Rheinland-Pfälzische
Technische Universität
Kaiserslautern
Landau

Master Thesis

DNN-basierte virtuelle Trajektoriengenerierung für autonome Fahrzeuge: Fokus auf die Berechnung des lokalen Referenzpfads

DNN-based Virtual Trajectory Generation for Autonomous Vehicles: Focus on Local Reference Path Computation

LEHRSTUHL FÜR MECHATRONIK IN MASCHINENBAU UND FAHRZEUGTECHNIK
FACHBEREICH MASCHINENBAU UND VERFAHRENSTECHNIK
RHEINLAND-PFÄLZISCHE TECHNISCHE UNIVERSITÄT KAISERSLAUTERN-LANDAU

Master Student:	Sarvesh Bhalchandra Telang
First Reviewer:	Prof. Dr.-Ing. Naim Bajcinca
Second Reviewer:	Dr. Sandesh Hiremath

State Date:	07.06.2024
Release Date:	09.12.2024
Matr.-Nr.:	422444

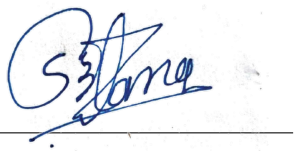
Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit eigenständig verfasst, nicht zu Prüfungszwecken an anderer Stelle eingereicht und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche bewusst genutzten Textausschnitte, Zitate oder Inhalte anderer Autoren sind ausdrücklich als solche gekennzeichnet

Declaration of Authorship

I hereby confirm that I have written this thesis independently, that I have not submitted it for academic assessment elsewhere, and that I have used only the resources indicated. Any extracts, quotations or ideas from other sources that have been knowingly used are clearly identified as such.

Kaiserslautern, December, 2024



Sarvesh Bhalchandra Telang

Preface

I would like to express my sincere gratitude to my professor Naim Bajcinca for accepting me to work on this thesis topic and giving me the opportunity to pursue this research. I would like to specially thank Dr. Sandesh Hiremath for his invaluable guidance, as well as the constant support throughout the course of this thesis. His insights and expertise have been instrumental in moving this work forward, helping me to overcome many challenges and greatly enriching my research experience. His encouragement and constructive feedback have broadened my understanding of the field.

I am also deeply grateful to my family and friends for their unwavering support, patience, and encouragement. Their belief in me has been a constant source of motivation, empowering me to persevere and complete this journey successfully.

Abstract

This research addresses the challenges of trajectory generation for autonomous vehicles by developing an efficient and modular framework to compute local reference paths for lane-keeping systems (LKS). The research focuses on designing a lightweight Deep Neural Network (DNN) to predict future lane coordinates from past sequential road lane data, enabling the generation of reliable reference paths. The primary objective is to integrate these virtual trajectories into an existing Model Predictive Control (MPC) simulation system and evaluate their robustness and efficiency during vehicle tracking.

Building on advancements in vision-based MPC approaches, the methodology pre-extracts lane coordinates using Ultra-Fast Structure-Aware Lane Detection (ULFD) and employs a Seq2Seq Transformer model for temporal regression. Unlike prior methods that rely on computationally intensive multi-modal data, raw visual inputs, or simple averaging techniques, this framework adopts a spatiotemporal inference approach to reduce computational overhead while maintaining robustness in dynamic driving scenarios, such as missing lanes, outliers, and lane changes.

Integration with an image-based MPC framework demonstrated smooth lateral motion control, reducing trajectory deviation to an average MAE of 0.57 m and a Fréchet distance of 0.87 m. Although longer forecast horizons resulted in marginal accuracy drops (2–3%), the system maintained strong performance under varying road conditions. Extensive experiments were conducted on datasets from TU Simple, CU Lane, and RPTU Universität, exploring various DNN architectures designed for multivariate time-series analysis. This thesis contributes to autonomous vehicle research by presenting an efficient and scalable trajectory generation framework that integrates seamlessly with model-based systems, enhancing the reliability and adaptability of lane-keeping solutions.

Keywords: Trajectory generation, Deep Neural Network (DNN), Lane-Keeping System (LKS), Autonomous Vehicles, Model Predictive Control (MPC)

Abbreviations

VTG	Virtual Trajectory Generation
DNN	Deep Neural Network
ADAS	Advanced Driver Assistance Systems
GPS	Global Positioning System
LKS	Lane Keeping System
MPC	Model Predictive Controller
PID	Proportional Integral Derivative
E2E	End-to-End
DCNN	Deep Convolutional Neural Network
IPOPT	Interior Point Optimizer
LSTM	Long Short Term Memory
RNN	Recurrent Neural Network
iMPC	Image-based Model Predictive Control
ULFD	Ultra Fast Structure-Aware Lane Detection
GRU	Gated Recurrent Unit
TCN	Temporal Convolutional Network
Seq2Seq	Sequence-to-Sequence
CARLA	Car Learning To Act
RGB	Red Green Blue
FOV	Field Of View
DOF	Depth Of Field

SARIMA	Seasonal Autoregressive Moving Average
DNN	Holt-Winters Exponential Smoothing
DGP	Data Generating Process
TFT	Temporal Fusion Transformer
NLP	Natural Language Processing
MHA	Multi-Head Attention
MAE	Mean Absolute Error
MSE	Mean Squared Error
RMSE	Root Mean Squared Error
MAD	Mean Absolute Deviation
KNN	K-Nearest Neighbour
RLR	Road Lane Re-construction
RPTU	Rheinland-Pfälzische Technische Universität

Contents

List of Figures

List of Tables

1. Introduction

Advancements in autonomous and semi-autonomous technologies have led to the adoption of various levels of autonomy, each incorporating a range of Advanced Driver Assistance Systems (ADAS) [12]. The high-level architecture commonly followed in these systems, as shown in figure 1.1, integrates perception, planning, and control modules. The perception module processes ego vehicle sensor data and environmental information to create a map and estimate the vehicle's state, including tasks such as lane detection, semantic segmentation, and depth estimation. This information feeds into the planning module, where trajectory generation is performed at both global and local levels. Finally, the control module executes the planned trajectory through vehicle kinematics and dynamics control. These interconnected components work in harmony to ensure safe and efficient autonomous navigation.

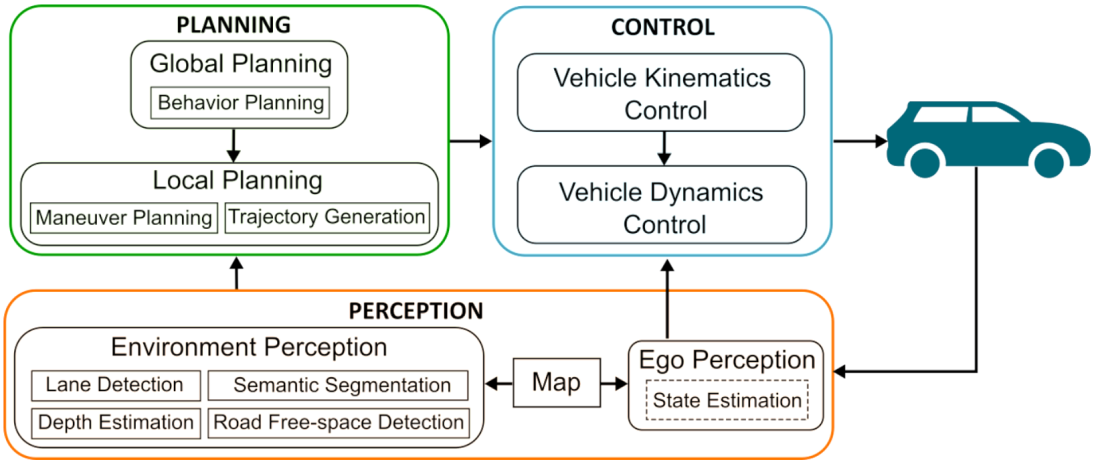


Figure 1.1: Architecture of Autonomous Navigation system. [1]

The autonomous navigation field organizes its techniques into two categories: planning and reacting [10]. Global path planning focuses on creating a comprehensive route that navigates the vehicle toward its target destination. Global planning assumes prior knowledge of the entire environment, including obstacle positions and lane boundaries [10].

On the other hand, reacting techniques fall under local path planning, which focuses on generating several local candidate paths based on real-time sensor data that represent the local environment [2], as illustrated in figure 1.2.

From the perspective of autonomous driving, lane keeping is a fundamental component of control, guiding the vehicle to maintain its position within the center of the lane while executing a planned path toward its destination. Lane keeping relies on accurate sensors to localize the vehicle with respect to lane boundaries and assess its lateral position. Smooth control mechanisms then ensure that the vehicle stays centered, correcting any drift. This technology significantly enhances safety and contributes to higher levels of autonomy [12].

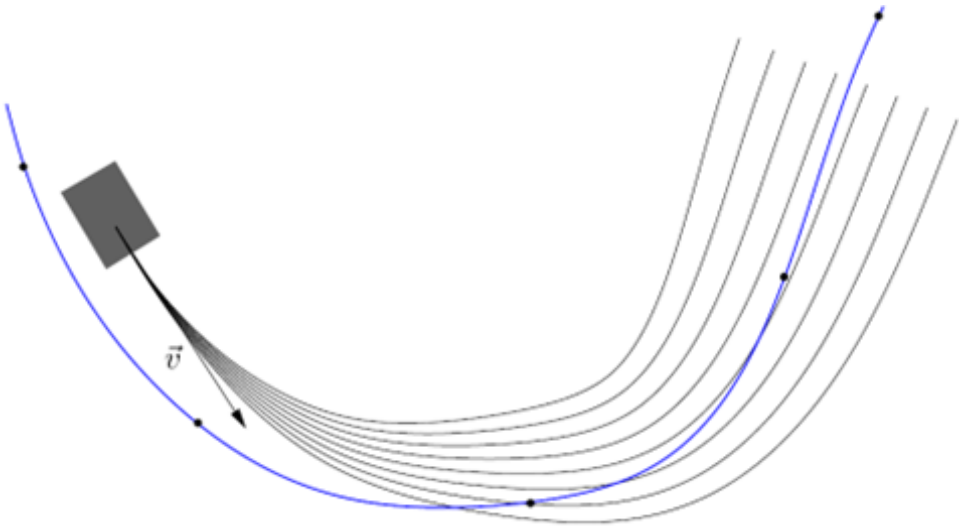


Figure 1.2: Global path and Local candidate paths. [2]

To maintain the vehicle's lane position, the control subsystem generates appropriate steering and acceleration/braking commands based on the planned path and desired speed. Various control techniques, including Proportional-Integral-Derivative (PID) Controllers, Full-state Feedback Controllers, Sliding Mode Controllers, and Adaptive and Fuzzy Controllers, have been applied to lane-keeping systems [12]. However, many of these methods independently design lateral (steering) and longitudinal (acceleration/braking) controls, overlooking their coupling effects, which can degrade performance at higher speeds or in curves [12].

In contrast, Model Predictive Control (MPC) offers an advanced approach by generating optimal control signals over a finite time horizon while incorporating vehicle dynamics and respecting constraints [12]. By leveraging motion models, MPC predicts the vehicle's future states and determines control commands through iterative constrained optimization [12]. It uses the planned path, commonly referred to as the reference path, as input and ensures the vehicle stays close to this trajectory [1].

This thesis proposes a novel method for generating the local reference path, which serves as an essential input to the MPC controller. Robust perception systems and reference path generation are critical for ensuring the safety and reliability of lane-keeping, particularly under varying environmental conditions and external dynamic factors. The approach presented in this work aims to enhance the performance of the reference path in lateral motion controllers, contributing to higher levels of safety and autonomy in autonomous vehicles.

1.1 State-of-the-Art

Trajectory planning systems frequently rely on a combination of high-definition maps or navigation maps and GPS sensors to localize and plan the vehicle's path toward its destination [12]. Mapping both the system's current state and its intended trajectory within a global coordinate framework simplifies the design of Model Predictive Control (MPC). However, GPS data is often unreliable and prone to noise, especially under challenging road conditions. For example, GPS signals can become highly inaccurate or even completely unavailable in urban environments, tunnels, or rural areas surrounded by dense trees, disrupting localization and trajectory planning systems. To address these limitations, vision-based localization and planning methods have been proposed in [12] and [1] as an alternative for tasks like lane following and lane keeping until GPS data becomes usable again. Figure 1.3 illustrates the overview of current practices and the proposed approach.

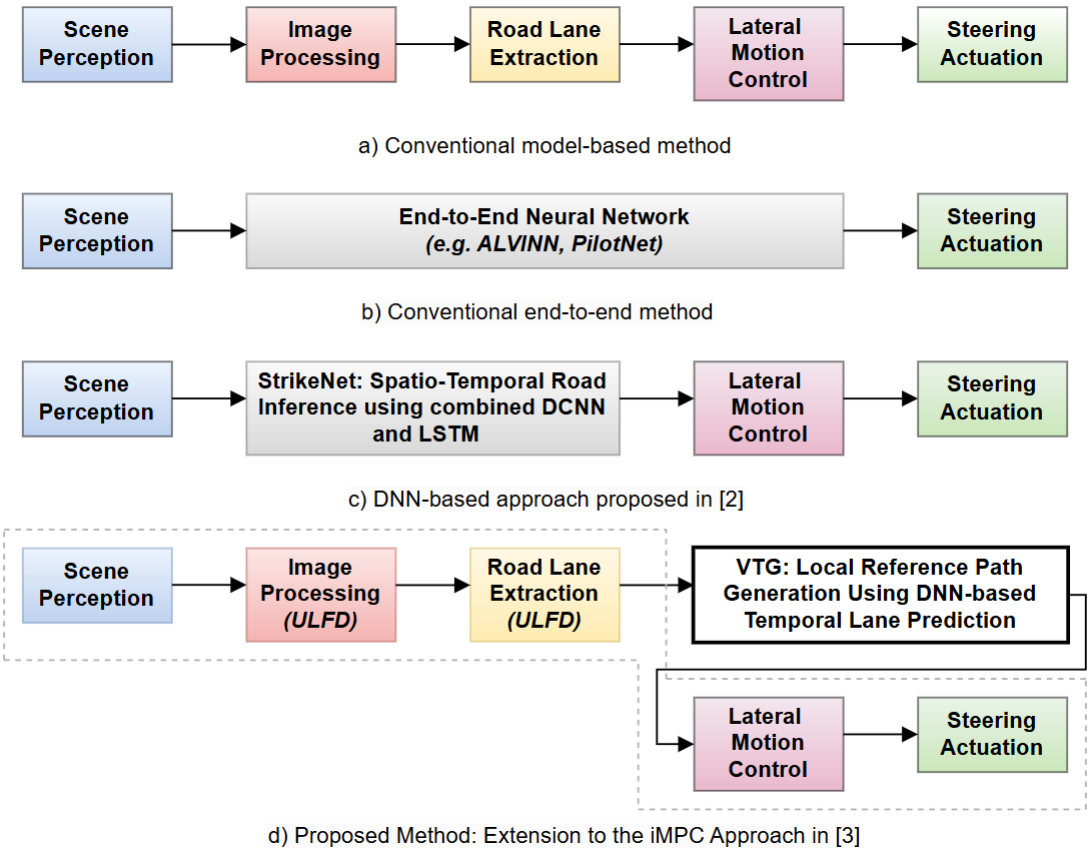


Figure 1.3: Overview of Current Practices and Proposed Approach. [3] [1]

Many Lane Keeping Systems (LKSs) rely on machine vision-based image processing to detect lanes in driving environments [3]. Traditional methods utilize hue contrast between lane markers and paved road surfaces (such as asphalt or concrete) to extract lane boundaries. Irrelevant pixels are then filtered out to emphasize lane features [3]. The effectiveness of such vision-based systems heavily depends on the sensor's ability to perceive the environment accurately. However, issues arise when road surfaces are dirty or lane markers are faded, making it difficult for the system to function correctly. In situations where lane markers are absent, LKS performance often deteriorates, potentially

leading to accidents. To address these challenges, Son et al. introduced predictive virtual lanes to mitigate short-term camera failures [13]. Additional approaches, such as GPS-based localization [14] and robust estimation of road coefficients [15], have been proposed for high-speed driving scenarios. Kang et al. explored fault-tolerant strategies for handling temporary vision failures [16]. While these techniques have proven useful for short durations, they are less effective for extended failures lasting beyond a few seconds. GPS-based methods are also limited in areas where signal accuracy is compromised, such as urban canyons or dense forests. Nonetheless, even when a precise tracking reference is unavailable, robust lane inference remains essential for ensuring safe LKS operations.

From the late 1980s to the mid-1990s, Pomerleau pioneered knowledge-based path-following systems, transitioning from one-to-one training to the empirical approaches seen in modern End-to-End (E2E) learning [17]. These systems modeled nonlinear relationships between road curvature and images without the need to extract specific features from the driving scene. Pomerleau also proposed an analytical approach involving "scanline" techniques based on curvature hypotheses [18]. Despite achieving high performance in scenarios without clear lane markings, these feed-forward systems faced challenges in estimating tracking references, which limited their stability and reliability.

Advancements in computing power during the 2010s accelerated the development of deep neural networks (DNNs) in autonomous driving. Many Deep Convolutional Neural Network (DCNN)-based E2E methods trained on virtual or real-world driving data emerged as prominent solutions [19], [20], [21], [22]. Subsequent models incorporating Long Short-Term Memory (LSTM) networks demonstrated further improvements by reducing steering jitter [23], [24]. Researchers have also employed Recurrent Neural Networks (RNNs) for advanced tracking control in autonomous aerial vehicles, combining model-based and neural network approaches for improved performance [25], [26]. DCNNs have shown promise in lane estimation using surround-view cameras [27], and encoder-decoder architectures have been applied for lane prediction in diverse driving scenarios [28], [29].

However, practical limitations persist. Steering control varies significantly based on vehicle dynamics, specifications, and individual driving styles, making it difficult to generalize E2E models [3]. Complex driving conditions often produce corner cases that are challenging to address using driver-dependent ground truths [3]. Additionally, E2E methods face difficulties in defining fault detection criteria, and their outputs are highly sensitive to small perturbations in pixel data [3]. These limitations make E2E outputs less robust and unreliable for direct vehicle control, emphasizing the need for alternative approaches to ensure safe and consistent operation.

1.2 Motivation

The motivation for this thesis stems from the growing reliance on autonomous vehicle systems to enhance safety, efficiency, and robustness in real-world environments. Traditional Model Predictive Control (MPC) approaches depend heavily on accurate state estimation in 3D coordinates, necessitating additional modules for perception and data transformation. While effective, these methods introduce significant complexity and computational overhead.

Building on the foundation laid in [1], which proposed an innovative approach to Image-based MPC (iMPC) for autonomous lane-following and lane-changing, this thesis seeks to

enhance the efficiency and precision of the trajectory generation approach utilized in [1]. A similar vision-based MPC approach for trajectory generation is explored in [12], but with a different method for generating the reference path. In [12], the lane centerline is derived from detected lane coordinates using an averaging method, which then serves as the reference path to guide the vehicle within lane boundaries. This traditional approach has been widely employed in previous research studies. However, this study proposes a novel method of computing the reference path by predicting lane coordinates from past sequential frames using a Deep Neural Network (DNN).

As shown in the figure 1.3, compared to the conventional model-based method, the proposed approach requires one additional step before lateral motion control. This approach of using prediction of future lane co-ordinates based on spatiotemporal road lane data to generate a reference path is also followed in [3] and [24]. The method from [24] uses an end-to-end neural network approach. While in [3], the reference path is generated through a process termed Road Lane Reconstruction (RLR) using a deep neural network model named StrikeNet. StrikeNet combines a Deep CNN and LSTM to infer road lane model coefficients directly from raw sequential images. These coefficients are then used to compute the necessary state variables and control inputs for the Lane Keeping System (LKS). Along with time sequential images, the method in [3] also considers multi-modal data as an input to enhance robustness and accuracy in lane reconstruction. It integrates visual features extracted from raw image sequences using CNNs with temporal dependencies captured by LSTM networks. Additionally, it incorporates vehicle dynamics, such as velocity and yaw rate, to provide further context about the vehicle's state and its interaction with the road. While this design enables the system to perform robustly in challenging conditions, such as missing or faint lane markers, it introduces significant computational complexity.

Unlike [3], we required an approach that leverages pre-extracted road lane coordinates from Ultra-Fast Structure-Aware Lane Detection (ULFD), offering a computationally less intensive solution. Instead of relying on complex image-based feature extraction, the focus shifts to generating the reference path directly from these predicted lane coordinates using temporal regression, bypassing the conventional method of averaging left and right lane curves to compute the center-line. We required a method that is highly efficient and adaptable that relies solely on past road lane x-y coordinates, designed specifically for systems that cannot accommodate the added computational overhead of processing raw visual inputs or multi-modal data. The solution needed to feature a modular design that integrates seamlessly with model-based systems, meeting the demands for efficiency, reliability, and lightweight implementation.

1.3 Problem Definition

This thesis aims to design and train a lightweight Deep Neural Network (DNN) capable of predicting future lane positions based on previously observed road lane curves. These lane curves are derived from sequential driving scenes with the assistance of Ultra-Fast Structure-aware Lane Detection (ULFD) [4]. The predicted future lane coordinates are then used to generate virtual trajectories, which serve as the basis for computing a local reference path. The computed reference path acts as an essential input to an existing image-based MPC framework (iMPC) [1], which is responsible for the vehicle's lateral motion control. By accurately predicting lane positions and computing reliable reference

paths, the DNN enables the MPC system to guide the vehicle along its planned trajectory effectively and safely. This integration ensures smooth and precise path-following behavior, contributing to the overall robustness and efficiency of autonomous vehicle systems.

