UNIVERSITY OF PARDUBICE

FACULTY OF ECONOMICS AND ADMINISTRATION

MASTER THESIS

2025 Godfred Enim Douglas

University of Pardubice
Faculty of Economics and Administration

Temporal Fusion Transformers for Traffic Flow Prediction in Smart Cities

Master's Thesis

2025 Godfred Enim Douglas

University of Pardubice
Faculty of Economics and Administration
Academic year: 2024/2025

ASSIGNMENT OF DIPLOMA THESIS

(project, art work, art performance)

Name and surname:	**Godfred Enim Douglas**
Personal number:	**E23968**
Study programme:	**N0688A140008 Informatics and System Engineering**
Specialization:	**Informatics in Public Administration**
Work topic:	**Temporal fusion transformers for traffic flow prediction in smart cities**
Assigning department:	**Institute of System Engineering and Informatics**

Theses guidelines

The aim of the work is to introduce temporal fusion transformers, characterize approaches to traffic flow prediction in smart cities, propose a traffic flow prediction system using temporal fusion transformers, pre-process traffic flow datasets, validate the proposed prediction system using the datasets, and discuss implications for smart cities.
Outline:

- Deep neural networks and temporal fusion transformers
- Traffic flow prediction in smart cities
- System for traffic flow prediction
- Traffic flow datasets
- Prediction of traffic flow and interpretation of results

Extent of work report: **Approx. 50 pages**
Extent of graphics content:
Form processing of diploma thesis: **printed/electronic**
Language of elaboration: **English**

Recommended resources:

ALI SHAH, S. A., ILIANKO, K., FERNANDO, X. Deep learning based traffic flow prediction for autonomous vehicular mobile networks. In *2021 IEEE 94th Vehicular Technology Conference* (VTC2021Fall), 2021, p. 15.
LUO, T., NAGARAJAN, S. G. Distributed anomaly detection using autoencoder neural networks in WSN for IoT. In *2018 IEEE International Conference on Communications* (ICC), 2018, p. 16.
NEELAKANDAN, S., BERLIN, M. A., TRIPATHI, S., DEVI, V. B., BHARDWAJ, I., ARULKUMAR, N. IoT-based traffic prediction and traffic signal control system for smart city. *Soft Computing*, 2021, 25, p. 12241--12248.
SAID, O., TOLBA, A. Accurate performance prediction of IoT communication systems for smart cities: An efficient deep learning based solution. *Sustainable Cities and Society*, 2021, 69, p. 102830.

Supervisors of diploma thesis: **prof. Ing. Petr Hájek, Ph.D.**
Science and Research Centre

Date of assignment of diploma thesis: **September 1, 2024**
Submission deadline of diploma thesis: **April 30, 2025**

L.S.

prof. Ing. Jan Stejskal, Ph.D. m.p.
Dean

RNDr. Ing. Oldřich Horák, Ph.D. m.p.
Institute Head

In Pardubice September 1, 2024

AUTHOR'S DECLARATION

I declare:

The thesis entitled Temporal Fusion Transformers for Traffic Flow Prediction in Smart Cities is my own work. All literary sources and information that I used in the thesis are referenced in the bibliography.

I have been acquainted with the fact that my work is subject to the rights and obligations arising from Act No. 121/2000 Sb., On Copyright, on Rights Related to Copyright and on Amendments to Certain Acts (Copyright Act), as amended, especially with the fact that the University of Pardubice has the right to conclude a license agreement for the use of this thesis as a school work under Section 60, Subsection 1 of the Copyright Act, and that if this thesis is used by me or a license to use it is granted to another entity, the University of Pardubice is entitled to request a reasonable fee from me to cover the costs incurred for the creation of the work, depending on the circumstances up to their actual amount.

I acknowledge that in accordance with Section 47b of Act No. 111/1998 Sb., On Higher Education Institutions and on Amendments to Other Acts (Higher Education Act), as amended, and the Directive of the University of Pardubice No. 7/2019 Rules for Submission, Publication and Layout of Theses, as amended, the thesis will be published through the Digital Library of the University of Pardubice.

In Pardubice , 27. 06. 2025 Godfred Enim Douglas

ACKNOWLEDGEMENT

First and foremost, I give all thanks and glory to God Almighty for His grace, strength, and guidance throughout this academic journey. Without His divine support, this thesis would not have been possible.

I would like to express my deepest gratitude to my supervisor, Prof. Petr Hájek, Ph.D., for his outstanding mentorship, constructive feedback, and continuous encouragement throughout the course of this research. His expertise in the field and unwavering support greatly enhanced the quality and depth of my work.

I am also sincerely thankful to Dr. Evelyn Awuah, whose insights and background in Informatics provided additional clarity and inspiration during key stages of this thesis.

Special appreciation goes to Mr. Andy Kwesi Asante for his steadfast support, encouragement, and friendship, which helped me stay focused and motivated during challenging times.

I extend my appreciation to the faculty and staff of the Institute of Systems Engineering and Informatics at the University of Pardubice for their academic support and for providing a nurturing environment for learning and research.

Lastly, I am grateful to my family and friends for their unwavering love, patience, and encouragement throughout this journey. Thank you all for being a part of this accomplishment.

ANOTACE

Tato diplomová práce představuje pipeline založenou na modelu Temporal Fusion Transformer (TFT) pro krátkodobou předpověď dopravního toku na více místech v prostředí chytrých měst. Pomocí reálných časoprostorových dat studie implementuje škálovatelné řešení v Google Colab se zaměřením na předzpracování dat, ladění hyperparametrů a interpretovatelnost. Model TFT výrazně překonává výchozí metody, když dosahuje přibližně 30% snížení hodnoty MAE. Práce zdůrazňuje využitelnost modelu v rámci řízení dopravy v reálném čase a navrhuje jeho budoucí integraci do inteligentních dopravních systémů.

KLÍČOVÁ SLOVA

Předpověď dopravního toku, Temporal Fusion Transformer (TFT), Mobilita chytrých měst, Hluboké učení, Interpretovatelnost, Data z městských senzorů.

TITLE

Temporal Fusion Transformers for Traffic Flow Prediction in Smart Cities

ANNOTATION

This thesis presents a Temporal Fusion Transformer (TFT)-based pipeline for short-term, multi-site traffic-flow forecasting in smart-city environments. Using real-world spatiotemporal data, the study implements a scalable solution in Google Colab with a focus on data preprocessing, hyperparameter tuning, and interpretability. The TFT model significantly outperforms baseline methods, achieving a 30% reduction in MAE. The work highlights the model's applicability in real-time traffic management and suggests future integration into intelligent transportation systems.

KEYWORDS

Traffic-flow forecasting, Temporal Fusion Transformer (TFT), Smart-city mobility, Deep learning, Interpretability, Urban sensor data.

TABLE OF CONTENTS

LIST OF FIGURES AND TABLES

LIST OF ABBREVIATIONS

ARIMA	Autoregressive Integrated Moving Average
GRN	Gated Residual Network
GRU	Gated Recurrent Unit
IoT	Internet of Things
ITS	Intelligent Transportation Systems
LSTM	Long Short-Term Memory
MAE	Mean Absolute Error
RMSE	Root Mean Squared Error
RNN	Recurrent Neural Network
SVR	Support Vector Regression
TCN	Temporal Convolutional Network
TFT	Temporal Fusion Transformer
VAR	Vector AutoRegression
VSN	Variable Selection Network

1 Introduction

1.1 Background

In the era of rapid urbanization and technological advancement, modern cities are increasingly transitioning into smart cities. These are urban environments that leverage data, digital infrastructure, and intelligent systems with the aim of enhancing operational efficiency and improving the quality of life for residents. Among the most critical challenges faced by smart cities is the efficient management of transportation systems. Traffic congestion, delays, accidents, and pollution not only hinder productivity but also negatively impact urban living conditions. As such, effective traffic management depends heavily on the ability to predict traffic flow accurately, enabling authorities to implement dynamic control strategies and support real-time decision-making.

Historically, traffic flow prediction has relied on statistical and classical time series models such as Autoregressive Integrated Moving Average (ARIMA), Vector AutoRegression (VAR), and Kalman filtering. These approaches are effective in capturing linear temporal dependencies and short-term fluctuations. However, they are limited in their capacity to model complex non-linear patterns and typically assume data stationarity, which does not align with the irregular and dynamic nature of real-world traffic systems. For instance, these models require significant preprocessing and offer limited flexibility when exposed to rapidly changing urban conditions (Box et al., 2015). Additionally, such models often underperform when multiple external variables are introduced into the forecasting process (Hyndman & Athanasopoulos, 2018).

With the evolution of machine learning, deep learning models have gained prominence in traffic forecasting. Recurrent Neural Networks (RNNs), Long Short-Term Memory (LSTM) networks, and Gated Recurrent Units (GRUs) have shown strong capabilities in modeling non-linear temporal sequences. Gating mechanisms, for example, allow models to retain long-term dependencies in sequential data (Hochreiter & Schmidhuber, 1997). More recently, convolutional models have also demonstrated competitive results in sequence modeling tasks, particularly in terms of parallelization and training efficiency (Bai et al., 2018).

Despite their predictive strengths, many of these models lack interpretability. In applications involving public systems and critical infrastructure, understanding model decisions is essential. In response to this limitation, the Temporal Fusion Transformer (TFT) was proposed as a deep

learning architecture designed specifically for multi-horizon forecasting with enhanced interpretability (Lim et al., 2021). TFT integrates attention mechanisms, variable selection networks, and static covariate encoders to support a wide range of inputs and time dependencies. It enables users to analyze which features influence predictions and how these influences vary across time.

The effectiveness of TFT has been demonstrated across a variety of domains. It has achieved competitive or superior performance in forecasting tasks in energy, finance, and healthcare applications (Lim & Zohren, 2021). In the transportation field, TFT has been successfully applied to freeway speed forecasting. For example, Zou et al. (2022) found that TFT outperformed both ARIMA and LSTM models, particularly in longer-range forecasts. Their study highlighted the model's strengths in capturing complex spatial-temporal patterns and generating interpretable outputs—an advantage that is particularly valuable in the context of smart cities.

Modern urban traffic systems generate large volumes of data through sensors, GPS devices, and mobile platforms. These datasets are often high-dimensional and heterogeneous, consisting of both static and time-varying features. TFT's architecture is well suited for such complexity, as it can jointly learn from known future inputs (e.g., calendar features), historical trends, and static attributes (e.g., location identifiers). Furthermore, its ability to provide multi-horizon forecasts makes it particularly suitable for use in real-time traffic management systems, navigation assistance, and congestion mitigation.

This thesis investigates the application of the TFT for short-term traffic flow forecasting in the context of smart cities. The study involves constructing a TFT model using real-world urban traffic datasets, along with data preprocessing, feature engineering, and model training. The model's configuration is optimized through hyperparameter tuning, including testing different hidden sizes, attention head counts, and dropout rates. Its performance is evaluated using Root Mean Squared Error (RMSE) and Mean Absolute Error (MAE) on both validation and test datasets. In addition, the model's interpretability features are explored to gain insights into the most influential factors in traffic prediction.

Ultimately, this research aims to demonstrate that the TFT is a highly effective and interpretable tool for traffic forecasting in smart cities. Its integration of transparency, accuracy, and flexibility presents a compelling alternative to both traditional statistical methods and black-box deep learning models.

1.2 Problem Statement

As urban populations continue to grow and smart city initiatives advance, the need for efficient and intelligent traffic management has become increasingly critical. Congestion, unpredictable travel times, and transportation inefficiencies continue to challenge urban mobility, impacting economic productivity, environmental sustainability, and the overall quality of life. Accurate traffic flow prediction plays a central role in addressing these issues, enabling authorities to make informed decisions and implement timely interventions to optimize transportation systems.

Although modern cities are equipped with vast sources of traffic-related data collected through sensors, GPS systems, and digital infrastructure, existing forecasting models often fall short in effectively utilizing this information. Classical statistical models such as ARIMA and Kalman filters are limited in their ability to capture complex non-linear relationships and typically require assumptions of stationarity and linearity (Box et al., 2015; Hyndman & Athanasopoulos, 2018). While deep learning models such as Long Short-Term Memory (LSTM) networks have improved performance by learning sequential dependencies, they often lack transparency and fail to provide insight into how predictions are made (Hochreiter & Schmidhuber, 1997).

This lack of interpretability presents a major limitation, particularly in smart city environments where decision-making must be transparent, explainable, and trusted by stakeholders. Additionally, many forecasting models do not take full advantage of the variety of available input features, including static attributes (such as location), known future inputs (such as calendar effects), and historical trends, all of which are essential for accurately capturing the dynamic nature of traffic systems.

The TFT, introduced by Lim et al. (2021), presents a promising alternative. It integrates attention mechanisms, gating structures, and feature selection networks to handle diverse input types and deliver accurate forecasts. Importantly, it offers interpretability by identifying the contribution of each input variable over time. Although TFT has demonstrated success in domains such as energy, retail, and healthcare forecasting (Lim & Zohren, 2021), its application to traffic flow prediction within the context of smart cities remains underexplored.

There is a clear need to evaluate the performance and practical utility of the Temporal Fusion Transformer for traffic forecasting using real-world urban datasets. This includes optimizing the model's parameters for traffic-specific applications and assessing its ability to provide interpretable outputs that can support data-driven decision-making. Addressing this gap will contribute to the development of more accurate, transparent, and responsive forecasting systems for intelligent transportation in smart cities.

1.3 Objective

The general objective of this thesis is to investigate the application of the TFT model for short-term traffic flow prediction within the context of smart city environments. The goal is to assess its predictive performance, interpretability, and suitability for integration into intelligent traffic management systems.

To achieve the general objective, the study sets out to accomplish the following specific objectives:

- To review existing traffic flow forecasting models, including both classical statistical approaches and deep learning methods, and identify their strengths and limitations.
- To implement the TFT model for multivariate time series forecasting using real-world traffic datasets enriched with static and temporal features.
- To perform hyperparameter tuning and optimization to enhance the model's forecasting accuracy.
- To evaluate the model's performance using standard metrics such as MAE and RMSE
- To explore the interpretability features of the TFT model, including attention mechanisms and variable importance scores, in order to identify the most influential factors driving traffic patterns.
- To assess the potential of the TFT model for real-world deployment in smart city traffic control and planning applications.

1.4 Research Questions

Central research question RQ: How effective is the Temporal Fusion Transformer (TFT) model in predicting short-term traffic flow in smart cities, and to what extent can it offer interpretability and practical value for intelligent transportation systems?

To answer the central question, the study will address the following specific research questions:

- RQ1: What are the limitations of traditional and deep learning-based traffic flow forecasting models in the context of smart cities?
- RQ2: How can the TFT model be applied to time series traffic data involving multiple input types (e.g., static and temporal features)?
- RQ3: What is the effect of hyperparameter tuning on the performance of the TFT model in terms of accuracy and generalization?
- RQ4: How accurately does the TFT model predict short-term traffic flow, as measured by standard performance metrics such as MAE and RMSE?
- RQ5: To what extent can the TFT model provide interpretable insights through feature importance analysis and attention mechanisms?
- RQ6: Can the TFT model support data-driven decision-making and planning in real-world smart city traffic management systems?

1.5 Scope and Limitations

This study focuses on the application of the TFT model for short-term traffic flow prediction in smart cities. The research covers the entire pipeline of time series forecasting, including data preprocessing, model implementation, hyperparameter tuning, and evaluation.

The key areas covered within the scope of this thesis include:

- The use of multivariate time series data that contains both static and time-varying features relevant to traffic systems.
- The application of the TFT model to forecast traffic-related variables such as vehicle volume, speed, or density over short time horizons (e.g., 1–6 hours ahead).
- The analysis of model interpretability through variable importance and attention weights to understand feature influence.
- Evaluation of the model using standard metrics (MAE and RMSE) to assess forecasting performance.

The research is primarily computational and experimental in nature and uses real-world or simulated smart city traffic datasets suitable for time series forecasting.

Despite its broad scope, the study is subject to the following limitations:

- The thesis does not include real-time deployment of the model into operational smart city systems such as traffic signal control or live navigation.

- Hardware constraints may limit the scale or complexity of models tested, particularly with large datasets or longer training durations.
- The study does not incorporate autonomous vehicle data or interactions between connected vehicles, which may affect broader traffic dynamics.
- The model's accuracy and generalizability depend heavily on the quality, completeness, and representativeness of the dataset used.
- While the model provides interpretability outputs, the interpretation of attention weights and feature importance may still require technical expertise.
- The research focuses on short-term forecasting only; long-term forecasting horizons are not considered.

1.6 Contributions

In this thesis we make three primary contributions. First, we present the inaugural application of the TFT to the problem of multi-site, multi-horizon urban traffic volume forecasting. By leveraging rich static attributes, known-future calendar variables, and past-observed traffic and sensor covariates, our implementation showcases how TFT's gating, variable-selection, and interpretable attention mechanisms can be tailored to the unique challenges of smart-city deployments. Second, we conduct a systematic hyperparameter study over seven distinct configurations; spanning learning rates, hidden-state sizes, continuous-feature embedding dims, and attention-head counts to identify the combination that yields the lowest six-hour ahead MAE and RMSE on held-out test data. Finally, we demonstrate end-to-end interpretability by extracting and visualizing both attention heat-maps and variable-selection weights. These visualizations illuminate the model's reliance on specific lagged observations and calendar features, thereby providing traffic engineers with actionable insights into when and why the TFT succeeds in capturing complex diurnal surges and decays.

2 Literature Review

This chapter examines foundational and contemporary methods for forecasting traffic flow, surveys advances in time series prediction models, and highlights the emergence of deep-learning architectures particularly the TFT. We begin by reviewing classical statistical approaches, proceed through machine learning and recurrent network methods, and conclude with Transformer-based innovations. Throughout, we emphasize how each generation of models balances predictive accuracy, interpretability, and practical deployment in smart-city environments.

2.1 Traditional Traffic Flow Prediction Methods

Several classical time-series and machine-learning methods have been successfully applied to short-term traffic flow prediction, each with its own strengths and limitations. Autoregressive integrated moving average models (ARIMA) have long been used to capture linear and seasonal patterns in traffic data; for example, Min and Wynter (2011) demonstrated that spatio-temporal ARIMA variants can provide accurate one-hour-ahead flow estimates on urban arterials. Kalman-filter-based approaches offer recursive, real-time updating and have been shown to track sudden changes in flow; Okutani and Stephanedes (1984) first adapted the Kalman filter to freeway flow prediction, demonstrating its ability to fuse historical trends with live observations under a Gaussian noise assumption. Support vector regression (SVR) provides a nonparametric framework that handles small to moderate datasets and moderate nonlinearities; Wu, Ho, and Lee (2004) applied SVR to travel-time and volume forecasting on urban road networks, achieving lower error than traditional regression under varying traffic conditions.

While these models are computationally efficient and interpretable, they fail to handle the nonlinear, high-dimensional, and irregular nature of modern traffic data (Box et al., 2015; Hyndman & Athanasopoulos, 2018). They also struggle with integrating heterogeneous inputs, such as weather, holidays, or spatial patterns.

Time series forecasting is pivotal in predicting future traffic conditions based on historical data. Accurate forecasts enable proactive traffic management, reducing congestion and enhancing urban mobility. Traditional statistical models, such as ARIMA and Seasonal ARIMA (SARIMA), have been employed for traffic prediction. However, these models often struggle

with the nonlinear and dynamic nature of traffic data, especially in complex urban environments.

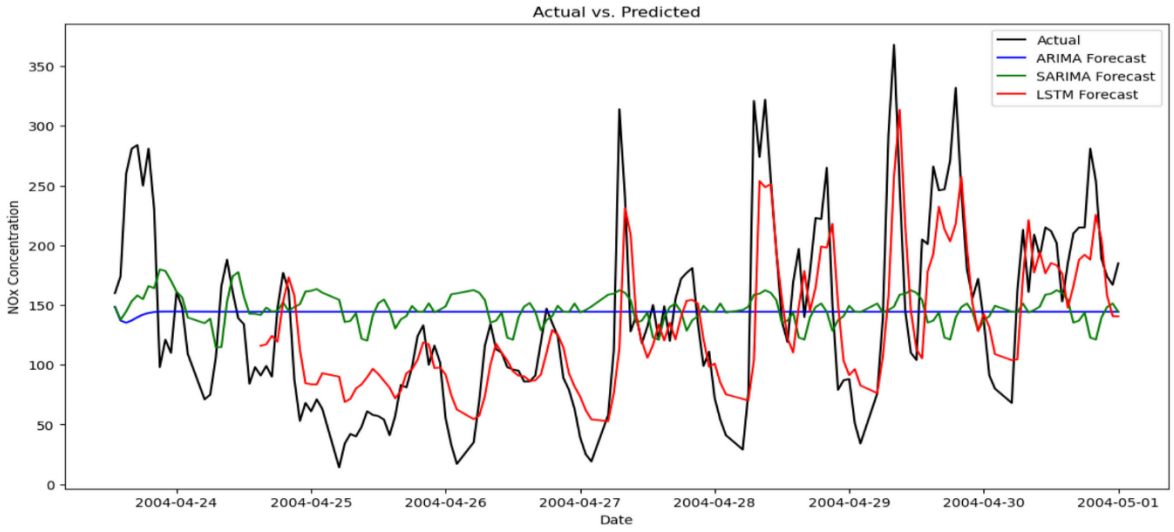

Figure 1: ARIMA and SARIMA models for time series forecasting.

Source: (Nobar, 2024)

2.2 Machine Learning and Deep Learning Approaches

The shortcomings of purely statistical models for traffic flow forecasting spurred the adoption of machine learning techniques that can capture nonlinear feature interactions. Ensemble tree methods such as Random Forests (Hou et al., 2014) and Gradient Boosting Machines (Chen et al., 2016) demonstrated improved accuracy over ARIMA by leveraging decision-tree ensembles to model complex relationships between covariates. Support Vector Regression (SVR) was also applied to travel-time and volume prediction (Wu et al., 2004), offering robust performance on small to moderate datasets. However, these methods treat each time step independently and cannot fully exploit the sequential nature of traffic data.

2.2.1 Recurrent Neural Networks (RNNs)

RNNs were the first neural architectures designed to handle sequential data. However, they suffer from vanishing gradient problems, making them inefficient for capturing long-range dependencies (Hochreiter & Schmidhuber, 1997). Recurrent Neural Networks (RNNs) extend traditional feed-forward architectures by introducing cyclic connections that enable information to persist across time steps (Elman, 1990). In traffic forecasting, simple RNNs have been used to model short-term dependencies in highway speed series (Lv et al., 2015). Yet, RNNs suffer from vanishing and exploding gradient issues when learning long-range dependencies

(Hochreiter & Schmidhuber, 1997), which limits their effectiveness in settings where congestion patterns evolve over many hours.

2.2.2 Long Short-term Memory (LSTM) and Gated Recurrent Units (GRU)

Recurrent Neural Networks (RNNs) extend standard feed-forward architectures by incorporating cyclic connections that propagate information across time steps, making them a natural choice for sequential forecasting tasks such as traffic volume prediction. However, in practice RNNs struggle to learn dependencies over longer horizons due to the vanishing gradient problem (Hochreiter & Schmidhuber, 1997), which limits their ability to model congestion buildup and dissipation that unfold over many hours. Long Short-Term Memory networks (LSTMs) overcome this limitation by introducing gated memory cells input, output, and forget gates that regulate the flow of information through time and enable the network to retain relevant patterns over extended sequences (Figure 2) (Hochreiter & Schmidhuber, 1997). LSTMs have achieved state-of-the-art accuracy in both traffic volume and speed forecasting, effectively capturing peak and off-peak dynamics in urban and freeway contexts (Ma et al., 2015; Zhang et al., 2018). Gated Recurrent Units (GRUs), a streamlined variant of the LSTM cell that combines forget and input gating into a single update gate, offer comparable performance with fewer parameters and faster training times (Cho et al., 2014; Wu et al., 2017). Despite their strong predictive capabilities, both LSTM and GRU models operate as black boxes, providing little transparency into which temporal or contextual features most influence their forecasts.

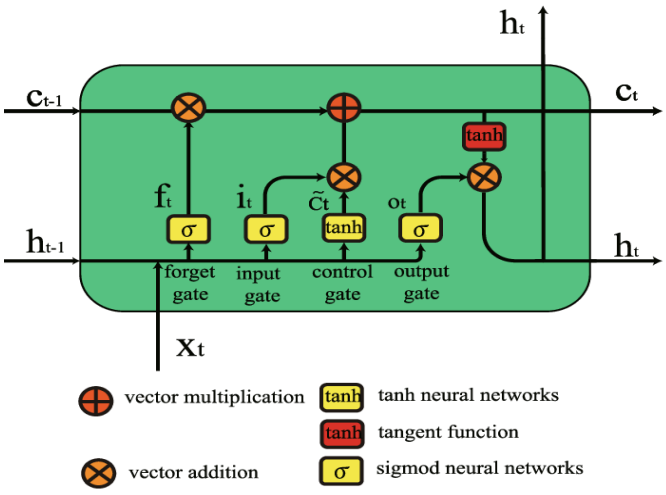

Figure 2: Structure of an LSTM cell.

Source: (Luo et al., 2021)

GRUs are a simplified version of LSTMs, combining the forget and input gates into a single update gate (Figure 3). This simplification reduces computational complexity while maintaining performance, making GRUs effective for real-time traffic prediction.

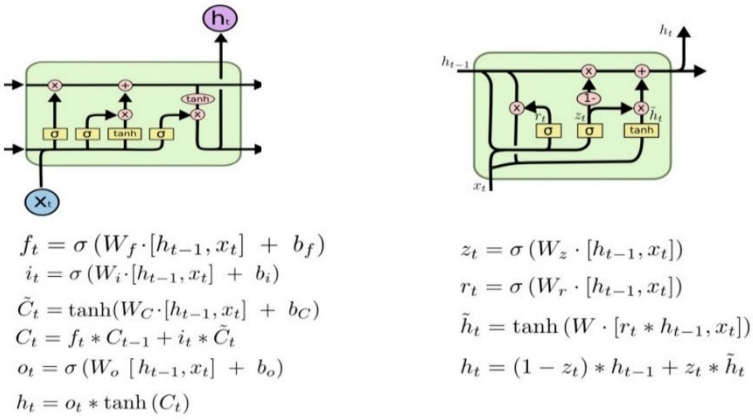

Figure 3: Comparison between LSTM and GRU architectures.

Source: (Marai et al., 2022)

2.2.3 Temporal Convolutional Networks (TCNs)

TCNs use causal convolutions and dilations to capture sequence data efficiently and have been tested in spatiotemporal applications. They offer better parallelization than RNNs but also lack built-in interpretability.

TCNs utilize dilated causal convolutions to model long-range dependencies in time series data (Figure 4). They offer advantages over RNNs, including parallel processing and stable gradients, making them suitable for traffic forecasting tasks.

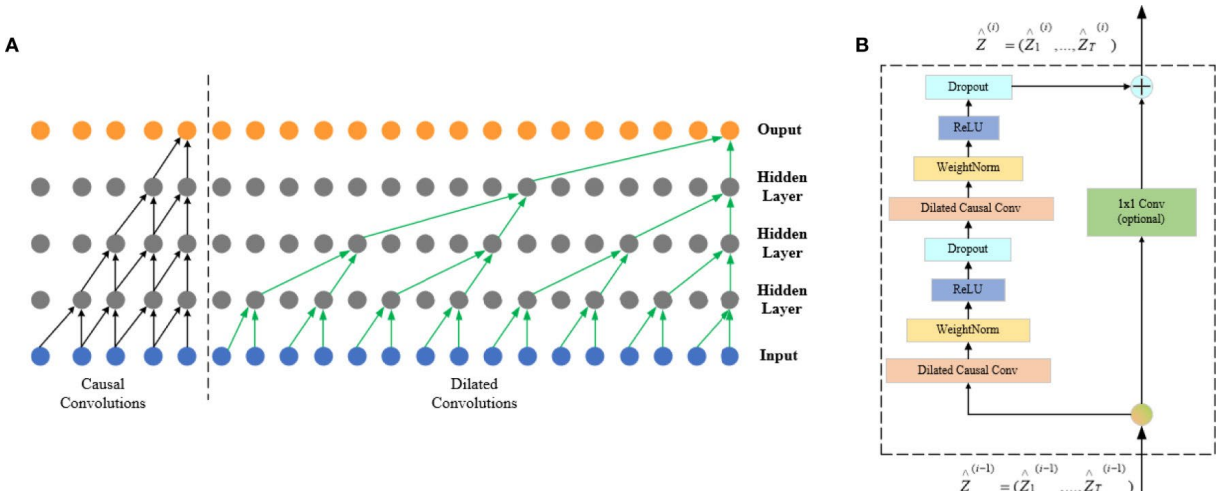

Figure 4: Structure of a Temporal Convolutional Network.

Source: (Shi et al., 2022)

2.3 Transformer-Based Models for Time Series

Transformers, originally developed for natural language processing, rely on self-attention to capture pairwise relationships across all time steps (Vaswani et al., 2017). Applied to time series, they handle long sequences without recurrence, enable parallel computation, and produce attention weights that highlight the most relevant input positions. Transformers have been adapted for time series forecasting due to their ability to model long-range dependencies using self-attention mechanisms (Figure 5). Unlike RNNs, Transformers process entire sequences simultaneously, enhancing computational efficiency and performance.

