": len(history.history['loss']),
            "max_epochs": epochs,
            "patience": patience,
            "final_train_loss": float(history.history['loss'][-1]),
            "final_val_loss": float(history.history['val_loss'][-1]),
            "model_metrics": {
                "mse": float(mean_squared_error(y_actual, y_pred)),
                "rmse": float(np.sqrt(mean_squared_error(y_actual, y_pred))),
                "mae": float(mean_absolute_error(y_actual, y_pred)),
                "r2_score": float(r2_score(y_actual.flatten(), y_pred.flatten()))
            }
        }

        self.metrics = metrics

        # Save results
        self._save_training_results(history, metrics, timestamp, y_actual, y_pred)

        return history, metrics

    except Exception as e:
        print(f"Error during training: {e}")
        return None, None
```

SOURCE CODE FOR GAT

LOAD DATA

```
def load_data(self):
    """Load and preprocess the speed and distance data"""
    print("Loading speed data...")
    self.speed_data = pd.read_excel(self.speed_data_path)

    # Clean speed data
    self.speed_data = self.speed_data.replace([np.inf, -np.inf], np.nan)
    self.speed_data = self.speed_data.ffill().bfill()

    # Convert day to datetime
    if not pd.api.types.is_datetime64_any_dtype(self.speed_data['day']):
        self.speed_data['day'] = pd.to_datetime(self.speed_data['day'])

    # Create timestamp column
    self.speed_data['timestamp'] = self.speed_data.apply(
        lambda row: row['day'] + pd.Timedelta(seconds=row['interval']), axis=1)

    print("Loading detector distances...")
    self.detector_distances = pd.read_excel(self.detector_distances_path)

    # Clean distance data
    self.detector_distances = self.detector_distances.replace([np.inf, -np.inf], np.nan)
    self.detector_distances = self.detector_distances.ffill().bfill()

    # Build graph for path finding
    self.build_graph()

    print("Data loaded and cleaned successfully.")
```

BUILD GRAPH:

```
class TrafficDataPreprocessor:
    def build_graph(self):
        """Build optimized graph representations for path finding"""
        print("Building network graphs...")

        try:
            # Create regular graph
            self.graph = nx.Graph()

            # Get unique detector IDs
            unique_detectors = set()
            for _, row in self.detector_distances.iterrows():
                # Ensure detector IDs are strings
                det1 = str(row['Detector1'])
                det2 = str(row['Detector2'])
                unique_detectors.add(det1)
                unique_detectors.add(det2)

            # Create a mapping from detector ID to integer index
            self.detector_to_idx = {det: i for i, det in enumerate(sorted(unique_detectors))}
            self.idx_to_detector = {i: det for det, i in self.detector_to_idx.items()}

            # Add edges with weights from detector distances
            for _, row in self.detector_distances.iterrows():
                # Convert to string to ensure consistency
                det1 = str(row['Detector1'])
                det2 = str(row['Detector2'])
                self.graph.add_edge(
                    det1,
                    det2,
                    weight=row['Distance (meters)']
                )

            # Create MST graph using Kruskal's algorithm
            self.mst_graph = nx.minimum_spanning_tree(self.graph)

            print(f"Network graphs built successfully:")
            print(f"- Main graph: {self.graph.number_of_nodes()} nodes, "
                f"{self.graph.number_of_edges()} edges")
            print(f"- MST graph: {self.mst_graph.number_of_nodes()} nodes, "
                f"{self.mst_graph.number_of_edges()} edges")

            # Calculate and store basic graph metrics
            self.graph_metrics = {
                'diameter': nx.diameter(self.graph),
                'average_shortest_path': nx.average_shortest_path_length(self.graph),
                'average_degree': sum(dict(self.graph.degree()).values()) /
                                  self.graph.number_of_nodes()
            }

            print("\nGraph metrics:")
            print(f"- Network diameter: {self.graph_metrics['diameter']:.2f} meters")
            print(f"- Average path length: "
                f"{self.graph_metrics['average_shortest_path']:.2f} meters")
            print(f"- Average node degree: {self.graph_metrics['average_degree']:.2f}")

        except Exception as e:
            print(f"Error building graphs: {e}")
            raise
```

PREPARE TRAINING DATA:

```
def prepare_training_data(self):
    """Prepare data for GAT training"""
    print("Preparing training data...")