1.3.1 Mathematical Formulation

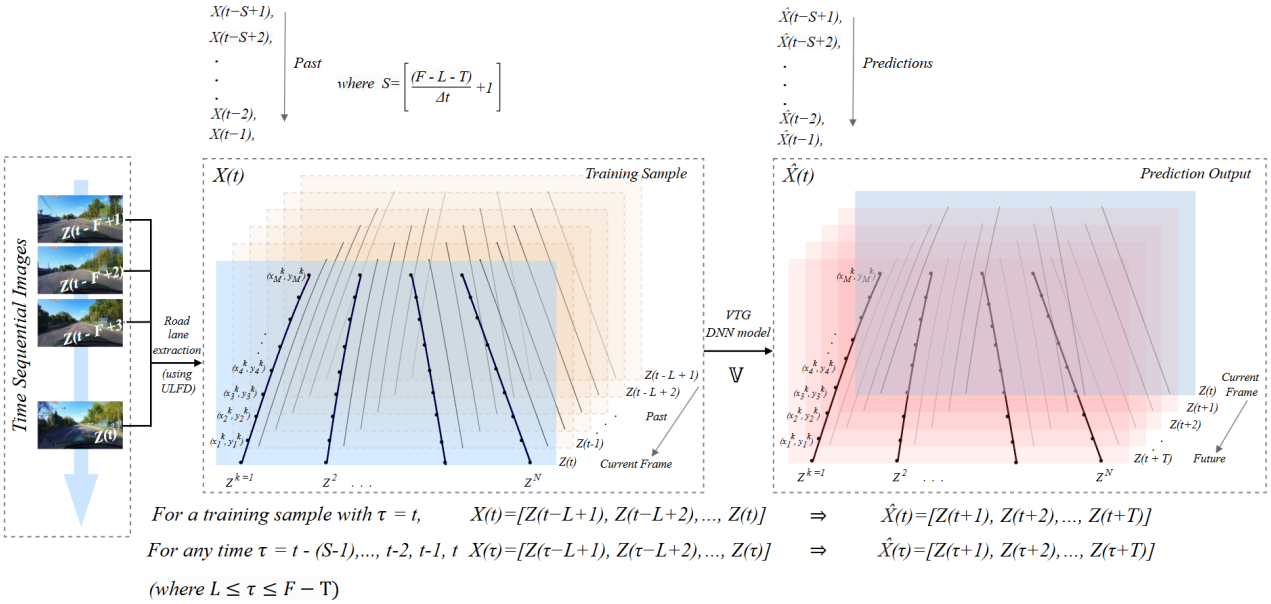


Figure 1.4: Mathematical Formulation.

Let Z represent an individual frame having multiple road lanes, with N being the total number of lanes present in that frame:

$$Z = \{Z^1, Z^2, \ldots, Z^N\}$$

Each lane Z^k in Z is defined by a set of M coordinates: "

$$Z^k = \left\{\left(x_1^k, y_1^k\right), \left(x_2^k, y_2^k\right), \ldots, \left(x_M^k, y_M^k\right)\right\}$$

" with $M, N \in \mathbb{N}$.

Let $X(t)$ denote a sequence of past lane detections over L sequential frames up to time t:

$$X(t) = (Z(t-L+1), Z(t-L+2), \ldots, Z(t))$$

where $L \in \mathbb{N}$ is the sequence length.

The goal is to predict a sequence of future lane coordinates $\hat{X}(t)$ over a forecast horizon of T frames: "

$$\hat{X}(t) = (Z(t+1), Z(t+2), \ldots, Z(t+T))$$

" where $T \in \mathbb{N}$ represents the number of frames to be predicted.

For example, consider a scenario with $N = 4$ lanes per frame, each defined by $M = 16$ coordinates, a sequence length of $L = 9$ frames, and a forecast horizon of $T = 4$ frames:

- The set of lanes in each frame is:

$$Z = \{Z^1, Z^2, Z^3, Z^4\}$$

- Each lane Z^k consists of:

$$Z^k = \left\{\left(x_1^k, y_1^k\right), \left(x_2^k, y_2^k\right), \ldots, \left(x_{16}^k, y_{16}^k\right)\right\}$$

- The input sequence of past lane detections is: "

$$X(t) = (Z(t-8), Z(t-7), \ldots, Z(t))$$

"

- The predicted sequence of future lane coordinates is:

$$\hat{X}(t) = (Z(t+1), Z(t+2), Z(t+3), Z(t+4))$$

The primary objective is to develop a DNN V that maps the input sequence $X(t)$ to the predicted future sequence $\hat{X}(t)$:

$$V : X \to Y, \quad X(t) \mapsto V\left(X(t)\right) = \hat{X}(t)$$

Now, let F be the total number of sequential frames available. The total number of samples S can be calculated by:

$$S = \frac{F - L - T}{\Delta t} + 1$$

where Δt is the stride or step size between sequences. Assuming a stride of 1 between samples to utilize all possible sequences, then S would be given by:

$$S = F - L - T + 1$$

For any time τ where $L \leq \tau \leq F - T$, the overall input sequence for all training samples could be represented as:

$$X(\tau) = [Z(\tau - L + 1), Z(\tau - L + 2), \ldots, Z(\tau)].$$

Correspondingly, the predicted future sequence will then be represented by:

$$\hat{X}(\tau) = [Z(\tau + 1), Z(\tau + 2), \ldots, Z(\tau + T)].$$

1.3.2 Thesis Contribution

The key contributions of this thesis are outlined as follows:

- Proposed a DNN-based approach for generating virtual trajectories to serve as input for lane-keeping and path-following systems.
- Introduced an improved method for computing local reference paths by predicting future lane positions based on temporal sequences of past lane coordinates. The approach ensures robustness and accuracy, particularly in dynamic road conditions, including missing lane markings, outliers, and inaccuracies during lane detection.
- Explored, compared, and tested various state-of-the-art DNN architectures—such as RNNs, LSTMs, GRUs, Transformers, and Temporal Convolutional Networks (TCNs)—specifically suited for multivariate time series problems. The Seq2Seq Transformer model was identified as the most effective, offering high accuracy and robustness.
- Integrated the computed reference paths into an existing image-based Model Predictive Control (MPC) framework [1], enabling precise and smooth lateral motion control. This solution enhances the reliability and efficiency of autonomous vehicle navigation systems while maintaining modularity for seamless integration with existing model-based systems.
- By utilizing pre-extracted road lane coordinates from Ultra-Fast Structure-Aware Lane Detection (ULFD) [4], the approach avoids computationally intensive image-based feature extraction methods and use of multi-modal features, making it efficient and suitable for resource-constrained systems.

1.4 Thesis Outline

The detailed outline of the remaining contents is as follows: Section II provides an overview of foundational concepts, including methods from the existing framework such as ULFD and the iMPC system, traditional and deep learning-based approaches for reference path generation, and an introduction to the CARLA simulator. Section III describes the implementation process, covering data preparation, curve fitting, feature engineering, and data normalization for training. Section IV outlines the experimental framework, detailing the setup, training methodology, loss metrics, and deployment of the trained model in CARLA for real-time inference. Section V evaluates the model using metrics such as Fréchet Distance and curvature loss, presenting both quantitative and qualitative analyses of the predicted reference paths. Section VI highlights the results, focusing on model performance in offline and simulation environments while validating the robustness of the generated trajectories. Finally, Section VII concludes the thesis by summarizing key findings, discussing limitations, and proposing directions for future research.

2. Background

This chapter provides an overview of the core concepts and methodologies that form the foundation of this thesis. It explores the application of deep learning to reference path generation, the role of simulation platforms like CARLA for testing and evaluation, and the framing of lane prediction as a deep learning problem. Additionally, it delves into the fundamentals of time series and state-of-the-art deep learning neural network techniques specifically designed for processing sequential data. An overview of the datasets used for training is also presented. Together, these elements establish the groundwork for the design, implementation, and evaluation of the proposed trajectory generation framework.

2.1 Existing Framework

This section outlines the existing framework that forms the foundation of this thesis. It leverages Ultra-Fast Structure Aware Lane Detection [4], a method designed to efficiently detect lanes and extract lane coordinates under challenging conditions. Additionally, the image-based MPC (iMPC) approach from [1], introduced in the previous chapter, is explored in detail. The section also provides an introduction to the Frenet coordinate system and examines traditional methods commonly employed for reference path generation, establishing a basis for the advancements proposed in this work.

2.1.1 Ultra-fast Structure Aware Lane-Detetction (ULFD)

Modern lane detection methods predominantly utilize pixel-wise segmentation, which encounters challenges in complex scenarios such as severe occlusion and extreme lighting, and struggles with computational efficiency. These methods require processing every pixel in an image to determine whether it belongs to a lane, leading to high computational costs and slower processing times that are unsuitable for real-time applications. Furthermore, in conditions where visual cues are minimal or absent—referred to as "no-visual-clue" scenarios—pixel-wise segmentation often fails to accurately detect lanes due to its reliance on local features.

In [4], the authors propose Ultra Fast Structure-aware Deep Lane Detection, a novel method inspired by human perception, focusing on contextual and global information

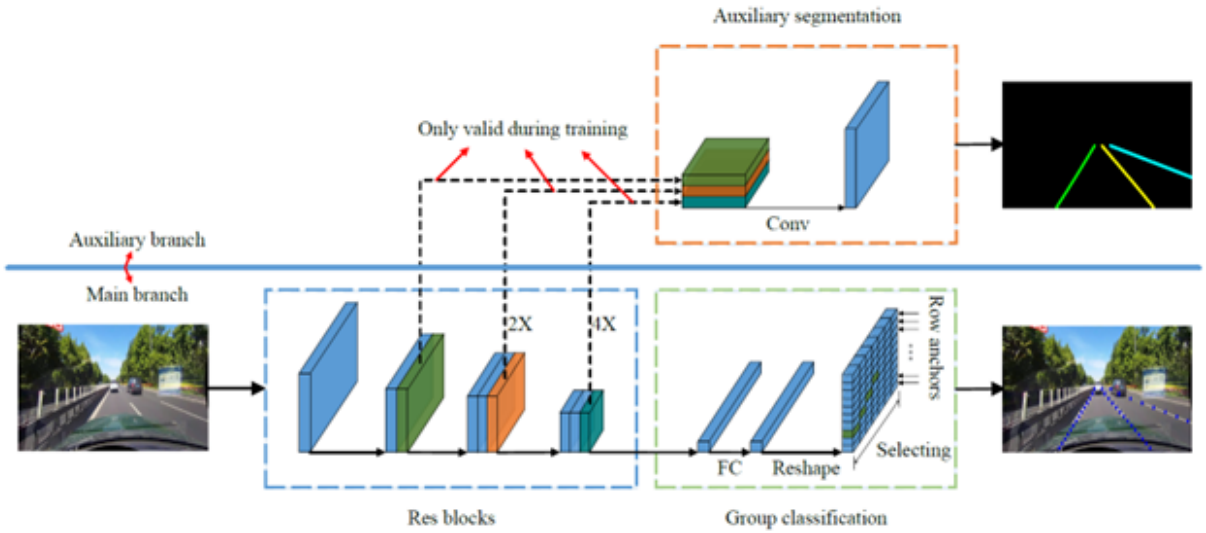


Figure 2.1: Ultra Fast Structure-Aware Lane Detection. [4]

for lane detection, particularly under challenging conditions. This method redefines lane detection framing it as a row-based selection task leveraging global features, hence minimizing computational overhead but with compromising the accuracy. It utilizes a larger receptive field to capture contextual information from the entire image, allowing it to infer lane positions even in the absence of direct visual cues, as demonstrated in the main branch block of Fig. 3.

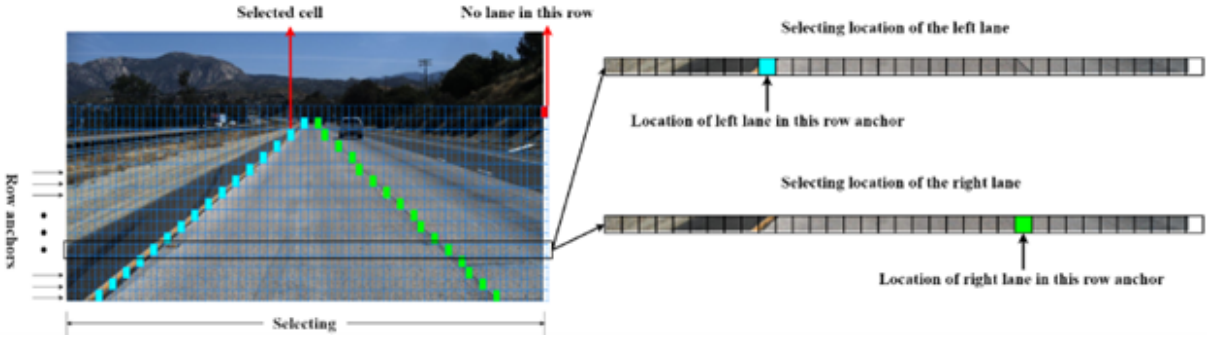


Figure 2.2: Row-wise selection in ULFD method.

The key innovation is the transformation of lane detection into a row-based selection problem rather than pixel-wise segmentation. The image is divided into a set of predefined horizontal lines called row anchors. For each row anchor, the method selects the most probable lane positions by evaluating global feature maps. This reduces the computational complexity because the model only needs to process information at specific rows instead of every pixel. By focusing on predefined row anchors, the computational load is significantly reduced compared to traditional segmentation methods that require dense computations across the entire image. Figure 2.2 illustrates the row-based selection process for lane detection. Row anchors represent predefined horizontal locations, and lane positions are determined at these anchors by selecting specific grid cells. Figure 2.3 compares the proposed row-based selection formulation with traditional pixel-wise segmentation. At left, the proposed method selects lane locations at predefined rows (row anchors) using

grid cells, focusing only on critical locations and reducing computation. In contrast, the right side shows the traditional segmentation method, which classifies every pixel across the entire image, leading to higher computational costs. This approach effectively addresses "no-visual-clue" conditions by leveraging global features with a large receptive field, allowing it to infer lanes even under occlusion or extreme lighting.

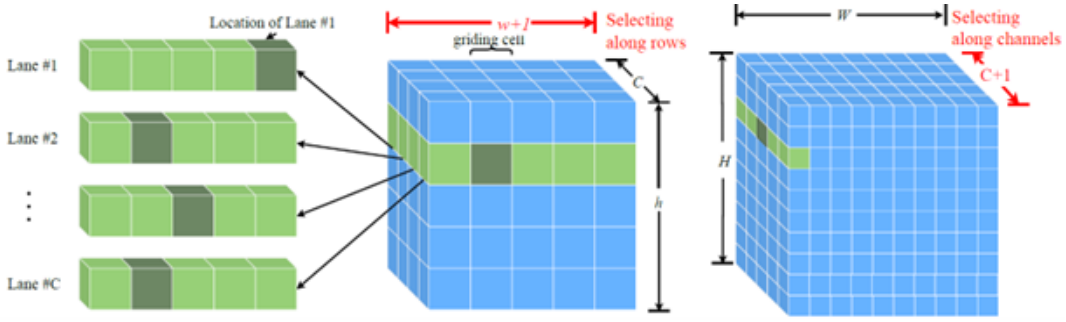


Figure 2.3: ULFD formulation vs Conventional Segmentation. [4]

2.1.2 Lane Keeping in Autonomous Driving Using Model Predictive Control

This section describes the existing work performed in [1] over the kinematic MPC controller framework, which serves as the foundation for this thesis. The primary goal of this thesis is to integrate the virtual trajectory generated herein into the framework proposed in [1], thereby enhancing its path-following capabilities. The role of the MPC is to generate commands for controlling the vehicle's lateral and longitudinal dynamics so that the vehicle closely follows the given reference path at a desired speed. More specifically, the MPC generates a sequence of steering angle and acceleration values to track the reference while satisfying the physical constraints of the actuators.

The MPC controller in [1] utilizes a kinematic model introduced in [5] for the planar motion of a ground vehicle. The state of the vehicle and the reference path are represented in camera coordinates (O_c), while the error dynamics are expressed in the Frenet frame (O_s) of the reference path. The yaw error (ψ_e) is defined as the angular difference between the vehicle's heading (ψ_p) and the orientation of the reference path (ψ_s), given by:

$$\psi_e = \psi_p - \psi_s. \tag{2.1}$$

The dynamics of the yaw error are formulated as:

$$\dot{\psi}_e = \alpha - \frac{v_t \cos(\psi_e)\kappa}{1 - y_e\kappa}, \tag{2.2}$$

where α is the yaw rate, v_t is the velocity at the center of gravity (CoG), y_e is the lateral error, and $\kappa = \frac{1}{\rho}$ denotes the curvature of the reference path with ρ as the radius of curvature.

The lateral error, representing the perpendicular distance of the vehicle from the reference path, is described by the equation:

$$\dot{y}_e = v_t \sin(\psi_e). \tag{2.3}$$

These equations are part of the vehicle kinematic model (VKM), which provides a comprehensive description of the vehicle's motion. The full dynamics include:

$$\dot{v}_t = a_t, \quad \dot{\psi}_p = \alpha, \quad \dot{s} = \frac{v_t \cos(\psi_e)}{1 - y_e \kappa}, \tag{2.4}$$

$$\dot{x} = v_t \cos(\psi_p), \quad \dot{y} = v_t \sin(\psi_p), \tag{2.5}$$

where a_t is the longitudinal acceleration, s is the arc length along the reference path, and $[x, y]$ are the Cartesian coordinates of the vehicle.

This model is compactly expressed as:

$$"\dot{X} = F(X, U)", \tag{2.6}$$

here $X = [v_t, \psi_p, s, \psi_e, y_e, x, y]^T$ represents the state vector, and $U = [a_t, \alpha]^T$ is the control vector.

This thesis contributes to this framework by integrating the generated virtual trajectory as the reference input, enabling dynamic adaptation to real-time lane-detection inputs and improving the controller's performance.

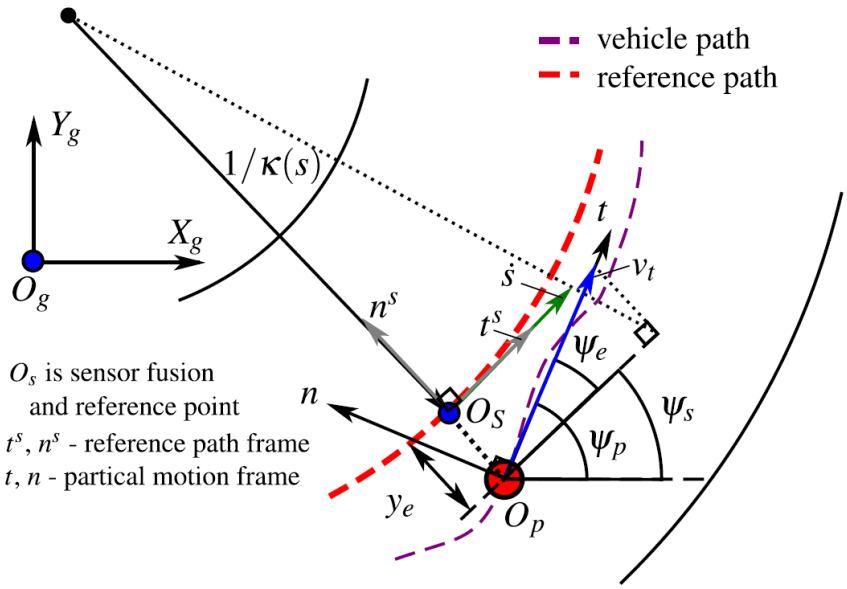


Figure 2.4: Kinematic MPC Model: Curvilinear Motion Description. [5]

2.2 Traditional approach for Reference Path generation

As previously discussed, in [12], the reference path is derived by averaging detected lane coordinates to estimate the lane center line—an approach widely adopted in prior research. This section delves deeper into such traditional techniques, providing a detailed analysis of their results. These methods can be categorized into two main approaches, with real-time lane detection-based methods computing the reference path directly from detected lane coordinates in real-time, while spatiotemporal frame-based methods utilize features

extracted from sequential driving scenes to generate the reference path. Before delving into specific methods, it is important to first examine the rationale behind the use of real-time lane detection approaches.

In [12], it is assumed that the lane marking position does not change significantly between consecutive frames, particularly when the camera's frame rate is high. In other words, the locations of lane line pixels in the current frame are assumed to be approximately the same as those in the previous frame, with only minor deviations. The error between the detections in the current and previous frames is calculated, and if the deviation exceeds an acceptable margin, the new detection is ignored. Otherwise, the detected line is averaged with previous detections to produce the final result. The assumption of no significant change in lane positions during consecutive frames was derived from [30], which proposed a lane detection method based on pairs of consecutive frames rather than a sequence of multiple spatiotemporal frames.