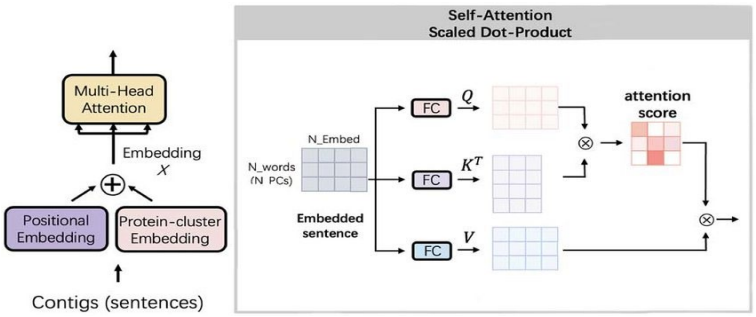

Figure 5: Self-attention mechanism in Transformer models.

Source: (Shang et al., 2022)

2.3.1 Advantages of transformers

- Handle long sequences without recurrence.
- Enable parallel computation.
- Provide attention weights that highlight key parts of the input.

2.3.2 Limitations in early transformers

Initial transformer models lacked capabilities for:

- Handling known future inputs (e.g., holidays, weather forecasts).
- Processing static features (e.g., road segment ID).
- Generating prediction intervals to express uncertainty.

These gaps motivated the development of more tailored architectures for time series forecasting.

2.4 Summary of the Literature Gap

While extant literature has demonstrated the effectiveness of both classical statistical techniques (e.g., ARIMA, Kalman filters) and modern deep-learning architectures (including RNNs, LSTMs, GRUs, TCNs, and even Transformer variants) for short-term traffic forecasting, no

study to date has brought these advances together in a unified framework tailored for multi-site, multi-horizon volume prediction in urban networks. Critically, the majority of prior work either:

a. focuses on a single location or sensor,
b. limits prediction to one-step-ahead horizons,
c. treats complex models as black boxes, providing little insight into which temporal or contextual factors drive their forecasts.

This thesis addresses that gap by:

1. Systematically tuning a TFT across seven distinct hyperparameter configurations for simultaneous forecasting at dozens of intersections.
2. Extending its gaze to six-step (six-hour) ahead horizons.
3. Exposing its internal decision logic through attention- and variable-selection heatmaps thereby delivering both state-of-the-art accuracy and transparent interpretability in urban traffic volume forecasting.

3 Temporal Fusion Transformer

In this chapter, we present methods the key architectural components and mechanisms underpinning the TFT. Each subsection introduces one of the major building blocks, accompanied by a numbered figure illustrating its structure. The TFT is a specialized architecture designed for interpretable multi-horizon time series forecasting. TFT integrates several components to handle complex temporal patterns and provide insights into model decisions.

Developed by Lim et al. (2021), the TFT addresses the limitations of earlier models. It is designed for interpretable multi-step time series forecasting, supporting a wide range of data types and forecasting horizons.

Key features:

- Variable selection networks: Dynamically select relevant features for each time step.
- Static covariate encoders: Process unchanging inputs like location or sensor ID.
- Gated residual networks: Improve learning capacity while controlling overfitting.
- Self-attention layers: Identify long-term dependencies.
- Prediction intervals: Offer probabilistic forecasts, not just point estimates.
- Interpretability: Attention and feature importance values show which inputs contributed most.

TFT has demonstrated state-of-the-art performance in applications ranging from energy demand to financial time series, with emerging evidence of its superiority over LSTM and ARIMA in freeway speed forecasting (Zou et al., 2022). Crucially, TFT's built-in interpretability supports transparent decision making and fosters trust in operational deployments.

Smart cities rely on intelligent transportation systems (ITS) that integrate data from various sources to optimize traffic flow, reduce congestion, and enhance safety. Real-time forecasting plays a key role in:

- Adaptive traffic signal control.
- Dynamic route guidance.
- Emergency response planning.

TFT's ability to combine static, dynamic, and contextual data aligns well with the multi-source nature of smart city datasets. Its interpretability also supports data transparency, which is critical for public trust and policy accountability.

Study by Zou et al. (2022) show that TFT outperforms LSTM and ARIMA in freeway speed prediction, while also offering explainable outputs, an essential benefit for deployment in operational environments.

3.1 Key components of TFT

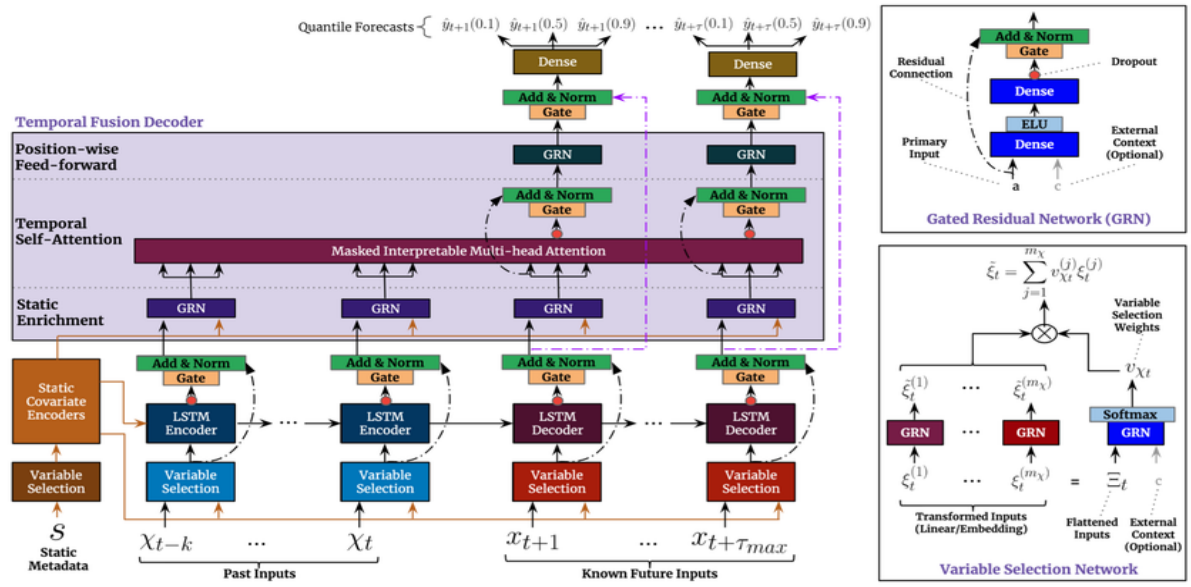

Figure 6:Architecture of the Temporal Fusion Transformer.

Source: (Srivastava et al., 2022)

The TFT architecture (Figure 6) incorporates a suite of advanced components designed to address the challenges of multivariate time series forecasting in complex environments like smart cities. These components ensure the model's ability to extract meaningful patterns, manage heterogeneous input types, and offer interpretable predictions. Each module within the TFT plays a distinct and critical role in delivering both performance and insight.

a) Variable Selection Networks (VSNs)

Variable Selection Networks are essential to the TFT's capacity for adaptive modeling. Unlike traditional methods that process all input variables equally, VSNs dynamically learn the relevance of each variable at each time step. This allows the model to filter out noise and emphasize features that contribute significantly to accurate predictions.

The selection process operates over three input types: static features (such as sensor location), past time-varying features (e.g., previous traffic volumes), and known future features (e.g., weather forecasts or holidays). Each type is passed through an embedding or encoding layer and processed using a Gated Residual Network. The results are combined using a softmax function to produce variable importance weights, guiding the model toward the most informative signals. This not only improves forecasting accuracy but also provides interpretability by highlighting which inputs matter most in different contexts.

b) Gated Residual Networks (GRNs)

GRNs form the computational backbone of several TFT submodules. They support nonlinear transformations of input data while preserving the flow of relevant information through skip connections and gating mechanisms. The gating system allows the model to modulate how much new information is introduced at each layer and how much of the previous context should be retained.

GRNs are particularly effective in modeling complex relationships among input variables, such as interactions between time of day, road type, and weather conditions in traffic systems. Their inclusion improves gradient flow during training and helps stabilize learning in deep architectures.

c) Static Covariate Encoders

Static covariate encoders process features that remain constant throughout the prediction horizon, such as geographic location or road classification. These encoders produce context vectors that condition the entire model architecture. The encoded static information is used to tailor both the variable selection and temporal attention processes, enabling the TFT to account for differences in behaviour across locations or environments.

For instance, traffic dynamics on a rural road may differ significantly from those in an urban intersection. By embedding this information early in the model, TFT can customize its forecasting logic based on context.

d) Temporal Self-Attention Mechanism

The self-attention mechanism in TFT enables the model to capture long-term dependencies in time series data. Unlike recurrent architectures that process data sequentially, self-attention allows the model to consider all time steps in parallel, assigning weights to past inputs based on their relevance to the current prediction.

In traffic forecasting, self-attention is particularly beneficial in identifying patterns like rush hour cycles or recurring congestion periods. This mechanism enhances the model's ability to incorporate distant but relevant temporal signals, contributing to improved long-horizon forecasts.

e) Prediction Intervals

TFT is designed not only to produce point estimates but also to quantify uncertainty through prediction intervals. This is achieved using Quantile Loss, which enables the model to learn different quantiles of the output distribution. The result is a range of plausible future values rather than a single prediction.

Prediction intervals are vital for traffic management in smart cities, where uncertainty is inherent due to unpredictable events such as accidents, sudden weather changes, or public gatherings. By providing upper and lower bounds alongside median predictions, TFT supports risk-aware decision-making and more resilient urban planning.

3.2 Application of TFT in Smart City Traffic Management

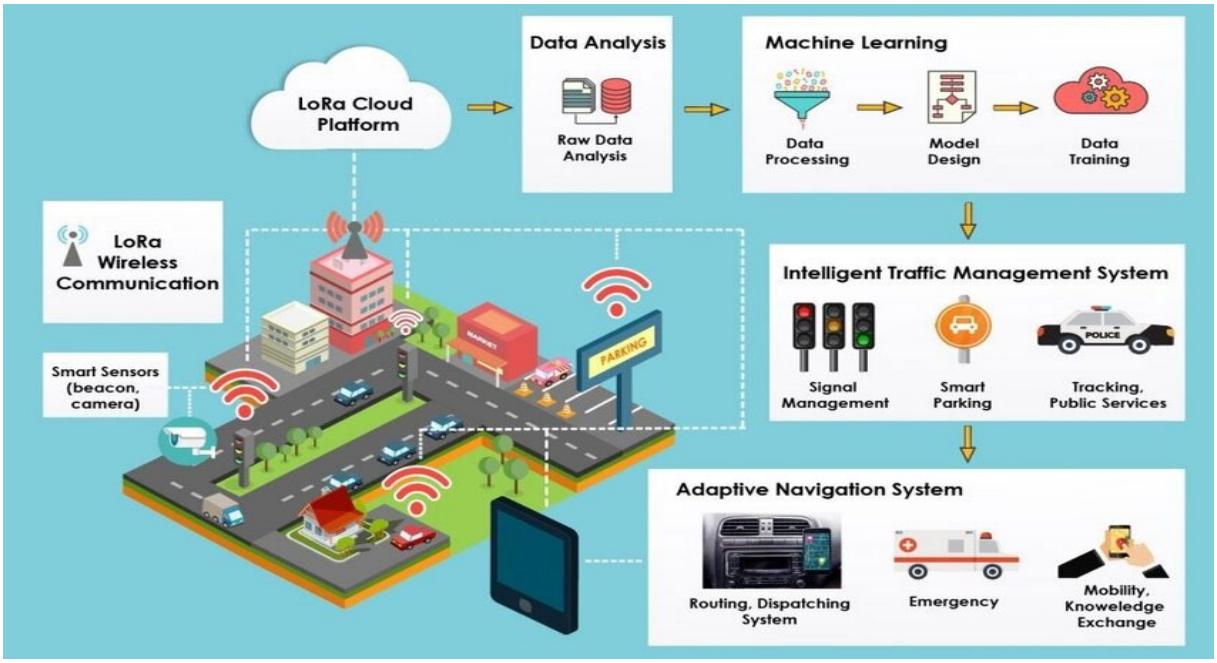

Figure 7: Smart traffic control architecture using AI.

Source: (Goenawan et al., 2024)

Firstly, TFT enables real-time traffic prediction, which supports the dynamic regulation of traffic signals and congestion control measures (Figure 7). This allows city planners and traffic

management systems to respond proactively to fluctuations in traffic volume, thereby improving flow and reducing delays.

Secondly, the model seamlessly integrates with Internet of Things (IoT) devices, such as traffic cameras, road sensors, and connected vehicles. These devices provide continuous streams of data including vehicle counts, speeds and environmental conditions that TFT can leverage for more accurate and context-aware forecasting. This integration ensures that the predictive system remains informed and responsive to the real-world dynamics of urban roadways.

Lastly, TFT exhibits strong scalability, making it suitable for large-scale deployment in growing urban environments. As cities expand and the complexity of traffic systems increases, TFT can accommodate higher volumes of data and maintain performance across diverse traffic scenarios. This makes it a robust choice for future-oriented, data-driven smart city ecosystems.

3.3 Data Preprocessing for TFT

The effective deployment of the TFT model for traffic flow prediction relies on the availability and careful preparation of diverse data types (Figure 8). These inputs ensure that the model captures both the temporal dynamics and contextual factors that influence urban traffic behaviour.

Firstly, historical traffic data forms the backbone of the TFT model. This includes time-stamped records of traffic volume, speed, and occupancy for specific road segments or intersections. Such data allows the model to learn temporal patterns, such as rush hour peaks, weekend lulls, and seasonal fluctuations in traffic flow.

Secondly, the model benefits from static features, which remain constant over time. These include structural attributes such as road width, number of lanes, speed limits, and the geographic location of sensors. Incorporating these variables helps the model contextualize traffic behaviour based on physical infrastructure and location-specific characteristics.

In addition, exogenous variables are factors that originate outside the traffic system and are essential for improving the accuracy of traffic flow predictions. These variables include weather conditions such as rainfall, temperature, and visibility, along with external influences like public events (for example, concerts or sports games) and holidays. Such factors can significantly alter normal traffic patterns by causing sudden changes in vehicle flow. Incorporating these variables into the model allows it to make more accurate predictions by accounting for disruptions and unusual conditions.

Finally, data preprocessing is essential to ensure data quality and compatibility with deep learning workflows. This process involves several key steps: handling missing values through interpolation or imputation techniques, normalizing numerical features to bring them to a comparable scale, and encoding categorical variables (e.g., day of the week, location ID) using techniques such as one-hot encoding or embeddings. Proper preprocessing ensures that the TFT model receives clean, structured input conducive to efficient training and accurate forecasting.

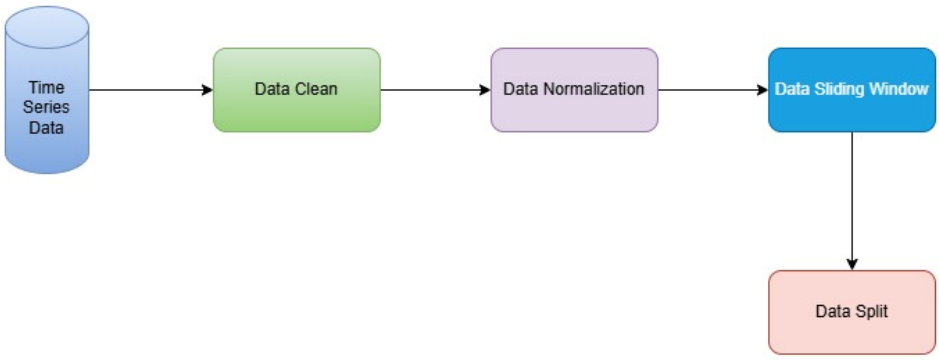

Figure 8:Data preprocessing pipeline for time series data.

Source: Author's own work.

3.4 Mathematical Formulation of TFT

Below, we detail each component of TFT in a mathematically rigorous way, referring to established literature for architectural choices and loss functions (Lim et al., 2020; Wen et al., 2017).

3.4.1 Variable Selection Networks

Since the relative importance of different covariates can vary both across samples and across time steps, TFT uses a variable selection network to learn adaptive weights for each input feature at every time step. Let there be m time varying covariate embeddings at time step t, denoted by $\{z_t^{(1)}, \dots, z_t^{(m)}\}$. Concatenate them into a single vector

$$z_t = \left[z_t^{(1)};\ z_t^{(2)};\ \dots;\ z_t^{(m)}\right] \in R^{m \times d}, \tag{1}$$

where d is embedding dimension for each feature. A gated residual network (GRN) block with context c_s (derived from static covariates) produces raw importance scores

$$v_t = \text{Softmax}\big(\text{GRN}(z_t, c_s)\big) = \left[v_t^{(1)}, v_t^{(2)}, \dots, v_t^{(m)}\right]^T. \quad (2)$$

Subject to $\sum_{i=1}^{m} v_t^{(i)} = 1$. Each component $v_t^{(i)}$ nonnegative and indicates the relative importance of feature i at time t. The selected representation at time *t* is then

$$z_t = \sum_{i=1}^{m} v_t^{(i)} z_t^{(i)}. \quad (3)$$

By dynamically adjusting $\{v_t^{(i)}\}$, the network can focus on the most relevant inputs and suppress noise from irrelevant variables (Lim et al., 2020; Arik & Pfister, 2019).

3.4.2 Gated Residual Network (GRN)

The gated residual network forms the core building block of TFT. Given an input vector $a \in R^d$ and an optional context vector $c \in R^d$, a GRN computes

$$u = \sigma(\mathrm{W}_4[a; c] + b_3) \in R^d, \quad (4)$$

$$g = \text{ELU}(W_1 a + W_2 c + b_1) \in R^d, \quad (5)$$

$$h = W_3 g + b_2 \in R^d, \quad (6)$$

$$GRN(a, c) = \text{LayerNorm}(a + u \odot h) \in R^d, \quad (7)$$

where $W_1, W_2, W_3, W_4 \in R^{d \times d}$ are weight matrices, $b_1, b_2, b_3 \in R^d$ are bias vectors, $\sigma(\cdot)$ is the sigmoid activation, and " $\odot$ " denotes elementwise multiplication. The gating vector u allows the network to modulate how much of the residual component *h* is used verses preserving the original input a. Layer normalization is applied to the sum to stabilize training. When no context c is provided, one simply sets $c = 0$. GRNs are used throughout TFT for feature transformation, variable selection scoring, static encoding, feed-forward layers, and gating in the self-attention block (Lim et al., 2020).

3.4.3 Static Covariate Encoding

Static covariates s (for example, lane count and geographic zone) are encoded into one or mosre context vectors that influence multiple TFT layers. Denote the static input dimension by d_s. A static encoder projects s into four distinct context vectors of dimension d_{model}:

$$c_s = \text{GRN}_s(s), \qquad (for\ variable\ selection) \quad (8)$$

$$c_h = \text{GRN}_h(s), \qquad (to\ initialize\ LSTM\ hidden\ state) \quad (9)$$

$$c_c = \text{GNR}_c(s), \qquad (for\ local\ temporal\ processing) \quad (10)$$

$$c_e = \mathrm{GRN}_e(s). \qquad (for\ static\ enrichment\ before\ attention). \qquad (11)$$

The resulting context vectors c_s, c_h, c_c, c_e modulate the variable selection, initialize LSTM states, and condition subsequent layers, thereby enabling site-specific dynamics to be captured throughout the network (Lim et al., 2020).

3.4.4 Local Processing via Sequence-to-Sequence LSTM

To capture short-term temporal patterns such as traffic surges due to a nearby event or the decay of congestion spikes, TFT employs a sequence-to-sequence long short-term memory (LSTM) layer. The procedure is as follows:

1. **Prepare input sequence**: At a forecast origin t, form a unified sequence of length $L + H$ by concatenating:
   - For indices $n \in -L + 1, \ldots, 0$, use the selected covariates vector z_{t+n} (which represents either past observed covariates or past model outputs).
   - For indices $n \in 1, \ldots, H$, use the encoded known-in-andvance covariate k_{t+n}.
2. **LSTM encoder**: Feed the sequence $\{z_{t-L+1}, \ldots, z_t\}$ of length L into an LSTM encoder whose initial hidden and cell states are set to c_h (from Section 4.3.3). The encoder processes all L time steps, producing a final hidden state h_t^{enc}.
3. **LSTAM decoder**: Initialize a separate LSTM decoder at time step t using hidden state h_t^{enc}. The decoder then consumes the sequence $\{k_{t+1}, \ldots, k_{t+H}\}$ of length H. Each step n of the decoder outputs a hidden representation $h_t^{\mathrm{dec},(n)}$.
4. **Gated skip connection**: For each position $n \in \{-\mathrm{L}+1, \ldots, \mathrm{H}\}$, merge the LSTM output with the original input $z_t^{(n)}$ via a gated residual network:

$$\boldsymbol{h_t'^{(n)}} = \mathrm{LayerNorm}\left(\boldsymbol{z_t^{(n)}} + \mathrm{GLU}\left(\boldsymbol{h_t^{\mathrm{lt},(n)}}\right)\right) \qquad (12)$$

   where GLU is a gated linear unit (Dauphin et al., 2017). This skip connection preserves original input information and allows the LSTM to apply nonlinear corrections only where needed.

The result of this sequence-to-sequence layer is a set of locally enhanced temporal features $\{h_t'^{(-L+1)}, \ldots, h_t'^{(H)}\}$, each of the dimension d_{model}. These features encode short-term dependencies in both past and known future inputs (Lim et al., 2020).

3.4.5 Static Enrichment Layer

While the local LSTM layer captures short-range temporal dynamics, static site attributes may modify how these dynamics unfold. Therefore, each enhanced temporal feature $h_t'^{(n)}$ is further processed by a static enrichment GRN, which integrates the static context vector c_e:

$$e_t^{(n)} = \mathrm{GRN}_{\mathrm{enrich}}\left(h_t'^{(n)}, c_e\right), \quad n \in \{-L+1, \ldots, H\}. \tag{13}$$

This produces static-enriched temporal features $\{e_t^{(-L+1)}, \ldots, e_t^{(H)}\}$. By conditioning on c_e, the network can learn how site-specific characteristics (for example, number of lanes or typical congestion patterns) modulate short-term dynamics (Lim et al., 2020).

3.4.6 Interpretable Self-Attention Layer

To capture long-range temporal dependencies such as weekly traffic cycles, holiday effects, or delayed surge propagation, TFT employs a multi-head self-attention mechanism over the static-enriched temporal features. Unlike canonical Transformer self-attention, TFT's interpretable multi-head attention shares the same value projection across all heads and aggregates attention scores additively, ensuring that importance weights remain interpretable.

Formally, let $N = L + H$, and stack the static-enriched vectors into a matrix:

$$E_t = \left[e_t^{(-L+1)}, e_t^{(-L+2)}, \ldots, e_t^{(H)}\right] \in R^{N \times d}. \tag{14}$$

For each of m heads, define learned projections $W_Q^h, W_K^h \in \mathbb{R}^{d_{model} \times d_{attn}}$ and a shared value projection $W_V \in \mathbb{R}^{d_{model} \times d_{model}}$. Compute:

1. Keys and queries for head *h*

$$Q^h = E_t W_Q^h, \quad K^h = E_t W_K^h \tag{15}$$

2. Scaled dot-product attention for each head *h*

$$A^h = softmax\left(\frac{\left(Q^h (K^h)^T\right)}{} sqrt(d_{attn})\right) \in R^{N \times N}. \tag{16}$$

3. Apply shared value projection:

$$V = E_t W_V \in R^{N \times d_{model}} \tag{17}$$

4. Head-wise output:

$$H^h = A^h V \in R^{N \times d_{model}} \tag{18}$$

5. Aggregate Across Heads and Final Projection:

$$H = \sum_{\{h=1\}}^{\{m\}} H^{(h)},\ Y_t = H\,W_O \in R^{N\,x\,d_{model}}, \quad (19)$$

where $W_O \in R^{d_{model}x\,d_{model}}$ is a learned output projection.

6. Gated skip connection and layer normalization:

   For each time position $n \in \{-L+1, \ldots, H\}$, merge the self-attention output with the input $e_t^{(n)}$ via

   $$a_t^n = LayerNorm\big(e_t^n + GLU(Y_t^n)\big), \quad (20)$$

   where $Y_t^{(n)}$ is the nth row of Y_t and GLU denotes a gated linear unit (Dauphin et al., 2017).

The result $\{a_t^{-L+1}, \ldots, a_t^{(H)}\}$ constitutes the set of attention-enhanced temporal features. Because heads share value projections, attention scores remain interpretable: one can examine $A^{(h)}$ to see which past or future positions influenced each prediction (Lim et al., 2020).

3.4.7 Position-Wise Feed-Forward and Output Layers

Following the attention layer, TFT applies an additional position-wise feed-forward network—another gated residual network (GRN) to each $a_t^{(n)}$. Specifically:

$$f_t^n = GRN_{ff(a_t^n)},\ n \backslash in \{-L+1, \ldots, H\}. \quad (21)$$

A final gated skip connection then merges $a_t^{(n)}$ with $f_t^{(n)}$ producing $h_t^{(n)} \in R^{d_{model}}$:

$$h_t^n = LayerNorm\big(a_t^n + GLU(f_t^n)\big). \quad (22)$$

For positions corresponding to the forecast horizon $(n = 1, \ldots, H)$, each $h_t^{(n)}$ is passed through a linear quantile output head to produce a set of quantile predictions:

$$\{y\}_{\{t+n\}}^q = W_q h_t^n + b_q, \qquad q \in \{0.1, 0.5, 0.9\}, \quad (23)$$

where $W_q \in R^{\{1\,\times d_{model}\}}$ and $b_q \in R$ are learned parameters for the qth quantile. When training with quantile loss, the model simultaneously optimizes all target quantile using the following loss for a single sample at horizon n:

$$L_{q(y,\hat{y})} = \max\big(q(y - \hat{y}), (q - 1)(y - \hat{y})\big). \quad (24)$$

Summing this quantile loss over all horizons yields the overall training objective (Wen et al., 2017).

4 Data and Methods

In this chapter, we present a comprehensive account of the data sources, feature-engineering steps, and modeling framework used to build and assess a multi-horizon traffic forecasting model for a smart-city pilot. The chapter is structured into four principal sections:

i. **Dataset description and preprocessing**

We begin by detailing the raw traffic data, its origin, structure and contents. We then explain how each field is transformed (for example, by assigning a time index, extracting calendar features, and encoding categories) so that the dataset is ready for forecasting. This section clarifies exactly which measurements were collected (hourly volumes, lane counts, sub-hourly distributions, lagged covariates, etc.) and how missing or out-of-scope entries are handled.

ii. **Problem formulation and baseline model**

Next, we formally state the forecasting objective given a rolling window of the previous L hours of traffic and auxiliary covariates (both past-observed and future-known), predict the next H hours of volume at each location. We then introduce a simple "persistence" baseline, which repeats the most recent observed value across the full forecast horizon, as a reference point that any advanced model must outperform.

iii. **TFT methodology**

We describe the architecture of the TFT in depth, covering its key components, variable selection networks, gated residual blocks, static covariate encoders, masked multi-head self-attention, and quantile output layers. We then explain how each block fits together to produce interpretable, high-performance multi-step forecasts. In particular, we clarify how static, past-observed, and future-known inputs are combined, how the model dynamically weights time-varying features at each look-back step, and how the final decoder generates a full distribution of future values (for example, 10th, 50th, and 90th percentiles) at each of the H future time steps.

iv. **Evaluation strategy and performance metrics**

Finally, we outline our end-to-end training and evaluation pipeline: splitting the data into training, validation, and test segments; constructing overlapping sequences of length $L + H$; training with early stopping based on validation quantile loss; and then measuring performance on held-out data. We define and

explain the use of MAE and RMSE as our primary metrics. We also describe how loss curves (epoch-wise training versus validation) and location-wise actual-versus-predicted plots are generated to assess convergence behaviour and spatial forecasting accuracy.

By the end of this chapter, the reader will understand:

- exactly which raw data fields are available and how they are transformed into model-ready inputs,
- the precise mathematical objective (forecasting H future steps given L past observations),
- the rationale for including a simple persistence baseline,
- the inner workings of the TFT; why each block exists and how it contributes to interpretability and accuracy,
- and the concrete steps we follow to split, train, and evaluate models, including which error metrics are computed and why.

The detailed model-training code itself appears in the Appendix B; this chapter focuses on the conceptual and procedural foundations of our approach without exposing low-level implementation details.

4.1 Dataset Description and Preprocessing

We assembled an hourly traffic-volume database by collecting counts from multiple detector sites deployed throughout our smart-city pilot network. Each observation includes a static site identifier (for example, an intersection or freeway segment) and the number of parallel lanes at that location. To capture fine-grained within-day variability, we record twenty-four sub-hourly distribution flags; one for each hour block which can represent, for example, the fraction of vehicles passing during each fifteen-minute interval. We then derive a full timestamp by adding the elapsed-hours index to a fixed reference date. From that timestamp we extract calendar covariates such as day of week, a binary weekend indicator, the hour block itself, a coarser time-of-day category (Night for hours 0 to 5, Morning for 6 to 11, Afternoon for 12 to 17, and Evening for 18 to 23), the calendar month, and a mapped season label (Winter, Spring, Summer, or Fall) (Lim et al., 2020; Wen et al., 2017). In order to provide the model with recent context, we also include up to ten past-observed measurements of traffic volume and any additional sensor readings that are only known at or before the current hour.

Prior to model fitting we apply a four-step preprocessing pipeline designed to guarantee that each input sequence of length L plus H contains exactly those covariates legitimately available at forecast time. First, we sort all records by site and elapsed-hours and then re-index each group to produce a "relative time" counter running from zero to T for every location. Second, we convert this counter into a true datetime and derive all calendar covariates so that future positions in the forecast horizon correctly incorporate known information. Third, we convert all categorical fields: site identifier, time-of-day category, and season into integer codes for learned embedding vectors. Finally, we remove any training rows with missing traffic values to avoid introducing incomplete cases. By strictly separating static inputs, known-future calendar features, and past-observed measurements, we ensure that no data beyond the forecast origin can influence the predictions, thereby preserving the validity of our multi-hour evaluation.