    # Handle NaN values in the speed data
    self.speed_data = self.speed_data.ffill().bfill()

    grouped = self.speed_data.sort_values(['detid', 'timestamp'])
    X, y = [], []

    try:
        for detector, group in grouped.groupby('detid'):
            # Get detector distances
            detector_distances = self.detector_distances[
                (self.detector_distances['Detector1'] == detector) |
                (self.detector_distances['Detector2'] == detector)
            ]
            avg_distance = detector_distances['Distance (meters)'].mean()
            if np.isnan(avg_distance):
                avg_distance = detector_distances['Distance (meters)'].median()
            if np.isnan(avg_distance):
                avg_distance = 1000

            # Normalize distance
            normalized_distance = avg_distance / 1000  # Convert to kilometers

            # Get base feature values
            features = group[self.feature_columns].values

            # Create sequences with sliding window
            for i in range(len(features) - self.sequence_length):
                sequence = features[i:i + self.sequence_length]

                # Calculate time features
                timestamp = group.iloc[i]['timestamp']
                time_of_day = timestamp.hour / 24.0
                day_of_week = timestamp.weekday() / 7.0
```

```
            # Create sequence with all features
            sequence_with_features = np.column_stack((
                sequence,  # Base features
                np.full((self.sequence_length, 1), normalized_distance),  # Distance
                np.full((self.sequence_length, 1), time_of_day),  # Time of day
                np.full((self.sequence_length, 1), day_of_week)  # Day of week
            ))

            target = [
                features[i + self.sequence_length, 4],  # speed
                features[i + self.sequence_length, 1],  # flow
                features[i + self.sequence_length, 2],  # occupancy
                i * 5  # time offset
            ]

            X.append(sequence_with_features)
            y.append(target)

    if not X or not y:
        raise ValueError("No valid sequences could be created from the data")

    X_array = np.array(X, dtype=np.float32)
    y_array = np.array(y, dtype=np.float32)

    # Scale features
    n_samples = X_array.shape[0]
    n_timesteps = X_array.shape[1]
    n_features = X_array.shape[2]

    print(f"Input shape before scaling: {X_array.shape}")

    X_reshaped = X_array.reshape((n_samples * n_timesteps, n_features))
    X_scaled = self.scaler_X.fit_transform(X_reshaped)
    X_scaled = X_scaled.reshape((n_samples, n_timesteps, n_features))

    # Scale targets
    y_scaled = self.scaler_y.fit_transform(y_array)

    # Split data
    X_train, X_test, y_train, y_test = train_test_split(
        X_scaled, y_scaled, test_size=0.2, random_state=42
    )

        # Convert to PyTorch tensors
        X_train = torch.tensor(X_train, dtype=torch.float32)
        X_test = torch.tensor(X_test, dtype=torch.float32)
        y_train = torch.tensor(y_train, dtype=torch.float32)
        y_test = torch.tensor(y_test, dtype=torch.float32)

        print(f"Training data shape: {X_train.shape}, Target data shape: {y_train.shape}")
        return X_train, X_test, y_train, y_test

    except Exception as e:
        print(f"Error preparing training data: {e}")
        return None, None, None, None
```

```python
def kruskal_shortest_path(self, start_detector, end_detector):
    """Find shortest path using Kruskal's algorithm"""
    def find(parent, i):
        if parent[i] == i:
            return i
        return find(parent, parent[i])

    def union(parent, rank, x, y):
        xroot = find(parent, x)
        yroot = find(parent, y)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
        else:
            parent[yroot] = xroot
            rank[xroot] += 1

    # Get all edges with weights
    edges = []
    for (u, v, d) in self.graph.edges(data=True):
        edges.append((u, v, d['weight']))