Another study, [31], introduced a spatiotemporal sequence-based approach for lane detection, demonstrating slightly better performance than [30] in tunnel sequences. The results from [31] indicate that the assumption made in [12] holds true primarily for consecutive frames but becomes less reliable when there is an increase in the time interval among frames because of thecomputational delays or when the difference between frames is large at high vehicle speeds. To elaborate further, the approach presented in [30] identifies lanes by evaluating inter-frame similarity to ensure location consistency of detected lanes and the estimated vanishing point across consecutive frames. While this method achieves an overall detection accuracy exceeding 90%, its performance drops significantly when Gaussian noise with a SD of 5 is applied to the vanishing point. [30]. The second paper [31] employs a spatiotemporal image-based approach, leveraging temporal consistency across scan-lines to detect lanes. It is more robust to issues like missing lanes and lens flares, outperforming frame-based methods. However, it struggles with abrupt changes, such as pitch angle variations due to speed bumps, though it recovers effectively once normal conditions resume. While the vanishing point-based method excels in handling noisy environments with enhanced geometric constraints, the spatiotemporal method demonstrates better resilience to occlusion and missing data by integrating temporal features, showing promise in computational efficiency and robustness.

Similar to [12], many state-of-the-art methods [1][2][32][6][33] derive the center-line from the detected lane markings of either the current or previous frame, subsequently utilizing it for lane-keeping control. In [34], the way-points of the center-line and the steering angle are used to define the steering vector. The steering angle is computed based on point 1 x_1, y_1, representing the closest point to the vehicle, and point 2 x_2, y_2, which is the midpoint of the center-line. This is given by:

$$\theta = \arcsin\left(\frac{x_1 - x_2}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}\right) \tag{2.7}$$

The steering vector is considered as the short-term trajectory or reference path for lane-keeping. Figure shows the general trajectory algorithm followed in [6] and [33]. In this, the spline fitting using arc-length parametrization is used to represent the road lanes along with the center line. Then for the localization of the vehicle on the center line the vehicle position is mapped from the Cartesian to the Frenet coordinate system. This will be

explained in next section. This study follows the similar approach as this, however, the candidate paths are generated using DNN based approach.

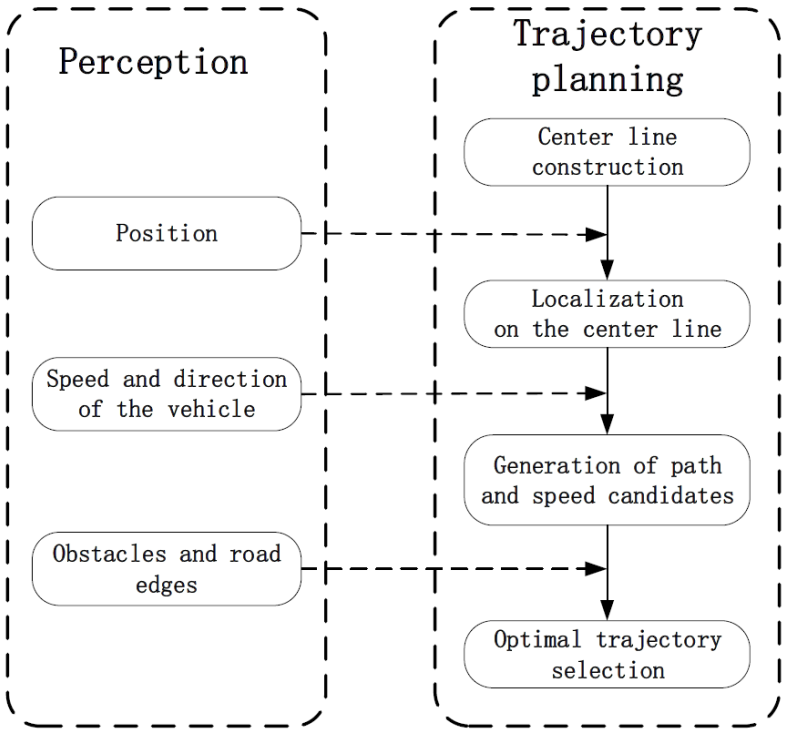


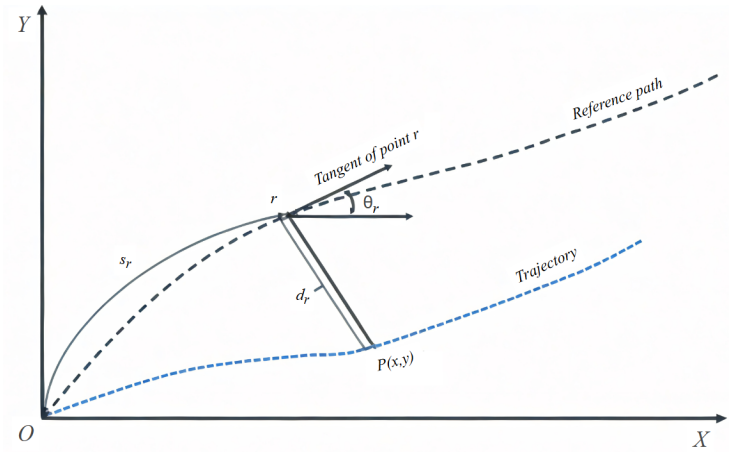
Figure 2.5: Traditional Trajectory Planning Algorithm. [6]

Figure 2.5 illustrates the general trajectory algorithm outlined in [6] and [33]. This approach employs spline fitting with arc-length parametrization to represent the road lanes, including the center-line. The vehicle's position is then localized on this center-line by mapping its Cartesian coordinates into the Frenet coordinate system [33]. In next section, this co-ordinate system will be explained in more details.

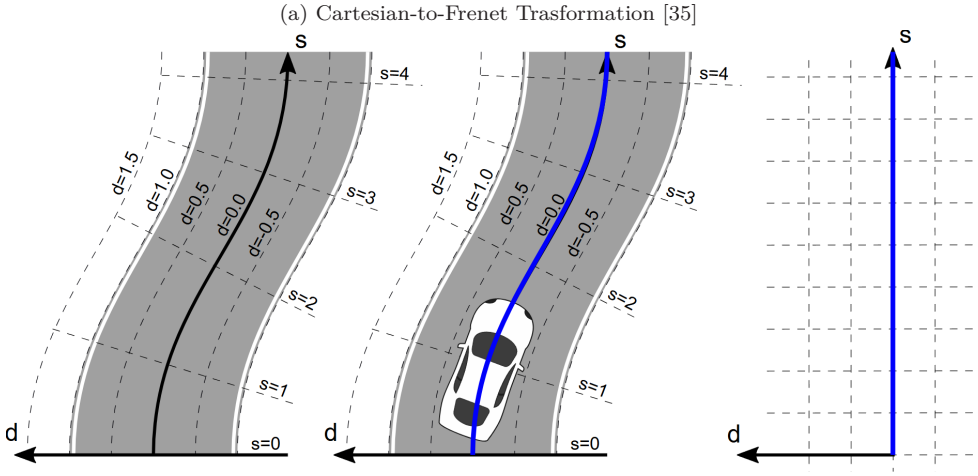
While this thesis follows a similar methodology, it introduces a novel aspect by generating candidate paths using a DNN-based approach, diverging from traditional methods. This aligns with the broader objective of this study, as discussed earlier, to adopt a vision-based MPC approach for lane-keeping control.

2.2.1 Frenet Co-ordinate System

Most of the methods discussed above, along with our approach, propose the use of the Frenet coordinate system for localization of the vehicle. This co-ordinate system offers a more intuitive way to represent the relationship between a vehicle's position and the road lane compared to traditional Cartesian coordinates [35].



(a) Cartesian-to-Frenet Trasformation [35]



(b) Frenet Path with s and d parameters [36]

Figure 2.6: Kinematic MPC Model

As shown in the figure 2.6b, It uses two variables, s and d, to describe the vehicle's location. The s coordinate, also referred to as longitudinal displacement, indicates the distance traveled along the road, while the d coordinate, known as lateral displacement, represents the side-to-side position of a vehicle relative to the reference path [41]. The figure 2.6a illustrates the relationship between the Frenet and Cartesian coordinate systems [35]. This transformation is mathematically defined as:

$$\begin{cases} s_p = s_r \\ d_p = \pm\sqrt{(x_p - x_r)^2 + (y_p - y_r)^2} \end{cases} \qquad \begin{cases} x_p = x_r - d\sin(\theta_r) \\ y_p = y_r + d\cos(\theta_r) \end{cases} \tag{2.8}$$

Here, r and p represent the reference point on reference path and the current vehicle position, respectively, while θ_r denotes the angle between the tangent of the reference line and the x-axis.

By leveraging the s-axis, Frenet coordinates simplify the mathematical representation of a reference path, providing an efficient way to describe its progression along the road [36]. This path guides vehicles along a smooth, curvature-continuous route. However, to prevent collisions, planners must account for static and dynamic objects, which are typically outside the scope of the reference path [36].

2.3 Deep Learning based approach for Reference Path Generation

Deep neural network-based methods offer a promising solution for addressing the complex dynamics and parametric uncertainties in vehicle lateral motion control [23]. Recent research in deep learning for Lane Keeping Systems (LKS) has largely focused on end-to-end learning approaches, including convolutional neural networks (CNN) [22], reinforcement learning [37], and imitation learning [38]. However, these methods often overlook temporal dependencies in input data, leading to steering angle calculations based on limited information. Furthermore, CNN-based end-to-end models are highly sensitive to image shifts and the motion of surrounding objects [39], making them unsuitable for direct application in lane keeping assist.

DNN models, particularly those tailored for time-series forecasting, excel in capturing temporal dependencies and patterns [40]. Additionally, modern DNN architectures, comprising "Long Short-Term Memory (LSTM), Recurrent Neural Networks (RNN), Gated Recurrent Units (GRU), and Transformers", offer the ability to process sequential data efficiently and model complex multi-variate relationships across time [40]. Unlike traditional methods that focus on instantaneous lane detection or basic geometric assumptions, integrating historical and spatial information enables more robust and accurate predictions. By training on large datasets with diverse road and environmental conditions, deep learning models demonstrate resilience to noisy inputs and missing lane markings. Techniques such as attention mechanisms in Transformer models enhance the model's ability to emphasize on the most pertinent aspects of the data, improving robustness even in challenging scenarios like occlusions or lane ambiguities [40].

2.4 CARLA: Car Learning to Act

This research utilizes "CARLA (Car Learning to Act)", A publicly available simulator designed for urban driving scenarios, to develop, train, and evaluate autonomous driving systems. CARLA creates a dynamic virtual environment and provides a straightforward interface for interaction between the simulated world and an autonomous agent [41]. The platform operates on a server and client architecture, with the server handling the simulation and scene rendering, while the client, implemented in Python, manages interactions via sockets. The client sends control commands, such as steering, acceleration, and braking, as well as meta-commands, and receives sensor data in response [41].

CARLA offers extensive flexibility in configuring an agent's sensor suite. Users can mount various sensors, including RGB cams and pseudo sensors for depth as well as semantic segmentation [41]. The client specifies the number, type, and positioning of cams, along with detailed parameters like 3D location, orientation relative to the coordinate system of vehicle, FOV, and depth of field. This flexibility makes CARLA a versatile tool for autonomous driving research [41].

2.5 Time Series as Deep Learning problem

Before discussing state-of-the-art model architectures, the terminology of time series is introduced. Time series analysis organizes data points chronologically, with each point

corresponding to a specific time [40]. Variations in the data arise from factors such as trends, stationarity, periodicity, seasonality, volatility, and noise. In this context, the time series is framed as a regression problem and can be expressed as:

$$\hat{y}_{i,t+1} = f(y_{i,t-k:t}, x_{i,t-k:t}, s_i) \tag{2.9}$$

Here, the predicted value for the next time step is represented as $\hat{y}i, t+1$. The series "$yi, t-k:t = y_{i,t-k}, \ldots, y_{i,t}$" refers to the observed target elements, while "$x_{i,t-k:t} = x_{i,t-k}, \ldots, x_{i,t}$" represents exogenous inputs that could affect the target. Both are considered within a back-casting window covering the last k time steps.

2.5.1 Uni-variate vs Multi-variate Time Series Problem

To determine the appropriate data preparation strategy, the type of time series is first identified. Common types include Univariate Input-Univariate Output, Multivariate Input-Univariate Output, and Multivariate Input-Multivariate Output [42]. The first approach is the simplest and most traditional forecasting setup, involving a single input time series and a single output time series [42]. This type of problem is often addressed using classical statistical methods such as SARIMA or HWES [42]. Initially, this thesis experimented with this approach by training a model on a single time series using racetrack data and applying it to the lane sequence of a single lane. The maps with multiple lane segments were structured into a single time series with sliding window approach. However, this approach failed to capture the inter-dependencies between multiple lanes from individual frames during turns, which are critical for understanding their correlation.

The next type, Multivariate Input-Univariate Output, is highly suited for deep learning models [42]. Deep neural networks excel at identifying complex predictive patterns in multivariate time series, which classical methods often overlook [42]. However, the problem addressed in this thesis falls under the category of Multivariate Input-Multivariate Output, the most challenging type. This approach involves inter-dependencies among multiple variables, making it highly complex and time-intensive, requiring efficient data handling and advanced modeling techniques [42].

2.5.2 Data Generating Process (DGP)

Time series data comprises observations recorded sequentially over time, with each series generated by an underlying mechanism referred to in statistical literature as the Data Generating Process (DGP) [7]. These processes can be broadly categorized as deterministic or stochastic. Deterministic processes exhibit predictable behavior, governed by precise mathematical formulations that result in consistent trends, such as uniform increments or decrements over time [7].

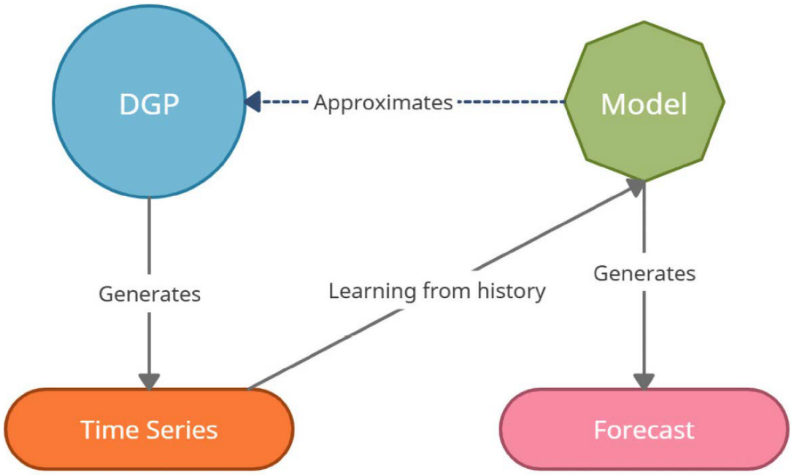


Figure 2.7: Data Generating Process (DGP). [7]

In contrast, stochastic processes, which account for most real-world time series, including those utilized in this study, describe systems that evolve over time with inherent randomness, yet exhibit discernible probabilistic patterns [7]. For instance, stochastic processes can model phenomena like weather changes, where variations occur randomly but follow underlying trends and probabilities. This study follows a general DGP structure, as depicted in Figure 2.7. The figure depicts the relationship between the DGP, the model, and forecasting. The DGP iteratively produces sets of time series data, while the model approximates the DGP by learning patterns from historical data to produce forecasts [7].

2.5.3 Time Delay Embedding for Regression Problems

This thesis adopts the time-delay embedding method from [7] to transform time series data into a regression framework. Consider a time series with L time steps, where T represents the most recent observation, and earlier observations are indexed as $T-1, T-2, \ldots, T-L$, , as illustrated in Figure 2.8. In theory, forecasting could condition each observation on all previous observations, but this is computationally infeasible for large L. To address this, a finite memory model or Markov model is used, restricting the forecasting function to the last M observations ($M < L$), which is often referred to as the autoregressive order or the size of the memory window.

The time-delay embedding process involves using a sliding-window of length M to split the series into sections of uniform length [7]. For instance, with $M = 3$, the first subsequence consists of "$T-3, T-2, T-1$", and T is the target. Sliding the window backward by one step generates the next subsequence "$(T-4, T-3, T-2)$" and its target $(T-1)$. This process continues until the entire time series is segmented, resulting in a dataset where each feature vector (of size M) corresponds to a single target. This method aligns the dataset into a structure suitable for machine learning models.

Each feature in the transformed dataset is assigned a semantic meaning based on its lag relative to the target [7]. For example, the right-most feature column, which is 1 step behind, referred to as Lag 1. The second column represents 2 time steps behind the target (Lag 2), and so on. Generalizing this, a feature n steps behind the target is labeled as Lag n. This transformation encodes the auto-regressive structure of the time series into a

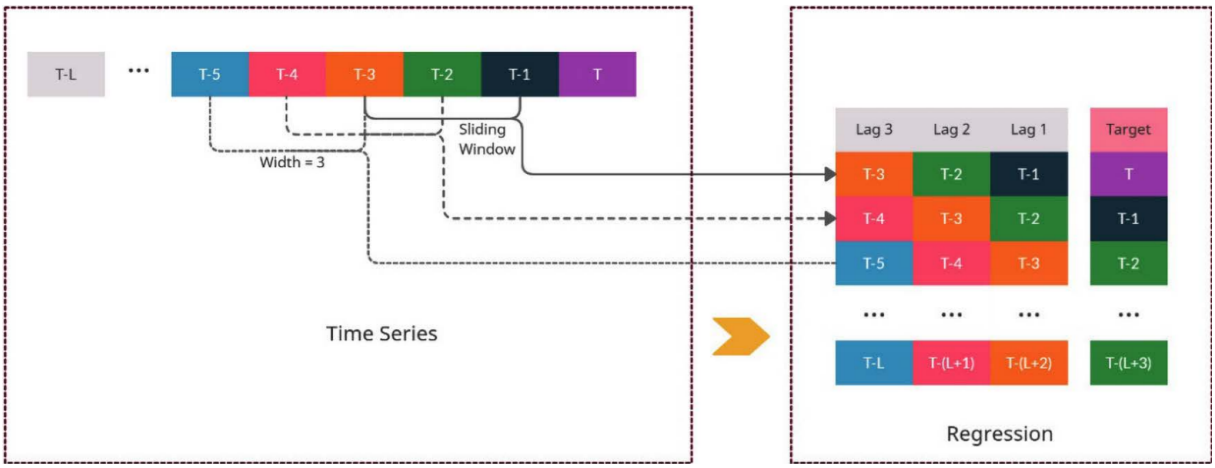


Figure 2.8: Transforming Time Series into Regression via Sliding Window. [7]

form compatible with standard regression models, enabling efficient learning of temporal dependencies [7].

The approach described above, as outlined in [7], adheres to a multivariate input–univariate output structure. However, in this thesis, the methodology is extended to a multivariate-to-multivariate framework by incorporating multiple target sequences and their associated lags. This adaptation addresses the specific demands of the problem, focusing on capturing inter-dependencies among multiple sequences. Further details of this technique, tailored to the specific challenges of this thesis, are presented in the implementation section.

2.6 Deep Learning Models for Time Series Forcasting

This section aims to explore and analyze state-of-the-art approaches for modeling time series data. It provides an overview of commonly used DNN models for time series forecasting and examines key architectural frameworks for training input data. The models examined include "LSTMs, RNNs, GRUs, Sequence-to-Sequence Transformer models, Temporal Convolutional Networks (TCNs), and Temporal Fusion Transformers (TFTs)". Each model is designed with specific objectives in mind, and their performance varies depending on the context and design goals. The performance of these models is examined in the experiments chapter using the same time series dataset and identical hardware computing environment to ensure a fair comparison.

2.6.1 Recurrent Neural Networks (RNNs)

Elman [8] introduced RNNs, a neural architecture specifically developed for processing sequential data, such as time series and NLP data. Unlike traditional feedforward networks, Recurrent neural networks incorporate recurrent connections, allowing data to propagate across time intervals. This enables RNNs to utilize the previous "hidden state (h_{t-1})" along with the recent input (x_t) to calculate the recent hidden state (h_t), capturing temporal dependencies and contextual relationships in the sequence.

The hidden state serves as a memory unit that updates at each time step to retain information from prior inputs. It is computed using the formula:

$$h_t = \tanh(W_h h_{t-1} + W_x x_t + b) \tag{2.10}$$

where "W_h and W_x referred to the weight matrices of the hidden state and input, b is the bias term, and tanh is the hyperbolic tangent activation function. The output at time step t, denoted as y_t", is calculated as:

$$y_t = W_o h_t + b_o \tag{2.11}$$

Here, W_o termed as the output weight matrix, and b_o is called the bias term. This architecture enables RNNs to effectively process variable-length sequences and model complex temporal patterns within the data [8]. The internal structure of RNN cell is shown in the figure 2.9

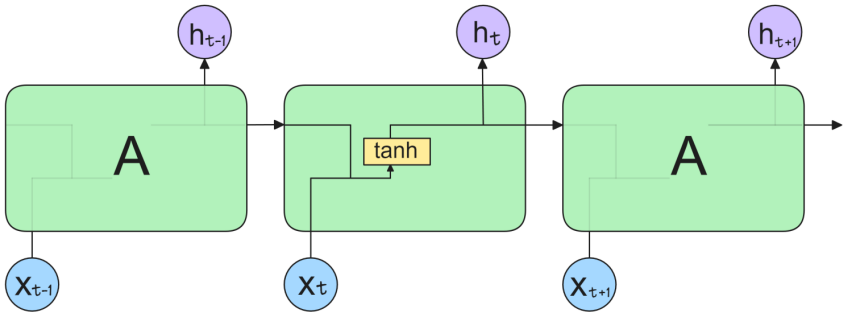


Figure 2.9: Internal Structure of RNN. [8]

2.6.2 Long Short-Term Memory Networks (LSTMs)

These networks incorporate memory cells to store long-term information alongside the hidden state, acting as a "gradient highway" to mitigate gradient loss during training [7]. These cells are managed by three primary gates:

- **Input Gate (I_t)**: Determines how much data from the recent input (x_t) and the past "hidden state (h_{t-1})" to store in the memory cell. It uses a learnable weight matrix, bias term, and sigmoid activation to produce values between 0 and 1. This is represented by the leftmost sigmoid (σ) in the figure 2.10.