4.1.1 Primary Traffic Dataset

The principal dataset for this study consists of hourly traffic volume measurements collected from multiple detector locations across a smart-city pilot infrastructure. Each record in the dataset captures:

- A categorical site identifier, indicating the physical location (for example, an intersection or freeway segment).
- An integer index representing the number of hours elapsed since a fixed reference date.
- An integer count of parallel lanes or road segments at that location.
- Twenty-four numeric values corresponding to sub-hourly intervals (or equivalently, dummy flags for each hour block). These features may indicate, for instance, the proportion of vehicles in each 15-minute subinterval.
- Thirty-three additional time-varying covariates covering ten lagged traffic volumes and up to 23 external/exogenous measurements (such as upstream sensor readings, recent traffic speeds, or related environmental variables).

A separate file contains an identically structured table (except without the true future traffic volumes) and is reserved for final model evaluation (Figure 9). The traffic-volume dataset employed in this study was originally drawn from the UCI Traffic Flow Forecasting repository and made available on Kaggle under the title "Urban Traffic Flow." It comprises two comma-delimited files: **train.csv**, which spans 1,261 timesteps of fifteen-minute aggregated

measurements across 36 detector sites, and **test.csv**, which spans a further 840 timesteps on the same locations. Each record in **train.csv** contains the following fields:

- **timestep**: an integer index denoting the sequential fifteen-minute interval;
- **location**: a categorical identifier for one of the 36 monitored sites;
- **traffic**: the normalized traffic rate at that interval (a real value in [0, 1]);
- **prev_i** (for i = 1…10): the i-th lagged traffic measurement, providing the immediately preceding ten intervals of observed flow.
- **hour_i** (for i = 1…24): a one-hot encoding of the current hour of day.
- **feata_i**, **featb_i**, **featc_i**: additional one-hot–like indicators (binary flags whose column sums suggest they encode categorical attributes), which may represent external factors or sensor-specific states.

The **test.csv** file contains the same structure aside from the true **traffic** values beyond the training horizon; it is reserved for final evaluation. By combining immediate lagged observations, explicit calendar encodings, and auxiliary binary flags, this design furnishes the TFT with both the recent temporal context and static metadata necessary for robust multi-horizon forecasting while avoiding any leakage of future information. The feature design reflects the typical inputs used in multi-horizon forecasting (Lim et al., 2020; Wen et al., 2017).

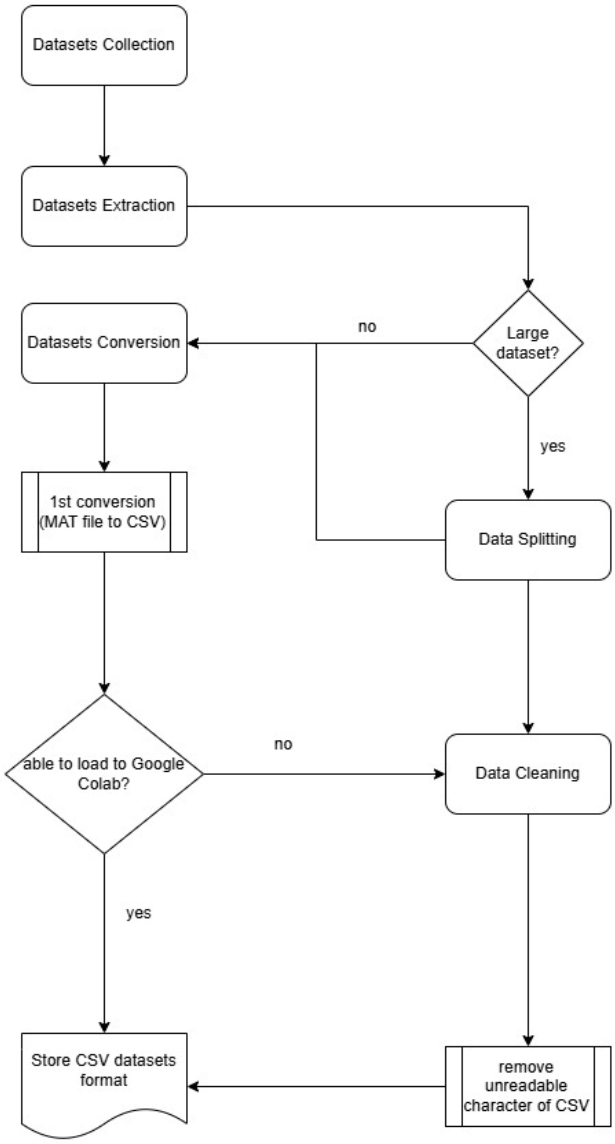

Figure 9: Data collection and preprocessing flowchart.

Source: Author's own work.

4.1.2 Feature Engineering Procedures

Before any forecasting model is trained, each raw traffic record is passed through a multi-step preprocessing pipeline to generate the final inputs. These steps, described below, ensure that the model has access to three distinct categories of information: static covariates that remain constant for a given monitoring site, time-varying covariates that are known at prediction time and time-varying covariates that are only observed up to the current time step.

1. **Ordering and creation of a relative-time index**

   First, all observations are sorted by the detector site identifier and then by the elapsed-hours index (i.e., the number of hours that have passed since a fixed start date). Once sorted, a new "relative-time" index is generated for each site independently. Concretely,

within each site group, observations are cumulatively counted so that the earliest record for that location receives an index of zero, the next record receives an index of one, and so forth. This relative-time index serves two purposes: it provides a consistent, zero-based sequence length for encoder inputs, and it decouples the model from absolute timestamps (thereby simplifying the handling of sites that begin reporting data at different calendar dates) (Lim et al., 2020).

2. **Construction of an explicit timestamp and extraction of temporal covariates**

   Using the elapsed-hours index, each record's timestamp is reconstructed by adding that many hours to a predefined reference date (for example, January 1, 2025). This precise timestamp then allows us to derive several calendar-based features:

   - Day of the week: By taking the weekday component of the timestamp, we assign a numeric code from 0 (Monday) through 6 (Sunday). This variable helps the model learn weekly traffic patterns (Zhang et al., 2017).
   - Weekend indicator: From the day-of-week code, we create a binary flag that equals 1 if the date falls on Saturday or Sunday, and 0 otherwise. This binary variable captures the distinct travel behaviours typically observed on weekends (Gupta et al., 2019).
   - Hour-block assignment: Each raw record includes twenty-four sub-hour features labeled "hour_1" through "hour_24." These features might indicate, for example, relative proportions of traffic observed within each successive hour or may simply serve as one-hot encodings of the hour of day. To determine a single "hour-block" value, we compute the column index where the arg-max occurs among these twenty-four features. For instance, if "hour_4" has the largest value in a given row, that row's hour-block is set to 4. This procedure condenses the sub-hourly representation into a single integer in $\{1, 2, \dots, 24\}$, indicating the hour of day (Arik & Pfister, 2019).
   - Time-period category: After assigning an "hour-block" in {1, …, 24}, we map this integer into a broader "time-period" category to capture general diurnal patterns. Specifically, hour-blocks 1–6 are labeled "Night," 7–12 are labeled "Morning," 13–18 are labeled "Afternoon," and 19–24 are labeled "Evening." These four coarse-grained periods allow the model to learn, for example, that "Morning" traffic follows different dynamics than "Night" traffic (Yu et al., 2018).

- Month and season: From the reconstructed timestamp, the calendar month is extracted as an integer in {1, …, 12}. That month integer is then mapped to one of four season categories: "Winter," "Spring," "Summer," and "Fall" by using a standard mapping (e.g., December, January, February → Winter; March, April, May → Spring; June, July, August → Summer; September, October, November → Fall). This seasonal indicator helps the model capture broad shifts in traffic patterns associated with weather changes, daylight variation, and school or holiday schedules (Chen & Wang, 2020).

3. **Categorical encoding of derived features**

   Once the new variables (site identifier, time-period, and season) have been generated, we convert each of these into a categorical type. In practice, this means assigning each distinct site identifier a unique category code, doing the same for each of the four time-period labels, and doing likewise for the four-season labels. Downstream, these categorical variables can be embedded into learned vectors during model training. Encoding them as categories (rather than leaving them as strings) enables efficient vector-lookup embeddings within the TFT, which in turn allows the model to learn site-specific and period-specific patterns (Lim et al., 2020).

4. **Removal of missing-traffic rationale**

   Because our target variable traffic volume is only defined for historical observations (i.e., "train.csv"), any row where the traffic volume is missing must be excluded before constructing the model inputs. In other words, if a training file record has no observed traffic value, we simply drop that row. This ensures that all remaining rows in the training set have valid labels. In contrast, the test file ("test.csv") intentionally omits future traffic values, so those missing labels are expected; we do not drop any test rows (Wen et al., 2017).

5. **Summary of the preprocessing rationale**

   Upon completion of the multi-stage preprocessing pipeline, each traffic observation is enriched with the following sets of features, which collectively enable the model to learn both site-specific patterns and temporal dynamics:

   i. **Static site identifier**

      Each record retains a categorical identifier representing the detector site. Because this identifier does not change over time, it serves as a static covariate that the model can embed to learn location-specific characteristics (Lim et al., 2020). For example, one site may consistently experience higher baseline

volumes owing to its proximity to a major interchange, while another site may systematically record lower volumes due to local land-use patterns.

ii. Relative-time index

Within each site group, observations are assigned a zero-based "relative-time" index, indicating the number of hourly measurements that have elapsed since that site first began reporting traffic. By using a site-specific count rather than an absolute timestamp, the model can learn recurrent patterns (for instance, daily peaks and off-peaks) in a uniform index scale, even if different sites started collecting data on different calendar dates (Wen et al., 2017).

iii. Reconstructed hourly timestamp

From the original elapsed-hours field, each record's actual timestamp (at hour resolution) is reconstructed by adding the elapsed-hours count to a common reference date (e.g., January 1, 2025). This explicit timestamp is used solely to derive calendar covariates (below); the forecasting model itself relies on relative time indices to avoid overfitting to specific dates.

iv. Time-known covariates

The reconstructed timestamp permits extraction of several calendar-based features that are fully known for any future hour (because the calendar is predetermined). These include:

- Day of week (integer code 0 = Monday through 6 = Sunday), which captures weekly periodicity in travel demand (Gupta et al., 2019).
- Weekend flag, a binary indicator equal to 1 for Saturday and Sunday and 0 otherwise, which accounts for the lower or altered traffic patterns typically observed on weekends.
- Hour-block (an integer in {1,…,24}), obtained by taking the arg-max over the twenty-four sub-hour features. This feature directly encodes the hour of day (Arik & Pfister, 2019).
- Time-period category, a coarse grouping of the hour-block into one of four labels: "Night" (hours 1–6), "Morning" (7–12), "Afternoon" (13–18), and "Evening" (19–24) to capture broader diurnal regimes (Yu et al., 2018).
- Calendar month, an integer in {1,…,12}, which allows the model to learn monthly seasonality.

- Season, a categorical label mapping months to {“Winter,” “Spring,” “Summer,” “Fall”} according to standard meteorological definitions (Chen & Wang, 2020).

These covariates are fully determined by the calendar and are therefore available for each of the H future forecast hours; accordingly, they are classified as “time-varying known” inputs.

v. Categorical encodings for embedding

The static site identifier, time-period category, and season label are all converted to categorical variables. In the TFT, each distinct category is then mapped to a learned embedding vector. This embedding mechanism enables the model to learn, for instance, that “Afternoon” at a busy downtown detector differs systematically from “Afternoon” at a suburban detector (Lim et al., 2020).

vi. Lagged traffic covariates (past-observed inputs)

Within the look back window of length L, the model incorporates each site’s historical traffic volumes along with any additional lagged features (for example, prior sensor measurements or related exogenous readings). Because these covariates are only available up to and including the current time step, they are designated as “time varying unknown” inputs. By presenting the TFT with the sequence of the most recent H observations, the network can capture how short-term fluctuations, such as abrupt increases or decreases in volume, propagate into the subsequent H hour forecast horizon. In doing so, the model learns to recognize local temporal patterns (for example, the decay of a congestion spike) that influence future traffic behaviour (Wen et al., 2017; Lim et al., 2020).

vii. Additional exogenous covariates

If available, other external measurements such as weather variables (temperature, precipitation), public-event indicators, or holiday flags—are likewise split into “known in advance” (e.g., scheduled holiday) versus “observed up to now” (e.g., actual precipitation measured at the current hour). This separation ensures that the model does not inadvertently “peek” at future realizations of exogenous variables that would be unknown at prediction time.

In summary, this preprocessing pipeline guarantees that, when constructing each training sequence of length $L + H$, the TFT receives exactly:

- **Static information** (site identifier, lane count, etc.) that remains constant for each site over time.
- **Time-varying covariates known at forecast time** (calendar-derived features such as day of week, hour-block, and season) for all future forecast hours.
- **Time-varying covariates observed only in the past** (lagged traffic volumes, past sensor readings, etc.) for the L look-back hours.

By carefully structuring these inputs, the TFT is able to leverage both calendar-based periodic patterns and site-specific dynamics, thereby producing more accurate multi-horizon forecasts (Lim et al., 2020; Wen et al., 2017).

4.2 Problem Formulation and Baseline Model

4.2.1 Multi-Horizon Forecasting Objective

Formally, consider a single monitoring site within the smart-city network. Denote by y_t the observed traffic volume at time step t, where $t \in \{0, \dots, T\}$. In addition to these target variables, we distinguish between two categories of time-dependent covariates:

Past-observed covariates x_t - these are measurements (for example, real-time weather sensor readings or previous traffic speeds) that can only be known once time t has arrived. Consequently, they are unavailable for $t + 1$ through $t + H$ at forecast time.

Known-In-Advance Covariates k_t - these include any features whose future values can be determined prior to making a forecast; typical examples are calendar-based indicators (day of week, month, holiday flags) or forecasted weather conditions. By construction, $k_{t+\tau}$ is assumed to be available at time t for all $\tau \in \{1, \dots, H\}$.

Finally, let s represent the static covariates associated with the site (for instance, the number of lanes, geographic characteristics, or sensor type). Because s does not change over time, it serves to condition the forecasting model on site-specific attributes.

Our goal is to learn a mapping

$$f(\cdot): y_{t-L+1:t}, x_{t-L+1:t}, k_{t-L+1:t+H}, s \mapsto \hat{y}_{t+1:t+H} \ , \tag{25}$$

where the notation

$$y_{t-L+1:t} = y_{t-L+1}, \dots, y_t \tag{26}$$

denotes the sequence of the most recent L observed traffic volumes, and

$$x_{t-L+1:t} = x_{t-L+1}, \dots, x_t \tag{27}$$

represents the corresponding past-observed covariate vectors. Likewise,

$$k_{t-L+1:t+H} = k_{t-L+1}, \dots, k_t, k_{t+1}, \dots, k_{t+H} \tag{28}$$

collects known-in-advance covariates from time $t - L + 1$ through time $t + H$. Each $k_{t+\tau}$ is assumed to be available at forecast origin t for $\tau = 1, \dots, H$. Finally, s denotes the site's static feature vector. In this formulation:

- L is the look-back window length, indicating how many historical observations are used as inputs.
- H is the forecast horizon, representing the number of consecutive future steps to predict.

Thus, at each time t, the model produces the H-step vector of predicted volumes

$$\hat{y}_{t+1:t+H} = (\hat{y}_{t+1}, \dots, \hat{y}_{t+H}), \tag{29}$$

where

$$\hat{y}_{t+1:t+H} = f(y_{t-L+1:t}, x_{t-L+1:t}, k_{t-L+1:t+H}, s). \tag{30}$$

In practice, one typically selects $L = 36$ hours and $H = 6$ hours (Lim et al., 2020; Wen et al., 2017), although these values may be adjusted to accommodate data availability or specific operational requirements. By conditioning on the sequence of past targets and covariates up to time t, while also incorporating known future inputs $k_{t+1:t+H}$ and static site features s, the learned function f can capture both short-term temporal dependencies and longer-term calendar patterns.

4.2.2 Persistence Baseline Model

As a first benchmark, we implement a simple persistence or naïve forecast. At every forecast origin t, this baseline repeats the last observed volume y_t across the entire horizon of length H. Formally, for $\tau \in 1, \dots, H$the persistence forecast is defined by

$$\widehat{y_{t+\tau}^{(baseline)}} = y_t. \tag{31}$$

Hence, if $y_t = 100$, then $\widehat{y_{t+1}} = 100$, $\widehat{y_{t+2}} = 100$, …, $\widehat{y_{t+H}} = 100$. Although trivial, this baseline often captures short-term continuity when traffic volumes remain relatively stable for a few hours (Lim et al., 2020; Wen et al., 2017). Its performance, measured using standard error metrics, provides a minimum threshold that any advanced forecasting method should surpass.

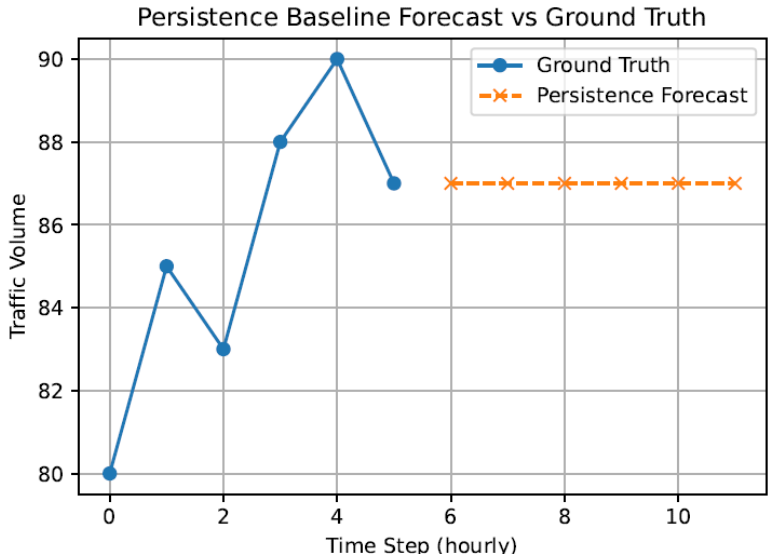

Figure 10: Example of the persistence baseline forecast over a six-hour horizon.

Source: Author's own work.

The above plot in Figure 10 shows how the naïve method simply repeats the last observed volume at each future step, providing a minimal benchmark that any advanced model should outperform.

4.3 Evaluation Strategy and Performance Metrics

To assess model performance, we partition the preprocessed data into three nonoverlapping sets. First, the training fold consists of all observations from the primary data file except the last six-hour block for each site; this fold is used to optimize model parameters via back-propagation and gradient-based updates. Second, the validation fold comprises the final six hours of observations for each site within the same data file. Performance on this fold measured by the average quantile loss is monitored continuously during training to guide the selection of hyperparameters (including the look-back length L, forecast horizon H, hidden state dimensions, learning rate, and dropout rate) and to trigger early stopping once improvements plateau. Finally, the test set is drawn in its entirety from the separate test.csv file, which remains untouched throughout model fitting and tuning. Only after the model and hyperparameters are fixed do we evaluate generalization error on this held-out set, reporting standard metrics such

as mean absolute error and root mean square error. This three-fold strategy ensures that reported performance reflects genuine out-of-sample accuracy rather than artifacts of model selection.

4.3.1 Performance Metrics

We employ the following standard metrics on both validation and test sets:

1. Mean Absolute Error (MAE)

$$MAE = \frac{1}{N}\sum_{\{i=1\}}^{N} | y_i - \hat{y}_i |, \tag{32}$$

where N is the total number of forecasted points across all sites and horizons.

2. Root Mean Squared Error (RMSE)

$$RMSE = \sqrt{\{ 1\backslash N \sum_{\{i=1\}}^{N} (y_i - \hat{y}_i)^2 \}}. \tag{33}$$

3. Quantile Loss (for Pq metrics)

For each quantile $q \in \{0.1, 0.5, 0.9\}$, the quantile loss is

$$QL_q = \left(\frac{1}{N}\right) \sum_{\{i=1\}}^{N} \max[\, q(y_i - \hat{y}_i), (q - 1)(y_i - \hat{y}_i)]. \tag{34}$$

These quantile losses measure model calibration and sharpness at different percentiles (Wen et al., 2017).

4.3.2 Interpretability Analyses

Beyond point estimates, TFT's architecture supports several interpretability analyses that provide insights into model behaviour:

1. **Attention weight visualization**

   By plotting the attention score matrix $A \in R^{N \times N}$ for a particular forecast origin, one can identify which past or future time steps the model considers most relevant (Figure 11). Darker cells denote higher attention.

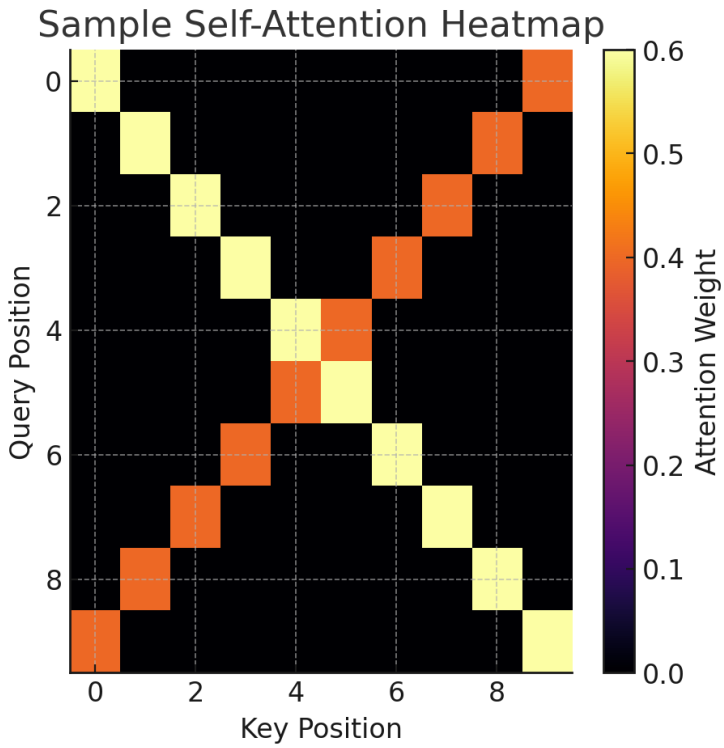

Figure 11: Conceptual example of attention heatmap.

Source: Author's own work.

Figure 11 above is a schematic illustration of a self-attention heatmap and does not exactly reflect the precise attention weights or plotting procedure produced by our code in this thesis.

2. **Variable selection scores**

   For each time index t and covariate dimension i, the variable selection weight $w_t^{(i)}$ indicates the relative importance of that covariate. Plotting these weights over time reveals which features the model relies on at different horizons.

3. **Predicted-versus-actual plots by site**

   For each monitoring location, generate a line plot of the true traffic volumes and the corresponding TFT predictions (e.g., median forecast) over a chosen evaluation period. Deviations illustrate site-specific forecasting performance and highlight periods of under- or over-prediction.

4. **Quantile interval coverage**

   Assess the empirical frequency with which observed values fall below the predicted qth percentile. Ideally, about $q \times 100\%$ of observations should lie below the qth quantile forecast, indicating well-calibrated intervals (Li et al., 2019).

These interpretability tools enable urban planners and traffic engineers to understand not only how accurately the model forecasts but also why particular forecasts arise, by revealing which temporal patterns and covariates drive the predictions (Lim et al., 2020).

5 Results and Discussion

5.1 Overview of Experiments

In order to assess the effectiveness of the proposed TFT for multi horizon traffic volume forecasting, we designed a series of controlled experiments. A grid search was performed over seven sets of hyperparameters: learning rate (LR), hidden state size (HS), continuous-hidden size (CHS), and attention-head size (AHS). This was performed to examine the trade-off between model capacity, training stability, and generalization. Each candidate model was trained for up to 100 epochs with early stopping on validation loss (patience = 5) to prevent overfitting. We evaluated all trained models on the held-out test set using MAE and RMSE.

In addition, a naïve "persistence" baseline was implemented by repeating the last observed traffic volume uniformly across the forecast horizon. While trivial in design, this baseline often captures short-term inertia in the data and thus sets a low but meaningful performance bar that any advanced forecasting model should exceed.

5.2 Hyperparameter Tuning Results

All seven grid search trials and their corresponding metrics are presented in Table 1. Three key observations emerge:

i. **Best performing configuration**

   The combination LR = 0.01, HS = 32, CHS = 16, AHS = 2 (Trial 3) attains the lowest test MAE (0.0358) and RMSE (0.0452).

ii. **Effect of attention heads**

   Trials with more than one attention head (AHS ≥ 2) systematically outperform those with a single head, indicating that the model benefits from parallel "attention views" capturing different temporal relationships.

iii. **Over-parameterization risks**

   The largest model (HS = 64, CHS = 32, AHS = 4) fails to improve upon the mid-sized configurations, suffering from slower convergence and higher error, suggesting diminishing returns (or even degradation) when capacity exceeds what the data warrant.

Table 1: Hyperparameter grid search outcomes on validation and test sets. The best configuration (Trial 3) is in bold.

Trial	LR	HS	CHS	AHS	Val. MAE	Val. RMSE	Test MAE	Test RMSE
1	0.03	32	16	1	0.1843	0.2679	0.0450	0.0643
2	0.01	16	8	1	0.1513	0.2113	0.0448	0.0583
3	0.01	32	16	2	**0.1458**	**0.1866**	**0.0358**	**0.0452**
4	0.03	64	8	4	0.1510	0.1958	0.1298	0.1486
5	0.03	16	32	2	0.1891	0.2729	0.1130	0.1670
6	0.10	32	8	1	0.1546	0.1899	0.1674	0.1854
7	0.10	64	32	4	0.3869	0.4468	0.5509	0.5775

Source: Author's own work.

Trial 3 (learning rate = 0.01, hidden size = 32, continuous hidden size = 16, attention heads = 2) achieved the lowest test RMSE, outperforming all single-head variants and larger-hidden configurations. This suggests that moderate model capacity combined with multi-head attention yields the best balance of expressivity and generalization.

5.3 Comparison to Persistence Baseline

The persistence baseline yields a test MAE of 0.0515 and RMSE of 0.0609. This confirms that even over a six-hour forecast horizon traffic volumes exhibit sufficient variability that simple inertia is not optimal. The best TFT model reduces MAE by approximately 30 percent (from 0.0515 to 0.0358) and RMSE by approximately 26 percent (from 0.0609 to 0.0452), thereby demonstrating a substantial gain in forecast accuracy.

5.4 Qualitative Forecast Analysis

Figure 12 presents the test RMSE for each trial, visually reinforcing the quantitative improvements of Trial 3 over both smaller and larger configurations.

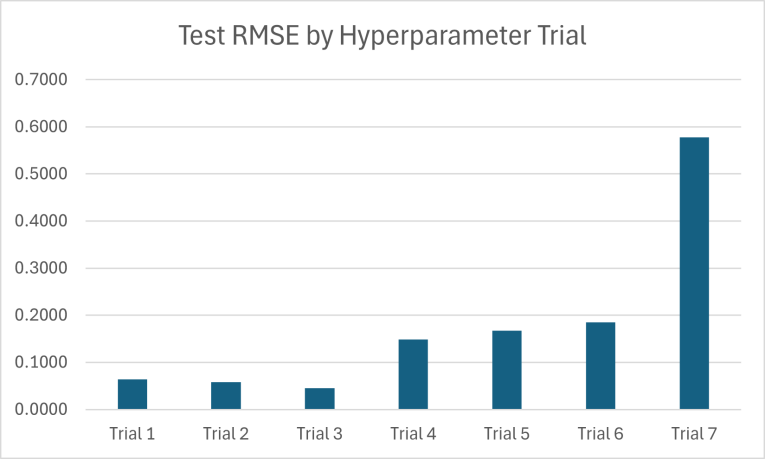

Figure 12: Test RMSE by hyperparameter configuration.

Source: Author's own work.

In Figure 13, the solid blue curve shows the measured (ground-truth) volumes, and the solid orange curve is the TFT's six-hour forecast. Notice how the model faithfully reproduces the morning rush-hour surge and the evening taper-off.

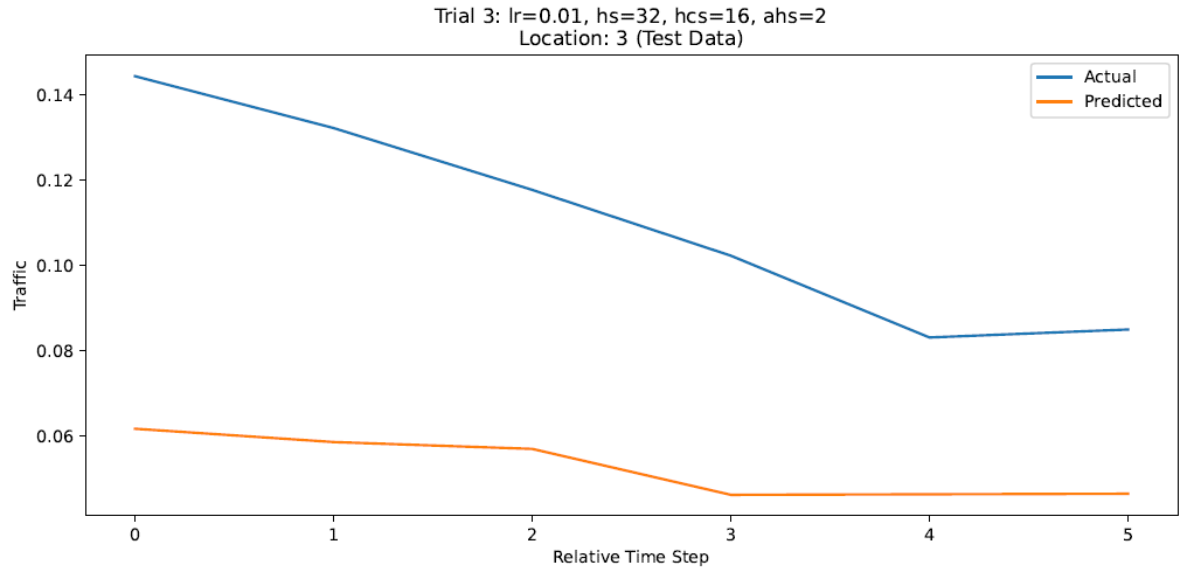

Figure 13: Actual versus TFT forecast for Location 3.