    # Sort edges by weight
    edges.sort(key=lambda x: x[2])

    vertices = list(self.graph.nodes())
    parent = {v: v for v in vertices}
    rank = {v: 0 for v in vertices}

    mst_edges = []
    for u, v, w in edges:
        if find(parent, u) != find(parent, v):
            mst_edges.append((u, v, w))
            union(parent, rank, u, v)

    # Create a new graph with MST edges
    mst = nx.Graph()
    for u, v, w in mst_edges:
        mst.add_edge(u, v, weight=w)

    try:
        path = nx.shortest_path(mst, start_detector, end_detector, weight='weight')
        distance = sum(mst[path[i]][path[i+1]]['weight'] for i in range(len(path)-1))
        return path, distance
    except nx.NetworkXNoPath:
        return None, None
```

BUILD MODEL

```
class SimpleGATModel(nn.Module):
    def __init__(self, input_dim, hidden_dim=32, output_dim=4, dropout=0.3):
        super(SimpleGATModel, self).__init__()

        # Feature extraction layers
        self.feature_extractor = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Dropout(dropout)
        )

        # Simple attention mechanism
        self.attention = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim),
            nn.Tanh(),
            nn.Linear(hidden_dim, 1)
        )

        # Output layer
        self.output_layer = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        # Use only the last few timesteps for prediction
        batch_size = x.size(0)
        seq_len = x.size(1)

        # Process each timestep
        features = self.feature_extractor(x.view(batch_size * seq_len, -1))
        features = features.view(batch_size, seq_len, -1)

        # Apply attention
        attention_weights = F.softmax(self.attention(features).squeeze(-1), dim=1)
        context = torch.bmm(attention_weights.unsqueeze(1), features).squeeze(1)

        # Final prediction
        output = self.output_layer(context)

        return output
```

TRAIN MODEL:

```
def train_new_model(preprocessor):
    print("\nModel Training Configuration")
    print("-" * 40)
    try:
        # Get training parameters
        epochs = int(input("Enter maximum epochs (default 20): ") or "20")
        batch_size = int(input("Enter batch size (default 128): ") or "128")
        patience = int(input("Enter early stopping patience (default 10): ") or "10")

        print("\nStarting model training...")
        print("This may take several minutes. Please wait...")

        start_time = time.time()

        # Prepare training data
        X_train, X_test, y_train, y_test = preprocessor.prepare_training_data()

        if X_train is None or X_test is None or y_train is None or y_test is None:
            print("Failed to prepare training data. Check your dataset.")
            return None

        # Get number of nodes in the graph
        num_nodes = preprocessor.graph.number_of_nodes()

        # Create model with optimized architecture
        model = GATTrafficPredictionModel(
            nfeat=X_train.shape[2],  # Number of features
            nhid=64,                 # Hidden layer features
            nclass=y_train.shape[1], # Number of output classes
            dropout=0.3,             # Moderate dropout
            nheads=4,                # Reduced number of attention heads
            num_nodes=min(num_nodes, 20)  # Limit number of nodes
        )

        # Move model to GPU if available
        model = model.to(device)

        # Create trainer
        trainer = GATTrainer(model, device, preprocessor)

        # Optimize data loading with efficient parameters
        train_dataset = torch.utils.data.TensorDataset(X_train, y_train)
        test_dataset = torch.utils.data.TensorDataset(X_test, y_test)

        # Use gradient accumulation for memory efficiency
        accumulation_steps = max(1, len(X_train) // (batch_size * 10000))

        train_loader = torch.utils.data.DataLoader(
            train_dataset,
            batch_size=batch_size,
            shuffle=True,
            num_workers=0,  # Set to 0 to avoid multiprocessing overhead
            pin_memory=True  # Enable pinned memory for faster GPU transfer
        )
```

```
test_loader = torch.utils.data.DataLoader(
    test_dataset,
    batch_size=batch_size,
    shuffle=False,
    num_workers=0,
    pin_memory=True
)