- **Forget Gate (F_t)**: Regulates how much data to discard from the memory cell's prior "state (C_{t-1})" using a similar mechanism. This is represented by the middle sigmoid (σ) in the figure 2.10.

- **Output Gate (O_t)**: Decides the quanity of the data from updated "cell state (C_t)" that contributes to the current "hidden state (h_t)", forming the cell's output. This is represented by the rightmost sigmoid (σ) in the figure 2.10.

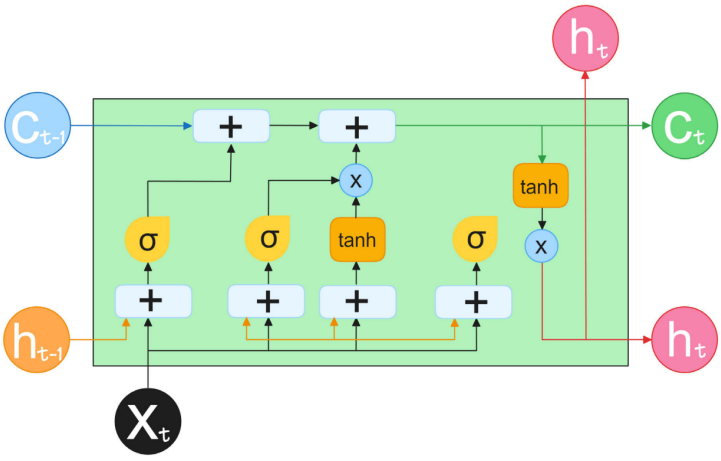


Figure 2.10: Internal Structure of LSTM. [7]

Each gate employs weight matrices for inputs and hidden states, along with biases, ensuring precise updates for effective long-term sequence modeling [7]. These innovations revolutionized sequential data processing, making LSTMs a cornerstone for deep learning tasks.

2.6.3 Gated Recurrent Units (GRUs)

The GRUs were developed as a streamlined alternative to the LSTM, designed to overcome its computational complexity [40]. By adopting a simplified architecture, the GRU reduces computational overhead, accelerates training, and promotes faster model convergence [40]. In a GRU, unlike the LSTM which uses three gates, there are only two gates [7]:

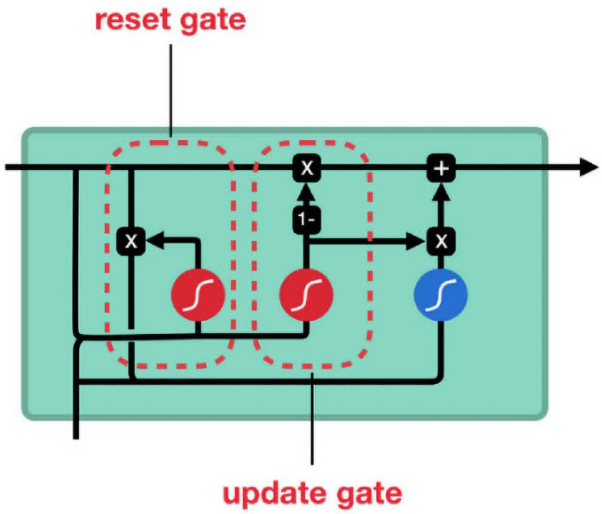


Figure 2.11: Internal Structure of GRU. [7]

- **Reset Gate** (R_t): This gate determines the extent to which the prior "hidden state (H_{t-1})" contributes to the candidate hidden state at the current time-step (t) [7].

- **Update Gate (U_t)**: This gate regulates how much of the prior "hidden state (H_{t-1})" is carried forward and also the quantity of data from the candidate hidden state ($\tilde{H}_t$) integrated into the new hidden state [7].

These simplified gates make GRU computationally lighter while still maintaining effective sequence modeling.

2.6.4 Sequence-to-Sequence Transformer Models

Encoder-decoder models have been around for a while, but transformer-based encoder-decoder architectures were first introduced by Vaswani et al. [9] in their groundbreaking paper, "Attention is All You Need." This revolutionized sequence processing by replacing recurrent networks with an architecture based solely on attention mechanisms and feed-forward networks. Unlike traditional RNN-based Seq2Seq models that process sequences sequentially, Transformers process entire sequences in parallel, enabling scalability to large datasets and faster computation [7].

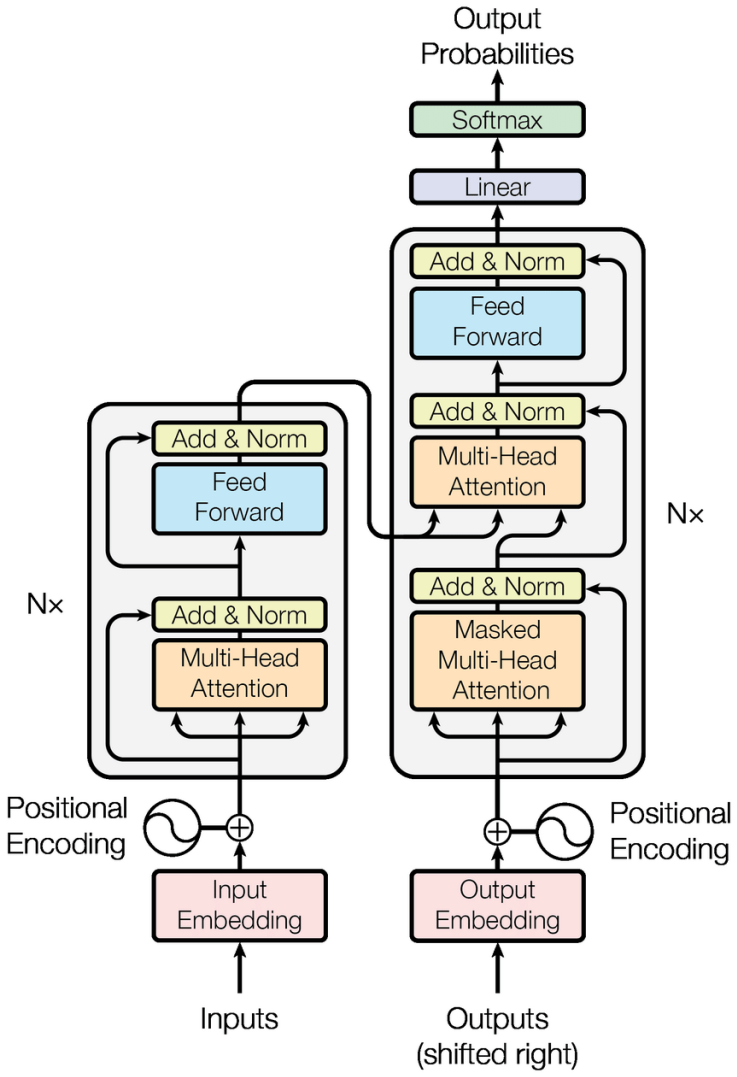


Figure 2.12: Seq-2-Seq or Vanilla Transformer. [9]

The Transformer comprises an encoder-decoder structure [9]. The encoder processes input embeddings enhanced with positional encodings, passing them through multiple blocks that contain Multi-Head Attention (MHA) layers and feed-forward networks. The decoder, while similar in design, introduces additional mechanisms to focus on relevant input while generating outputs. It combines self-attention layers and encoder-decoder attention layers, allowing it to effectively integrate information from the encoder's output [7].

To preserve sequence order during predictions, particularly in tasks like language translation or time series forecasting, the decoder employs masked self-attention [9]. This ensures the model does not "cheat" by accessing information from future timesteps, allowing predictions to rely solely on past and current data [7]. The parallelism, computational efficiency, and innovative use of attention mechanisms make Transformers highly effective, replacing traditional Seq2Seq models and becoming a cornerstone in NLP and sequence-based tasks [7].

The accompanying figure 2.12 provides a detailed view of the Transformer architecture. The encoder, shown on the left, consists of multiple layers, each containing two key components: an MHA mechanism and A feed-forward network applied independently at each position. These components are enhanced by residual connections and layer normalization, which improve gradient flow and ensure the model's stability during training. The decoder, on the right, mirrors the encoder's structure but introduces additional functionality. It incorporates masked MHA to prevent attention to future positions and an encoder-decoder attention layer that allows it to attend specifically to the encoder's output while generating predictions.

2.6.5 Temporal Fusion Transformer (TFT)

Lim et al. [30] introduced the Temporal Fusion Transformer (TFT), a model that integrates LSTM and Transformer architectures. The approach leverages LSTM's sequence modeling capabilities to preprocess input sequences, generating context-aware and time-sensitive representations at various time steps [40]. These representations are then passed to the Transformer layer, which employs Attention mechanisms to extract long-term dependencies, addressing the information loss often associated with sequence models. Unlike traditional Transformers [9], which rely on positional encoding, the TFT network uses a Sequence-to-Sequence layer, better suited for time series data as it captures local temporal patterns through recurrent connections [40].

2.6.6 Temporal Convolutional Networks (TCNs)

TCNs [56] are widely utilized for modeling time series data, leveraging causal convolutions to prevent information leakage [40]. Unlike RNNs, where predictions for later timesteps must wait for earlier ones, TCNs enable parallel processing by applying the same filter across each layer [43]. This allows TCNs to process long input sequences as a whole during both training and evaluation, rather than step-by-step as in RNNs [43].

A key advantage of TCNs is their distinct backpropagation path, which is independent of the temporal sequence direction. This design avoids the issues of exploding or vanishing gradients that commonly affect RNNs, a problem that necessitated the development of architectures like LSTMs and GRUs [43]. Furthermore, unlike LSTMs and GRUs, which

require substantial memory to store intermediate results for their multiple gates, TCNs share filters across layers, with backpropagation depending only on the network depth [43]. This efficient design makes TCNs particularly well-suited for handling long sequences.

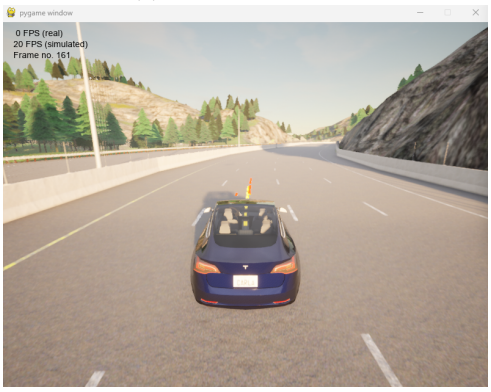
2.7 Dataset

This research leverages multiple datasets to train a robust model that captures diverse dynamic aspects for lane prediction. A key requirement for this problem is the sequential nature of the data. When considering multiple datasets, additional factors such as frame resolution or aspect ratio, frame capture rate (fps), and annotation structure used for edge cases (e.g., occlusions caused by vehicles, truncated markings at frame edges) must also be evaluated to verify the compatibility. If there are no significant differences in these attributes, multiple labeled datasets can be merged or used simultaneously. Therefore, before the data preparation phase, it is essential to assess which datasets are feasible for inclusion in the proposed approach. Figure 2.13 provides reference examples from the dataset's image data.

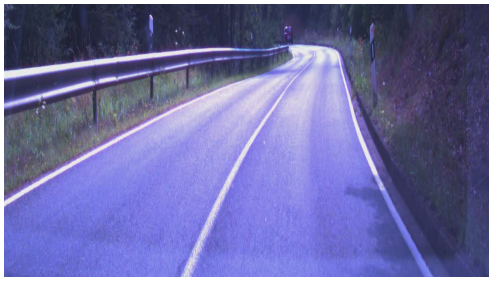
In this study, labeled lane coordinates from two prominent datasets, TuSimple [44] and CULane [45], were utilized:

(a) TUSimple example

(b) CULane example



(c) CARLA Town4 example



(d) RPTU Tripstadter example

Figure 2.13: Image Data Examples

- **TuSimple Dataset:** This dataset contains 6,408 road images captured on US highways, with a standardized resolution of 1280×720 [44]. It includes 3,626 training images, 358 validation images, and 2,782 test images [45]. The scenarios feature light traffic and clear lane markings [45]. However, occluded lane markings are not annotated, despite their relevance in real-world applications [45]. For this study,

two sequential subsets, `label_0313_1` and `label_0313_2`, were selected for their suitability for sequential analysis.

- **CULane Dataset:** The CULane dataset was created using cameras mounted on six vehicles recording videos during various driving scenarios in Beijing. Over 55 hours of videos were recorded, producing a total of 133,235 frames—over 20 times larger than the TuSimple dataset [45]. The data is split into 88,880 frames for training, 9,675 frames for validation, and 34,680 frames for testing, all at a resolution of 1640×590 [45]. This dataset covers challenging traffic conditions for lane detection. A subset, `driver23_30frame`, was chosen for its balanced representation of traffic and highway scenarios.

  The primary difference between the TuSimple and CULane datasets lies in their frame rates—20 fps and 30 fps, respectively. This variation is relatively small and contributes to creating a robust model by combining datasets with diverse conditions. In this study, the input dataset structure is designed for regression, specifically sequential lane coordinates. Consequently, factors such as weather conditions, noise, or distortions in visual features are not critical unless they directly alter the lane coordinates. Nonetheless, these factors could introduce slight variations in model performance if the nature of the lane coordinates is affected.

- **Custom Dataset from CARLA Simulation:** To ensure the model accounts for challenging scenarios, it is important to include stagnant frames from traffic scenarios and noisy frames with interrupted lane markings. The selected subset from the CULane dataset contains sufficient stagnant frames for effective learning. However, benchmark datasets generally lack significant instances of noisy or interrupted lane coordinates. To address this limitation, a custom dataset was created using CARLA simulation in Town 4. Lane coordinates were extracted using the Ultra-Fast Structure-Aware Lane Detection model, yielding approximately 4,000 frame iterations. These frames were collected and saved for further processing.

- **RPTU Universität Dataset:** The above-mentioned datasets collectively provide a substantial foundation for training. However, the objective was to capture a broader range of road dynamics, including missing lanes. To achieve this, a labeled dataset from RPTU Universität was also included. This dataset comprises approximately 1,000 images captured along Trippstadter Straße in Kaiserslautern, Germany. Unlike highway datasets, these images often feature fewer than four lanes, resulting in empty lane sequences, which are essential for addressing scenarios with missing lanes.

3. Implementation

The implementation of this research is structured into two main stages, as illustrated in Fig. 9: (I) Data Preparation and Model Training, and (II) Inference in the CARLA Simulation Environment. This approach integrates multiple datasets, advanced preprocessing methods, and a lightweight DNN model to achieve robust predictions from lane sequences and reference paths for autonomous driving applications. Each step is designed to address challenges such as missing data, noise, and dynamic road scenarios, ensuring a robust pipeline suitable for deployment in simulation and real-world environments.

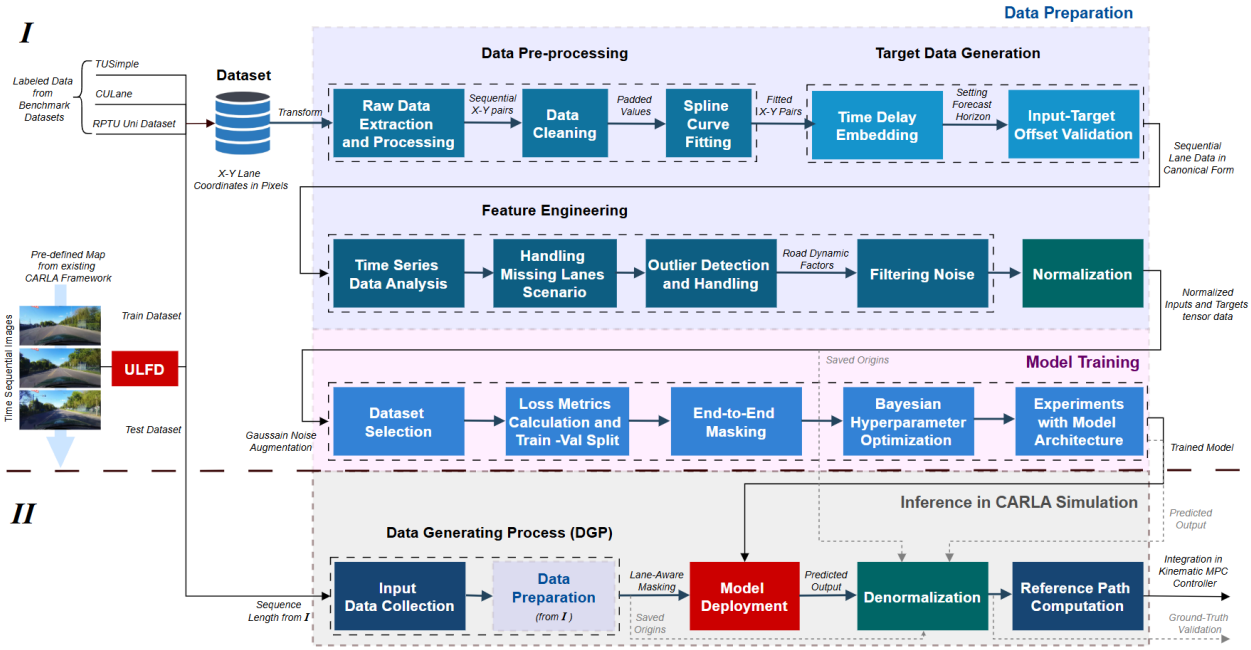


Figure 3.1: Methodology.

The first stage, Data Preparation, as shown in the upper section of Fig. 9, begins with extracting raw lane data from datasets such as TU Simple, CU Lane, and RPTU Universität Dataset. These datasets, which vary in structure and resolution, are unified

into a consistent format to ensure compatibility. The raw lane data, represented as X-Y pixel coordinates, undergoes transformation into sequential lane data. Following the initial data extraction, data cleaning and curve fitting techniques are applied to address missing values, irregularities in the number of coordinates, and lane point spacing, ultimately transforming the lane data into a uniform format suitable for training.

The next step involves target data generation using time-delay embedding, as explained in 2.5. In this step, sequential data is organized into sliding windows of a desired number of consecutive frames (sequence length). This approach captures temporal dependencies between frames, ensuring that the model learns to predict future lane positions based on historical trends. Feature Engineering, a subsequent critical component of the pipeline, addresses real-world challenges such as missing data, noise, and other dynamic road conditions, which will be discussed in detail later. This approach seeks to enhance the reliability of outcomes during model training while ensuring that the input data is both accurate and reflective of real-world driving conditions.

Normalization is then performed to standardize the lane data, as highlighted in Fig. 9. Using a shifted origin method, all lanes within a sequence are aligned to a common origin to enhance consistency and simplify the learning process. The Model Training phase, depicted in the lower section of Fig. 9, involves splitting the data into training and validation parts, structuring the input data as tensors, and optimizing the model for predictive performance. Various DNN architectures specified in Chapter 2, including LSTM, GRU, transformers, and hybrid models, are explored to effectively solve the multivariate time-series regression problem. Loss metrics, such as Mean Absolute Error (MAE) and curvature loss, are employed to evaluate the model's accuracy. Bayesian Hyperparameter Optimization (HPO) is employed as a tuner to refine the model's configuration, enabling it to achieve minimal training and validation losses. The model training process also incorporates masking techniques to exclude padded values during training and loss calculation, ensuring the model focuses solely on valid data points.

The second stage, Inference in the CARLA Simulation Environment, leverages the trained model to predict lane sequences in real-time, as shown in the bottom-most section of Fig. 9. The Data Generation Process (DGP), described in Chapter 2, is followed to generate the dataset. Real-time sequential frames are collected in the same sequence length used for training the model. The preprocessing pipeline established during the training phase is reused to process real-time detected frame data, ensuring consistency between training and inference. The trained model generates predictions for future frames, which are then denormalized to restore their original scale. Polynomial curve fitting is applied to the predicted lane sequences to ensure smoothness and precision in the generated reference path. Finally, a centerline is computed to serve as the reference path for the Model Predictive Control (MPC) framework.

Visualization of the results, including predicted lanes and the reference path, is performed using OpenCV. These visualizations provide insights into the model's performance and help identify any deviations caused by environmental noise or sensor inaccuracies. The overall pipeline, as summarized in Fig. 9, demonstrates the seamless integration of data preparation, model training, and real-time inference in a simulation environment. This robust and scalable framework paves the way for further development and deployment in real-world autonomous driving systems. Each stage of this pipeline will be explained in

greater detail in the following sections to provide a deeper understanding of the methods and techniques used.

3.1 Data Pre-processing

The primary objective of preprocessing in time-series forecasting is to transform the dataset into a canonical form that facilitates efficient training of a DNN model [42]. In this research, data preprocessing is a crucial step, as it ensures the raw lane data is converted into a consistent format compatible for training. Given the involvement of multiple datasets in this framework, it is evident that these datasets differ significantly in structure, resolution, the number of X and Y coordinates, lane lengths, lane indices, and the presence of missing values. This variation necessitates unification into a standardized format to enable effective training and prediction. First, a standardized format is defined to streamline this process. Based on a standard road scenario, benchmark datasets, and the previously established ULFD work within the CARLA Framework, the number of lanes in each frame is standardized to 4, the lane coordinates per segment are set to 16, the sequence length is defined as 9, and the initial forecast horizon is set to 1. These parameters ensure uniformity across the datasets and are represented as N=4, M=16, L=9, and T=1 (as shown in Fig. 2 from the problem statement). Additionally, since the ULFD model was primarily trained on TUSimple data, the standard frame resolution is chosen to be 1280x720 to align with this benchmark. The number of frames F varies across datasets, which presents the objective of extracting and forming sets of sequentially fitted X-Y pairs of lane coordinates from consecutive frames while maintaining a consistent sequence length. The preprocessing pipeline is designed to handle this variability while addressing challenges such as missing values, irregular lane lengths, and inconsistent data formats. By enforcing a standardized format, applying transformations, and normalizing the data to the standard resolution, the pipeline ensures that the input data is in a desired canonical form. This structured preprocessing lays the foundation for robust model training and reliable lane prediction in the context of time-series forecasting for autonomous driving systems.