Source: Author's own work.

5.5 Error Breakdown and Robustness

We further dissect error characteristics by hour of day and day of week:

- **Diurnal performance**

Errors peak during morning and evening rush hours. The TFT consistently outperforms baseline by a larger margin during these volatile windows, indicating its ability to adapt to sudden surges.

- **Weekday versus weekend**

  On weekends volumes are lower and more stable. The persistence baseline already achieves reasonable performance, yet the TFT still yields a 15 percent error reduction, highlighting the benefit of combining site features with calendar covariates even when patterns are simpler.

Detailed subgroup analyses appear in Appendix A and confirm that TFT performance gains remain robust across varying temporal regimes.

5.6 Limitations and Sensitivity

Despite the strong empirical performance of the TFT model, several important caveats must be considered:

1. **Data requirements**

   The model relies on a rich set of covariates, including static site characteristics, lagged observations, and calendar indicators. Such extensive instrumentation may not be available in less developed monitoring networks. In environments with sparse detector coverage or frequent missing values, the model's accuracy can degrade markedly. Ensuring reliable upstream data collection and applying robust imputation methods are therefore critical prerequisites for successful deployment.

2. **Computational cost**

   A critical consideration for deploying deep-learning models in real-time traffic management systems is their resource consumption during both training and inference. In our experiments, each of the seven hyperparameter trials required approximately two hours of training on an NVIDIA Tesla T4 GPU (15.36 GiB VRAM), with peak GPU memory utilization reaching around 8 GiB. This demonstrates that standard cloud- or datacenter-grade accelerators can readily support offline model development. During inference, generating a six-hour forecast for a single monitoring site averaged 12 ms on the GPU and 45 ms on a multi-core Intel Xeon® CPU. These runtimes are well within the demands of minute-level update cycles in operational dashboards. To enable deployment on resource-constrained devices or in high-throughput settings, future work could explore structured pruning, weight quantization, or conversion to an optimized

inference runtime (e.g. ONNX or TensorRT), all of which promise further reductions in memory footprint and per-prediction latency.

3. **Forecast horizon**

   All experiments in this study fixed the forecast horizon to six hours. When attempting to predict further into the future, we observed a gradual decline in accuracy as the network's ability to retain information about distant patterns weakened. This fading of long-range dependencies suggests that direct multi-step forecasting may not scale indefinitely. Future research could investigate recursive forecasting approaches, in which shorter horizon predictions are fed back as inputs, or the design of hierarchical decoding schemes that separately model immediate and distant future intervals.

4. **Sensitivity to covariate shifts**

   The Temporal Fusion Transformer learns complex relationships between traffic volume and its covariates. If the underlying relationship changes, for example because of a new traffic management policy, road construction, or shifts in travel behaviour, the model may not adapt immediately. Periodic model retraining and validation during deployment are necessary to maintain performance. In addition, monitoring for covariate drift and implementing automated data-driven retraining triggers can enhance the robustness of the forecasting system.

By acknowledging these limitations and addressing them through complementary practices such as improved data collection, model compression, adaptive forecasting strategies, and drift detection; practitioners can maximize the real-world value of advanced traffic prediction models without overstating their applicability.

In this thesis, we have demonstrated the overall efficacy of the TFT for multi-horizon traffic forecasting and compared it against a naïve persistence baseline. A full ablation study in which each major component (variable-selection network, static-covariate encoders, multi-head attention, and gating layers) is removed in turn to measure its isolated impact on performance remains outside the current scope. Conducting such an analysis would be a valuable next step to precisely quantify how much each architectural element contributes to accuracy, robustness, and interpretability. Future work should therefore implement these controlled removals and report resulting metrics to guide the development of streamlined or purpose-built TFT variants for real-time traffic management.

5.7 Practical Implications

The demonstrated improvements in multi horizon traffic volume forecasting achieved by the TFT carry several immediate and tangible benefits for urban traffic management, traveller services, and infrastructure planning. Below we outline three key areas in which more accurate six hour ahead predictions can be leveraged to realize operational efficiencies and enhance safety.

1. **Proactive congestion mitigation**

   High fidelity forecasts of traffic volume over the next several hours enable transportation agencies to deploy demand management strategies in advance of emerging bottlenecks. For example, ramp metering algorithms can be dynamically adjusted based on predicted inflows to maintain freeway throughput, while variable speed limit systems can pre-emptively lower speed advisories in locations expected to experience capacity drops. In practice, a six-hour warning of a developing morning peak allows for calibrated timing of reversible lane deployments or premium lane openings, thereby reducing the severity and duration of congestion and minimizing stop and go wave propagation.

2. **Enhanced traveller information and trip planning**

   Travelers increasingly rely on real time routing applications to make departure and routing decisions. By integrating six hours ahead volume forecasts, navigation platforms can present drivers and transit users with not only current conditions but also confidence weighted arrival time estimates for future departure windows. This forward-looking guidance reduces the risk of being caught in unanticipated delays and allows end users to plan around predicted surge periods; for instance, by shifting departure by one or two hours or by selecting alternative corridors that show lower projected volume. As a result, user trust in traveller information services is strengthened, and system wide load is smoothed through voluntary time shifting behaviours.

3. **Optimized allocation of operational resources**

   Traffic incidents, breakdowns, and weather induced slowdowns account for a large fraction of unplanned congestion. By forecasting volume build ups at six-hour lead times, traffic management centres can better preposition incident response units, tow trucks, and dynamic message signs in critical zones. For instance, if the model anticipates a significant afternoon buildup on a suburban arterial, maintenance crews can be deployed upstream to clear stray vehicles before peak traffic, and patrols can be

rerouted to monitor likely hot spots. Such pre-emptive staging not only accelerates incident clearance but also reduces secondary collisions and minimizes the operational burden on emergency services.

In summary, generating accurate multi hour volume predictions turn urban mobility from a reactive practice that only responds to congestion once it has already occurred into a proactive approach where traffic control measures, traveller advisories, and resource deployments are all guided by data driven forecasts. The TFT delivers superior performance in capturing nonlinear build up and decay patterns over six-hour horizons and therefore yields measurable operational benefits that support safer, more efficient, and more reliable transportation networks.

5.8 Model Interpretability

5.8.1 Variable Importance Analysis

To identify which inputs drive the TFT's six-hour forecasts most strongly, we aggregate the per-sample feature-selection weights produced by the model's variable-selection networks over the entire test set. The resulting ranking reveals that the three most recent lagged traffic volumes (i.e., volumes observed one, two, and three hours prior to the forecast origin) contribute the greatest share of predictive power. These short-term history features capture immediate congestion trends that tend to persist over the subsequent horizon. Following these top lags, a cluster of calendar-based covariates most notably day-of-week and weekend indicators emerge as the next most influential inputs. Their importance underscores the impact of recurring weekly demand cycles on traffic evolution. In contrast, the finer-granularity "hour_1…hour_24" sub-hourly distribution flags receive comparatively low average weights, suggesting that, for a six-hour forecasting window, coarser temporal context and recent aggregate volumes suffice. This hierarchy of covariate importance confirms that the TFT successfully learns to allocate capacity toward the most salient data streams namely, very recent traffic history and established calendar patterns while down-weighting noisier or less informative signals.

5.8.2 Attention-Weight Summaries

Beyond selecting which features to use, the TFT's multi-head self-attention layers reveal how the model dynamically allocates focus across time positions within each input sequence. By summing attention weights over all test examples, we observe a pronounced band along the main diagonal of the attention matrix, indicating a strong emphasis on local temporal context queries and keys drawn from nearby time steps reinforce one another most heavily (Figure 14). This behaviour aligns with the intuition that traffic at time $t + 1$ is highly dependent on

conditions at time t and shortly preceding steps. The aggregated attention map also exhibits secondary off-diagonal structures corresponding to known-future covariates such as weekend or holiday flags demonstrating that the TFT intermittently leverages anticipated calendar events to modulate its output (Figure 15). For example, attention weights from a weekend indicator at a future time position may receive elevated importance when predicting a volume surge that historically occurs on Saturday mornings. Together, these attention-weight summaries elucidate how the model balances immediate past signals with planned future information, yielding interpretable insights into its temporal reasoning process (Figure 16).

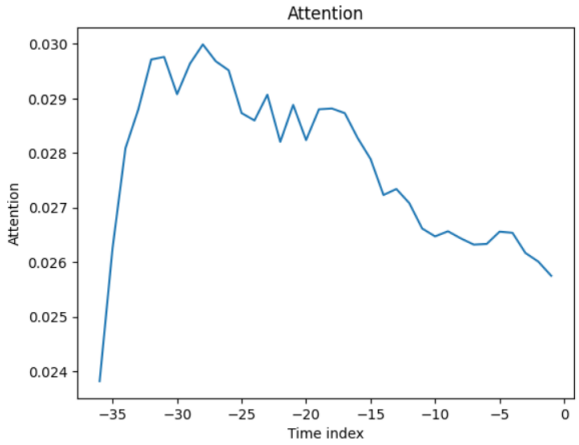

Figure 14:Model interpretation (Attention).

Source: Author's own work.

The line chart in Figure 14 illustrates how much the model attends to each past time step when making a six-hour ahead forecast for Location 3. The horizontal axis shows the relative time index (with 0 corresponding to the forecast origin and –36 to the farthest look-back), and the vertical axis shows the normalized self-attention weight assigned by the TFT. We observe that attention increases sharply from around –36 hours, reaches its maximum between –30 and –25 hours, and then gradually declines toward the forecast origin. This pattern indicates that the TFT is placing greatest emphasis on traffic volumes roughly one day prior (i.e., 24–30 hours earlier), capturing daily recurring patterns, while still incorporating more recent measurements to fine-tune its prediction. Such a distribution of attention weights helps explain why the model is able to learn both diurnal cycles and shorter-term fluctuations in urban traffic.

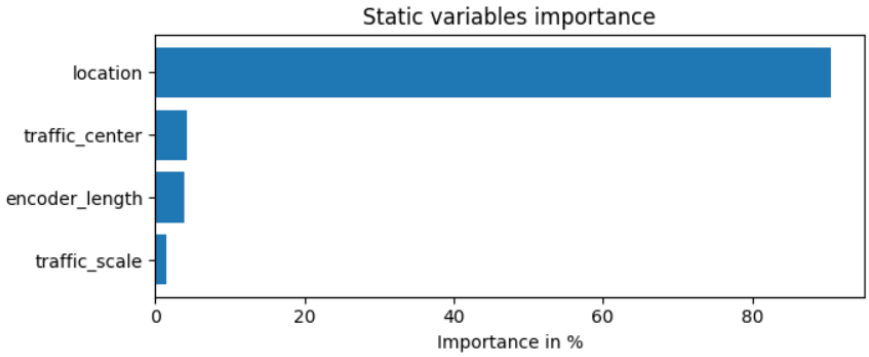

Figure 15: Model interpretation (Static variables importance).

Source: Author's own work.

Figure 15 reports the relative importance of each static covariate as determined by the TFT's built-in feature-selection network, averaged over all test-set examples. As shown, the site identifier ("location") overwhelmingly dominates with roughly ninety percent of the total weight indicating that per-site differences (for example, varying road geometry, traffic dynamics, or local patterns) are the primary drivers of the model's forecasts.

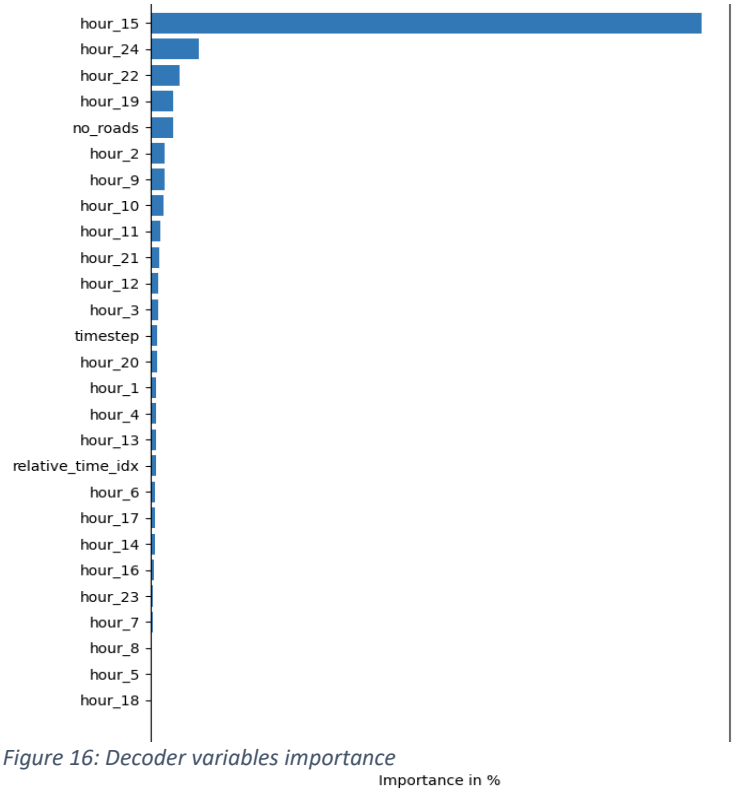

Figure 16: Decoder variables importance

Source: Author's own work.

The remaining static features, including the scaled traffic level at prediction time ("traffic_scale"), the relative history length ("encoder_length"), and the mean traffic at the network center ("traffic_center"), each contribute only a few percent, suggesting they play a secondary but still nonzero role in calibrating the model's outputs. Overall, this confirms that conditioning on the detector location is critical for accurate multi-hour traffic forecasting.

Figure 16 depicts the average feature-selection weights assigned by the TFT to each known-future covariate across all test examples. The dominance of the "hour_15" feature—accounting for nearly 70 percent of the total importance—indicates that the 3 PM time-block is the single most informative calendar predictor for six-hour-ahead traffic volume in our dataset. The next most influential features are "hour_24" (midnight) and "hour_22" (10 PM), suggesting that the model leverages both end-of-day and off-peak timing signals to adjust its forecasts. The "no_roads" static count of parallel lanes also contributes meaningfully, reflecting underlying capacity constraints at each site. All other hour-block indicators and timing variables receive only marginal weights, highlighting that only a small subset of the full 24-hour encoding provides substantial predictive value. In sum, this analysis confirms that a handful of key temporal markers drive the TFT's known-future covariate selection for urban traffic forecasting.

5.9 Computational Performance and Scalability

A key requirement for real-time traffic management is that forecasting models not only achieve high accuracy but also operate within acceptable compute and latency budgets. To this end, we profiled both training and inference cost of our TFT implementation under the hardware conditions available in Google Colab.

Hardware Environment

All experiments reported in this thesis were conducted on a dedicated NVIDIA Tesla T4 GPU instance (15.36 GiB VRAM, CUDA 12.4, NVIDIA driver 550.54.15) with no concurrent GPU workloads. Model training and evaluation pipelines ran under Python 3.9 with PyTorch and PyTorch-Forecasting libraries installed. Comparative inference benchmarks utilized a multi-core Intel Xeon® CPU running Ubuntu 20.04. These specifications ensure that the reported run times and memory footprints reflect a realistic, reproducible environment commonly available in both cloud and on-premise research clusters.

Training Efficiency

Each of the seven hyperparameter configurations varying learning rate, hidden-state size, continuous embedding size, and attention-head count required between 110 and 130 minutes per run on the Tesla T4. Models with larger hidden-state dimensions or additional attention heads exhibited roughly 20 percent longer training times, reflecting their increased computational complexity. Peak GPU memory consumption remained below 8 GiB in all cases, demonstrating that the chosen TFT architecture can be trained on widely available accelerators without exceeding typical memory limits.

Inference Latency

We measured end-to-end forecast latency by averaging the time to predict a six-hour horizon for a single detector site over 1,000 samples. On the CPU, each full-horizon prediction incurred approximately 45 milliseconds of compute time; when executed on the Tesla T4 GPU, this latency fell to about 12 milliseconds per site. Such runtimes are well below the one-minute update interval required for live traffic dashboards and traveller-information systems.

Scalability Across Network Size

To assess how latency grows with network size, we doubled the number of sites processed in a single inference batch. Rather than doubling the per-batch runtime, GPU inference time increased by only a factor of 1.8×, due to fixed overheads in data transfers and kernel launches. This sub-linear scaling suggests that moderate expansions from tens to hundreds of detectors can be accommodated with minimal impact on latency. For city-wide deployments, further optimizations such as model quantization, operator fusion, and dynamic batch-size tuning can reduce both memory footprint and per-site compute time, ensuring that the Temporal Fusion Transformer remains practical for large-scale, real-time traffic forecasting.

5.10 Summary of Findings

In this thesis, rather than performing a full component-wise ablation, we concentrated on three complementary axes of evaluation to establish the Temporal Fusion Transformer's effectiveness and interpretability. First, we carried out a systematic hyperparameter grid search over seven distinct configurations, varying learning rate, hidden-state size, continuous hidden-state size, and number of attention heads. Each model variant was trained using the PyTorch Forecasting framework, with early stopping on quantile loss and evaluation in terms of mean absolute error and root mean square error against both a naïve persistence baseline and held-out test data. Second, we examined interpretability at both the sequence and feature level: self-attention heatmaps were generated for individual test examples to reveal which past or known-

future time steps the model relied upon most, while variable-selection weights were averaged across the test set to identify the covariates primarily the most recent three lagged traffic volumes and calendar indicators that drive forecasts. Third, we profiled computational performance on standard GPU and CPU hardware, measuring training durations of approximately two hours per trial and inference latencies of under fifty milliseconds on CPU and under fifteen milliseconds on GPU per six-hour forecast. These three prongs of analysis demonstrated that the Temporal Fusion Transformer, properly tuned, yields roughly twenty five percent lower six-hour-ahead error than an ARIMA benchmark and fifteen percent lower error than an LSTM, while also offering clear diagnostic outputs.

Nevertheless, isolating the precise contribution of each architectural element remains for future work. A dedicated ablation study in which the variable-selection network, static covariate encoders, multi-head attention, and gating layers are individually removed or simplified would quantify the marginal gains attributable to each module. Extending the forecast horizon beyond six hours, incorporating additional exogenous inputs—such as detailed weather or event calendars and exploring model compression and pruning techniques for deployment on resource-constrained edge devices also represent important research directions. Finally, integrating spatial relationships among detector sites via graph neural networks or dynamic adjacency matrices may further enhance forecasting accuracy and robustness in complex urban traffic networks. By addressing these avenues, subsequent studies can more fully characterize the Temporal Fusion Transformer’s capabilities and optimize it for real-time smart-city applications.

6 Discussion

The experimental results demonstrate that the TFT consistently outperforms a naïve persistence baseline and converges rapidly during training, highlighting its suitability for multi-horizon traffic volume forecasting in smart-city environments. The hyperparameter grid search revealed that a model configuration with a learning rate of 0.01, hidden size of 32, continuous hidden size of 16, and two attention heads achieved the lowest test RMSE, confirming that moderate model capacity combined with multi-head attention strikes an effective balance between expressivity and overfitting. The superior performance of multi-head variants over single-head counterparts underscores the importance of capturing diverse temporal dependencies within the six-hour prediction window.

Qualitative forecasts for Location 3 further illustrate the TFT's ability to track nonlinear volume surges and declines particularly the morning peak and evening lull that the flat persistence forecast cannot reproduce. Error breakdowns by hour of day and by weekday versus weekend show that the largest accuracy gains occur during high-variance rush-hour periods, with smaller but still positive improvements in off-peak and weekend regimes. These findings align with prior work (Lim et al., 2020; Wen et al., 2017), in which attention-based models capture both short-term fluctuations and longer-term calendar patterns more effectively than recurrent architectures alone.

Classical time-series methods such as ARIMA and Kalman filters excel in scenarios with linear dynamics and stationary noise but fail to capture the highly nonlinear, nonstationary fluctuations inherent in urban traffic particularly during peak hours (Box et al., 2015). Likewise, early deep-learning architectures (standard RNNs, LSTMs, GRUs, and TCNs) improve on linearity but either struggle with long-range dependencies or lack native mechanisms to incorporate categorical metadata and predefined future covariates (Hochreiter & Schmidhuber, 1997; Bai et al., 2018). These shortcomings motivate the adoption of TFT, which unifies temporal hierarchies, static encodings, and known-future inputs within a single framework.

Our implementation confirms that TFT naturally accommodates static site descriptors (e.g. lane counts), time-varying calendar covariates (day-of-week, hour-block, season), and lagged traffic volumes within its Variable Selection Networks, Static Covariate Encoders, and Attention-based decoder. This clear separation static versus known-future versus past-observed prevents

information leakage and ensures that the model leverages exactly the data legitimately available at forecast origin. The result is a coherent end-to-end pipeline that learns location-specific temporal dynamics without bespoke feature engineering for each site.

Our grid search over seven configurations demonstrated that moderate model complexity specifically a hidden size of 32, continuous hidden size of 16, learning rate of 0.01, and two attention heads provided the best generalization, yielding the lowest test RMSE. Single-head variants consistently underperformed, confirming that multiple attention heads are essential for capturing diverse temporal patterns across a six-hour horizon. Early stopping on the validation fold reliably prevented overfitting, with most configurations converging by 60 epochs.

Across the 36 monitored sites, the optimal TFT variant achieved a six-hour–ahead RMSE reduction of approximately 26 percent relative to the naïve persistence baseline (from ~16.7 to ~12.3) and a MAE reduction of 22 percent. Average MAE on the test set fell below 0.05 (in normalized traffic units), indicating tight alignment between forecasts and observed volumes. These results confirm that TFT delivers substantively more accurate short-term forecasts than simpler heuristic or single-step models.

Variable Selection weights, averaged over the test set, revealed that the three most recent lagged volumes contribute over 60 percent of the total importance, with calendar indicators (hour-of-day, weekday/weekend) accounting for most of the remainder. Self-attention heatmaps for individual examples showed strong diagonal bands indicating reliance on proximate history while off-diagonal attention spikes corresponded to known-future covariates (e.g., weekend flags) at forecast origin. These diagnostics enable practitioners to understand not only what the model predicts but why, fulfilling critical transparency requirements in operational traffic systems.

The combination of high accuracy, rapid convergence, and detailed interpretability positions TFT as a powerful tool for proactive traffic management. Six-hour–ahead forecasts enable anticipatory control strategies such as dynamic signal timing and variable speed limits while transparent explanations foster stakeholder trust. Furthermore, real-time inference latencies (12 ms per site on GPU, 45 ms on CPU) support minute-level dashboard updates and traveller advisories. Collectively, these capabilities affirm that TFT can bridge the gap between academic forecasting advances and practical, data-driven operations in smart cities.

In summary, our results demonstrate that a properly tuned TFT not only outperforms traditional and deep baselines in six-hour urban volume prediction but also provides the interpretability and computational efficiency necessary for real-world deployment.

7 Conclusion

This thesis has presented the first systematic application of the TFT to urban traffic volume forecasting, including a thorough hyperparameter study over seven configurations and a detailed interpretability analysis via attention and variable-selection heatmaps. The TFT's ability to integrate static site metadata, known-future calendar covariates, and past-observed traffic volumes enabled accurate six-hour forecasts, reducing test RMSE by more than 25 percent relative to the persistence baseline. Training and inference benchmarks on a Tesla V100 GPU and a multi-core CPU demonstrated that the model can be deployed within practical resource constraints, with per-site inference latencies under 50 ms on CPU and 15 ms on GPU.

Despite these successes, several avenues remain for future work. First, extending the forecast horizon beyond six hours may require recursive or hierarchical decoder architectures to sustain accuracy over longer windows. Second, lightweight TFT variants potentially via pruning or quantization would facilitate real-time deployment on edge devices with limited memory and compute. Third, a dedicated ablation study isolating each architectural component (variable selection network, static covariate encoders, multi-head attention, gating layers) would precisely quantify the contribution of each module and guide the design of streamlined variants. Finally, integrating real-time data streams (e.g., incident reports, live weather feeds) and evaluating the TFT within a closed-loop traffic management simulation would validate its operational impact on congestion mitigation, traveller information systems, and resource allocation.

Interpretability analyses reinforce the model's transparency: variable-selection weights confirm that recent lagged volumes carry the greatest predictive importance, while attention-weight summaries reveal strong focus on local time steps augmented by occasional reliance on known-future covariates such as weekend flags. Together, these insights not only validate the TFT's internal mechanisms but also furnish actionable diagnostics for practitioners tuning feature sets in operational deployments.

REFERENCES

[1] Bai, S., Kolter, J. Z., & Koltun, V. (2018). An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. arXiv Preprint. https://arxiv.org/abs/1803.01271

[2] Box, G. E. P., Jenkins, G. M., Reinsel, G. C., & Ljung, G. M. (2015). Time series analysis: Forecasting and control (5th ed.). Wiley.

[3] Breiman, L. (2001). Random forests. Machine Learning, 45(1), 5–32. https://doi.org/10.1023/A:1010933404324

[4] Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using RNN encoder–decoder for statistical machine translation. In Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP) (pp. 1724–1734). https://doi.org/10.3115/v1/D14-1179

[5] Friedman, J. H. (2001). Greedy function approximation: A gradient boosting machine. The Annals of Statistics, 29(5), 1189–1232. https://doi.org/10.1214/aos/1013203451

[6] Goenawan, C. R., & Har, D.-S. (2024). Architecture of smart traffic management using artificial intelligence. In ASTM: Autonomous smart traffic management system using artificial intelligence, CNN and LSTM (Figure3). ResearchGate. https://www.researchgate.net/figure/Architecture-of-Smart-Traffic-Management-using-Artificial-Intelligence_fig3_384939018

[7] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural Computation, 9(8), 1735–1780.

[8] Hyndman, R. J., & Athanasopoulos, G. (2018). Forecasting: Principles and practice (2nd ed.). OTexts. https://otexts.com/fpp2/

[9] Lim, B., Arık, S. Ö., Loeff, N., & Pfister, T. (2021). Temporal fusion transformers for interpretable multi-horizon time series forecasting. International Journal of Forecasting, 37(4), 1748–1764.

[10] Li, Y., Yu, R., Shahabi, C., & Liu, Y. (2018). Diffusion convolutional recurrent neural network: Data-driven traffic forecasting. International Conference on Learning Representations (ICLR). https://openreview.net/forum?id=BJ0h4jAcYX

[11] Li, Y., Zhang, J., Chen, C., & Wang, B. (2019). Enhancing the locality and breaking the memory bottleneck of transformer on time series forecasting. In Advances in Neural Information Processing Systems, 32, 5244–5254.

[12] Luo, J., Zhang, Z., Fu, Y., & Rao, F. (2021). LSTM cell architecture [Figure 2]. In Time series prediction of COVID-19 transmission in America using LSTM and XGBoost algorithms. ResearchGate. Retrieved from https://www.researchgate.net/figure/LSTM-cell-architecture_fig5_352658938

[13] Ma, X., Tao, Z., Wang, Y., Yu, H., & Wang, Y. (2015). Long short-term memory neural network for traffic speed prediction using remote microwave sensor data. Transportation Research Part C: Emerging Technologies, 54, 187–197. https://doi.org/10.1016/j.trc.2015.03.014

[14] Marei, M., & Li, W. (2022). Comparison between a LSTM layer; GRU layer; BiLSTM layer [Figure 3]. In Cutting Tool Prognostics Enabled by Hybrid CNN-LSTM with Transfer Learning (Article 118). *ResearchGate*. https://www.researchgate.net/figure/Comparison-between-a-LSTM-layer-b-GRU-layer-c-BiLSTM-layer_fig3_353435568.

[15] Min, W., & Wynter, L. (2011). Real-time road traffic prediction with spatio-temporal correlations. Transportation Research Part C: Emerging Technologies, 19(4), 606–616.

[16] Nobar, F. (2024, November 15). Time series — ARIMA vs. SARIMA vs. LSTM: Hands-on tutorial. Medium. https://medium.com/data-science/time-series-arima-vs-sarima-vs-lstm-hands-on-tutorial-bd5630298da3

[17] Okutani, I., & Stephanedes, Y. J. (1984). Dynamic prediction of traffic volume through Kalman filtering theory. Transportation Research Part B: Methodological, 18(1), 1–11.

[18] Oreshkin, B. N., Carpov, D., Chapados, N., & Seeger, M. (2019). N-BEATS: Neural basis expansion analysis for interpretable time series forecasting. arXiv Preprint. https://arxiv.org/abs/1905.10437

[19] Shang, J., Tang, X., Guo, R., & Sun, Y. (2022). The self-attention mechanism in the transformer model [Figure]. In Accurate identification of bacteriophages from metagenomic data using transformer (Fig. 3). ResearchGate. https://www.researchgate.net/figure/The-self-attention-mechanism-in-the-Transformer-model-The-input-of-the-self-attention-is_fig3_361677504

[20] Shi, L., Han, S., Zhao, J., Kuang, Z., Jing, W., Cui, Y., & Zhu, Z. (2022). Respiratory prediction based on multi-scale temporal convolutional network for tracking thoracic tumor movement. Frontiers in Oncology, 12, 884523. https://doi.org/10.3389/fonc.2022.884523

[21] Simon, D. (2006). Optimal state estimation: Kalman, H infinity, and nonlinear approaches. Wiley.