# Modify the trainer to use more efficient training
history, metrics = trainer.train_model(
    X_train, X_test, y_train, y_test,
    epochs=epochs,
    batch_size=batch_size,
    patience=patience,
    train_loader=train_loader,
    test_loader=test_loader,
    accumulation_steps=accumulation_steps,
    verbose=True
)

# Generate timestamp for model and scaler files
# Generate timestamp for model and scaler files
timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
if metrics and 'timestamp' in metrics:
    timestamp = metrics['timestamp']

# Save the model
model_path = os.path.join(MODEL_DIR, f"gat_model_{timestamp}.pth")
torch.save(model.state_dict(), model_path)
print(f"Final model saved to {model_path}")

# Save the scalers
scaler_path = os.path.join(MODEL_DIR, f"scalers_{timestamp}.pkl")
scalers = {
    'scaler_X': preprocessor.scaler_X,
    'scaler_y': preprocessor.scaler_y
}
joblib.dump(scalers, scaler_path)
print(f"Scalers saved to {scaler_path}")


training_time = time.time() - start_time

if metrics:
    print("\nTraining Summary:")
    print("-" * 40)
    print(f"Training timestamp: {timestamp}")
    print(f"Training time: {training_time:.2f} seconds")
    print(f"Final training loss: {metrics.get('final_train_loss', 'N/A'):.6f}")
    print(f"Final validation loss: {metrics.get('final_val_loss', 'N/A'):.6f}")

    print("\nPrediction Metrics:")
    print(f"RMSE: {metrics.get('rmse', 'N/A'):.2f}")
    print(f"MAE: {metrics.get('mae', 'N/A'):.2f}")
```

```
print("\nTraining visualizations have been saved to:")
print(f"- {METRICS_DIR}")

# Get predictions for visualization
model.eval()
with torch.no_grad():
    y_pred = model(X_test.to(device)).cpu().numpy()
    y_actual = y_test.numpy()

# Create comprehensive visualization
plt.figure(figsize=(20, 15))

# 1. Training History Plot
plt.subplot(3, 2, 1)
plt.plot(history['train_loss'], label='Training Loss')
plt.plot(history['val_loss'], label='Validation Loss')
plt.title('Model Loss Over Time')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)

# 2. Speed Prediction Scatter Plot
plt.subplot(3, 2, 2)
plt.scatter(y_actual[:, 0], y_pred[:, 0], alpha=0.5)
plt.plot([y_actual[:, 0].min(), y_actual[:, 0].max()],
         [y_actual[:, 0].min(), y_actual[:, 0].max()],
         'r--', label='Perfect Prediction')
plt.title('Predicted vs Actual Speed')
plt.xlabel('Actual Speed')
plt.ylabel('Predicted Speed')
plt.legend()
plt.grid(True)

# 3. Flow Prediction Plot
plt.subplot(3, 2, 3)
plt.scatter(y_actual[:, 1], y_pred[:, 1], alpha=0.5)
plt.plot([y_actual[:, 1].min(), y_actual[:, 1].max()],
         [y_actual[:, 1].min(), y_actual[:, 1].max()],
         'r--', label='Perfect Prediction')
plt.title('Predicted vs Actual Flow')
plt.xlabel('Actual Flow')
plt.ylabel('Predicted Flow')
plt.legend()
plt.grid(True)
```

MAKE PREDICTIONS

```
# Calculate predictions for paths with enhanced weather consideration
paths_info = []
for path, distance, path_type in [
    (regular_path, regular_distance, "Regular"),
    (kruskal_path, kruskal_distance, "Kruskal")
]:
    if not path:
        continue

    try:
        # Prepare path features
        path_features = self._prepare_path_features(path, distance, current_time, weather_features)

        # Make prediction
        with torch.no_grad():
            path_features = path_features.to(self.device)
            predictions = self.model(path_features)
            predictions = predictions.cpu().numpy()

        # Inverse transform predictions
        predictions = self.scaler_y.inverse_transform(predictions)

        # Apply weather adjustments with more sophisticated model
        predictions = self._apply_weather_adjustments(predictions, weather_features)