3.1.1 Raw Data Extraction

The process begins with extracting raw lane data from multiple datasets, including TU Simple, CU Lane, and the RPTU Universität Dataset. The raw data, represented as X-Y pixel coordinates, is converted into a DataFrame format to enable efficient manipulation. To ensure uniformity, all frames are verified to contain exactly 4 lanes. Missing values or absent lanes are padded with a placeholder value of -2, as all valid pixel values are positive, making -2 a clear marker for invalid entries. While most datasets consistently have 4 lanes except for missing lanes, CU Lane presents instances with more than 5 lanes; in such cases, only the first 4 lanes are retained for training. The DataFrame is then sorted by timestamp to ensure the sequential ordering of frame coordinates across the entire dataset. To standardize the data further, frame resolution adjustments are applied. For example, datasets like CU Lane, originally with a resolution of 1640x590, are scaled to the standardized format of 1280x720 using non-uniform scaling factors. Although this introduces a change in the aspect ratio, the relative difference between lane positions of sequential frames remain proportional due to the consistent application of the scaling factors across the entire dataset.

3.1.2 Data Cleaning

During the data cleaning stage, invalid entries such as NaNs, infinite values, or missing values in occluded or missing lanes are systematically identified and replaced with the placeholder value (-2) for consistency. This step ensures that no invalid data points remain. Additionally, any out-of-bound coordinates—values falling outside the standardized resolution—are deemed invalid and replaced to maintain compatibility with the standardized format. Since the subsequent step involves curve fitting, it is critical to ensure that the X and Y arrays have lengths greater than 1. Arrays with only one element are treated as invalid, as they cannot be fitted to a curve. To prevent duplication, duplicate values within the arrays are eliminated by forming a union of the values beforehand. These steps collectively prepare the data for accurate curve fitting and downstream processing.

3.1.3 Curve Fitting with Arc-Length Parametrization

As discussed in the overview, curve fitting is critical for addressing inconsistencies in coordinate counts and lane point spacing. The raw data's X and Y arrays contained a random number of points per lane segment, necessitating standardization to a fixed number of points (M=16). This was achieved using curve fitting with an arc-length parametrization approach, adapted from [10]. The method involves first parameterizing the curve using cumulative distances (d) between waypoints, followed by reparameterization based on the arc length (s). Figure 3.2 illustrates this process, showing the fitted curve (blue) and cumulative distances (red). Both spline and polynomial curve fitting methods are compatible with this arc-length parametrization framework. However, the selection of technique depends on the stage of the workflow and the challenges presented by the data. During training, achieving accuracy in capturing local variations in lane structure is prioritized over dealing with outliers from lane detection. In contrast, during inference, computational efficiency, smoothness of curve and robustness to outliers are more critical.

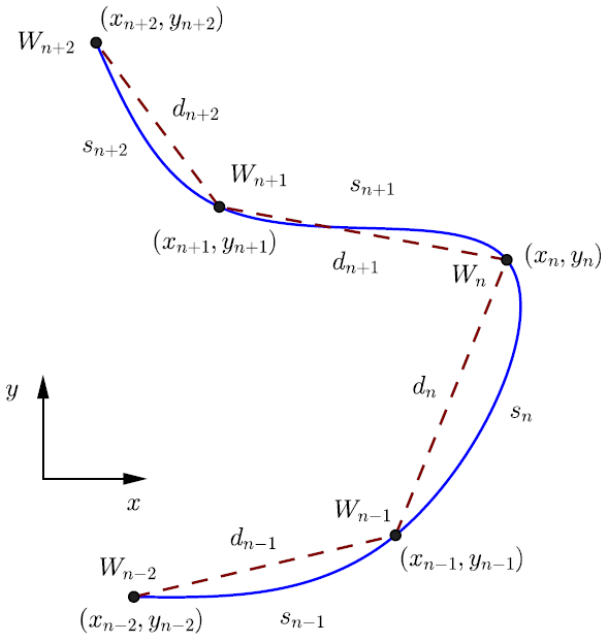


Figure 3.2: Curve Fitting with Arc-length Parametrization. [10]

3.1.3.1 Spline Curve Fitting for Model Training

During training, the focus is on enabling the model to learn from all potential variations in the data, including the presence of outliers. These outliers may represent unique or edge-case scenarios that the model must handle effectively during real-world deployment. Spline curve fitting is used in this phase because it joins all points, including outliers, without discarding or smoothing over them. While splines may not always fit the curve perfectly, their sensitivity to outliers ensures that the model is exposed to these variations. This decision is underpinned by the need for the model to learn from edge cases rather than relying solely on perfectly smooth data. By training on data that includes these variations, the model gains robustness in handling scenarios with noise or imperfect detections. Additionally, most outlier cases in the training data are filtered beforehand using the approach mentioned in the feature engineering section 3.2, making such occurrences rare. The combination of data filtering and spline fitting strikes a balance between realism and accuracy in the training dataset. Spline curve fitting involves parameterizing the curve with cumulative distances (d) and using these distances to interpolate the X and Y coordinates with a specified fit type (e.g., quadratic). Afterward, the arc length (s) is computed to re-parametrize the curve for uniform distribution of points. The algorithm followed for spline curve fitting is shown below.

Algorithm 3.1: Spline Curve Fitting with arc-length parametrization

function spline_curve_fitting (X, Y, n_points);
Input : X array, Y array, Number of points
Output : Interpolated curve with desired number of points
CALCULATE Cumulative distances between waypoints: $distance = \sqrt{dx^2 + dy^2}$;
PERFORM Interpolation of the curve using cumulative distances as the parameter: using `interp1d`;
if *length of array = 2* **then**
    'linear fit';
else
    'quadratic fit';
end
CALCULATE Arc lengths using differences between interpolated points;
PERFORM Last point estimation (Ensuring uniform length of array) by adding/subtracting mean of difference between arc lengths and cumulative distances;
PERFORM Interpolating again using the newly computed arc lengths with desired number of points;

3.1.3.2 Polynomial Curve Fitting

Inference requires a different approach, as the primary goal is to generate smooth, robust, and computationally efficient curves from model predictions. Unlike training, where outlier information may be valuable, inference aims to minimize the impact of outliers on the resulting trajectory. Polynomial curve fitting is highly suitable in this context because it is less sensitive to shape deviations caused by outliers. Even if the input data contains noisy points, the fitted polynomial curve remains stable and consistent, effectively ignoring the influence of outliers. An additional advantage of polynomial fitting is the ability to control

the smoothness of the curve by adjusting the degree of the polynomial. Higher degrees allow the curve to adapt more closely to intricate shapes, while lower degrees ensure broader smoothing. This flexibility makes polynomial fitting ideal for post-prediction processing, where stability and smoothness are prioritized over capturing outlier-induced variations. During inference, the curve is first parameterized using cumulative distances, followed by polynomial fitting of the X and Y coordinates. The resulting polynomial functions are re-parametrized using arc length (s) to ensure evenly distributed points along the curve. This method guarantees smooth, reliable trajectories that align closely with the road geometry, even when the input data contains minor imperfections. The algorithm followed for polynomial curve fitting is shown below.

Algorithm 3.2: Polynomial Curve Fitting with arc-length parametrization

1 function polynomial_curve_fitting $(X, Y, degree, n\_points)$;
Input : X array, Y array, Degree, Number of points
Output : Interpolated polynomial curve with desired number of points and degree
2 CALCULATE Cumulative distances between waypoints: $distance = \sqrt{dx^2 + dy^2}$;
3 PERFORM Polynomial curve fitting using cumulative distances with desired degree: using `np.polyfit`;
4 CALCULATE Arc lengths using differences between interpolated points;
5 PERFORM Last point estimation (Ensuring uniform length of array) by adding/subtracting mean of difference between arc lengths and cumulative distances;
6 PERFORM Interpolating again using the newly computed arc lengths with desired number of points;

3.1.4 Target Data Generation

The process of target data generation is a crucial step in the model training pipeline as it provides the ground truth required to compute training and validation losses. The optimizer adjusts the model weights based on these losses to improve predictive performance. In time-series forecasting, target data is generated by applying a sliding window to the input data. This method ensures that the model learns temporal dependencies by capturing patterns over a fixed sequence of frames while generating additional sequential data samples.

3.1.4.1 Time Delay Embedding

The next step involves implementing time delay embedding, as introduced in 2.5. This technique is used to address the challenge of limited data availability, which can lead to overfitting and poor model generalization. Incorporating extensive time series datasets can help mitigate overfitting by reducing the generalization error—the disparity between training and testing errors—but this approach is typically expensive and time-consuming [29]. To overcome this, time delay embedding generates additional sequential data using a sliding window approach. Instead of extending the length of the time series dataset, as shown in Figure 3.3, this method focuses on increasing its width by adding multiple overlapping time series [7]. This effectively increases the amount of data available for model training, improving its robustness and generalization capability.

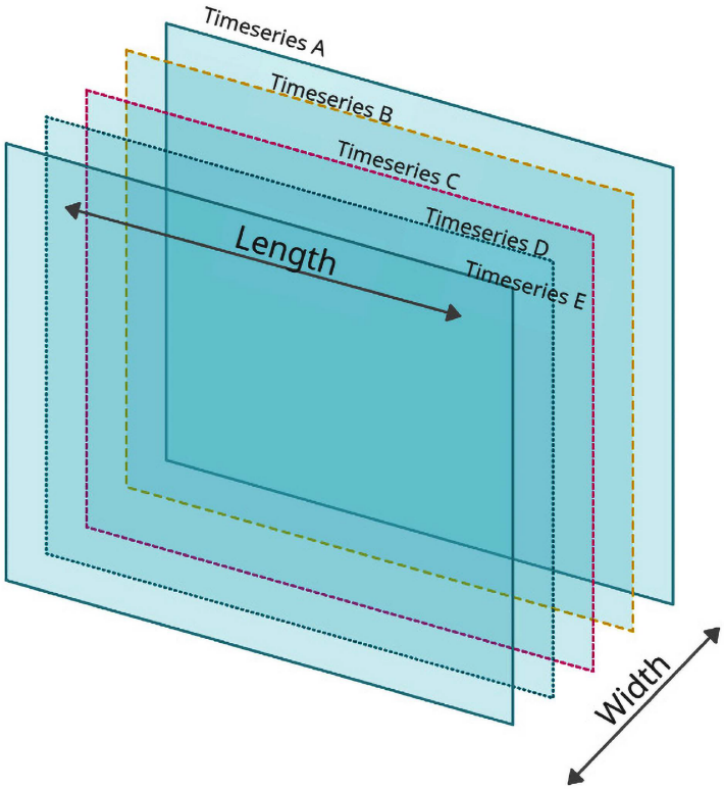


Figure 3.3: Time Delay Embedding. [7]

Each time series represents a training sample, consisting of a set of lane curves derived from a sequence of consecutive frames equal to the standardized sequence length L. According to the predefined format, L is set to 9 frames. Consequently, for each dataset, the data frame containing the complete time series is divided into smaller segments of 9 sequential frames. Before this segmentation, it is necessary to establish the forecast horizon (T). The forecast horizon determines how far into the future the model predicts and must be fine-tuned through trial and error based on model performance. Initially, T is set to 1 second, corresponding to 20 frames for most datasets (e.g., TUSimple [44]) and 30 frames for CU Lane [45], and later tested up to 4 seconds. This forecast horizon of 4s is double the average human perception-reaction time for braking (1.5 to 2 seconds) [46], providing a realistic prediction window for autonomous driving scenarios.

In time series terminology (as defined in 2.5), the sequence length corresponds to the window size, while the forecast horizon corresponds to the input-target offset. Using the problem-specific terminology from 1.3, F represents the total number of sequential frames in the dataset (complete time series length), which is divided into S time series samples. Figure 3.3 provides an overview, where the sequence length L defines the length of each time series, and the grid width represents the total number of samples S in the dataset.

3.1.4.2 Input-Target Offset Validation

In this step, input and target data are generated with an offset, which indicates how much ahead the target data is relative to the input data. In time-series forecasting, this offset is synonymous with the forecast horizon. Figure 3.4 illustrates the process followed to

generate the target data. As shown in this figure, the number of elements (Fi and Ft) along with the number of samples (Si and St) in the input and target data remain equal.

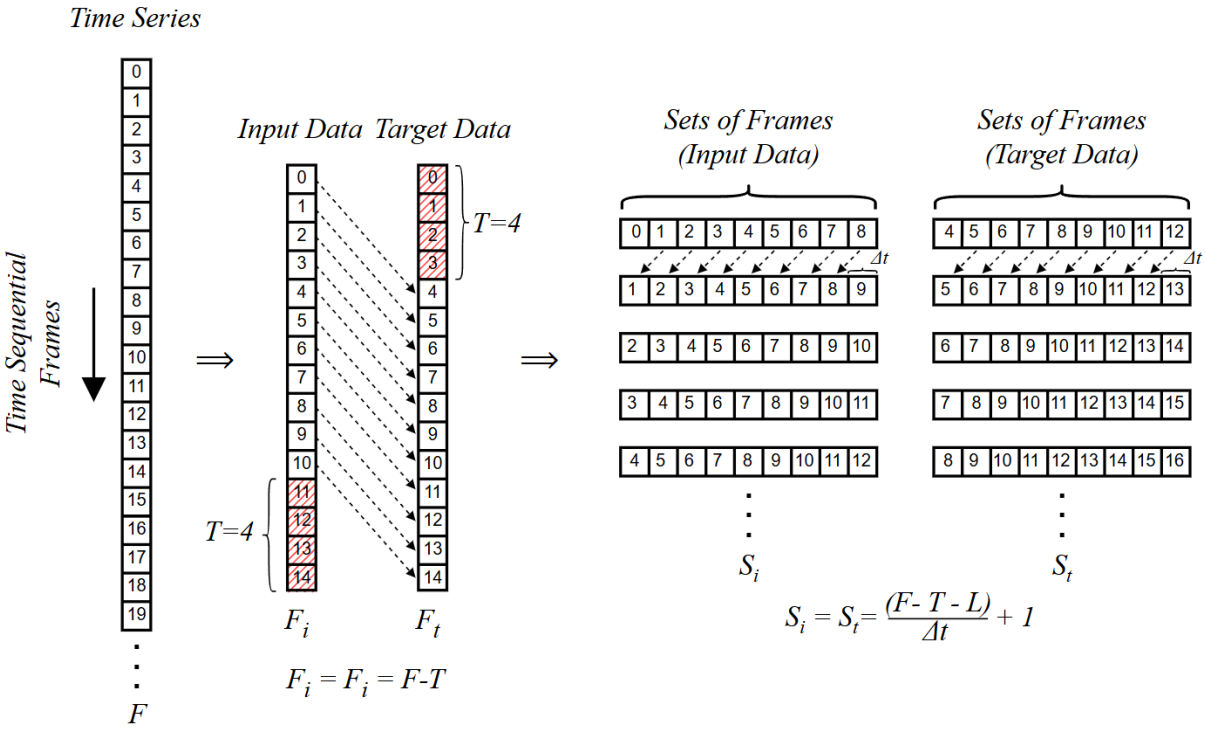


Figure 3.4: Formation of Input-Target Pairs.

The figure provides an overview of the entire process followed for each individual dataset. The raw time-series data is represented by sequential frames, denoted as $0, 1, 2, \ldots, F$, with the time index shown along the vertical axis to indicate the temporal order. According to the problem statement in Chapter 1, T is defined as the forecast horizon, which represents how far into the future the model should predict. In this example, $T = 4$, indicating that each frame in the target data should always be 4 frames ahead of the corresponding frames in the input data. The sequences should exhibit a consistent offset, meaning the target data should always follow the input data by the defined forecast horizon (T).

Each input-target pair must consist of sequentially ordered frames, where the input data represents past observations and the target data corresponds to future predictions. The input data is generated by removing the last T frames from the original time-series data, while the target data is created by removing the first T frames. This ensures that both input and target datasets have equal lengths, allowing for a consistent one-to-one correspondence between the input data (past observations) and the target data (future predictions). Further, these datasets are converted into input-target pairs with each set having 9 sequential frames. The formula used to compute the number of valid input-target pairs based on the sequence length (L), forecast horizon (T), and total number of frames (F) is given by:

$$S_i = S_t = \frac{F - T - L}{\Delta t} + 1 \tag{3.1}$$

Where:

- F is the total number of frames in the time-series dataset.
- T is the forecast horizon (the offset between input and target).
- L is the sequence length (number of frames per sequence).
- Δt is the time step between consecutive frames, which in this case always remains 1 to generate the maximum possible input-target pairs.

After generating the target data using the time-delay embedding method, it is essential to verify that the input-target pairs are correctly aligned. The offset between the input data (representing past lane positions) and the target data (representing future lane positions) must be consistent and properly applied. The validation step involves visualizing the processed data, where the lane coordinates are displayed alongside the raw data in video clips. This visualization ensures that the sequence of input and target pairs is correct, showing the desired offset and maintaining consistency.

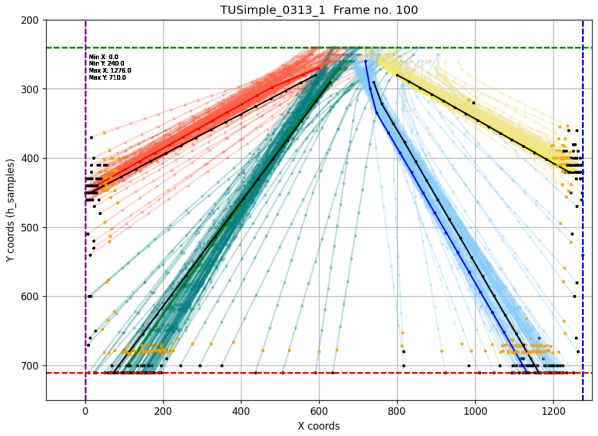
Based on this review, the model input data is finalized, ensuring that the model receives accurate data for training. At the end of this process, the output gives the input data and target data in a canonical form with the standardized format mentioned before. However, there are still some processes remaining, such as feature engineering and normalization, to further improve the dataset and prepare it for model training.

3.2 Feature Engineering

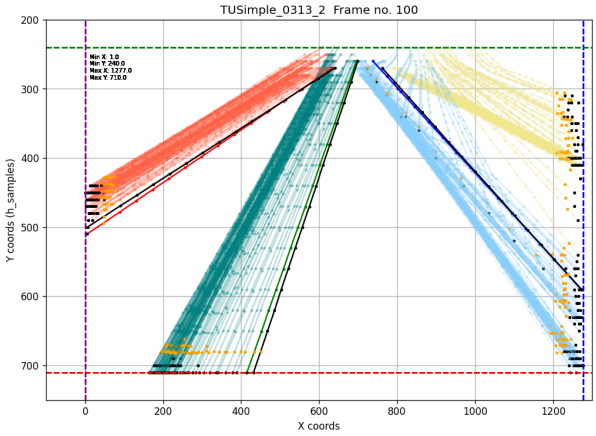
Feature engineering involves creating and refining features from data, often leveraging domain expertise, to facilitate a more effective and streamlined learning process [10]. In traditional machine learning, designing high-quality features is critical for achieving optimal model performance. This aspect of ML is highly problem-specific, requiring tailored solutions crafted uniquely for each scenario [10]. At this stage, the input and target time series data have been verified to be consistent and sequential. However, in real-world scenarios, various dynamic factors—such as missing lanes, outliers, road transitions, and environmental influences—can challenge the model's ability to generalize effectively. To build a robust model, it is crucial to account for these scenarios before training. This involves including data that reflects such cases directly in the training set. This necessity is one of the key reasons for leveraging multiple datasets during model training. This section outlines the problem-specific feature engineering techniques employed to address these challenges.

3.2.1 Time Series Data Analysis

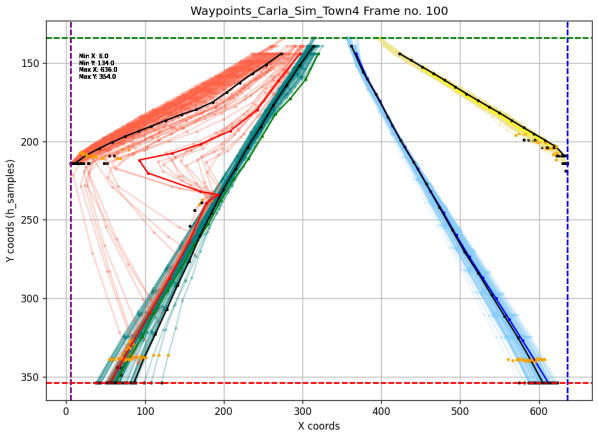
To determine which elements to filter out and the acceptable level of noise to include, it is essential to first analyze the time series data. This analysis is performed over the processed dataset after applying curve fitting, before it is grouped into sets of 9 frames for training. As mentioned in 2.7, the seven dataset sources include: two from TUSimple (label_0313_1 and label_0313_2), three from RPTU Universität (tripstadtersthin, tripstadthin, and tripstadtruck), one from CU Lane (driver23_30frame), and one from the existing ULFD CARLA simulation framework.