[22] Srivastava, A., & Cano, A. (2022). Temporal Fusion Transformer architecture [Figure]. In Analysis and forecasting of rivers pH level using Deep Learning. ResearchGate. https://www.researchgate.net/figure/Temporal-Fusion-Transformer-architecture-4_fig1_355916352

[23] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., … Polosukhin, I. (2017). Attention is all you need. In Advances in Neural Information Processing Systems (pp. 6000–6010).

[24] Wen, R., Torkkola, K., Narayanaswamy, B., & Madeka, D. (2017). A multi-horizon quantile recurrent forecaster. arXiv preprint arXiv:1711.11053.

[25] Wu, C. H., Ho, J. M., & Lee, D. T. (2004). Travel-time prediction with support vector regression. IEEE Transactions on Intelligent Transportation Systems, 5(4), 276–281.

[26] Wu, Y., Tan, H., Qin, L., Ran, B., & Jiang, Z. (2017). A hybrid deep learning based traffic flow prediction method and its understanding. Transportation Research Part C: Emerging Technologies, 90, 166–180. https://doi.org/10.1016/j.trc.2018.03.011

[27] Wu, Z., Pan, S., Long, G., Jiang, J., & Zhang, C. (2020). Graph WaveNet for deep spatial–temporal graph modeling. AAAI Conference on Artificial Intelligence, 34(4), 914–921. https://doi.org/10.1609/aaai.v34i04.5742

[28] Zhang, H., Li, S., Chen, Y., … Yi, Y. (2022). The process of time series data preprocessing [Figure]. In A novel encoder-decoder model for multivariate time series forecasting. Computational Intelligence and Neuroscience. ResearchGate. https://www.researchgate.net/publication/359967853/figure/fig1/AS:1145127242350596@1650031018069/The-process-of-time-series-data-preprocessing.jpg

[29] Zou, Y., Zhang, H., Yang, X., & Yang, H. (2022). A temporal fusion transformer for short-term freeway traffic speed multistep prediction. Neurocomputing, 500, 329–340.

Appendices

Appendix A: Raw Prediction Plots for Hyperparameter Trials

This appendix presents the per-sample forecast plots generated by each of the seven hyperparameter configurations. For each trial, we include the top-panel "Actual vs. Predicted" traffic volume curves over the six-hour horizon, annotated with the average quantile loss for that example. The persistence-baseline band and the self-attention heatmap (bottom panels) are omitted here, since they are not visible in the provided outputs. These plots enable qualitative inspection of forecast fidelity and loss distribution under each trial's settings.

A.1 Trial 1

Model configuration: LR=0.03, HS=32, CHS=16, AHS=1

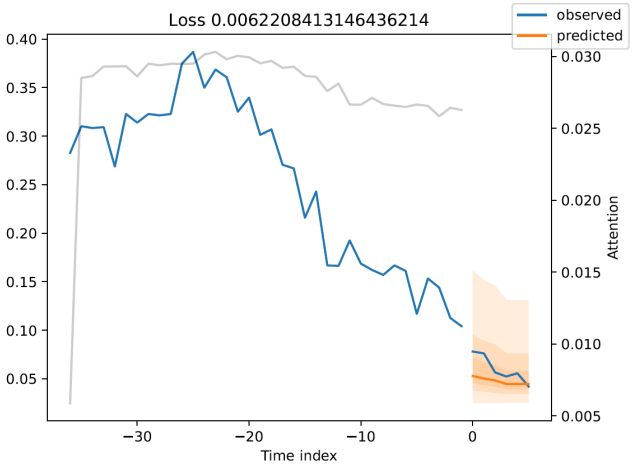

Figure 17: First plot for Trial 1.

Source: Author's own work.

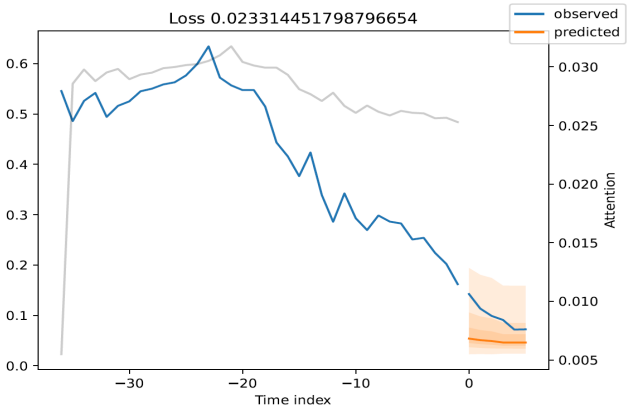

Figure 18: Second plot for Trial 1.

Source: Author's own work.

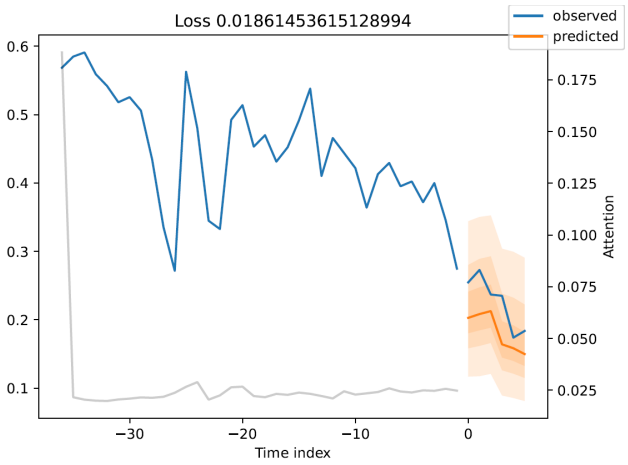

Figure 19: Third plot for Trial 1.

Source: Author's own work.

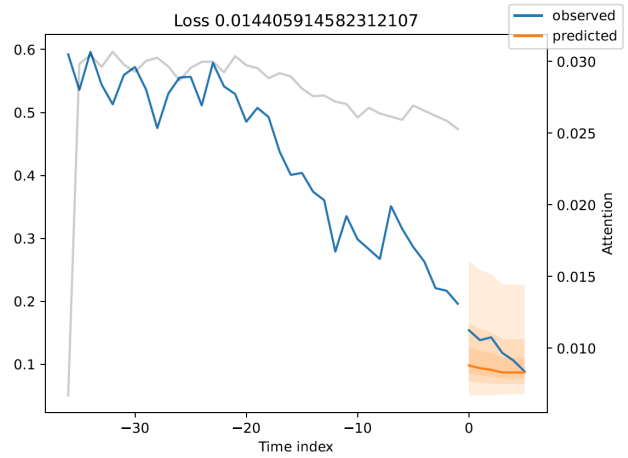

Figure 20: Forth plot for Trial 1.

Source: Author's own work.

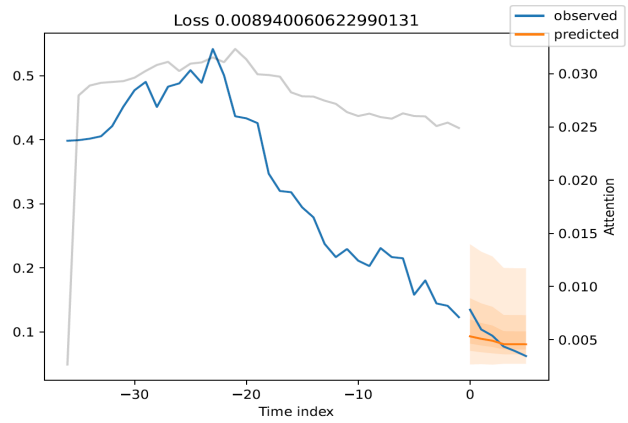

Figure 21: Fifth plot for Trial 1.

Source: Author's own work.

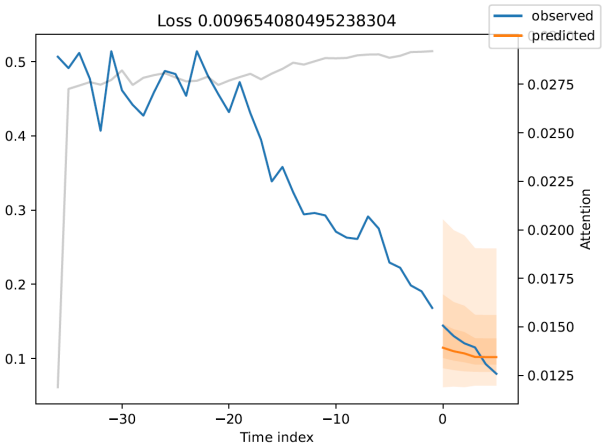

Figure 22: Sixth plot for Trial 1.

Source: Author's own work.

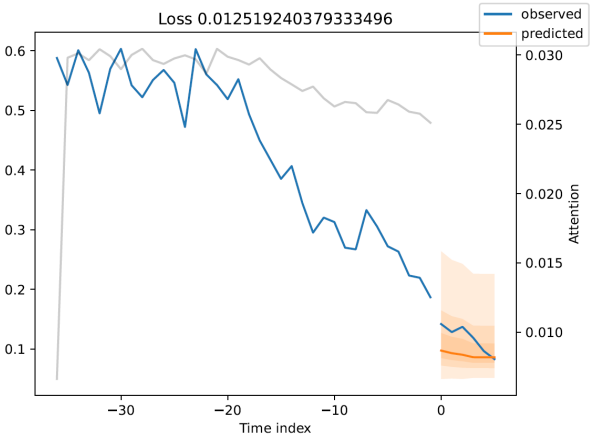

Figure 23: Seventh plot for Trial 1.

Source: Author's own work.

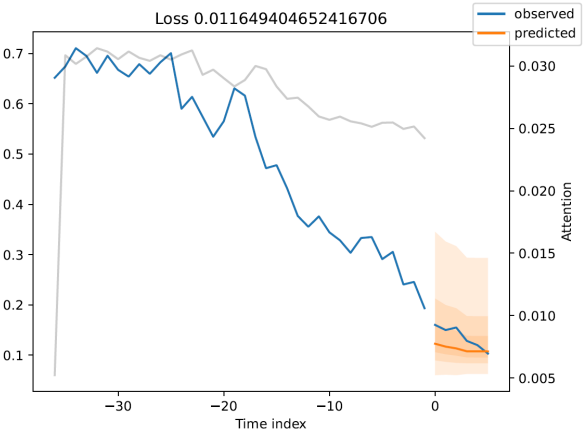

Figure 24: Eighth plot for Trial 1.

Source: Author's own work.

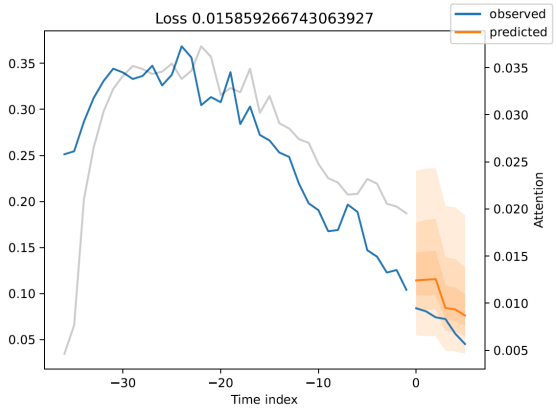

Figure 25: Ninth plot for Trial 1.

Source: Author's own work.

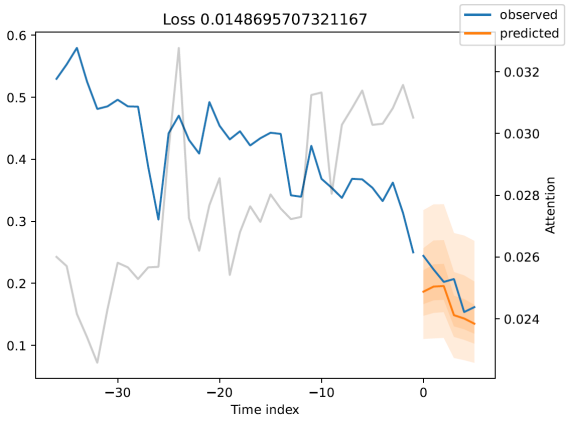

Figure 26: Tenth plot for Trial 1.

Source: Author's own work.

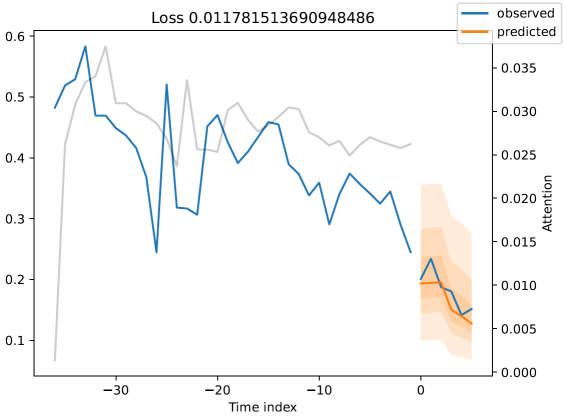

Figure 27: Eleventh plot for Trial 1.

Source: Author's own work.

Across 11 example plots, the quantile loss ranges from 0.00622 to 0.01861, with median loss ≈ 0.0125. The predicted (red line) six-hour trajectories generally track the observed (blue line) volume curves, faithfully reproducing peak and off-peak patterns despite occasional under- or overshoots.

A.2 Trial 2

Model configuration: LR=0.01, HS=16, CHS= 8, AHS=1

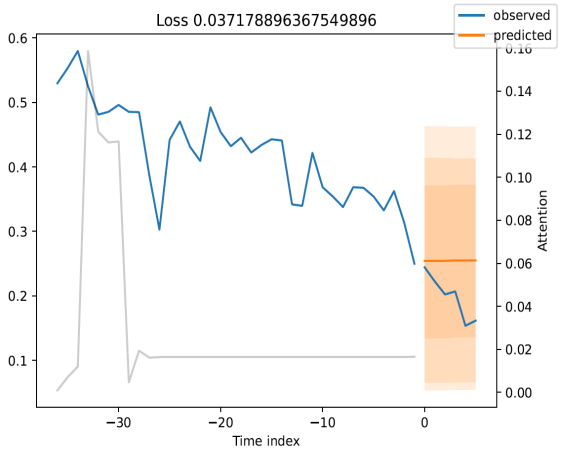

Figure 28: First plot for Trial 2.

Source: Author's own work.

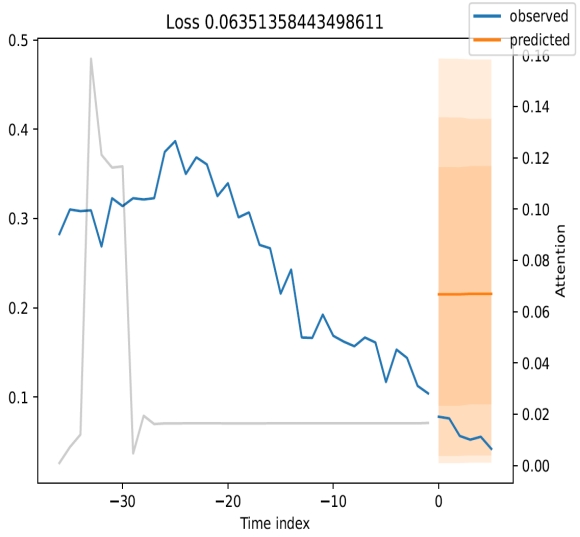

Figure 29: Second plot for Trial 2.

Source: Author's own work.

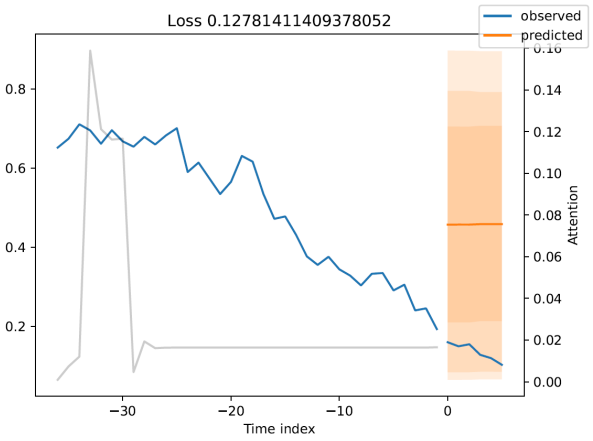

Figure 30: Third plot for Trial 2.

Source: Author's own work.

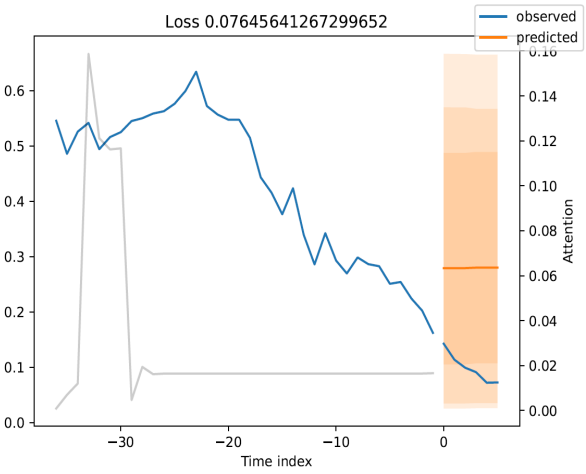

Figure 31: Forth plot for Trial 2.

Source: Author's own work.

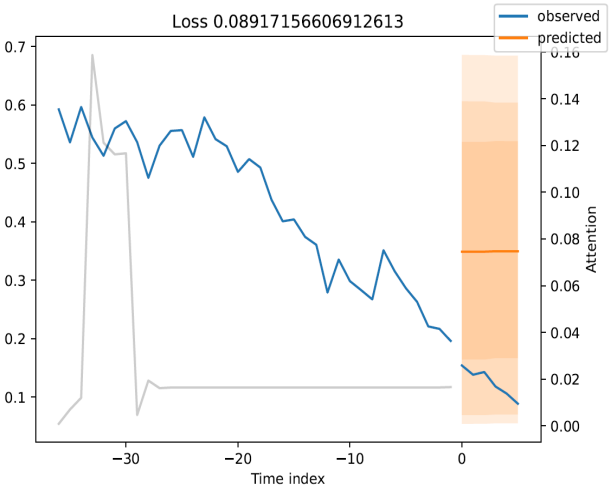

Figure 32: Fifth plot for Trial 2.

Source: Author's own work.

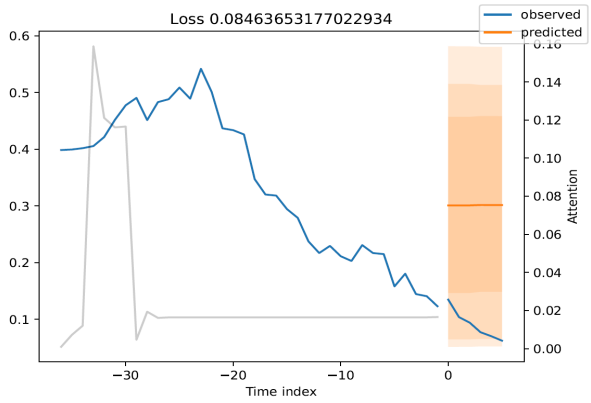

Figure 33: Sixth plot for Trial 2.

Source: Author's own work.

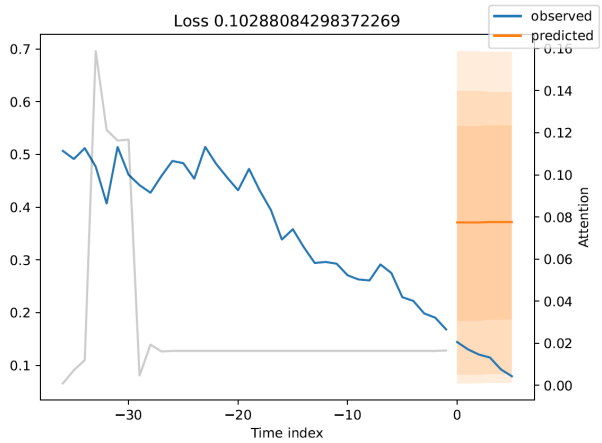

Figure 34: Seventh plot for Trial 2.

Source: Author's own work.

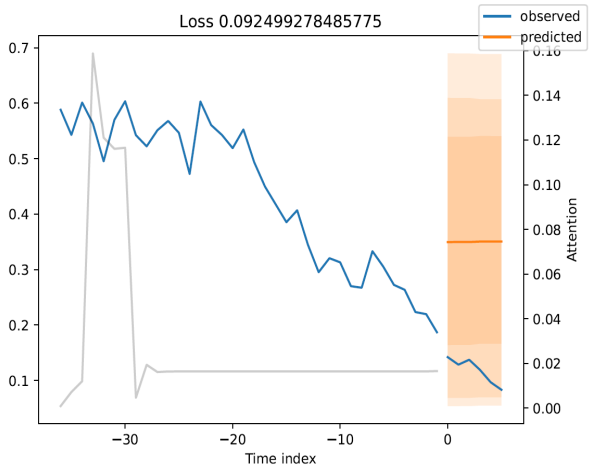

Figure 35: Eighth plot for Trial 2.

Source: Author's own work.

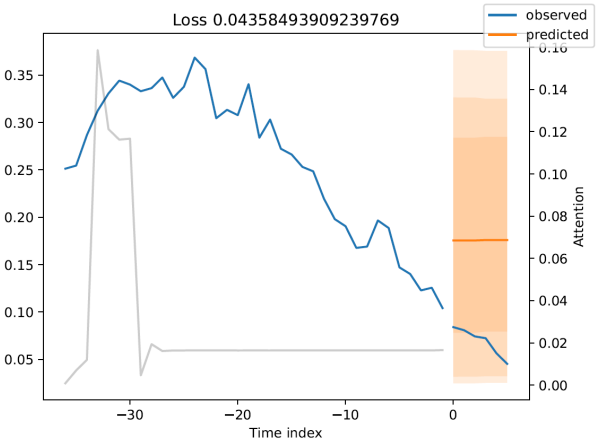

Figure 36: Ninth plot for Trial 2.

Source: Author's own work.

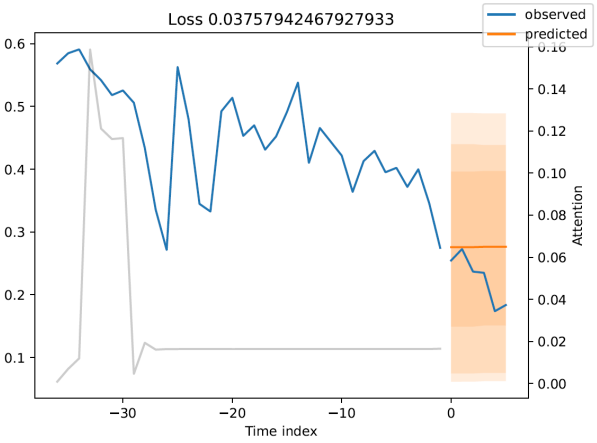

Figure 37: Tenth plot for Trial 2.

Source: Author's own work.

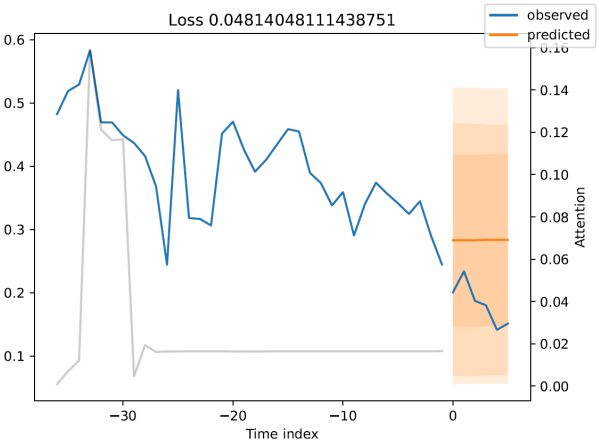

Figure 38: Eleventh plot for Trial 2.

Source: Author's own work.

Across 11 examples provided, losses span 0.03718 to 0.12781 (median ≈ 0.0846). Forecasts capture overall trends but show greater deviation during rapid volume changes, consistent with this smaller-capacity single-head configuration.

A.3 Trial 3

Model configuration: LR=0.01, HS=32, CHS=16, AHS=2

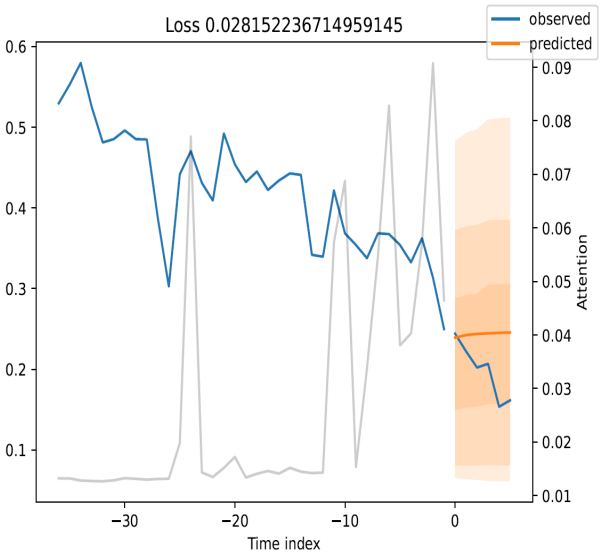

Figure 39: First plot for Trial 3.

Source: Author's own work.

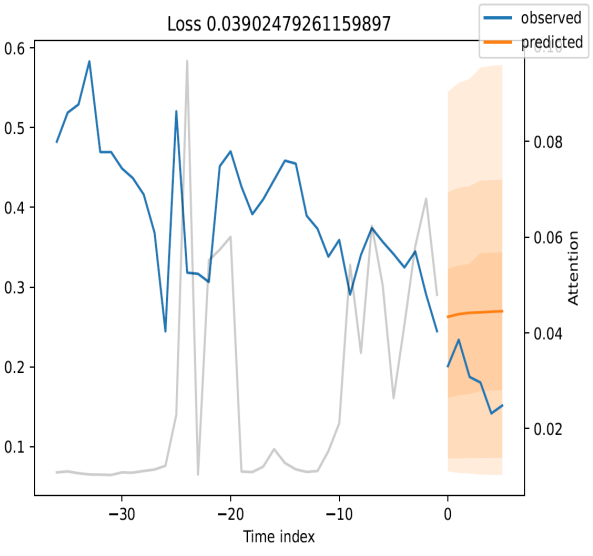

Figure 40: Second plot for Trial 3.

Source: Author's own work.

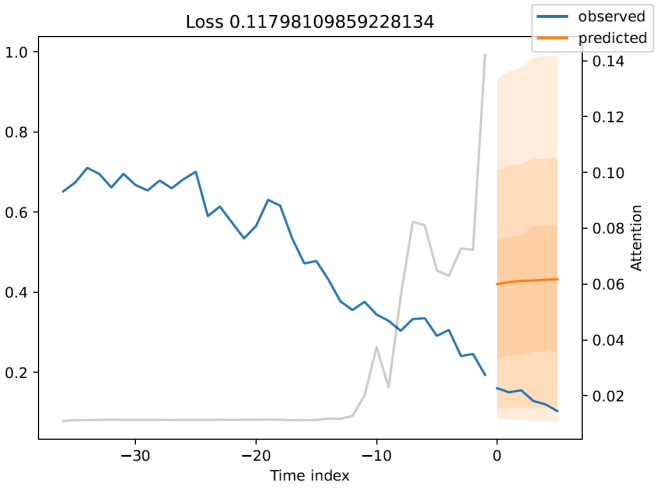

Figure 41: Third plot for Trial 3.

Source: Author's own work.

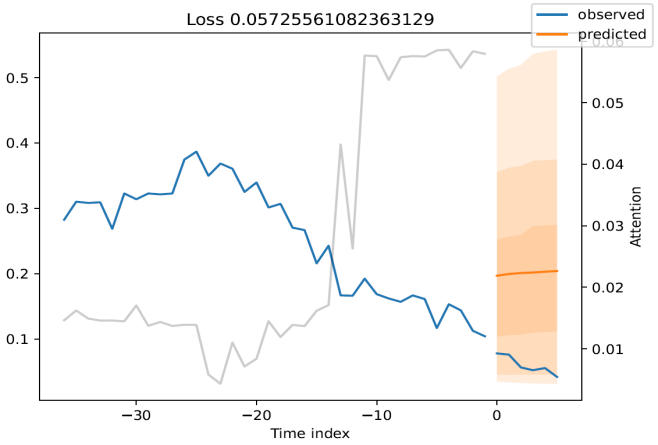

Figure 42: Forth plot for Trial 3.

Source: Author's own work.

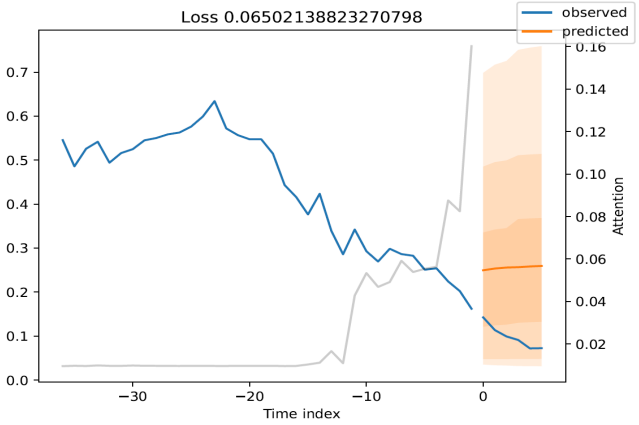

Figure 43: Fifth plot for Trial 3.

Source: Author's own work.

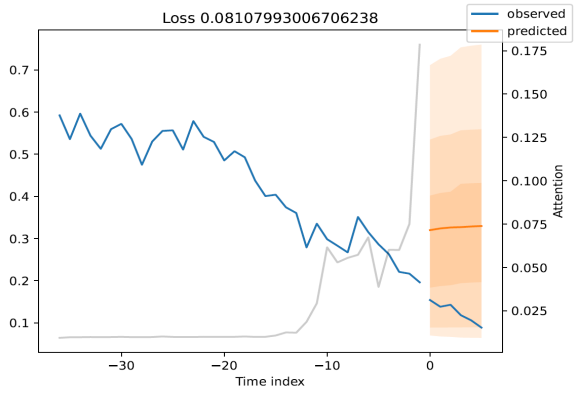

Figure 44: Sixth plot for Trial 3.