        # Calculate path metrics
        path_metrics = self._calculate_path_metrics(
            path,
            predictions,
            distance,
            current_time,
            weather_features
        )

        paths_info.append({
            "path_type": path_type,
            **path_metrics
        })

    except Exception as path_error:
        print(f"Error processing path {path_type}: {path_error}")
```

```
    # Choose best path
    if not paths_info:
        return {"error": "Could not calculate predictions for any path"}

    best_path = min(paths_info, key=lambda x: x["eta_minutes"])

    return {
        "start_detector": start_detector,
        "end_detector": end_detector,
        "best_path_type": best_path["path_type"],
        "path": best_path["path"],
        "total_distance_meters": best_path["distance"],
        "total_distance_km": best_path["distance"] / 1000,
        "predicted_speed_kmh": best_path["predicted_speed"],
        "predicted_flow": best_path["predicted_flow"],
        "predicted_occupancy": best_path["predicted_occupancy"],
        "eta_minutes": best_path["eta_minutes"],
        "current_time": current_time.strftime("%Y-%m-%d %H:%M:%S"),
        "estimated_arrival_time": best_path["arrival_time"].strftime("%Y-%m-%d %H:%M:%S"),
        "weather_conditions": weather_features,
        "alternative_paths": paths_info
    }
```

ANNEXURE 2

OUTPUT SCREENS

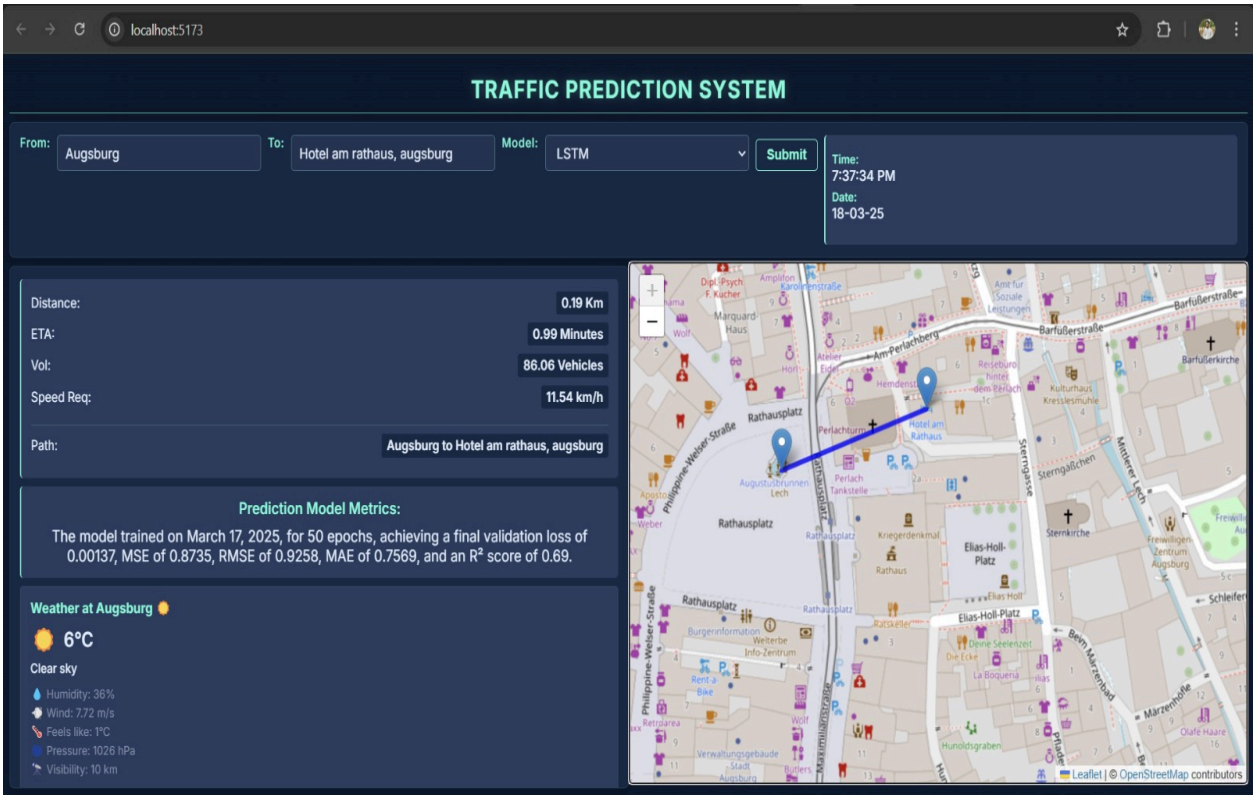

Figure A2.1 : LSTM OUTPUT

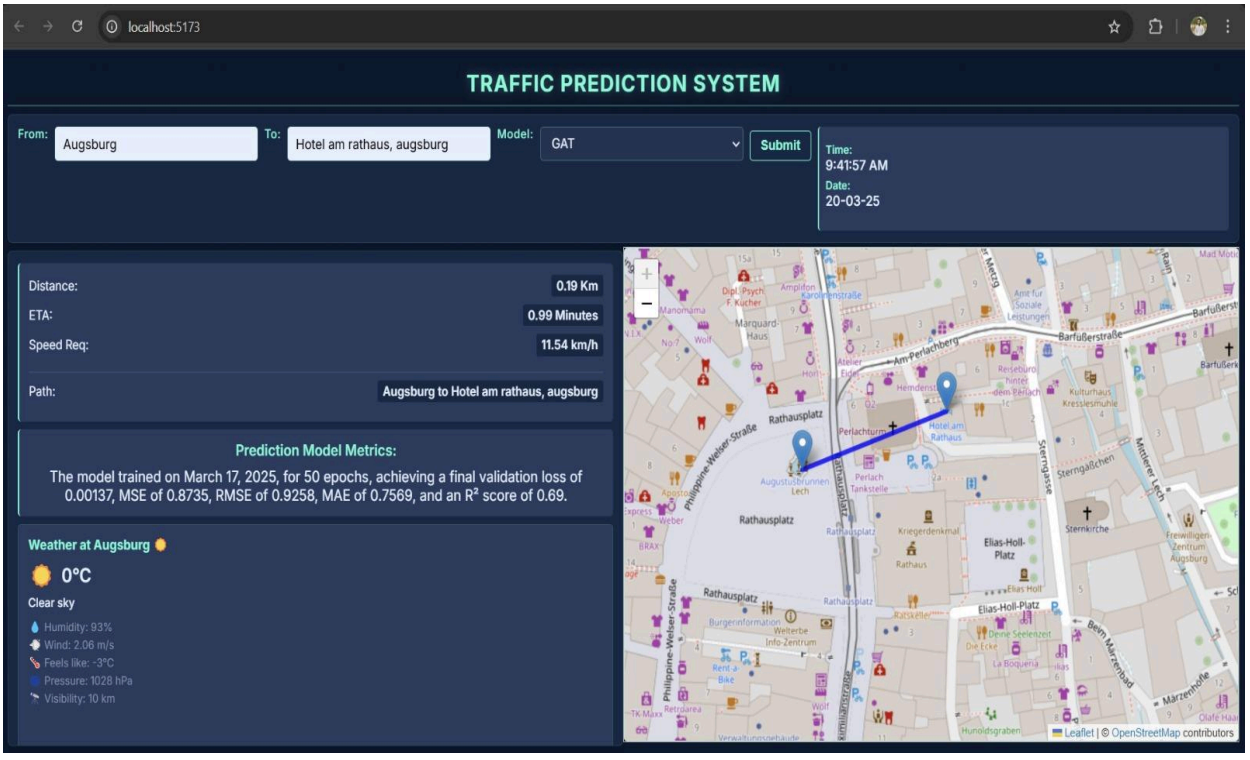

Figure A2.2 : GAT OUTPUT