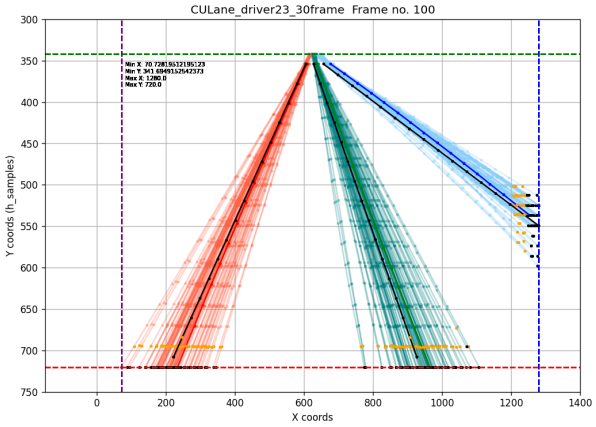
(a) TUSimple__0313__1



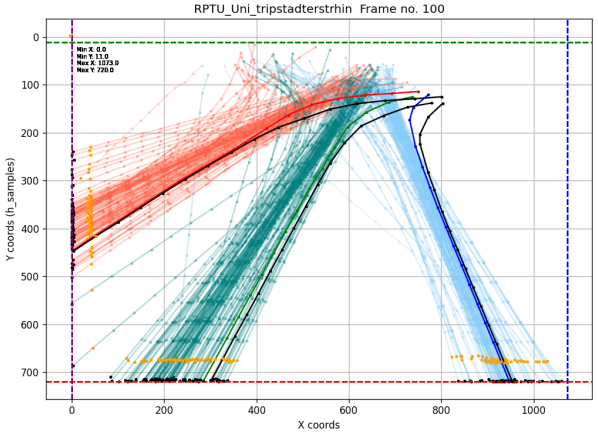
(b) TUSimple__0313__2



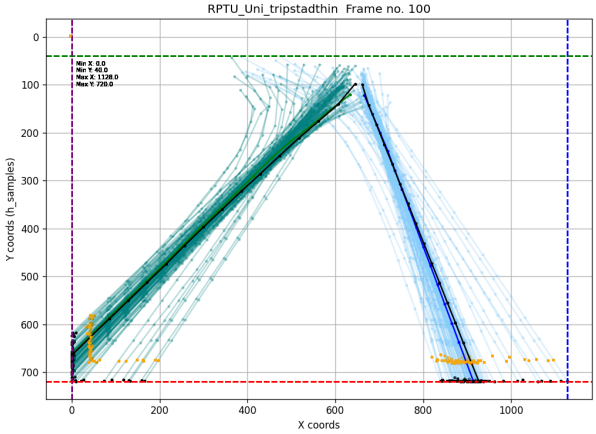
(c) waypoints__carla__town4



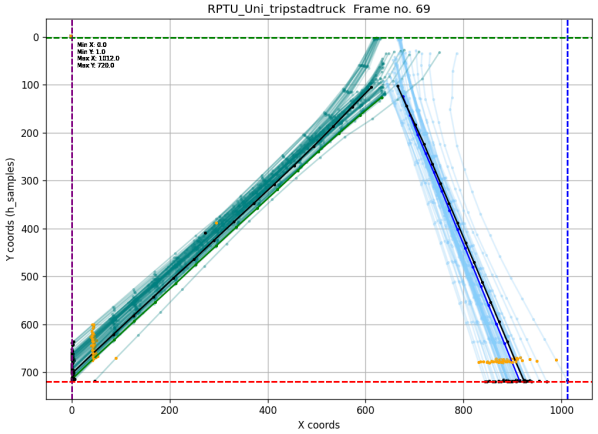
(d) CULane__driver23__30



(e) RPTU__tripstadterstrhin



(f) RPTU__tripstadthin



(g) RPTU__tripstadtruck

Figure 3.5: Overview of Datasets: First 100 sequential frames

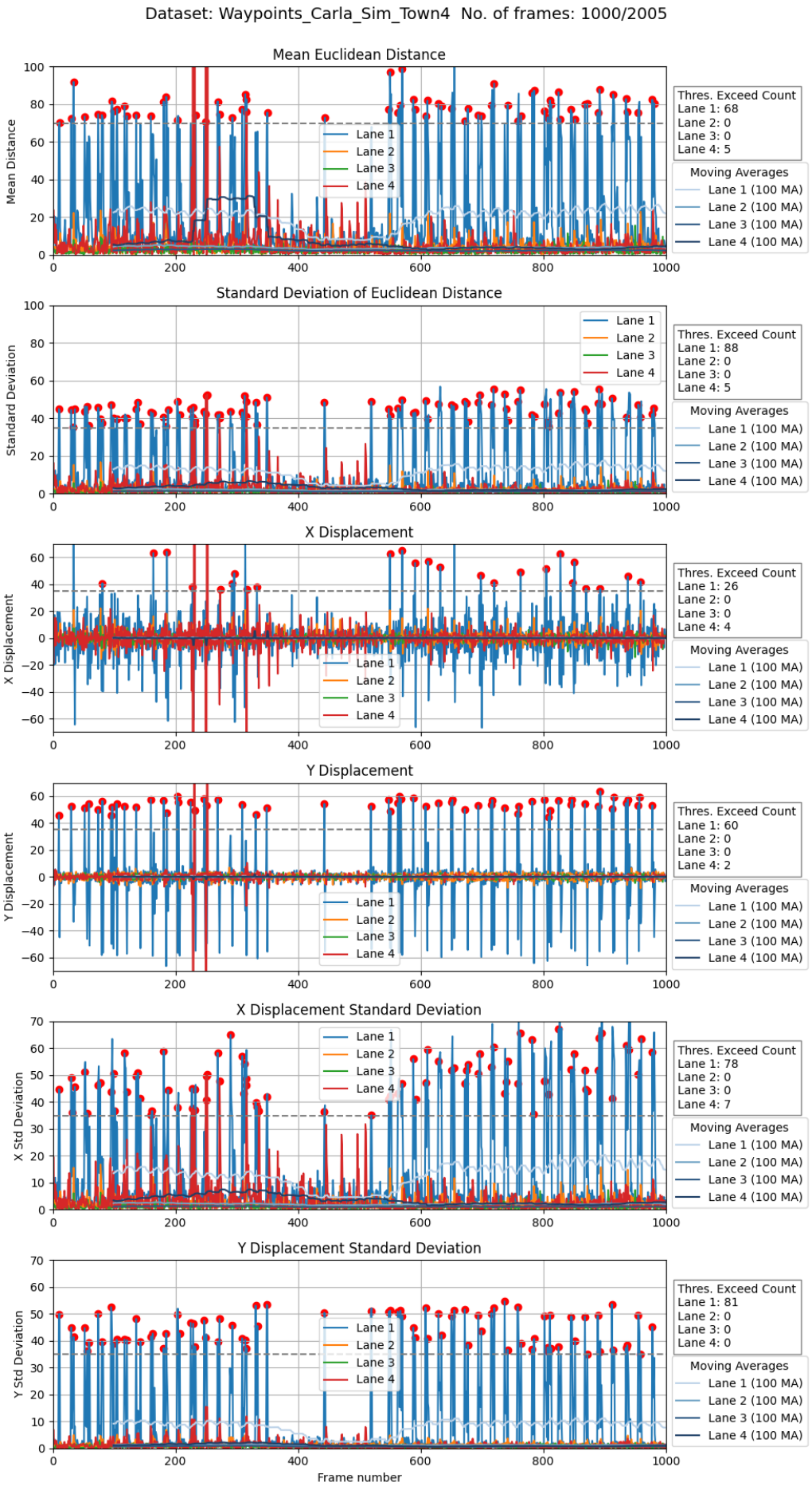


Figure 3.6: Time Series Analysis- Dataset from CARLA Simulation.

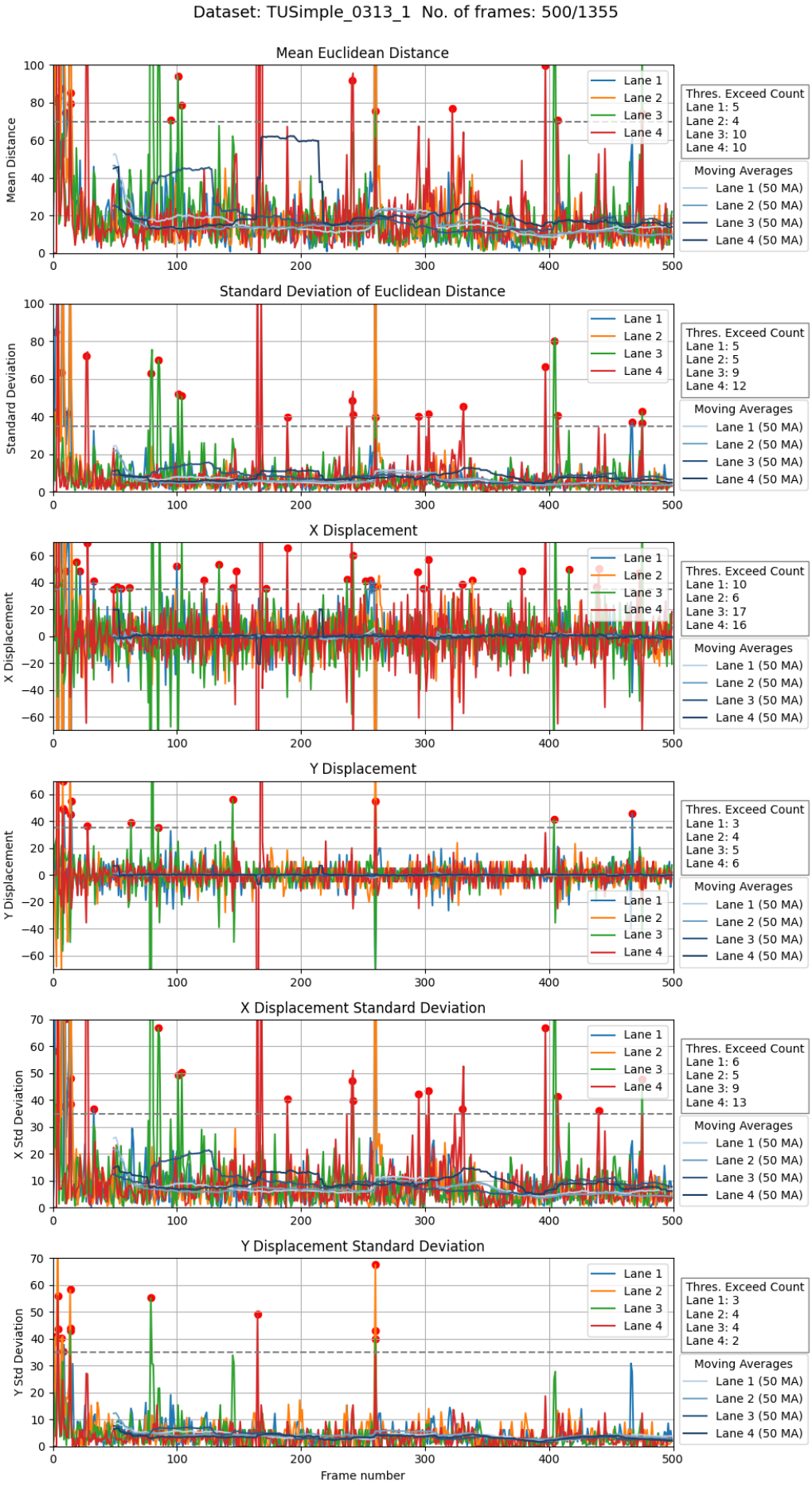


Figure 3.7: Time Series Analysis- TUSimple_0313_1

Figure 3.5 provides a visualization of the fitted lane coordinates from the first 100 frames of each dataset. Although this subset represents a small portion of the overall data, it effectively highlights key structural features.

In these figures, the four lanes are color-coded as- red for lane 1, green for lane 2, blue for lane 3, and yellow for lane 4. Bounding rectangles indicate the minimum and maximum values along the X and Y axes. Black lines represent the lane coordinates of the current frame, while the fully opaque colored lines represent the previous frame. The transparent colored lines in the background depict lane coordinates from earlier frames in the sequence, illustrating the temporal progression of the lanes. The black and orange points represent the first and second points of each lane curve, respectively, indicating the starting position and the initial direction of the lane. Note that missing values (-2) are hidden during visualization for clarity.

The dataset-specific observations reveal the distinct characteristics and challenges presented by each dataset. The TUSimple label_0313_1 3.5a dataset highlights lane change transitions of the ego vehicle, where the lanes progress smoothly over time, but transitions are noticeable along the X axis due to relative displacement between frames. Moreover, the TUSimple label_0313_2 3.5b dataset exhibits noise in lane 3 and lane 4 caused by lane bifurcation markings. Despite this noise, the dataset provides sufficient information to train robust models. The CARLA Simulation Framework dataset contains distorted or outlier values, particularly in lane 1, which reflect real-world challenges such as sensor inaccuracies or sudden environmental changes. The RPTU Universität tripstadtersthin 3.5e dataset features steeply curved lanes and instances where the fourth lane is missing in the first 100 frames, capturing dynamic scenarios often encountered in urban driving environments. The CU Lane driver23_30frame 3.5d dataset displays straight and consistent lane lengths with stable Y-coordinates, but the first lane is missing. The RPTU Universität datasets tripstadthin 3.5f and tripstadtruck 3.5g focus solely on ego lanes and the X-coordinates show significant horizontal displacement due to turns.

The analysis of the datasets provides key insights. It is evident that the relative difference between pairs of consecutive frames is minimal. However, there is a significant difference between the current frame and frames from 9 or 10 frames earlier, highlighting the importance of capturing long-term dependencies. In these datasets, the Y-coordinates remain relatively stable over time due to the consistent positioning of the camera frame, whereas the X-coordinates exhibit greater variability, particularly during left and right turns. However, in cases of steep turns, a slight increase in Y displacement is observed. Additionally, during occlusions, variable lengths of lane markings may occur, indicating some degree of interdependence between X displacement and Y displacement.

To understand more about the interdependencies, the statistical evaluation is performed over the sequential consecutive frames in each dataset, which provides crucial insights into the temporal dynamics of datasets, enabling the identification of key trends and outliers. This includes calculating metrics such as mean euclidean distance, standard deviation, and X and Y displacements for each lane across consecutive frames. These metrics quantify the lane shifts, noise levels, and overall consistency of lane markings. The examples are provided in figures 3.6 and 3.7, two of the seven datasets with consideration of almost half of the sequential frames. The figures for the rest of the datasets are provided in the appendix. The threshold values for displacement and standard deviation are set to 70

and 35, respectively. Observations from the waypoints_carla_simulation dataset clearly demonstrate that lane 1 has a significant number of outliers exceeding the threshold.

Dynamic scenarios such as missing lanes, outliers, and road transitions must be included in the training data to ensure the model generalizes well to real-world conditions. These scenarios are evident across the datasets, and their inclusion helps the model handle edge cases effectively. This analysis serves as the foundation for the further applied feature engineering techniques.

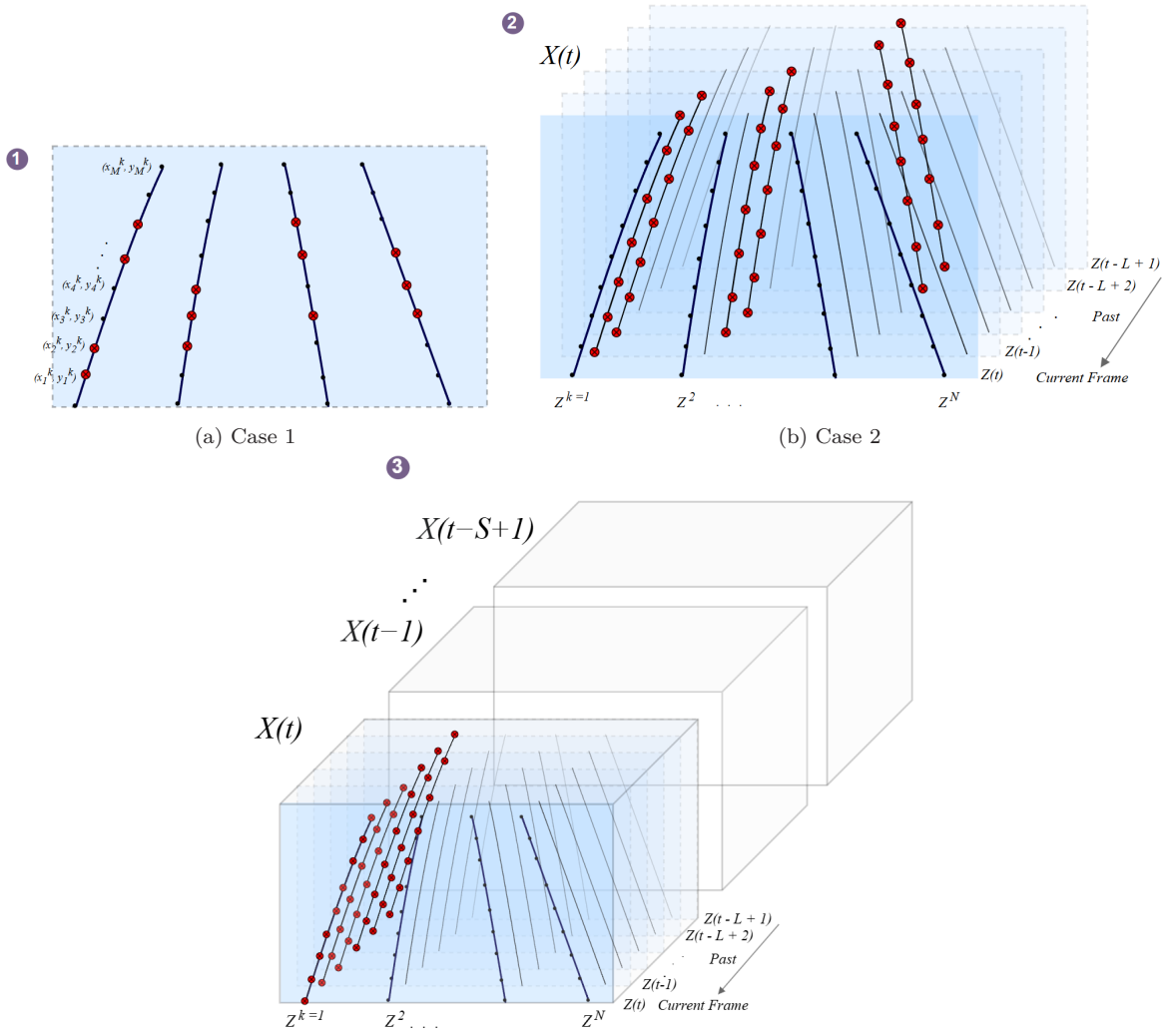
3.2.2 Handling Missing Values

Handling missing data is an inevitable challenge when working with large time-series datasets. To handle this effectively, it is first necessary to analyze what the missing-ness implies. In this problem, missing data can arise in three distinct scenarios. The first scenario, illustrated in Figure 3.8a, involves missing lane coordinates in individual lane curves, meaning certain M coordinates (shown with red cross-marks) are absent within a lane curve. During the data processing phase, such missing values are padded with -2. This padding scheme is also applied to invalid values, such as NaNs and out-of-bound data points, ensuring consistency across the dataset. This scenario is resolved during the curve-fitting process through interpolation, enabling the reconstruction of the lane curves. As a result, padded values remain only in cases where lanes are completely empty.

The second scenario, shown in Figure 3.8b, occurs when the lanes are missing in specific frame sequences of individual training samples ($X(t)$). This issue can arise due to errors in lane detection, environmental factors, or occlusions that prevent certain lanes from being detected. This scenario must be addressed carefully to ensure that the model can still extract meaningful patterns despite the incomplete data.

The third scenario, represented in Figure 3.8c, occurs when a specific lane is missing across the entire dataset. This persistent absence could result from consistent issues during data acquisition, such as sensor limitations or challenging road conditions. This situation poses a unique challenge because it reflects a systematic lack of information rather than occasional gaps, which could cause false positives.

Both scenarios can be effectively handled using the forward-filling method described in Algorithm 3.3. The algorithm processes individual samples with a tensor shape of 1,9,128, where each sample consists of 4 lane sequences with a shape of 1,9,32. For each lane sequence, it ensures that the first and last lanes from individual lane sequences are valid and do not contain entirely missing values. Forward filling is applied only when there are occasional gaps in the sequence, preventing incorrect filling of lanes that are entirely missing throughout. This distinction is critical as the absence of valid values in both the first and last lanes indicates a lack of evidence for the presence of lane markings, thereby avoiding false-positive responses. By ensuring that forward filling is performed selectively, the algorithm maintains the sequential consistency of the data while preventing the processing of entirely empty lanes. This approach ensures the dataset remains accurate and reliable, effectively addressing both scenarios of missing lane coordinates within individual sequences and missing lanes across frames.