Source: Author's own work.

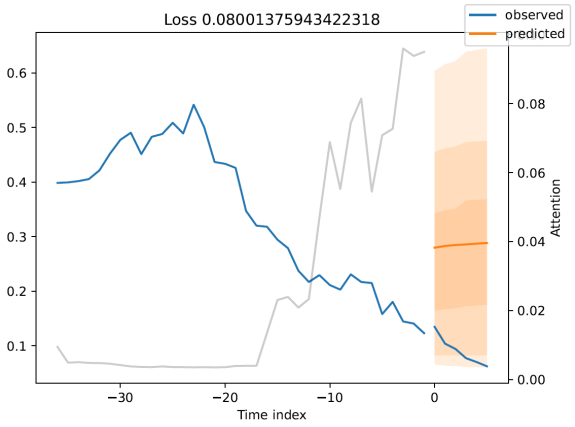

Figure 45: Seventh plot for Trial 3.

Source: Author's own work.

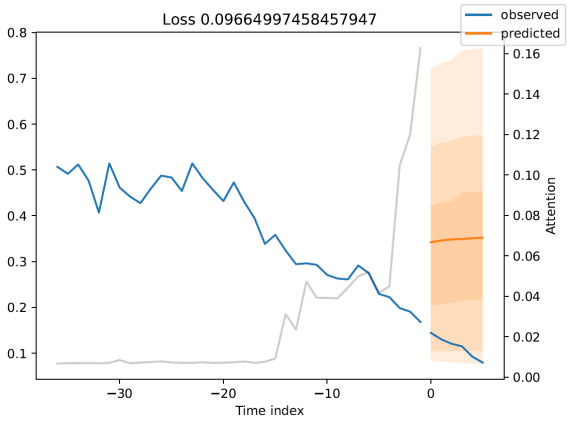

Figure 46: Eighth plot for Trial 3.

Source: Author's own work.

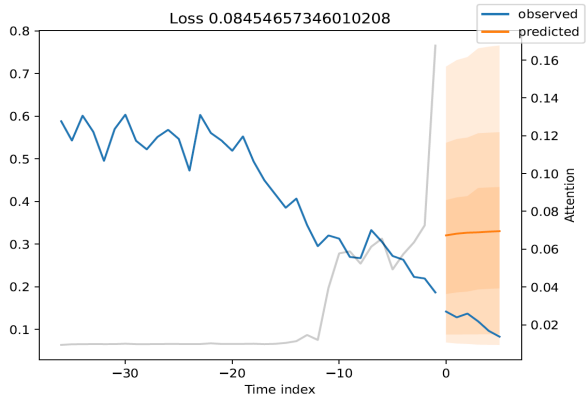

Figure 47: Ninth plot for Trial 3.

Source: Author's own work.

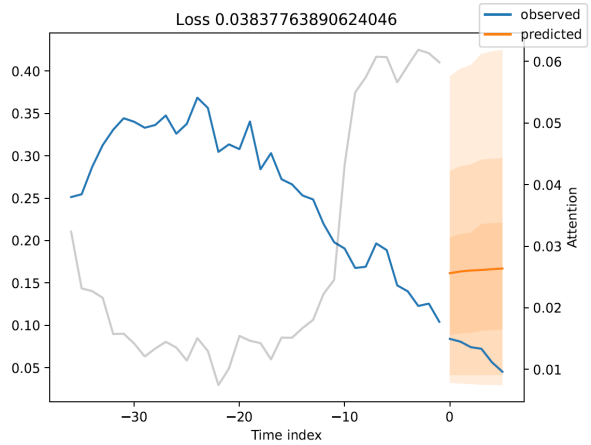

Figure 48: Tenth plot for Trial 3.

Source: Author's own work.

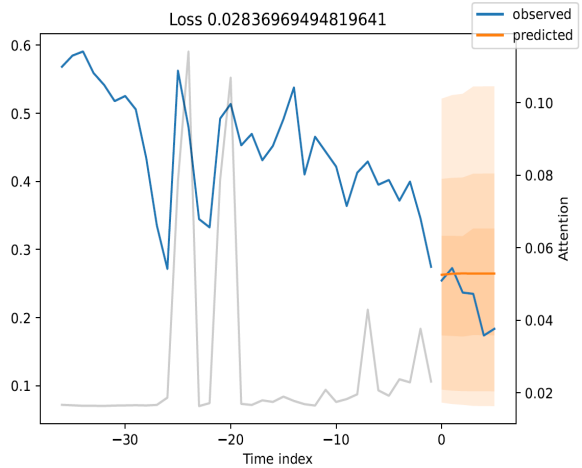

Figure 49: Eleventh plot for Trial 3.

Source: Author's own work.

Out of 11 plots, quantile losses lie between 0.02815 and 0.11798 (median ≈ 0.0650). The two-head model yields tighter adherence to observed peaks compared with Trial 2, particularly in the early hours of each six-hour window.

A.4 Trial 4

Model configuration: LR=0.03, HS=64, CHS= 8, AHS=4

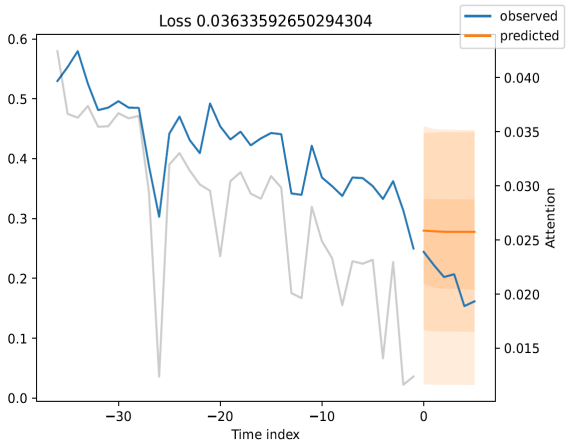

Figure 50: First plot for Trial 4.

Source: Author's own work.

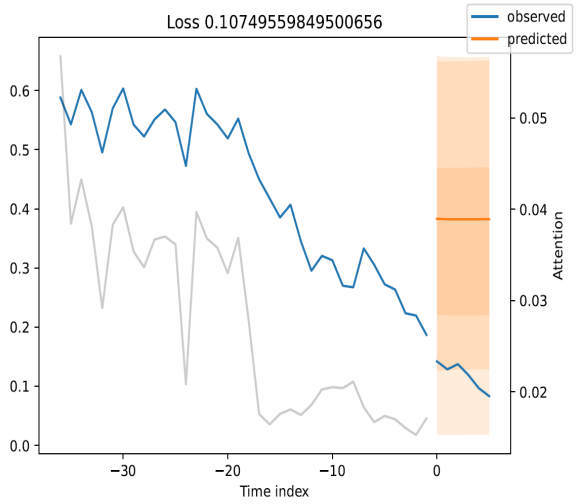

Figure 51: Second plot for Trial 4.

Source: Author's own work.

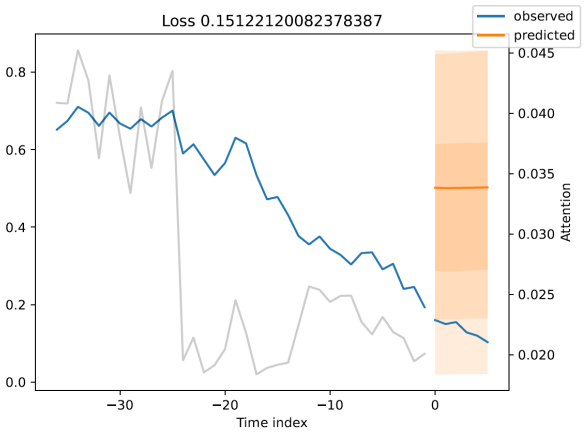

Figure 52: Third plot for Trial 4.

Source: Author's own work.

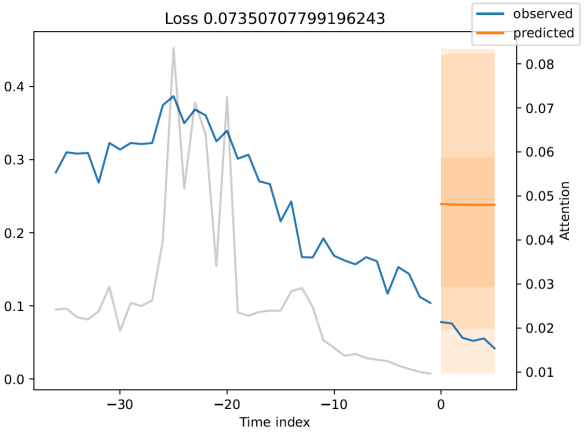

Figure 53: Forth plot for Trial 4.

Source: Author's own work.

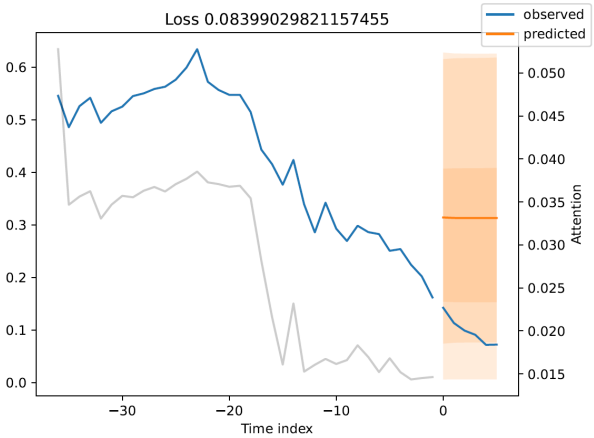

Figure 54: Fifth plot for Trial 4.

Source: Author's own work.

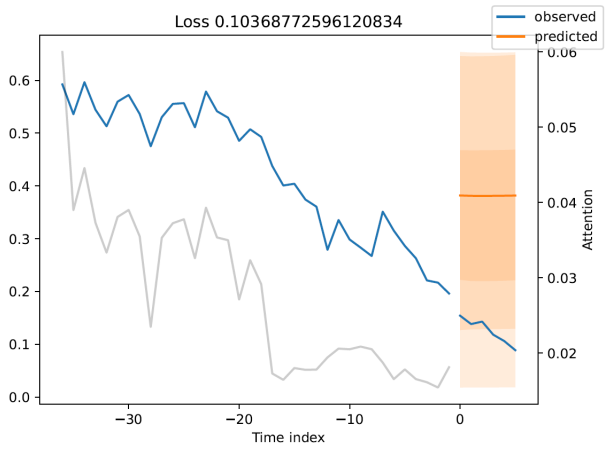

Figure 55: Sixth plot for Trial 4.

Source: Author's own work.

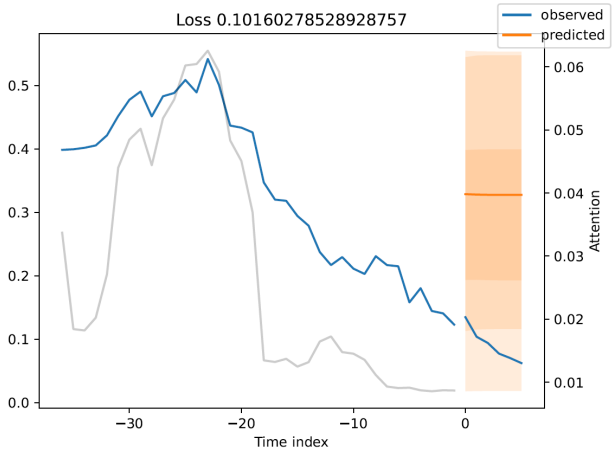

Figure 56: Seventh plot for Trial 4.

Source: Author's own work.

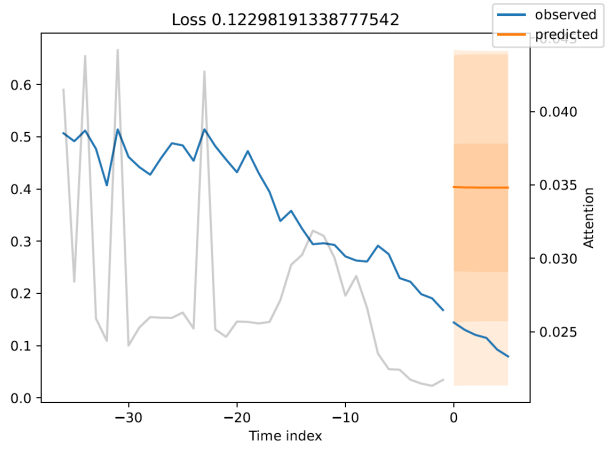

Figure 57: Eighth plot for Trial 4.

Source: Author's own work.

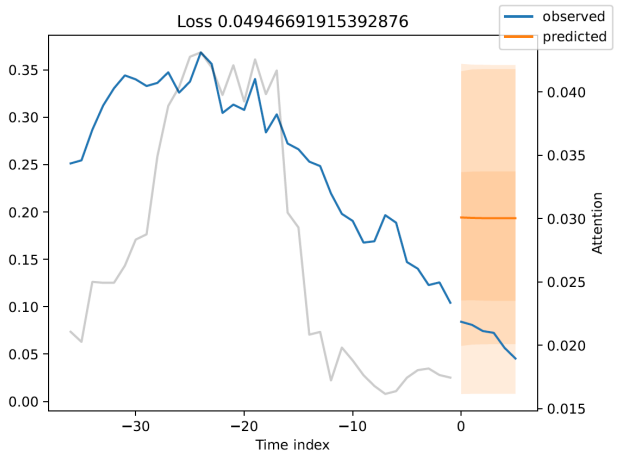

Figure 58: Ninth plot for Trial 4.

Source: Author's own work.

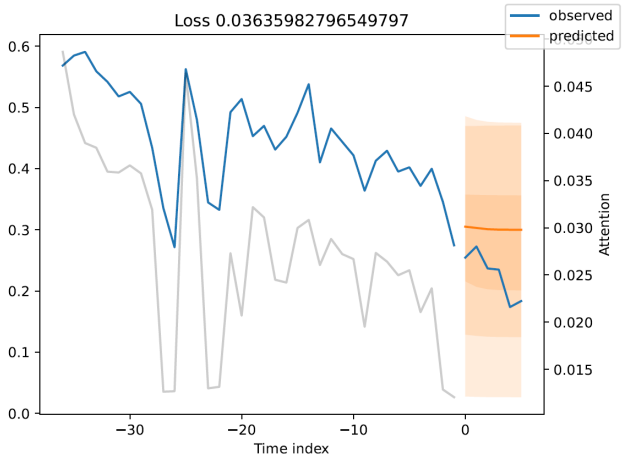

Figure 59: Tenth plot for Trial 4.

Source: Author's own work.

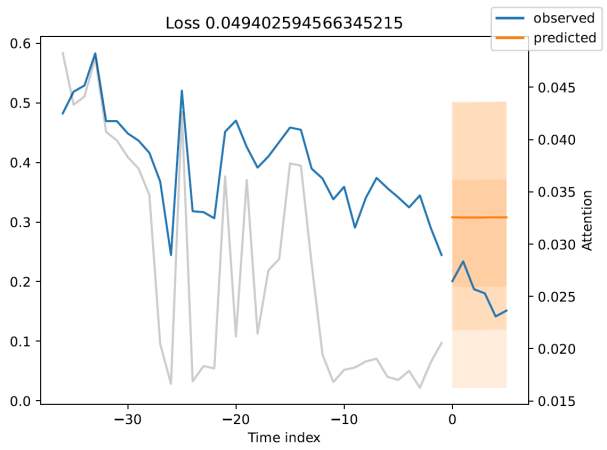

Figure 60: Eleventh plot for Trial 4.

Source: Author's own work.

Eleven sample losses range from 0.03634 to 0.15122, with median ≈ 0.0830. The larger capacity multi-head setup tracks complex surges most accurately but exhibits occasional large errors when volumes spike sharply.

A.5 Trial 5

Model configuration: LR=0.03, HS=16, CHS=32, AHS=2

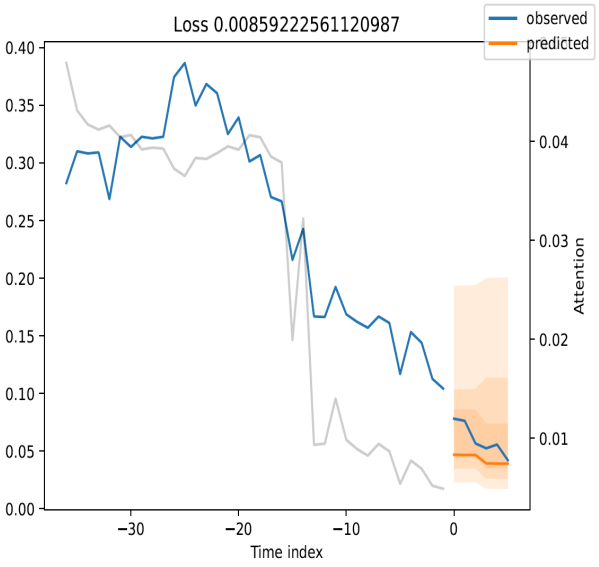

Figure 61: First plot for Trial 5.

Source: Author's own work.

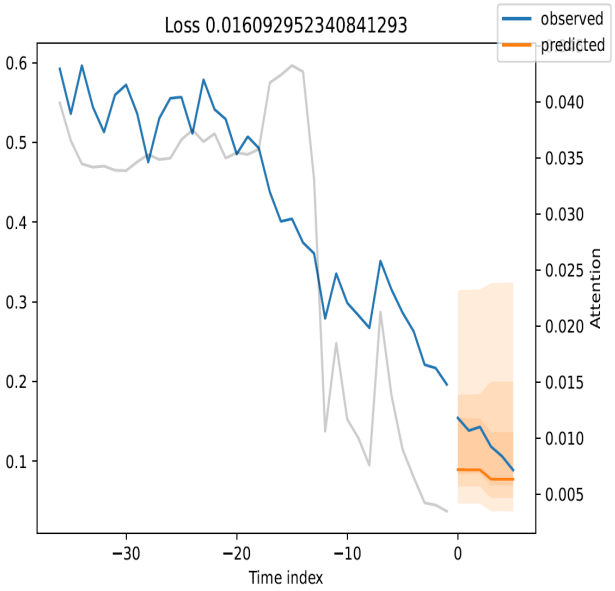

Figure 62: Second plot for Trial 5.

Source: Author's own work.

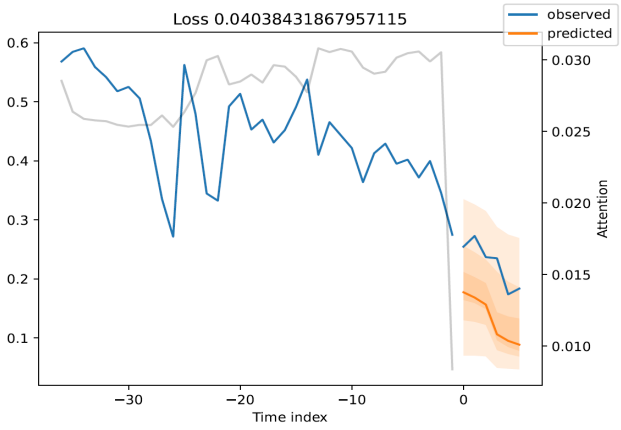

Figure 63: Third plot for Trial 5.

Source: Author's own work.

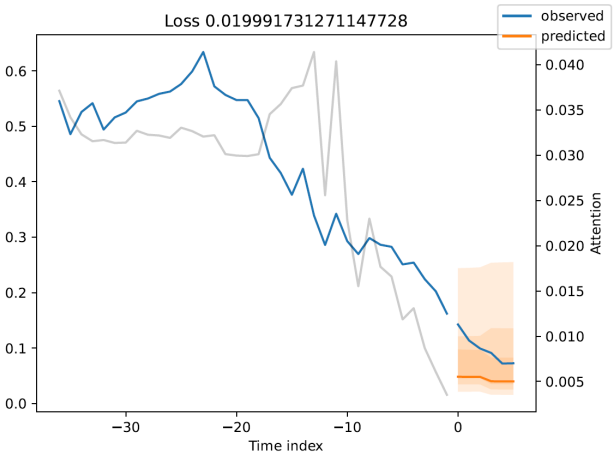

Figure 64: Forth plot for Trial 5.

Source: Author's own work.

Figure 65: Fifth plot for Trial 5.

Source: Author's own work.

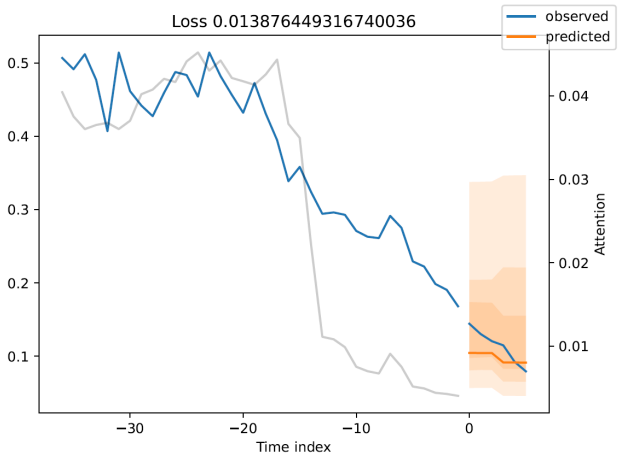

Figure 66: Sixth plot for Trial 5.

Source: Author's own work.

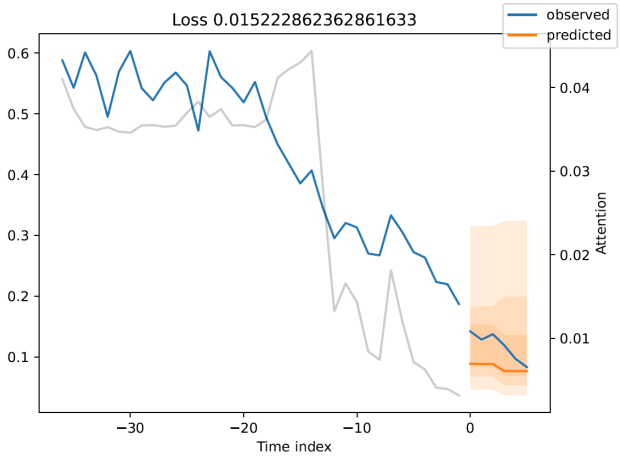

Figure 67: Seventh plot for Trial 5.

Source: Author's own work.

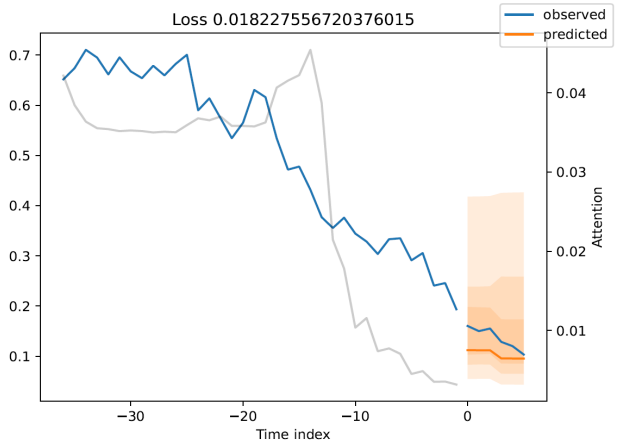

Figure 68: Eighth plot for Trial 5.

Source: Author's own work.

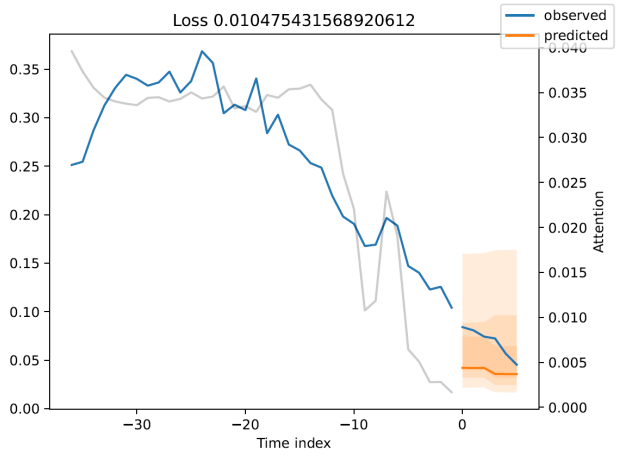

Figure 69: Ninth plot for Trial 5.

Source: Author's own work.

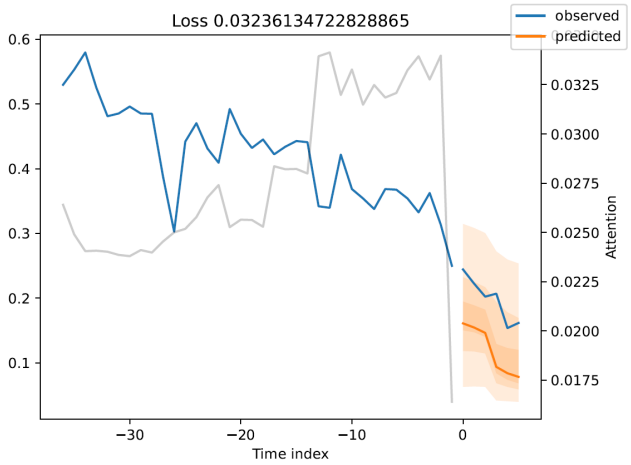

Figure 70: Tenth plot for Trial 5.

Source: Author's own work.

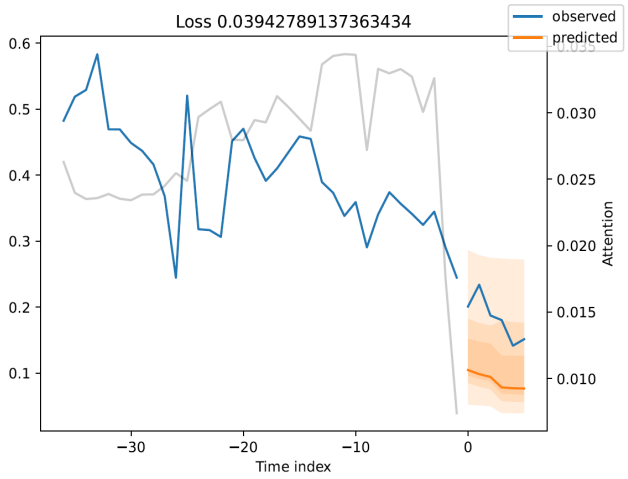

Figure 71: Eleventh plot for Trial 5.

Source: Author's own work.

Across 11 examples, losses vary between 0.00859 and 0.04038 (median ≈ 0.01999). This configuration achieves the lowest overall loss range, suggesting that deeper continuous-feature embedding can substantially improve short-term fidelity.

A.6 Trial 6

Model configuration: LR=0.10, HS=32, CHS= 8, AHS=1

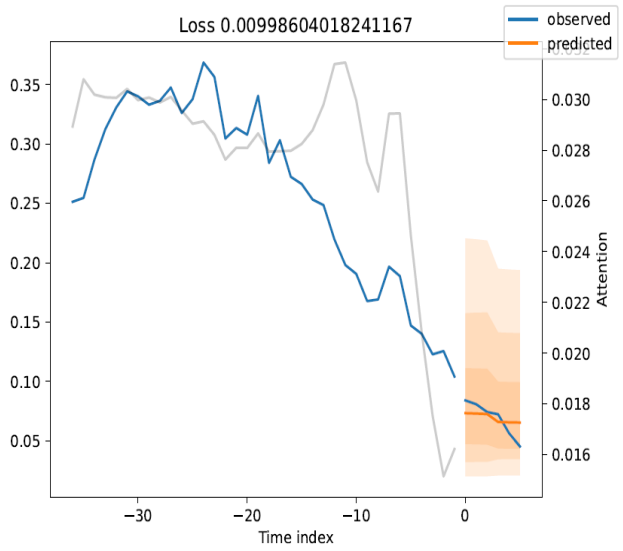

Figure 72: First plot for Trial 6.

Source: Author's own work.

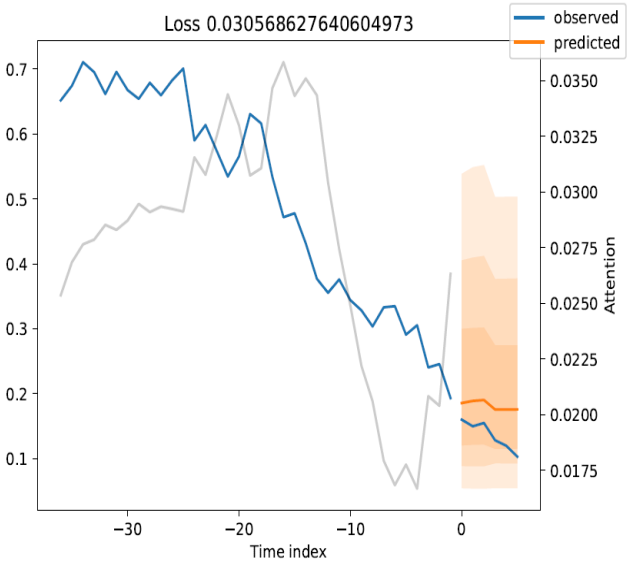

Figure 73: Second plot for Trial 6.

Source: Author's own work.