(a) Case 1

(b) Case 2

(c) Case 3

Figure 3.8: Handling Missing lanes: Overview of Cases 1, 2 and 3

Algorithm 3.3: Feature engineering: Forward Fill Missing Values

function Forward_fill_missing_values ;
Input : $l = [l^1, l^2, l^3, l^4]$ where $l^N = [Z^N(t), Z^N(t-1), ..Z^N(t-L+1)]$ (tensor shape: 1,9,128) for $L = 9$
Output : Forward filled l^1, l^2, l^3, l^4
for l^N *in* l *(tensor shape: 1,9,32)* **do**
    `// Identify invalid lanes`
    IDENTIFY Lanes with all padded values $[-2]$;
    if $\forall z \in Z^N(t), z \neq -2$ ***AND*** $\forall z \in Z^N(t-L+1), z \neq -2$ **then**
        `// Perform imputation for missing values`
        PERFORM Forward filling;
    else
        pass;
    end
end

3.2.3 Outlier Detection and Handling

During the time series analysis of lane sequences, it was observed that outliers could be effectively identified by computing the standard deviation of Euclidean distances between lanes in consecutive frames. However, relying solely on consecutive frames proved to be ineffective due to the presence of random invalid lanes within the sequences. The data generation process produces each sample iteratively, consisting of 9 frames in a sequence. This iterative nature often leads to scenarios where an invalid lane is mistakenly identified as the reference lane, resulting in incorrect distance calculations. Moreover, invalid lanes may appear continuously across multiple frames in a sequence or as multiple anomalies within a single frame. This highlights the need for a robust method to reliably identify a reference lane for Euclidean distance calculations.

To address these challenges, an algorithm 3.4 was developed to operate at the sample level, focusing on all individual frames within a sequence rather than relying on consecutive ones. In this approach, the mean Euclidean distance of each lane to all other lanes within the same sample is calculated. The lane with the smallest mean distance is selected as the reference lane, as it is considered the closest to the cluster formed by all other lanes in the frame. This method is conceptually similar to the k-nearest neighbor (KNN) classification technique used in clustering [11]. In KNN, as shown in the figure, the classification of an element is determined by the majority vote of its neighbors within a specified neighborhood (k) [11]. However, in this algorithm 3.4, the elements are lanes represented by group of points rather than single points. Here, the k value is reinterpreted as the mean Euclidean distance between corresponding pairs of curve points, rather than a fixed neighborhood size.

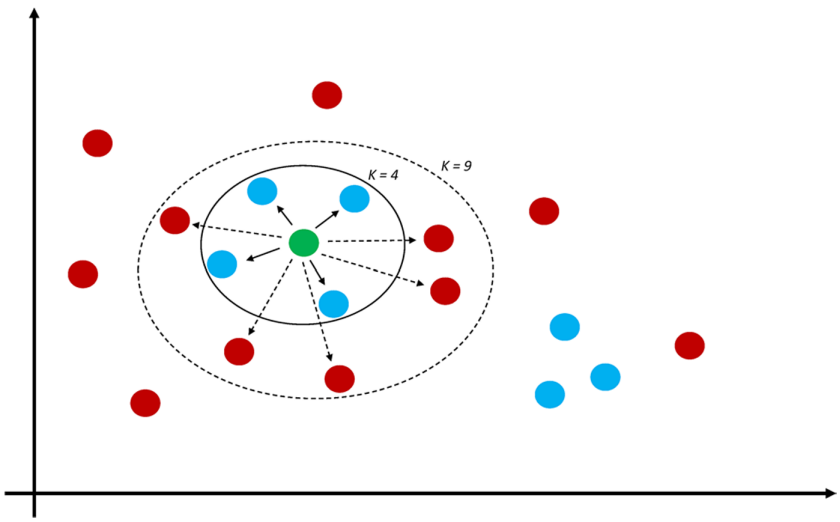


Figure 3.9: K-Nearest Neighbour Method. [11]

Once the reference lane is identified, it is used as a basis for computing the distances of all other lanes. The standard deviation is calculated based on these distances and a threshold criterion is applied to determine whether a lane is an outlier. The threshold value is chosen based on experimental analysis conducted across multiple datasets and validated using video clips and simulations performed in CARLA. If a lane is found to exceed this threshold, corrective measures are taken. If the outlier lane is not the first

Algorithm 3.4: Feature Engineering: Standard Deviation Filter

function Standard_Deviation_Filter;
Input : $l = [l^1, l^2, l^3, l^4]$, where $l^N = [Z^N(t), Z^N(t-1), \ldots, Z^N(t-L+1)]$ with $L = 9$ and each $l^N \in \mathbb{R}^{1\times 9\times 32}$
Output : Filtered lanes l^1, l^2, l^3, l^4 with filled or padded outlier lanes
for $l^N \in l$ **do**
  // Step 1: Compute pairwise Euclidean distances between l^N and all other lanes l^Q in l
  $d^N = [\text{Euclidean}(l^N, l^Q) \mid \forall l^Q \in l, Q \neq N]$;
  // Step 2: Compute the mean of pairwise distances across L frames
  $D^N = [\mu(d_1^N), \mu(d_2^N), \ldots, \mu(d_9^N)]$;
  // Step 3: Identify reference lane l_r^N with lowest mean distance
  $l_r^N = \arg\min_{l^M \in l}\{\mu(D^M)\}$;
  // Step 4: Compute Euclidean distances from each frame in l^N to the reference lane l_r^N
  $d_r^N = [\text{Euclidean}(Z^N(t-k), Z_r^N(t-k)) \mid k \in \{0, \ldots, L-1\}]$;
  // Step 5: Calculate standard deviation of distances
  $\sigma^N = \sigma(d_r^N) = [\sigma_1^N, \sigma_2^N, ..., \sigma_9^N])$;
  // Step 6: Check for lanes with σ^N exceeding threshold
  for σ^i *in* $\sigma^N \mid i \in \{0, \ldots, 9\}$ **do**
    $Z_{\text{valid}} = []$;
    if $\sigma^i >$ *threshold* **then**
      // Step 6a: Forward fill Z with previous VALID lane
      if $Z_{valid} \neq \emptyset$ **then**
        $Z = Z_{\text{valid}}$;
      end
      else
        // Step 6b: Step 6b: If no prior valid Z values exist, pad with (-2)
        $Z = [-2, -2, \ldots, -2]$*32;
      end
    end
    else
      $Z_{\text{valid}} = Z$;
    end
  end
end

lane in the sequence, forward filling is performed, where the lane is interpolated based on the preceding valid frame. If forward filling is not possible, the lane is replaced with a default padded value to maintain the sequence's structure.

3.2.4 Filtering Noise: Consideration of Road Dynamic Factors

In this step, the dataset is adopted to certain real-world issues which may occur during the inference. The samples exhibiting these issues are carefully identified and excluded from the training and validation sets to ensure robustness. These cases include lane change scenarios during vehicle maneuvers, lane bifurcations, stagnant or identical frames during

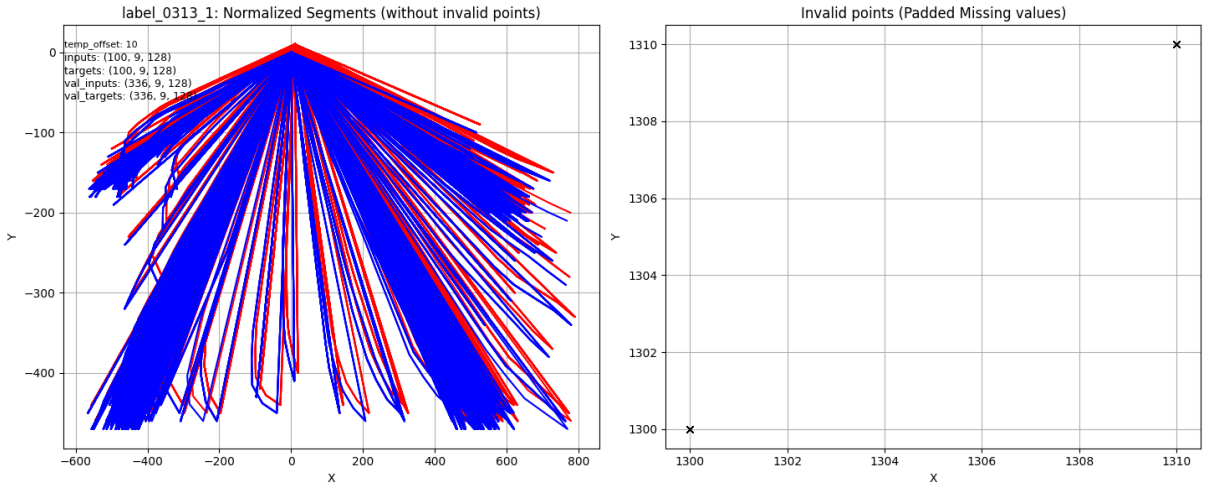
stable vehicle states, fluctuations in lane lengths (Y-coordinates), and environmental or sensor noise. Each of these cases poses unique challenges to the model, requiring specific pre-processing steps to mitigate their effects.

The first case involves lane change scenarios, which significantly disrupt the time-series patterns of lane data. During lane changes, the lateral (X-coordinate) displacement of the lanes increases substantially towards the side of the turn, deviating from the usual trends observed in the data. This disruption occurs because the lanes switch their positions, conflicting with the sequential structure on which the data preparation relies. Lane changes are inherently random and introduce complexity, making it difficult for the model to generalize effectively. Moreover, since the objective is to generate local candidate paths without considering obstacles, these lane transitions do not provide meaningful information for the model. To address this, lane change cases are identified through a combination of time-series analysis and visualization of the normalized dataset. Normalization, which is described in detail later, helps highlight these anomalies. For instance, lane change scenarios are flagged based on the difference between filtered and unfiltered normalized samples, enabling their exclusion from training. The figure 3.10 illustrates a normalized data example for the first 100 training samples, each with a tensor shape of (1, 9, 128). The upper portion displays the unfiltered dataset, while the lower portion presents the filtered dataset.

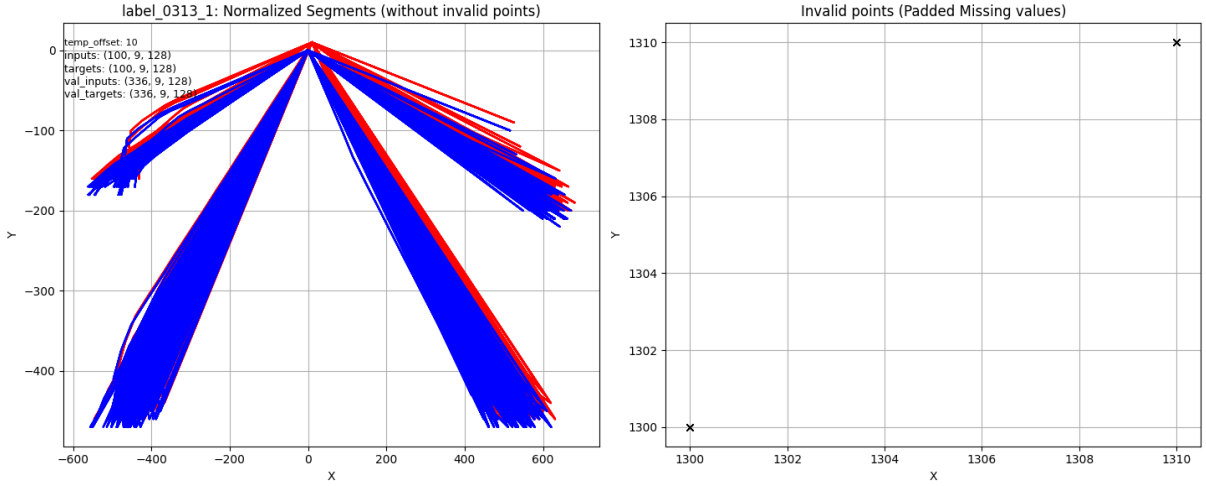
The second case concerns stagnant or identical frames that can occur during stable vehicle states, such as stationary traffic scenarios. To address these scenarios, the CULane [45] dataset, which includes diverse traffic scenarios, is incorporated into the training data. In such instances, duplicate values may appear across frames, leading to redundant data. This issue is mitigated during the data preprocessing phase, where duplicates are filtered out. For example, if 4 out of 9 frames in a training sample are identical, the model uses the remaining frames to make predictions. While this scenario does not heavily impact the time-series sequence, it is crucial for the model to account for it during training to improve reliability.

The third case involves fluctuations in lane length, specifically variations in the Y-coordinates of the lane markings. Such fluctuations typically arise due to interruptions in painted lane markings caused by occlusions, bifurcations, or inaccuracies in lane detection. Under normal conditions, changes in Y displacement values between consecutive frames are relatively consistent. However, extreme variations are occasionally introduced in the dataset to ensure the model learns to handle such cases. This approach allows the model to generalize better when encountering abrupt changes in displacement during inference.

The final case focuses on environmental or sensor noise, which is unavoidable in real-world scenarios. Extreme weather conditions or sensor malfunctions can introduce inconsistencies in the detected lane markings. While such noise cannot be entirely eliminated, the dataset incorporates measures to reduce its impact. For instance, Gaussian noise is augmented in the training data to simulate realistic sensor imperfections. However, it was observed that the model performs optimally with a Gaussian noise standard deviation of up to 0.5, maintaining an accuracy of approximately 95%. Beyond this threshold, the accuracy decreases significantly. These findings demonstrate the model's ability to tolerate a certain level of noise while maintaining performance.



(a) Unfiltered Normalized Data



(b) Filtered Normalized Data

Figure 3.10: Lane Change Scenario: Filtered vs Unfiltered Example

By addressing these cases systematically, the dataset is refined to better reflect real-world conditions, enabling the model to generalize effectively and maintain high accuracy during inference.

3.3 Normalization

Normalization is a pre-processing technique designed to minimize the impact of outliers and feature dominance by ensuring numerical consistency in the data [47]. This is accomplished by scaling or transforming features to a common range, ensuring that every feature contributes equally to the model in terms of its distribution range [47]. This step not only accelerates model convergence during training but also improves the model's robustness. Lane coordinate data is normalized by shifting the origin of all coordinates to zero and adjusting the range of pixel values. The process also accounts for padded missing values, ensuring that data integrity is maintained during model training and evaluation.

In this study, the purpose of normalizing coordinates is to shift all lane data to a common origin and bring the coordinate values to a consistent range. This makes the data invariant

to translation, ensuring the model focuses on lane shapes rather than absolute positions. The figure 3.11 shows the example for a single frame.

Original Range: $x \in [0, 1280],\ y \in [720, 0]$ (inverted Y-axis due to pixel representation).

Normalized Range: $x \in [-640, 640],\ y \in [-720, 0]$.

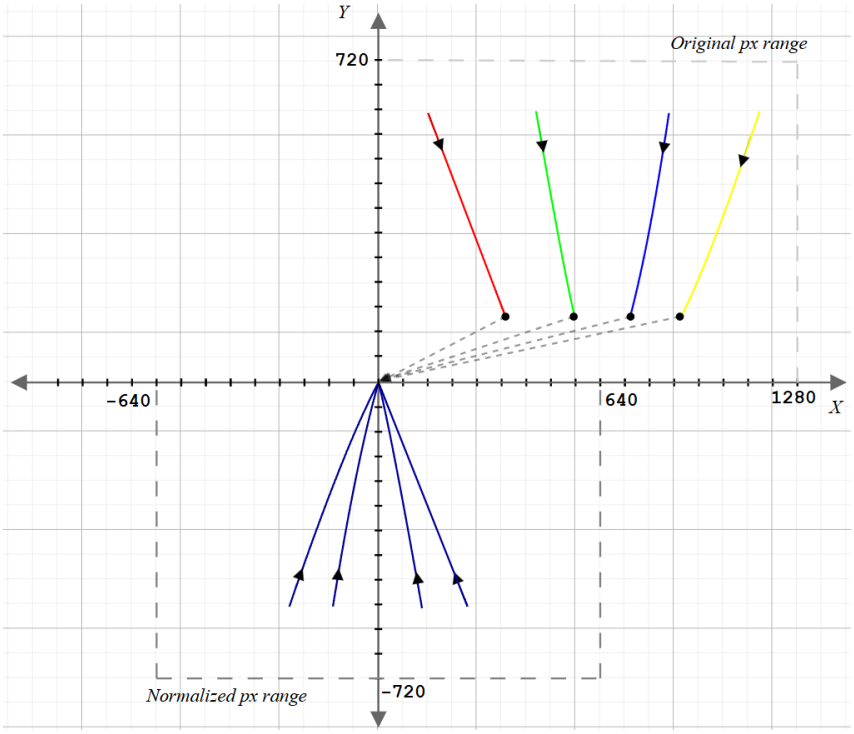


Figure 3.11: Normalization: Shifted Origin Method.

It ensures that the model learns meaningful patterns by simplifying the data distribution and reducing the effects of varying coordinate magnitudes.

The shifted origin method normalizes each coordinate relative to the first valid value (the origin) in a sequence. For each input tensor of shape $(1, 9, 32)$, the dimensions are defined as follows:

- **1**: Batch size, indicating that the operation is applied to one batch or sample at a time.
- **9**: Sequence length, representing the number of frames in the input.
- **32**: Features, consisting of 16 pairs of (x, y) coordinates for each frame. After normalization, the inputs and targets for each lane sequence $(1, 9, 32)$ are concatenated across four lanes, forming tensors of shape $(1, 9, 128)$ representing the training sample size.

The normalization process begins by extracting the first valid (x, y) coordinate pair in the sequence. These values are considered the origin for the respective frame. Next, the origin values are subtracted from all other coordinates in the frame, effectively shifting the

frame's coordinate system to align with the origin. To enable de-normalization, the origin values, c_x and c_y, are stored separately. These stored values are critical for reconstructing the original coordinates after the model processes the data.

For padded missing values, identified by the marker $(-2, -2)$, the origins are assigned a unique value of -1302. This value lies outside the x-y pixel range, ensuring that the padded values remain clearly distinguishable during model training. During the de-normalization process, these padded values are restored to their original -2 representation, maintaining the integrity of data.

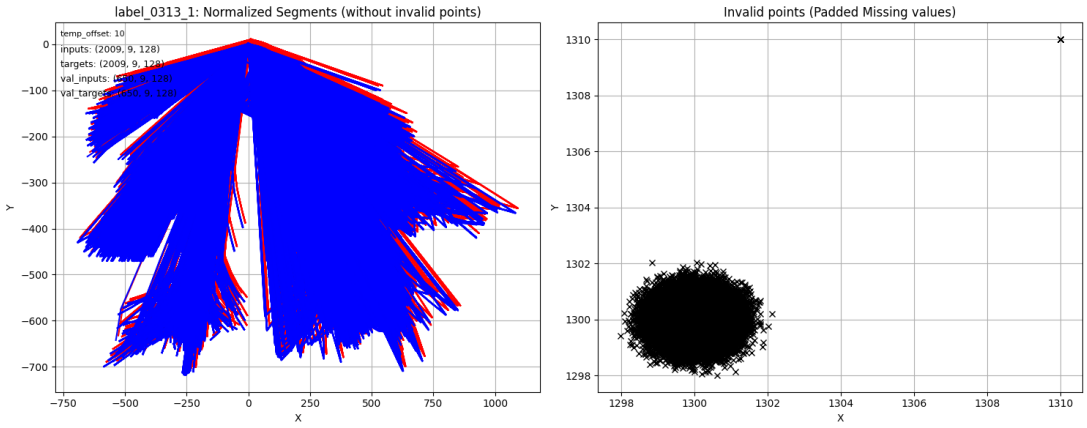


Figure 3.12: Normalized Final Data for Seq-2-Seq Model

The figure 3.12 illustrates the final normalized input data prepared for the Seq2Seq model. The plot on the right highlights the presence of added random Gaussian noise in the invalid values. These values are carefully constrained to remain below the threshold of 1280, ensuring they can be easily identified and masked during training.

4. Experiments

This chapter outlines the experimental framework developed to train, evaluate, and deploy the DNN model for generating reference path in autonomous driving scenarios. Building on the data preparation phase outlined in 3.1, this chapter progresses through model training, ground-truth evaluation, and integration within the CARLA simulation framework.

The data preparation phase outputs normalized input and target data, forming the basis for subsequent experiments. The structure and selection of this dataset are first detailed under experimental setup. Then the selected dataset is shown which is used for model training. The Model Training Process section elaborates on the training methodology, highlighting key aspects such as loss functions, handling missing values through end-to-end masking, and hyperparameter optimization.

Next, the methods followed for Ground-Truth Evaluation are explained, emphasizing the metrics and techniques used for both quantitative and qualitative evaluation of the trained models. These include curvature loss, Frechet distance, and the newly introduced similarity score. The chapter then presents a comprehensive Results and Analysis, comparing the performance of various model architectures.

Finally, the second phase of the implementation pipeline is addressed, focusing on deploying the trained model within the CARLA simulation environment. This section discusses real-time inference, reference path computation, and integration with a kinematic MPC controller, providing a complete overview of the model's deployment and application in autonomous driving scenarios.