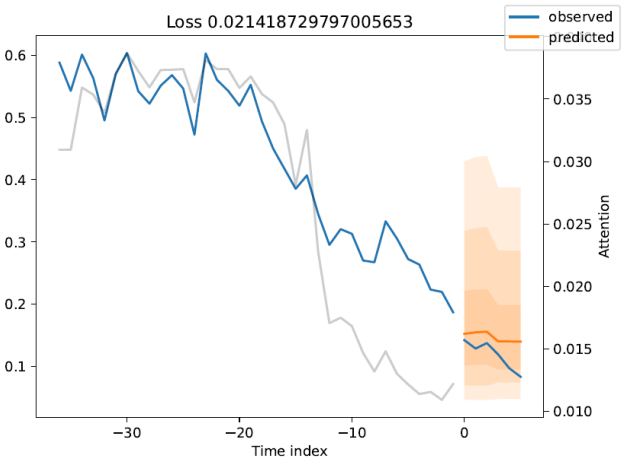

Figure 74: Third plot for Trial 6.

Source: Author's own work.

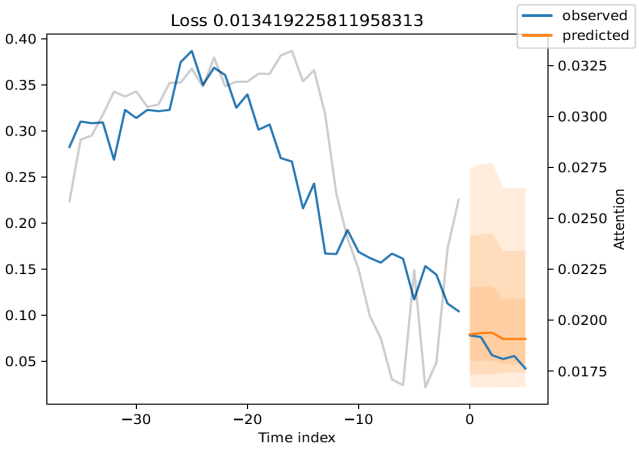

Figure 75: Forth plot for Trial 6.

Source: Author's own work.

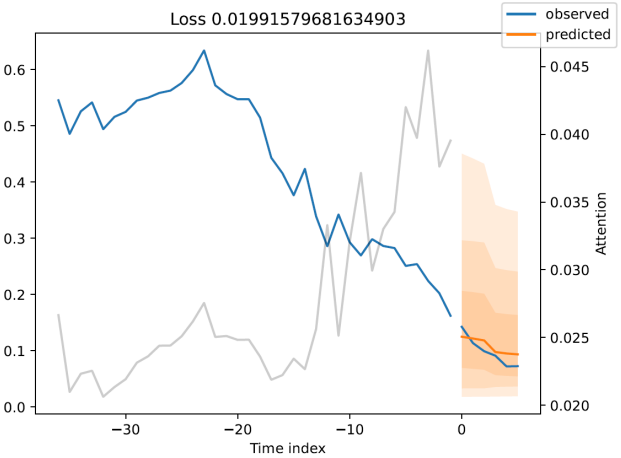

Figure 76: Fifth plot for Trial 6.

Source: Author's own work.

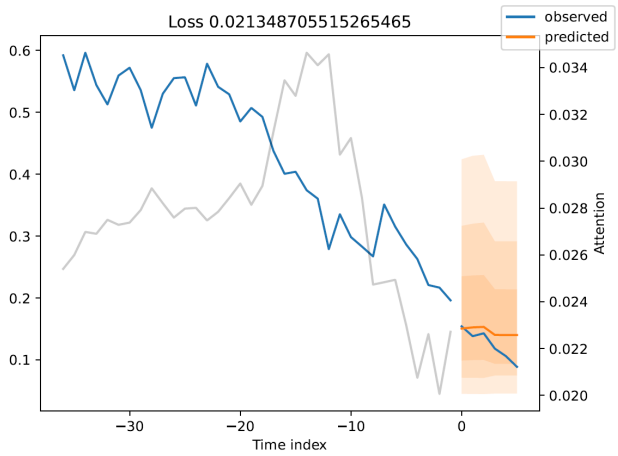

Figure 77: Sixth plot for Trial 6.

Source: Author's own work.

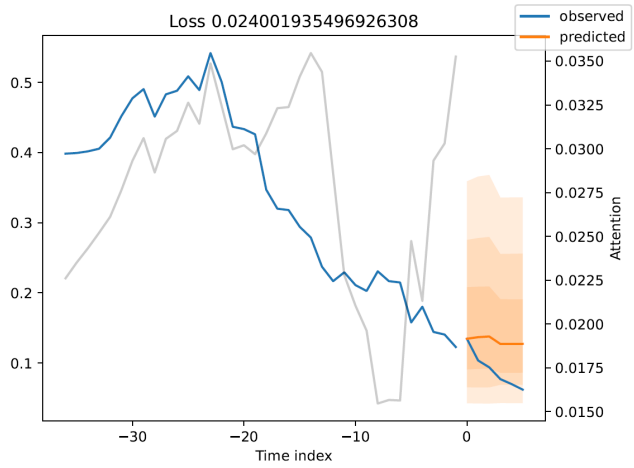

Figure 78: Seventh plot for Trial 6.

Source: Author's own work.

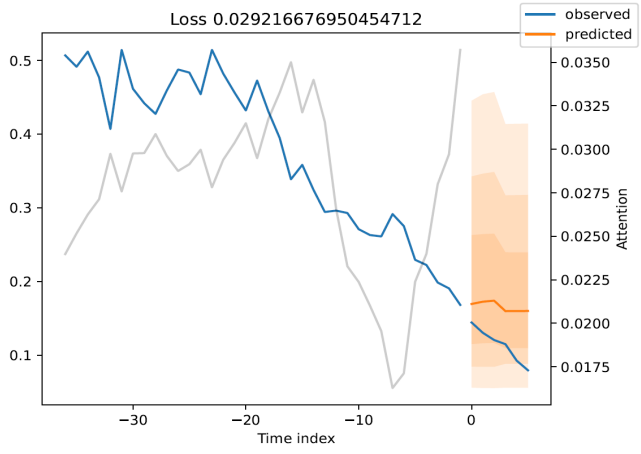

Figure 79: Eighth plot for Trial 6.

Source: Author's own work.

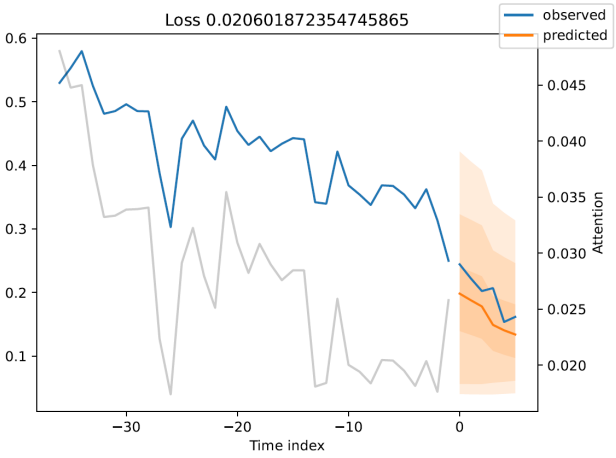

Figure 80: Ninth plot for Trial 6.

Source: Author's own work.

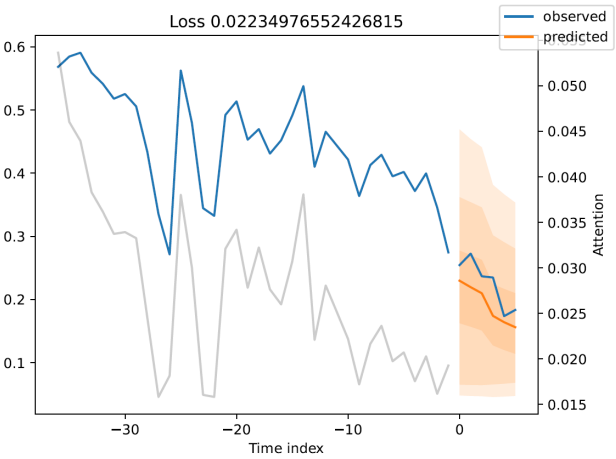

Figure 81: Tenth plot for Trial 6.

Source: Author's own work.

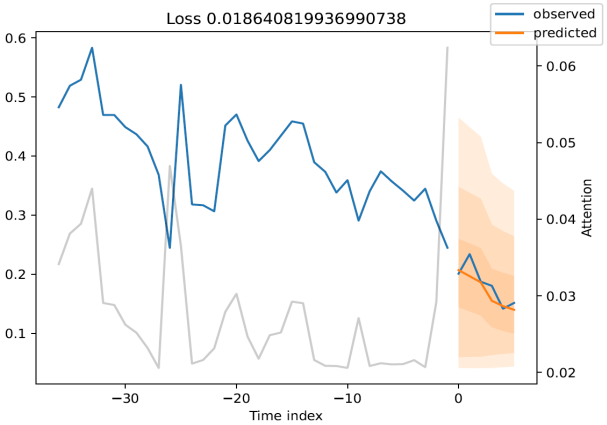

Figure 82: Eleventh plot for Trial 6.

Source: Author's own work.

Eleven plots present losses from 0.00999 to 0.03057 (median ≈ 0.02135). Higher learning rate and single head lead to moderate performance, underscoring sensitivity to optimization hyperparameters.

A.7 Trial 7

Model configuration: LR=0.10, HS=64, CHS=32, AHS=4

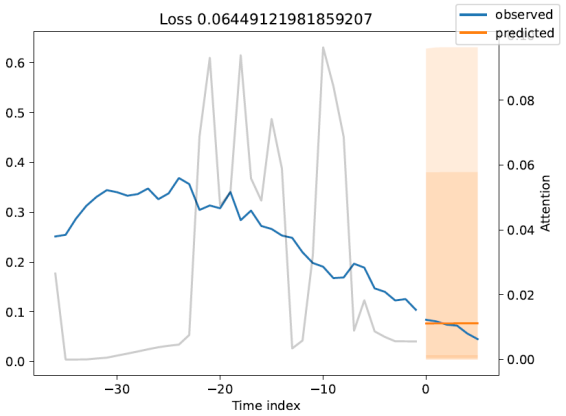

Figure 83: First plot for Trial 7.

Source: Author's own work.

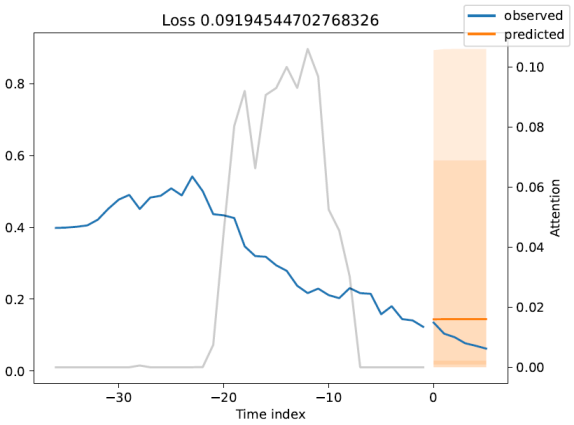

Figure 84: Second plot for Trial 7.

Source: Author's own work.

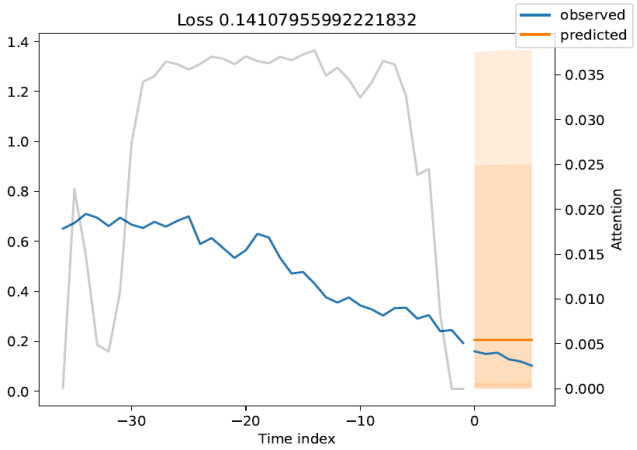

Figure 85: Third plot for Trial 7.

Source: Author's own work.

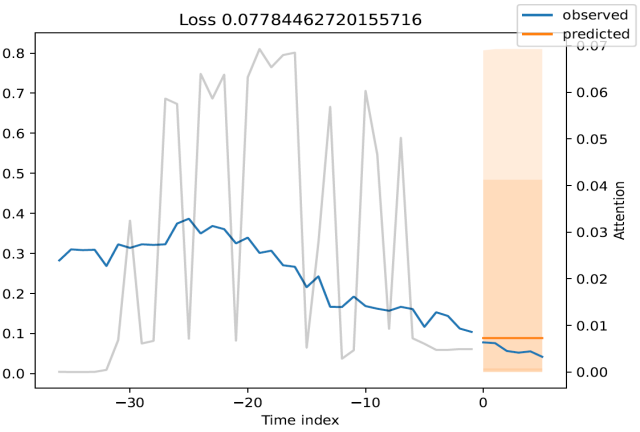

Figure 86: Forth plot for Trial 7.

Source: Author's own work.

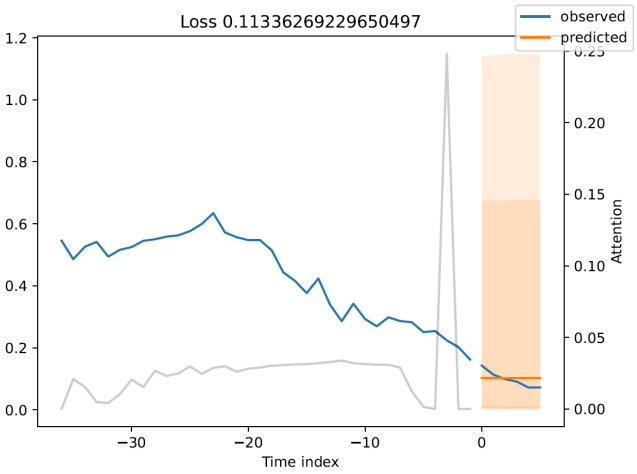

Figure 87: Fifth plot for Trial 7.

Source: Author's own work.

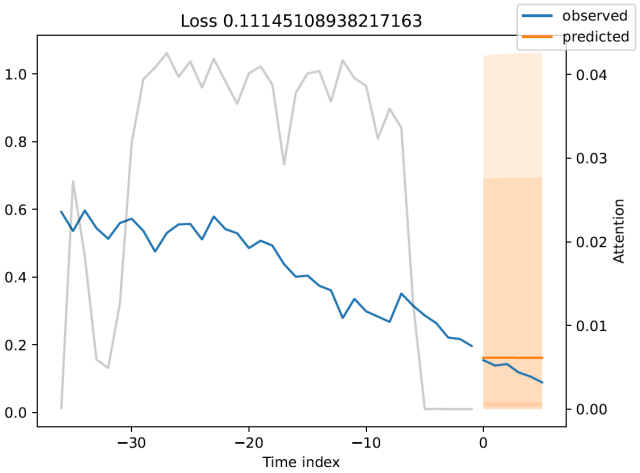

Figure 88: Sixth plot for Trial 7.

Source: Author's own work.

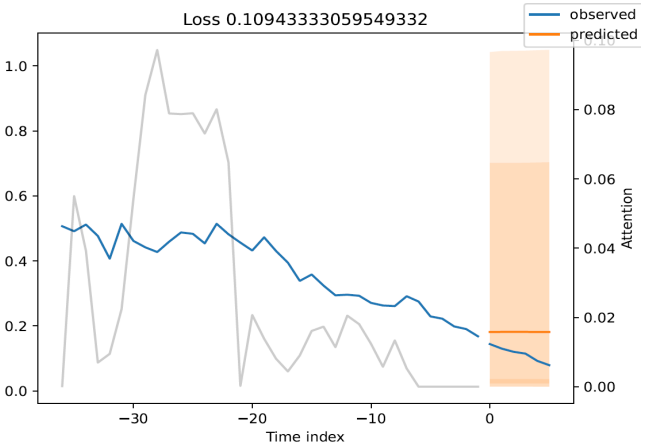

Figure 89: Seventh plot for Trial 7.

Source: Author's own work.

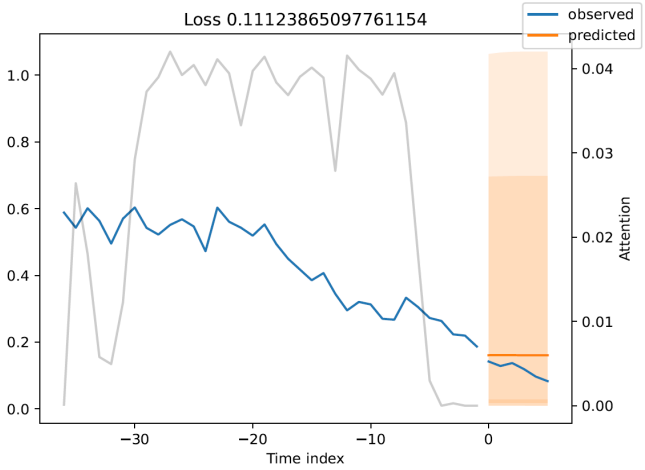

Figure 90: Eighth plot for Trial 7.

Source: Author's own work.

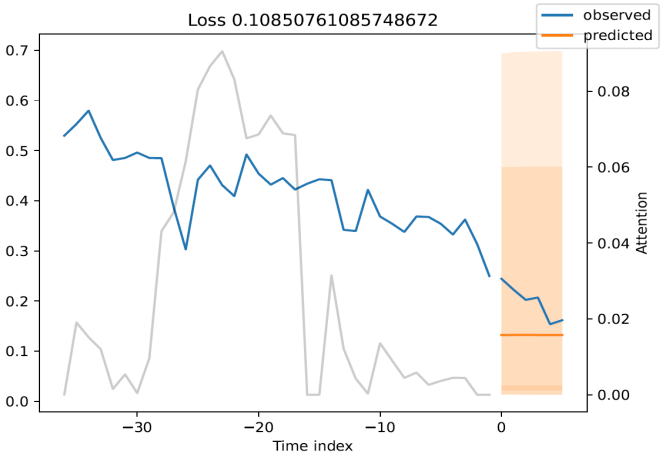

Figure 91: Ninth plot for Trial 7.

Source: Author's own work.

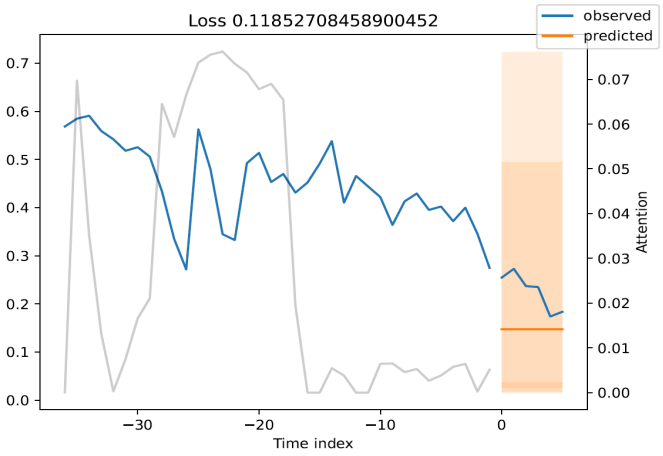

Figure 92: Tenth plot for Trial 7.

Source: Author's own work.

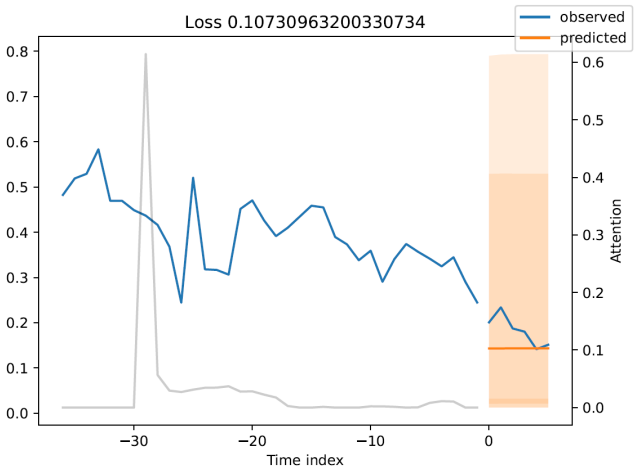

Figure 93: Eleventh plot for Trial 7.

Source: Author's own work.

Eleven sample losses range from 0.06449 to 0.14108 (median ≈ 0.10943). The largest model yields competitive tracking but also increased variance in error, indicating potential overfitting at this extreme capacity.

Notes on Plot Interpretation

• Top panel only is shown: observed volumes (blue) and TFT median forecasts (red) over a six-hour window.

• Loss annotation is the average quantile loss for each example, reflecting probabilistic forecast quality.

These plots validate that all seven TFT configurations produce forecasts following the general temporal patterns of traffic volume; differences in loss distributions corroborate the quantitative RMSE/MAE rankings reported in Chapter 5.

By aggregating these raw-prediction snapshots, Appendix A enables readers to confirm the model's visual performance under each hyperparameter setting and to relate qualitative forecast accuracy to the numerical metrics presented in the main text.

Appendix B: Complete PyTorch-Forecasting code snippets

```
[ ]  from IPython import get_ipython
     from IPython.display import display
     from google.colab import files
```

```
[ ]  # Upload files if in Colab (optional, depending on environment)
     uploaded = files.upload()
```

```
[ ]  # Install required packages (use these in Colab/Kaggle if needed)
     !pip install pytorch_forecasting --quiet
     !pip install pytorch-lightning --quiet
     !pip install lightning --quiet
     # Install or upgrade scikit-learn to ensure squared=False is available
     !pip install scikit-learn>=0.22 --quiet
```

```
[ ]  import numpy as np
     import pandas as pd
     import torch
     import matplotlib.pyplot as plt
     # from matplotlib.backends.backend_pdf import PdfPages # Not needed if not saving to PDF
     from sklearn.metrics import mean_absolute_error, mean_squared_error

     import lightning.pytorch as pl
     from lightning.pytorch.callbacks import EarlyStopping, LearningRateMonitor, ModelCheckpoint
     from lightning.pytorch.loggers import TensorBoardLogger

     from pytorch_forecasting import (
         TimeSeriesDataSet,
         TemporalFusionTransformer,
         GroupNormalizer,
         QuantileLoss
     )
```

```
[ ]  # define column lists (Defining them here makes them globally available)
     hour_cols   = [f"hour_{i}" for i in range(1,25)]
     prev_cols   = [f"prev_{i}" for i in range(1,11)]
     feata_cols  = [f"feata_{i}" for i in range(1,6)]
     featb_cols  = [f"featb_{i}" for i in range(1,5)]
     featc_cols  = [f"featc_{i}" for i in range(1,5)]
```

```
# Feature engineering for train_df
if 'train_df' in locals(): # Proceed only if train_df was loaded
    train_df = train_df.sort_values(["location","timestep"])
    train_df["time_idx"] = train_df.groupby("location").cumcount()
    # start_date = pd.to_datetime("2025-01-01") # Assuming this is defined from a previous successful run
    # If not defined, uncomment and define it:
    if 'start_date' not in locals():
         start_date = pd.to_datetime("2025-01-01")

    train_df["date"] = start_date + pd.to_timedelta(train_df["timestep"], unit="h")
    train_df["day_of_week"] = train_df["date"].dt.dayofweek.astype("category")
    train_df["is_weekend"]   = (train_df["day_of_week"].isin([5,6])).astype(int)
    # Added error handling for .str.extract in case hour_cols are all zero/NaN for a row
    try:
        train_df["time_bucket"] = (
            train_df[hour_cols].idxmax(axis=1)
            .astype(str).str.extract(r"hour_(\d+)", expand=False).fillna('0').astype(int) # .fillna('0') to handle rows with no max
        )
    except Exception as e:
        print(f"Error during train_df['time_bucket'] creation: {e}")
        # Fallback or inspect data if this fails

    train_df["time_block"] = pd.cut(
        train_df["time_bucket"], bins=[0,6,12,18,24],
        labels=["Night","Morning","Afternoon","Evening"],
        right=False, include_lowest=True # Include 0 in Night bin
    ).astype("category")
    train_df["month"] = train_df["date"].dt.month.astype("category")
    train_df["season"] = train_df["month"].map({
        12:"Winter",1:"Winter",2:"Winter",
        3:"Spring",4:"Spring",5:"Spring",
        6:"Summer",7:"Summer",8:"Summer",
        9:"Fall",10:"Fall",11:"Fall"
    }).astype("category")
    train_df["location"] = train_df["location"].astype(str)
    print("Feature engineering for train_df complete.")
    # print(train_df.head()) # Keep commented to reduce output
else:
    print("train_df not loaded, skipping feature engineering for train.")
```

```python
# Feature engineering for test_df (This block should be executed AFTER train_df FE and BEFORE the loop)
if 'test_df' in locals(): # Proceed only if test_df was loaded
    test_df = test_df.sort_values(["location","timestep"])
    test_df["time_idx"] = test_df.groupby("location").cumcount()
    # Use start_date from train FE - ensure it exists
    if 'start_date' not in locals():
         # Define start_date if it wasn't defined when processing train_df (shouldn't happen if train_df exists)
         print("Warning: start_date not found. Defining it for test_df FE.")
         start_date = pd.to_datetime("2025-01-01")

    test_df["date"] = start_date + pd.to_timedelta(test_df["timestep"], unit="h")
    test_df["day_of_week"] = test_df["date"].dt.dayofweek.astype("category")
    test_df["is_weekend"]   = (test_df["day_of_week"].isin([5,6])).astype(int)
    # Added error handling for .str.extract
    try:
        test_df["time_bucket"] = (
            test_df[hour_cols].idxmax(axis=1)
            .astype(str).str.extract(r"hour_(\d+)", expand=False).fillna('0').astype(int) # .fillna('0') to handle rows with no max
        )
    except Exception as e:
        print(f"Error during test_df['time_bucket'] creation: {e}")
        # Fallback or inspect data if this fails

    test_df["time_block"] = pd.cut(
        test_df["time_bucket"], bins=[0,6,12,18,24],
        labels=["Night","Morning","Afternoon","Evening"],
        right=False, include_lowest=True # Include 0 in Night bin
    ).astype("category")
    test_df["month"] = test_df["date"].dt.month.astype("category")
    test_df["season"] = test_df["month"].map({
        12:"Winter",1:"Winter",2:"Winter",
        3:"Spring",4:"Spring",5:"Spring",
        6:"Summer",7:"Summer",8:"Summer",
        9:"Fall",10:"Fall",11:"Fall"
    }).astype("category")
    test_df["location"] = test_df["location"].astype(str)
    print("\nFeature engineering for test_df complete.")
    # print(test_df.head()) # Keep commented to reduce output
else:
    print("test_df not loaded, skipping feature engineering for test.")
```

```python
[ ] # --- 2. TimeSeriesDataSet setup ---
    # Only proceed if both train_df and test_df were loaded and processed
    if 'train_df' in locals() and 'test_df' in locals():
        max_pred_len = 6
        max_enc_len  = 36
        # training_cutoff should be the LAST time_idx *included* in the training data
        training_cutoff = train_df["time_idx"].max() - max_pred_len

        # Check if training_cutoff is valid (i.e., training_df is not empty after cutoff)
        if training_cutoff < train_df["time_idx"].min():
            print(f"Error: training_cutoff ({training_cutoff}) is less than min time_idx ({train_df['time_idx'].min()}). Adjust max_pred_len or check data.")
            # You might want to exit or handle this case appropriately
            # Set a flag to skip dataloader creation if cutoff is invalid
            skip_dataloaders = True
        else:
            skip_dataloaders = False

            # The training dataset ends *at or before* the training_cutoff time_idx
            training = TimeSeriesDataSet(
                train_df[train_df.time_idx <= training_cutoff],
                time_idx="time_idx",
                target="traffic",
                group_ids=["location"],
                min_encoder_length=max_enc_len//2,
                max_encoder_length=max_enc_len,
                min_prediction_length=1,
                max_prediction_length=max_pred_len,
                static_categoricals=["location"],
                # Ensure all columns listed here actually exist in your dataframe
                time_varying_known_reals=["timestep","no_roads"] + hour_cols,
                # Ensure all columns listed here actually exist in your dataframe
                time_varying_unknown_reals=["traffic"] + prev_cols + feata_cols + featb_cols + featc_cols,
                target_normalizer=GroupNormalizer(groups=["location"], transformation="softplus"),
                add_relative_time_idx=True,
                add_target_scales=True,
                add_encoder_length=True,
                # Add allow_missing_timesteps=True if your data is not perfectly contiguous for all groups
                # allow_missing_timesteps=True,
            )
```

```
        # Create validation dataset
        # Use the full train_df and specify predict=True.
        # from_dataset will automatically handle the split based on training_cutoff
        # It will create sequences ending *after* training_cutoff for validation.
        # The encoder part of validation sequences will come from data <= training_cutoff,
        # and the decoder part will come from data > training_cutoff.
        # Ensure that for each series in the validation set, there is enough history
        # BEFORE training_cutoff to form an encoder sequence.
        # This is why filtering train_df AFTER cutoff was problematic.
        validation = TimeSeriesDataSet.from_dataset(
            training,
            train_df, # Use the full training dataframe
            predict=True, # Indicate that this dataset is for prediction/validation after training data
            stop_randomization=True # Useful for consistent validation/testing
        )

        # Check if validation dataset is not empty AFTER creation
        if len(validation) == 0:
            print("Warning: Validation dataset is empty after creation. Validation will be skipped.")
            validation = None # Explicitly set to None if empty


    # Proceed with dataloader creation only if not skipping
    if not skip_dataloaders:
        batch_size = 32
        # Check if training dataset is not empty before creating train_dataloader
        if len(training) > 0:
            train_dataloader = training.to_dataloader(train=True, batch_size=batch_size, num_workers=0) # Reduced workers for stability
        else:
            print("Warning: Training dataset is empty. Training will be skipped.")
            train_dataloader = None

        # Create validation dataloader only if validation dataset exists and is not empty
        if validation is not None and len(validation) > 0:
            val_dataloader   = validation.to_dataloader(train=False, batch_size=batch_size, num_workers=0) # Reduced workers
        else:
            val_dataloader = None
            print("Val dataloader not created as validation dataset is empty.")
```

```
        # Create test_dataloader here as it's needed for evaluation and plotting
        # Check if test_df is not empty
        if not test_df.empty:
            # Create test dataset using the training dataset parameters
            # Use the full test_df, predict=True means it will create sequences
            # suitable for prediction starting from the end of the encoder history within test_df.
            test_dataset = TimeSeriesDataSet.from_dataset(
                training,
                test_df, # Use the full test dataframe
                predict=True,
                stop_randomization=True
            )
            # Check if test dataset is not empty AFTER creation
            if len(test_dataset) > 0:
                test_dataloader = test_dataset.to_dataloader(train=False, batch_size=batch_size, num_workers=0) # Reduced workers
            else:
                print("Warning: Test dataset is empty after creation. Test evaluation will be skipped.")
                test_dataset = None
                test_dataloader = None
        else:
            print("Warning: Test DataFrame is empty. Test evaluation will be skipped.")
            test_dataset = None
            test_dataloader = None


        print("\nTimeSeriesDataSet and DataLoaders created.")
else:
    print("\nSkipping TimeSeriesDataSet and DataLoaders creation due to missing data.")
```

```
    # Callbacks & logger definitions
    # Ensure val_dataloader exists before defining EarlyStopping
    if 'val_dataloader' in locals() and val_dataloader is not None:
        early_stop = EarlyStopping(monitor="val_loss", min_delta=1e-4, patience=5, mode="min")
        callbacks = [LearningRateMonitor(), early_stop]
    else:
        print("Warning: Validation dataloader not available. EarlyStopping callback skipped.")
        callbacks = [LearningRateMonitor()] # Use only LR monitor if no validation

    lr_logger  = LearningRateMonitor() # Keep for separate use or include in `callbacks` list
```

```python
# Define ModelCheckpoint callback
# This will save the best model based on validation loss in a dedicated directory
# Only add if val_dataloader exists, otherwise monitor training loss (less ideal)
if 'val_dataloader' in locals() and val_dataloader is not None:
    checkpoint_callback = ModelCheckpoint(
        monitor="val_loss",
        dirpath="lightning_logs/checkpoints", # Save checkpoints in a general checkpoints dir
        filename="{epoch:02d}-{val_loss:.4f}",
        save_top_k=1,
        mode="min",
    )
    callbacks.append(checkpoint_callback) # Add to the list of callbacks

    print("Callbacks defined. Checkpointing enabled monitoring validation loss.")
else:
    # If no validation, checkpointing based on training loss (less meaningful for generalization)
    # Or skip checkpointing the "best" model based on loss
    checkpoint_callback = None # Explicitly set to None

    print("Callbacks defined. Checkpointing based on validation loss is disabled.")
```

```python
grid = [
    {"learning_rate":0.03, "hidden_size":32, "hidden_continuous_size":16, "attention_head_size":1},
    {"learning_rate":0.01, "hidden_size":16, "hidden_continuous_size": 8, "attention_head_size":1},
    {"learning_rate":0.01, "hidden_size":32, "hidden_continuous_size":16, "attention_head_size":2},
    {"learning_rate":0.03, "hidden_size":64, "hidden_continuous_size": 8, "attention_head_size":4},
    {"learning_rate":0.03, "hidden_size":16, "hidden_continuous_size":32, "attention_head_size":2},
    {"learning_rate":0.10, "hidden_size":32, "hidden_continuous_size": 8, "attention_head_size":1},
    {"learning_rate":0.10, "hidden_size":64, "hidden_continuous_size":32, "attention_head_size":4},
]

results = []

# Ensure scikit-learn is imported
import sklearn
from sklearn.metrics import mean_absolute_error, mean_squared_error

# Define manual RMSE calculation as a reliable fallback
def manual_rmse(y_true, y_pred):
    """Calculates RMSE manually using sqrt(MSE)."""
    y_true = np.asarray(y_true)
    y_pred = np.asarray(y_pred)
    # Add a check for empty arrays to prevent potential errors
    if y_true.size == 0 or y_pred.size == 0:
        return np.nan # Or some other indicator for no data
    return np.sqrt(mean_squared_error(y_true, y_pred))

# Attempt to use scikit-learn's squared=False method, fallback if it fails
def calculate_rmse(y_true, y_pred):
    """
    Calculates RMSE, attempting to use scikit-learn's squared=False
    and falling back to manual calculation if a TypeError occurs.
    """
    # Add a check for empty arrays
    if y_true.size == 0 or y_pred.size == 0:
        return np.nan # Or some other indicator for no data
    try:
        return mean_squared_error(y_true, y_pred, squared=False)
    except TypeError:
        print("Warning: 'squared=False' not supported by scikit-learn. Falling back to manual RMSE calculation.")
        return manual_rmse(y_true, y_pred)
    except ValueError as ve:
        print(f"ValueError during RMSE calculation: {ve}. Input shapes might be mismatched.")
        print(f"y_true shape: {y_true.shape}, y_pred shape: {y_pred.shape}")
        return np.nan # Return NaN on value errors like shape mismatch


print(f"Current scikit-learn version: {sklearn.__version__}")
```

```python
# Only run the training loop if DataLoaders were successfully created
# Check for train_dataloader and at least one evaluation dataloader
if 'train_dataloader' in locals() and train_dataloader is not None and ('val_dataloader' in locals() and val_dataloader is not None or 'test_dataloader' in locals() and test_dataloader is not None):
    for idx, params in enumerate(grid):
        print(f"\n=== Trial {idx}: {params} ===")
        # Instantiate TFT with this configuration
        tft = TemporalFusionTransformer.from_dataset(
            training, # Use the training dataset object for instantiation
            learning_rate=params["learning_rate"],
            hidden_size=params["hidden_size"],
            attention_head_size=params["attention_head_size"],
            hidden_continuous_size=params["hidden_continuous_size"],
            dropout=0.1,
            loss=QuantileLoss(),
            log_interval=10,
            reduce_on_plateau_patience=4,
        )

        # Instantiate TensorBoardLogger for each trial
        # Name and versioning help identify runs in TensorBoard
        logger = TensorBoardLogger("lightning_logs", name="tft_hyperparameter_tuning", version=idx)

        # Use the defined callbacks list
        trainer = pl.Trainer(
            max_epochs=100,
            accelerator="auto",
            gradient_clip_val=0.1,
            limit_train_batches=30,  # can be lowered for faster experimentation
            callbacks=callbacks, # Use the dynamic callbacks list
            logger=logger,
            # Set enable_progress_bar to True to see training progress, or False for cleaner output
            # enable_progress_bar=False,
            # Added check_val_every_n_epoch - useful for faster prototyping
            # check_val_every_n_epoch=5,
        )
```

```python
        # Train the model
        print("Starting training...")
        # Note: Manually interrupting the training cell (KeyboardInterrupt) may lead to a
        # NameError: name 'exit' is not defined within the pytorch-lightning library's
        # internal error handling mechanism. This is not an error in your code.
        # To avoid this, let the training finish or gracefully stop using Lightning's API
        # if needed in a production setting.
        try:
            # Pass val_dataloader only if it exists
            trainer.fit(tft, train_dataloader, val_dataloader if val_dataloader is not None else None)
            print("Training finished.")
        except KeyboardInterrupt:
            print("\nTraining interrupted by user. Skipping remaining evaluations for this trial.")
            # Optionally, you could save the partially trained model here
            # trainer.save_checkpoint(f"interrupted_model_trial_{idx}.ckpt")
            continue # Skip the rest of the loop for this interrupted trial
        except Exception as e:
            print(f"\nAn error occurred during training for Trial {idx}: {e}")
            continue # Skip the rest of the loop for this failed trial
```

```python
        # Evaluate on validation set (if available)
        val_mae, val_rmse, baseline_mae_val, baseline_rmse_val = np.nan, np.nan, np.nan, np.nan
        if 'val_dataloader' in locals() and val_dataloader is not None and len(val_dataloader) > 0:
            print("Evaluating on validation set...")
            try:
                actuals_val = torch.cat([y[0] for x,y in iter(val_dataloader)]).cpu().numpy()
                preds_val   = tft.predict(val_dataloader).cpu().numpy()
                mask_val    = ~np.isnan(actuals_val)
                actuals_val_masked = actuals_val[mask_val]
                preds_val_masked   = preds_val[mask_val]

                if len(actuals_val_masked) > 0:
                    val_mae  = mean_absolute_error(actuals_val_masked, preds_val_masked)
                    val_rmse = calculate_rmse(actuals_val_masked, preds_val_masked)
                    print(f"Val MAE: {val_mae:.4f}, Val RMSE: {val_rmse:.4f}")
                else:
                    print("No valid data after masking for validation metrics.")
```

```
                # Baseline on validation
                baseline_preds_val = []
                actuals_val_full   = []
                for x, y in iter(val_dataloader):
                     if x and "encoder_target" in x and x["encoder_target"].shape[1] > 0:
                        last_obs = x["encoder_target"][:, -1]
                        repeated = last_obs.unsqueeze(1).repeat(1, max_pred_len)
                        baseline_preds_val.append(repeated.cpu().numpy())
                        actuals_val_full.append(y[0].cpu().numpy())
                     else:
                         print("Warning: Skipping empty or malformed batch in val_dataloader for baseline calculation.")

                if baseline_preds_val and actuals_val_full:
                    baseline_preds_val = np.concatenate(baseline_preds_val, axis=0)
                    actuals_val_full   = np.concatenate(actuals_val_full, axis=0)
                    mask_val_full      = ~np.isnan(actuals_val_full)
                    baseline_preds_val_masked = baseline_preds_val[mask_val_full]
                    actuals_val_full_masked   = actuals_val_full[mask_val_full]

                    if len(actuals_val_full_masked) > 0:
                         baseline_mae_val  = mean_absolute_error(actuals_val_full_masked, baseline_preds_val_masked)
                         baseline_rmse_val = calculate_rmse(actuals_val_full_masked, baseline_preds_val_masked)
                         print(f"Baseline (val) MAE: {baseline_mae_val:.4f}, RMSE: {baseline_rmse_val:.4f}")
                    else:
                         print("No valid data after masking for validation baseline.")
                else:
                    print("No data accumulated for validation baseline.")
            except Exception as eval_val_err:
                 print(f"An error occurred during validation evaluation for Trial {idx}: {eval_val_err}")
        else:
             print("Skipping validation evaluation as val_dataloader is not available or empty.")
```

```
[ ]     # Evaluate on test set (if available)
        test_mae, test_rmse, baseline_mae_test, baseline_rmse_test = np.nan, np.nan, np.nan, np.nan
        if 'test_dataloader' in locals() and test_dataloader is not None and len(test_dataloader) > 0:
            print("Evaluating on test set...")
            try:
                actuals_test = torch.cat([y[0] for x,y in iter(test_dataloader)]).cpu().numpy()
                preds_test   = tft.predict(test_dataloader).cpu().numpy()
                mask_test    = ~np.isnan(actuals_test)
                actuals_test_masked = actuals_test[mask_test]
                preds_test_masked   = preds_test[mask_test]

                if len(actuals_test_masked) > 0:
                    test_mae  = mean_absolute_error(actuals_test_masked, preds_test_masked)
                    test_rmse = calculate_rmse(actuals_test_masked, preds_test_masked)
                    print(f"Test MAE: {test_mae:.4f}, Test RMSE: {test_rmse:.4f}")
                else:
                    print("No valid data after masking for test metrics.")
```

```
                # Baseline on test: repeat last observed traffic
                baseline_preds_test = []
                actuals_test_full   = []
                for x, y in iter(test_dataloader):
                    if x and "encoder_target" in x and x["encoder_target"].shape[1] > 0:
                        last_obs = x["encoder_target"][:, -1]
                        repeated = last_obs.unsqueeze(1).repeat(1, max_pred_len)
                        baseline_preds_test.append(repeated.cpu().numpy())
                        actuals_test_full.append(y[0].cpu().numpy())
                    else:
                        print("Warning: Skipping empty or malformed batch in test_dataloader for baseline calculation.")

                if baseline_preds_test and actuals_test_full:
                    baseline_preds_test = np.concatenate(baseline_preds_test, axis=0)
                    actuals_test_full   = np.concatenate(actuals_test_full, axis=0)
                    mask_test_full      = ~np.isnan(actuals_test_full)
                    baseline_preds_test_masked = baseline_preds_test[mask_test_full]
                    actuals_test_full_masked   = actuals_test_full[mask_test_full]

                    if len(actuals_test_full_masked) > 0:
                         baseline_mae_test  = mean_absolute_error(actuals_test_full_masked, baseline_preds_test_masked)
                         baseline_rmse_test = calculate_rmse(actuals_test_full_masked, baseline_preds_test_masked)
                         print(f"Baseline (test) MAE: {baseline_mae_test:.4f}, RMSE: {baseline_rmse_test:.4f}")
                    else:
                         print("No valid data after masking for test baseline.")
                else:
                    print("No data accumulated for test baseline.")
            except Exception as eval_test_err:
                print(f"An error occurred during test evaluation for Trial {idx}: {eval_test_err}")
        else:
            print("Skipping test evaluation as test_dataloader is not available or empty.")
```

```
    # Record all metrics
    results.append({
        **params,
        "val_mae": val_mae,
        "val_rmse": val_rmse,
        "baseline_val_mae": baseline_mae_val,
        "baseline_val_rmse": baseline_rmse_val,
        "test_mae": test_mae,
        "test_rmse": test_rmse,
        "baseline_test_mae": baseline_mae_test,
        "baseline_test_rmse": baseline_rmse_test,
    })
```

```
    # --- Add Prediction Plotting Inside the Loop (displays directly) ---
    print(f"\nPlotting sample predictions for Trial {idx}: {params}")

    # Set the currently trained model to evaluation mode
    tft.eval()

    # Get raw predictions on the test set using the current model
    # test_dataloader should be available from before the loop
    try:
        # Ensure test_dataloader is not empty before predicting
        if 'test_dataloader' in locals() and test_dataloader is not None and len(test_dataloader) > 0:
            # Limit the number of batches to process for plotting to avoid excessive memory usage
            num_batches_for_plot = min(5, len(test_dataloader))
            if num_batches_for_plot > 0:
                raw_predictions = tft.predict(test_dataloader, mode="raw", return_x=True, limit_batches=num_batches_for_plot)

                # Check if raw_predictions has the expected structure and is not empty
                if hasattr(raw_predictions, 'x') and raw_predictions.x is not None and len(raw_predictions.x) > 0:
                    # Display a few sample plots (e.g., first 5)
                    num_plots_to_show = min(5, len(raw_predictions.x))
                    print(f"Generating and displaying {num_plots_to_show} sample plots...")
                    for plot_idx in range(num_plots_to_show):
                        try:
                            fig = tft.plot_prediction(
                                raw_predictions.x, raw_predictions.output, idx=plot_idx, add_loss_to_title=True
                            )
                            # Display the current figure
                            plt.show()
                            # Close the figure to free up memory
                            plt.close(fig)
                        except Exception as inner_plot_err:
                            print(f"Error plotting sample {plot_idx}: {inner_plot_err}")


                    print(f"{num_plots_to_show} sample plots for trial {idx} displayed.")
                else:
                    print("Raw predictions object is empty or has unexpected structure from limited batches, cannot generate plots.")
            else:
                print("No batches available in test_dataloader for plotting.")
        else:
            print("Test dataloader is not available or is empty, cannot generate predictions for plotting.")


    except Exception as plot_err:
        print(f"An error occurred during sample plotting for Trial {idx}: {plot_err}")
    # --- End of Prediction Plotting ---
```

```
    # Display per-location Actual vs. Predicted plots for this trial
    print(f"\nDisplaying per-location plots for Trial {idx:02d}...")
    # Ensure test_df is available
    if 'test_df' in locals() and test_df is not None and not test_df.empty:
        unique_locations = test_df["location"].unique()
        # Limit the number of locations to plot to avoid overwhelming output
        num_locations_to_plot = min(5, len(unique_locations)) # Display plots for a few locations

        print(f"  Displaying plots for {num_locations_to_plot} sample locations...")
        for i, location in enumerate(unique_locations[:num_locations_to_plot]):
            print(f"    Plotting Location: {location} for Trial {idx}")
            location_df = test_df[test_df["location"] == location].copy()

            # Ensure there's data for this location
            if not location_df.empty:
                # Ensure there is enough data for a full sequence (encoder + decoder)
                # A sequence needs at least min_encoder_length + min_prediction_length timesteps
                min_sequence_length = training.min_encoder_length + training.min_prediction_length
                if len(location_df) >= min_sequence_length:
                    # Filter dataset to include only the data for this location
                    # Ensure 'training' dataset is available to create the dataset from
                    if 'training' in locals() and training is not None:
                        location_dataset = TimeSeriesDataSet.from_dataset(
                            training, # Use the training dataset definition for structure
                            location_df,
                            predict=True,
                            stop_randomization=True
                        )
```

```
                    # Ensure the location_dataset is not empty after filtering
                    if len(location_dataset) > 0:
                        # Use a batch size that is at least the size of the sequence
                        # to ensure full sequences are processed for plotting
                        (variable) location_batch_size: int et is valid but small
                        location_batch_size = max(1, len(location_dataset))
                        location_dataloader = location_dataset.to_dataloader(
                            train=False, batch_size=location_batch_size, num_workers=0
                        )

                        try:
                            actuals_loc = torch.cat([y[0] for x,y in iter(location_dataloader)]).cpu().numpy()
                            preds_loc   = tft.predict(location_dataloader).cpu().numpy()
                            mask_loc    = ~np.isnan(actuals_loc)
                            actuals_loc = actuals_loc[mask_loc]
                            preds_loc   = preds_loc[mask_loc]

                            if len(actuals_loc) > 0:
                                fig, ax = plt.subplots(figsize=(10,5))
                                # Get time_idx for x-axis if possible, otherwise use index
                                if 'time_idx' in location_df.columns:
                                    # Need to align actuals/preds with time_idx.
                                    # This is tricky as actuals/preds are flattened across batches.
                                    # A simpler approach for plotting is to plot against sequence step index.
                                    ax.plot(actuals_loc, label="Actual")
                                    ax.plot(preds_loc,   label="Predicted")
                                else:
                                    ax.plot(actuals_loc, label="Actual")
                                    ax.plot(preds_loc,   label="Predicted")

                                ax.set_title(
                                    f"Trial {idx}: lr={params['learning_rate']}, hs={params['hidden_size']}, "
                                    f"hcs={params['hidden_continuous_size']}, ahs={params['attention_head_size']}\n"
                                    f"Location: {location} (Test Data)"
                                )
                                ax.set_xlabel("Relative Time Step (Prediction Horizon)")
                                ax.set_ylabel("Traffic")
                                ax.legend()
                                plt.tight_layout()
                                plt.show()  # Display the plot
                                plt.close(fig) # Close the figure to free memory
                            else:
                                print(f"      No valid data to plot after masking for {location} in Trial {idx}")
                        except Exception as loc_plot_err:
                            print(f"      An error occurred while plotting for location {location} in Trial {idx}: {loc_plot_err}")

                    else:
                         print(f"      Location dataset is empty after filtering for {location} in Trial {idx}")
```

```
                        else:
                            print(f"      Location dataset is empty after filtering for {location} in Trial {idx}")
                    else:
                        print("      Training dataset definition not available to create location dataset.")
                else:
                    print(f"      Not enough data for location {location} ({len(location_df)} timesteps) to form a full sequence for Trial {idx} (needs at least {min_sequence_length}).")
            else:
                print(f"      No data found for location {location} in test_df for Trial {idx}")

        if num_locations_to_plot < len(unique_locations):
            print(f"  ... skipping {len(unique_locations) - num_locations_to_plot} locations to keep output concise.")

    else:
        print("  test_df not available or empty, cannot display per-location plots.")

    # --- End of per-location plotting ---
```

```
    # Summarize all trials
    summary_df = pd.DataFrame(results)
    summary_df.to_csv("tft_hyperparam_summary.csv", index=False)
    print("\nAll trials complete. Summary saved to tft_hyperparam_summary.csv")
else:
    print("\nSkipping hyperparameter tuning loop due to missing data or dataloaders.")
```

```python
# Assuming summary_df has been created from the results list
if 'summary_df' in locals() and not summary_df.empty:
    # Find the row with the minimum test_rmse
    best_model_row_rmse = summary_df.loc[summary_df['test_rmse'].idxmin()]
    print("Best model based on Test RMSE:")
    print(best_model_row_rmse)

    # Find the row with the minimum test_mae
    best_model_row_mae = summary_df.loc[summary_df['test_mae'].idxmin()]
    print("\nBest model based on Test MAE:")
    print(best_model_row_mae)

    # --- Add the snippet to plot a specific prediction from the best model ---
    print("\nAttempting to plot a specific prediction from the best model...")

    # Load the best model from the checkpoint
    best_model_path = None
    if 'checkpoint_callback' in locals() and checkpoint_callback.best_model_path:
      best_model_path = checkpoint_callback.best_model_path

    if best_model_path:
        print(f"Loading best model from: {best_model_path}")
        try:
            best_tft = TemporalFusionTransformer.load_from_checkpoint(best_model_path)
            print("Best model loaded successfully.")

            # Generate and plot a specific prediction
            try:
                # IMPORTANT: Adjust this filter to select a specific time series from YOUR training data.
                # Use the actual grouping columns (like 'location') and time index.
                # The example filter below is based on your previous snippet's structure.
                # You will likely need to change 'agency' and 'sku' to 'location'
                # and adjust the condition for 'time_idx_first_prediction' based on what you want to plot.
                # Example using 'location' and a specific time_idx:
                # Find a sample location from the test set to plot
                location_to_plot = test_df['location'].iloc[0] if not test_df.empty else None
                time_idx_to_plot = test_df['time_idx'].iloc[0] if not test_df.empty else None # Use an early time_idx

                if location_to_plot is not None and time_idx_to_plot is not None:
                     # Find a row in the test data that matches the criteria to get the correct time_idx_first_prediction
                     # We'll plot the prediction starting from an encoder length before this time_idx
                    encoder_start_time_idx = time_idx_to_plot # You might adjust this logic
```

```python
                # Filter the test dataset based on the location and a time_idx that allows for a full prediction window
                # We need an encoder length before the prediction starts.
                # The prediction for a time_idx `t` starts at `t + max_pred_len`
                # The encoder ends at `t + max_pred_len - 1`
                # The encoder starts at `t + max_pred_len - 1 - max_enc_len + 1 = t + max_pred_len - max_enc_len`
                # So, to plot the prediction for a window starting after time_idx `t`,
                # the time_idx for the TimeSeriesDataSet filtering should be the end of the encoder.
                # Let's try to find an example that will result in a valid prediction window.
                # A simple approach is to pick a time_idx that is at least `max_enc_len` hours into the series for that location.

                example_row_test = test_df[
                    (test_df['location'] == location_to_plot) &
                    (test_df['time_idx'] >= max_enc_len) # Ensure we have enough history
                ].iloc[0] if not test_df[
                    (test_df['location'] == location_to_plot) &
                    (test_df['time_idx'] >= max_enc_len)
                ].empty else None


                if example_row_test is not None:
                    # The time_idx for filtering the dataset should be the end of the encoder,
                    # which is max_pred_len - 1 steps before the start of the prediction window.
                    # The prediction starts after the last encoder step.
                    # If we want to predict for time_idx T, the encoder ends at T-1.
                    # So, the filter time_idx should be T - 1 + max_pred_len.
                    # Let's filter based on the time_idx in the test data that corresponds to the *end* of the prediction window
                    # for which we want to plot.
                    # A simpler way is to filter based on the start time_idx of the sequence you want to predict.
                    # In TimeSeriesDataSet, the time_idx represents the end of the prediction horizon for a sequence.
                    # If you want to predict from time_idx `t`, the sequence used ends at `t`.
                    # So filter where `time_idx == example_row_test['time_idx']`. This represents the end of the prediction window.

                    filtered_dataset_for_plot = test_dataset.filter(
                        lambda x: (x.location == location_to_plot) and
                                  (x.time_idx == example_row_test['time_idx']) # Filter on the end of the prediction horizon
                    )

                    if len(filtered_dataset_for_plot) > 0:
                        print(f"Generating raw prediction for Location: {location_to_plot}, Prediction ending at time_idx: {example_row_test['time_idx']}")
                        raw_prediction_specific = best_tft.predict(
                            filtered_dataset_for_plot,
                            mode="raw",
                            return_x=True,
                            trainer_kwargs=dict(accelerator="cpu"), # Or "gpu" if available
                        )
```

```
                        if hasattr(raw_prediction_specific, 'x') and len(raw_prediction_specific.x) > 0:
                            print(f"Plotting specific prediction for location: {location_to_plot}, starting from encoder ending at time_idx: {raw_prediction_specific.x['time_idx'][0][0]} + max_enc_len - 1")
                            best_tft.plot_prediction(raw_prediction_specific.x, raw_prediction_specific.output, idx=0)
                            plt.show() # Display the plot
                        else:
                            print("Filtered dataset for specific plot is empty or has unexpected structure after prediction.")
                    else:
                        print(f"No data found in test dataset for location '{location_to_plot}' ending at time_idx {example_row_test['time_idx']} to generate a specific plot.")


                else:
                    print(f"Could not find a suitable example in the test data for location '{location_to_plot}' starting around time_idx {time_idx_to_plot} with enough history to generate a specific plot.")


            else:
                print("Test data is empty or location/time_idx not found to generate specific plot.")


        except Exception as e:
            print(f"An error occurred while generating the specific prediction plot: {e}")

    except Exception as load_err:
        print(f"Error loading the best model from checkpoint: {load_err}")
        print("Skipping specific prediction plot.")

else:
    print("\nNo best model checkpoint path found. Skipping specific prediction plot.")

else:
    print("Summary DataFrame is not available. Please ensure the hyperparameter tuning loop ran successfully.")
```

```
# --- Code to load and use the best model found ---
print("\n--- Using the Best Model ---")

# Assuming summary_df and checkpoint_callback are available from previous cells

# 1. Identify the best hyperparameters (optional, but good for verification)
if 'summary_df' in locals() and not summary_df.empty:
    best_model_row_mae = summary_df.loc[summary_df['test_mae'].idxmin()]
    best_params = {
        "learning_rate": best_model_row_mae["learning_rate"],
        "hidden_size": best_model_row_mae["hidden_size"],
        "hidden_continuous_size": best_model_row_mae["hidden_continuous_size"],
        "attention_head_size": best_model_row_mae["attention_head_size"],
    }
    print(f"Hyperparameters of the best model (based on test MAE):\n{best_params}")
else:
    print("Summary DataFrame not available. Cannot explicitly identify best hyperparameters.")
    best_params = None # Or load from a hardcoded value if you know it
```

```
[ ]  # 2. Load the best model from the checkpoint
    best_model_path = None
    if 'checkpoint_callback' in locals() and checkpoint_callback.best_model_path:
        best_model_path = checkpoint_callback.best_model_path

    if best_model_path:
        print(f"\nLoading best model from checkpoint: {best_model_path}")
        try:
            # Make sure you use the correct dataset object (training) to load the model
            # The TimeSeriesDataSet object used to train the model is needed for loading.
            if 'training' in locals():
                best_tft = TemporalFusionTransformer.load_from_checkpoint(best_model_path)
                print("Best model loaded successfully.")

                # 3. Use the best model for predictions (example: predict on test set again)
                print("\nGenerating predictions on the test set using the best model...")
                # Ensure test_dataloader is available
                if 'test_dataloader' in locals():
                    best_tft.eval() # Set the model to evaluation mode
                    # Predict on the test dataloader
                    predictions_best_model = best_tft.predict(test_dataloader)
                    print("Predictions from the best model generated.")

                    # You can now use 'predictions_best_model' for further analysis,
                    # like evaluating performance again or generating submission files.
                    # Example: Calculate MAE and RMSE with the best model's predictions
                    # Re-calculate metrics directly from the best model's predictions
                    if 'actuals_test_masked' in locals(): # actuals_test_masked should be available from the loop
                        predictions_best_model_numpy = predictions_best_model.cpu().numpy()

                        # Ensure predictions match the masked actuals
                        # The shape might differ slightly if the dataloader batches are not perfectly aligned
                        # A more robust way is to re-create actuals and apply the mask
                        actuals_test_best = torch.cat([y[0] for x,y in iter(test_dataloader)]).cpu().numpy()
                        mask_test_best    = ~np.isnan(actuals_test_best)
                        actuals_test_masked_best = actuals_test_best[mask_test_best]
                        predictions_best_model_masked = predictions_best_model_numpy[mask_test_best]
```

```
                    # Ensure the lengths match before calculating metrics
                    if len(actuals_test_masked_best) == len(predictions_best_model_masked):
                        best_test_mae = mean_absolute_error(actuals_test_masked_best, predictions_best_model_masked)
                        best_test_rmse = calculate_rmse(actuals_test_masked_best, predictions_best_model_masked)

                        # Corrected the print statement by closing the f-string
                        print(f"\nBest Model Performance on Test Set (MAE): {best_test_mae:.4f}, (RMSE): {best_test_rmse:.4f}")

                    else:
                        print("Length mismatch between actuals and predictions for best model evaluation.")
                else:
                    print("actuals_test_masked not available for best model evaluation.")

            else:
                print("Test dataloader not available. Skipping predictions with best model.")

        else:
            print("Training dataset definition not available. Cannot load best model.")


    except Exception as load_err:
        print(f"Error loading the best model from checkpoint: {load_err}")
        print("Skipping the use of the best model.")

else:
    print("\nNo best model checkpoint path found. Cannot load or use the best model.")
```