4.1 Experimental Setup for Model Training

This section provides a comprehensive overview of the experimental setup, including the dataset structure, selected datasets for training, the training-validation split, and the overall system configuration. The experiments were conducted on two systems: a Linux system equipped with 4 NVIDIA Tesla V100-DGXS GPUs (each with 32 GB of RAM, running CUDA version 10.2) and a Windows device equipped with NVIDIA GeForce GTX 1650 GPU (8 GB of RAM, running CUDA version 12.1). PyTorch was utilized for both

Table 4.1: Dataset Summary

Dataset	Batch Size %	Inputs Shape	Targets Shape	Val Inputs Shape	Val Targets Shape
TUSimple_0313_1	100	(820, 9, 128)	(820, 9, 128)	(257, 9, 128)	(257, 9, 128)
TUSimple_0313_2	100	(953, 9, 128)	(953, 9, 128)	(302, 9, 128)	(302, 9, 128)
RPTU_Uni_trippstadthin	100	(156, 9, 128)	(156, 9, 128)	(53, 9, 128)	(53, 9, 128)
RPTU_Uni_trippstadterstrhin	100	(220, 9, 128)	(220, 9, 128)	(74, 9, 128)	(74, 9, 128)
RPTU_Uni_trippstadtruck	100	(41, 9, 128)	(41, 9, 128)	(14, 9, 128)	(14, 9, 128)
CULane_driver23_30frame	100	(2745, 9, 128)	(2745, 9, 128)	(773, 9, 128)	(773, 9, 128)
way_pts_carla_town04	100	(1493, 9, 128)	(1493, 9, 128)	(498, 9, 128)	(498, 9, 128)
Final Merged	N/A	(6428, 9, 128)	(6428, 9, 128)	(1971, 9, 128)	(1971, 9, 128)

the model training loop and data preparation, while Bayesian optimization was employed for tuning model parameters.

As described in the problem statement 1.3, the primary objective is to develop a deep neural network (DNN) V that maps an input sequence $X(t)$ to its predicted future sequence $\hat{X}(t)$, defined as:

$$V : X \to Y, \quad X(t) \mapsto V(X(t)) = \hat{X}(t).$$

The output of the data preparation phase consists of normalized 3D tensors for inputs and targets, formatted as (B, L, C), where B denotes the batch size, L indicates the sequence length, and C refers to the features or coordinate dimensions. The overall available processed data is summarized in 4.1. It should be noted that this table shows the datasets available for a forecast horizon $T = 4$. The total number of available datasets slightly increases for $T = 1$.

Each training sample $X(t)$ is structured with a shape of $(1, 9, 128)$, representing a single sample or batch with a sequence length of $L = 9$ and 128 features. As discussed in the feature engineering section, each sample is formed by concatenating four lane sequences l^1, l^2, l^3, l^4, each with a shape of $(1, 9, 32)$:

$$l = [l^1, l^2, l^3, l^4],$$

where each l^N represents an individual lane sequence with time sequential images and is defined as:

$$l^N = [Z^N(t), Z^N(t-1), \ldots, Z^N(t-L+1)],$$

with $l^N \in \mathbb{R}^{1\times 9\times 32}$.

4.1 presents a total of 7 datasets, comprising 8,399 samples. These datasets are divided into training and validation sets using a 75-25% split. The selection of datasets for model training is guided by experimentation with batch size percentages. For instance, 4.2 showcases the dataset selected for the best-performing trained model, highlighting the inclusion of Gaussian noise and its corresponding characteristics. Each dataset is allocated a specific batch size percentage, with 50% of the samples from each dataset randomly selected and augmented with Gaussian noise to introduce variability. The noise is applied exclusively to the input data with a predefined standard deviation, ensuring that the variability aligns with the expected distribution of real-world data. Validation sets remain untouched by noise to provide an unbiased evaluation of the model's performance.

For example, the dataset TUSimple_0313_1 has a batch size percentage of 40%, resulting in 328 total samples for training, with 164 (50%) of these samples randomly augmented

Table 4.2: Selected Dataset for Model Training

Dataset	Batch Size %	Gaussian Noise	Inputs Shape	Targets Shape	Val Inputs Shape	Val Targets Shape
TUSimple_0313_1	40	164/328 samples	(328, 9, 128)	(328, 9, 128)	(102, 9, 128)	(102, 9, 128)
TUSimple_0313_2	40	190/381 samples	(381, 9, 128)	(381, 9, 128)	(120, 9, 128)	(120, 9, 128)
RPTU_Uni_trippstadthin	100	78/156 samples	(156, 9, 128)	(156, 9, 128)	(53, 9, 128)	(53, 9, 128)
RPTU_Uni_trippstadterstrhin	100	110/220 samples	(220, 9, 128)	(220, 9, 128)	(74, 9, 128)	(74, 9, 128)
RPTU_Uni_trippstadtruck	100	20/41 samples	(41, 9, 128)	(41, 9, 128)	(14, 9, 128)	(14, 9, 128)
CULane_driver23_30frame	5	68/137 samples	(137, 9, 128)	(137, 9, 128)	(38, 9, 128)	(38, 9, 128)
way_pts_carla_town04	50	373/746 samples	(746, 9, 128)	(746, 9, 128)	(249, 9, 128)	(249, 9, 128)
Final Merged	N/A	1003/2009 samples	(2009, 9, 128)	(2009, 9, 128)	(650, 9, 128)	(650, 9, 128)

with Gaussian noise. The best-performing model, a Seq2Seq Transformer, uses only 30% of the total available data for training due to its high computational requirements. However, for other models like LSTM, RNN etc. almost 70% of available data is utilized for the experiments.

4.2 Model Training Process

This section provides a comprehensive explanation of the methodologies and techniques utilized for training the proposed deep neural network. The section outlines the strategies employed to ensure robust and efficient learning, especially in the context of handling real-world challenges such as missing data, outlier scenarios, and varying data characteristics.

It begins with a discussion of the loss metrics used to guide training, including primary metrics like MSE, RMSE, MAE, and Huber Loss, as well as additional terms such as curvature loss and polynomial coefficient loss. Strategies for handling missing values during training are detailed, focusing on masking techniques and their integration with transformer-based models, alongside a comparison of different masking mechanisms. The section also explores the choice of training hyper-parameters, covering optimizer selection, training loop configuration, and hyper-parameter tuning via Bayesian optimization. Finally, the structure and execution of the training loop and the experiments conducted to optimize the model are explained, with results summarized through trends in training and validation performance.

4.2.1 Loss Metrics

The primary objective of any classical machine learning model is to optimize its chosen evaluation metrics while minimizing the associated loss. In time series forecasting, the loss function plays a critical role in both machine learning and deep learning models. It serves as a benchmark for assessing the model's performance, guiding the selection of parameters by driving the minimization of the loss function. A detailed evaluation of loss functions for time series problems has been provided in [48]. Based on this analysis, "Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and Huber Loss" were selected as metrics for this time series regression model. These metrics are chosen for their effectiveness in addressing critical aspects of time series forecasting. MSE and RMSE effectively handle large errors by penalizing them more heavily, which is crucial for capturing significant deviations in predictions [48]. MAE provides a straightforward interpretation and is less sensitive to outliers, making it robust for datasets with moderate noise [48]. Huber Loss combines the strengths of both MSE and MAE, adapting to varying

data distributions while being robust to outliers [48]. Throughout the training process, these metrics are extensively tested to minimize the loss and optimize model performance. A brief explanation of each metric will follow.

4.2.1.1 Primary Loss: MSE, RMSE, MAE or Huber loss

Mean Absolute Error (MAE) or L1 loss is a straightforward and computationally efficient metric that offers a balanced evaluation of model performance [48]. It is mathematically defined as:

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i| \tag{4.1}$$

However, it employs a linear scoring procedure, where all errors are weighted uniformly in the computation of the mean. This linearity, while simple, can result in the gradient being too steep, potentially causing the optimization process to overshoot the minima during back-propagation [48].

The Mean Squared Error (MSE), also known as L2 quadratic loss, applies a squared penalty to errors and is mathematically defined as:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{4.2}$$

By squaring the errors, MSE assigns greater weight to outliers, which can significantly influence the model's performance. This property creates a smoother gradient for smaller errors, allowing optimization algorithms to effectively minimize the loss and achieve optimal parameter values [48]. As the squared error ensures non-negativity, the MSE value always ranges between 0 and infinity, growing exponentially with increasing errors [48]. While this loss has been preferred in most of the experiments during training, its sensitivity to outliers required careful consideration when working with noisy dataset like waypoints_carla_town4.

RMSE, computed from the square root of the MSE, measures the average magnitude of errors [48]. It is applied across various features, primarily to assess whether the increase in features improved the model predictions. It is mathematically defined as:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2} \tag{4.3}$$

Huber loss offers a seamless combination of quadratic and linear scoring, making it a robust choice for many regression tasks [48]. It incorporates a hyperparameter, δ, which determines the transition point between quadratic and linear behavior [48]. For errors smaller than δ, the loss follows a quadratic form, while for errors larger than δ, it transitions to a linear form [48]. Mathematically, it is expressed as:

$$L_\delta = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{if } |y - \hat{y}| < \delta, \\ \delta\left(|y - \hat{y}| - \frac{1}{2}\delta\right) & \text{otherwise.} \end{cases} \tag{4.4}$$

This approach seamlessly integrates the advantages of Mean Squared Error (MSE) for handling smaller deviations and Mean Absolute Error (MAE) for managing larger discrepancies, ensuring robustness to outliers while maintaining sensitivity to smaller deviations. However, it has been observed that this loss becomes computationally demanding in transformer models due to its conditional logic and the large size of datasets.

The primary loss ($\mathcal{L}_1$) is chosen by experimenting with one of the four loss functions mentioned above.

4.2.1.2 Curvature Loss and Polynomial Coefficients Loss

During training, results were visualized after each experiment to evaluate how well the predicted lane curves aligned with the target data. It was observed that even when primary losses were minimized, the model struggled to generalize effectively on steep curves. This limitation arose because primary losses primarily focus on minimizing the magnitude of errors without considering curvature. To address this issue, curvature loss was introduced, which is calculated using the standard parametric equation:

$$\mathcal{L}_2 = \frac{|x'(t) \cdot y''(t) - x''(t) \cdot y'(t)|}{\left((x'(t))^2 + (y'(t))^2\right)^{3/2}} \tag{4.5}$$

In addition to primary loss and curvature loss, a third loss, known as the polynomial coefficient loss, was also considered during experimentation. This involves fitting a polynomial to both the predicted and target curves, extracting their corresponding polynomial coefficients, and calculating mean difference between them. The number of coefficients depends on the degree of the polynomial used for curve fitting.

These three loss components—primary loss, curvature loss, and polynomial coefficient loss—were combined using weighting factors to create a composite loss function. The overall loss is represented as:

$$\mathcal{L} = \omega_1 \cdot \mathcal{L}_1 + \omega_2 \cdot \mathcal{L}_2 + \omega_3 \cdot \mathcal{L}_3 \tag{4.6}$$

where ω_1, ω_2, and ω_3 are weighting factors that adjust the priority assigned to each loss during training. By fine-tuning these factors, the model's performance is effectively optimized across various scenarios.

4.2.2 Handling Missing Values During Training

In this step, masking is applied to the padded values to ensure the model focuses exclusively on valid sequential data while ignoring invalid entries. During normalization, missing values are padded with 1300 (outside the pixel range of 1280 x 720) to clearly distinguish them from valid data points. Following this, Gaussian noise is added to the normalized coordinates, altering the coordinates by a controlled amount. As a result, these augmented values can now be effectively identified and masked using the criterion of values >1280.

To handle these invalid values during training, two methods are employed:

- **Masking During Loss Calculation:**
  In this approach, the mask is applied directly during the loss calculation after the model processes the data. This allows the model to train on all input values, including the invalid ones. However, during the loss calculation, these values are masked, ensuring that the optimizer disregards incorrect predictions for these values and updates the model weights based only on the loss from valid values. This method proved highly effective for sequential models such as RNNs, LSTMs, GRUs, and their variants, as these models inherently capture temporal dependencies.

- **Masking During Training for Transformer Models:**
  Transformer models and their variants, which distribute attention across all elements in the dataset, were less effective with the first approach. To address this, masking is applied during training itself, ensuring the model attends only to valid sequences. This is achieved using the `src_key_padding_mask`[49], which will be explained in the subsequent section.

4.2.2.1 Masked Self-Attention in Transformer Models

Transformer models utilize two primary masking techniques: src_mask and src_key_padding_mask [49]. Although both are designed to control the attention mechanism, they serve distinct purposes and are applied in different scenarios. The src_mask is used to define token-to-token attention patterns within a sequence [49]. It operates at the sequence level and is typically implemented as a square matrix, where each entry indicates whether one token can attend to another [49]. For example, in autoregressive tasks like text generation, src_mask enforces causality by allowing each token to attend only to itself and the tokens that precede it, ensuring that future information is not used during prediction [49].

In contrast, the src_key_padding_mask is used to handle sequences of varying lengths by ignoring padding tokens during attention computation [49]. It operates at the batch level and is structured as a binary matrix where each entry specifies whether a token is padding (`True`) or valid (`False`) [49]. This ensures that padded positions, which contain no meaningful information, do not contribute to the attention mechanism.

In summary, while the src_mask defines pairwise relationships within the sequence, the src_key_padding_mask focuses on excluding irrelevant padded tokens across all sequences in a batch.

4.2.2.2 Application of src key padding mask during training

In this problem, the src key padding mask is applied to exclude the padded values. This mask operates at the batch level in a format of [batch_size, seq_length]. For instance, if we consider a tensor with the shape (1, 9, 128), a binary mask of shape (1, 9) is applied. The accompanying figure 4.1 illustrates this clearly, showing an example where the 2nd and 3rd frames out of 9 contain missing lanes (indicated in red).

It should be noted that as the 2D binary mask is applied to the 3D tensor, it ignores entire frame from the sequence rather than specific invalid lanes within a frame. This is because the concatenation of 4 lanes, each with 32 features, forms a single array with 128 features. However, the model does not entirely discard the sample during training.

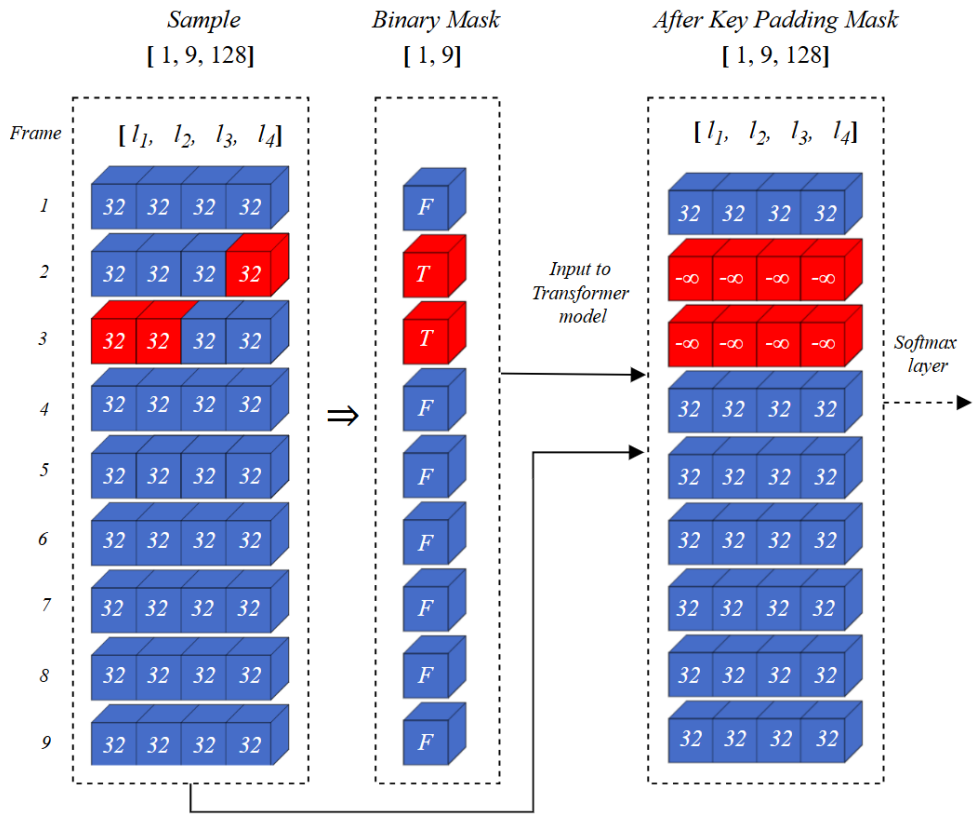


Figure 4.1: Application of src key padding mask during training

Instead, it processes the sample as having 9 frames but ensures that the 2nd and 3rd frames do not influence computations during the self-attention mechanism.

The self-attention mechanism calculates compatibility or attention scores between all frames in the sequence. These scores determine how much each frame or token should attend to others in the sequence. The src_key_padding_mask assigns a large negative value (e.g., $-\infty$) to the attention scores corresponding to the 2nd and 3rd frames. When the softmax function is applied to the attention scores, the scores for the 2nd and 3rd frames become effectively zero, ensuring these positions do not contribute to the attention weights.

For each token, the attention weights associated with the 2nd and 3rd elements are ignored, meaning their information does not propagate through the self-attention computation. The model processes the sequence as if only the valid frames at the 1st, 4th, 5th, ..., 9th positions exist, effectively skipping the padded elements.

This methodology is followed during training; however, during inference, a slight change is made to the binary mask. This adjustment will be explained in the next section.

4.2.2.3 Lane-Aware Masking during Inference

This masking technique is applied only during inference, where priority is given to the ego lanes while using the key padding mask. As discussed, the previously mentioned masking technique ignores an entire frame in the sequence even if a missing lane exists in just one lane sequence l^N within the sample. This approach works well during training as it helps the model learn effectively from complete data.

During feature engineering, forward filling is performed to address missing lanes. If missing lanes persist in the training data after this process, it is most likely a scenario where either the 1st and last lanes in the sequence are both missing, or the entire lane sequence is empty. In such cases, it is preferable for the model to skip these scenarios during training to avoid introducing noise.

However, during inference, if the model ignores an entire frame, it can still predict ego lanes using the available frames in the sequence. On the other hand, if only non-ego lanes are empty, this should not affect the prediction results for ego lanes. Furthermore, if the dataset contains only ego lanes with non-ego lanes consistently absent, the model should focus on predicting values for the ego lanes alone.

To address these issues, a lane-aware masking technique is introduced. As shown in the figure 4.2, unlike the previous approach, the binary mask values are set to true only if missing lanes occur in the ego lanes; otherwise, they remain false.

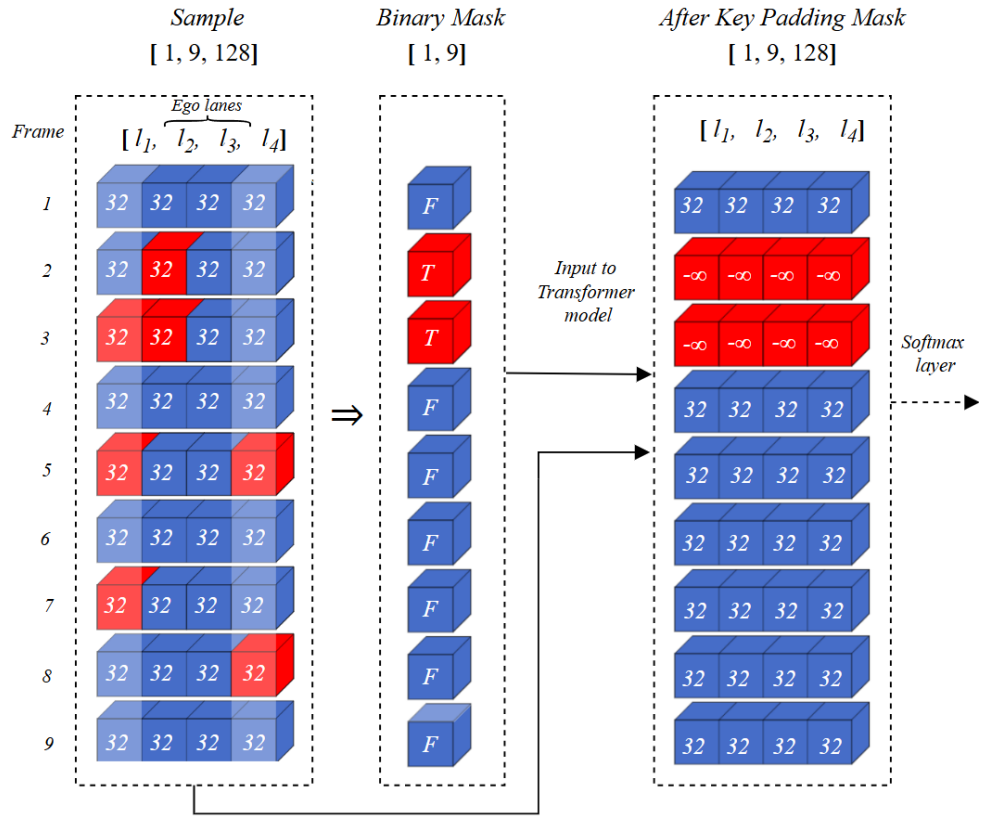


Figure 4.2: Lane-Aware Masking using src key padding mask

Lane-aware masking is not used during model training because it could adversely affect the prediction performance for non-ego lanes. This is because the model would become less robust to irregularities in non-ego lanes, as training would lack exposure to full-frame inconsistencies due to data limitations.

4.2.2.4 Bayesian Hyperparameter Optimization

After setting up training loops for models RNN, LSTM, GRU, and Transformers, Hyperparameter Optimization (HPO) is performed to fine-tune model settings for optimal

performance [50]. HPO can be manual, relying on domain expertise, or automated using methods like Bayesian Optimization or Grid Search [50]. For this task, Bayesian Optimization is chosen for its efficiency. Unlike Random and Grid Search, which test numerous configurations randomly, Bayesian Optimization uses past evaluations to guide future searches, ensuring faster convergence. The process begins by defining the search space, encompassing all possible hyperparameter combinations [42], to systematically explore and identify the best configuration. This involves setting up parameter ranges for standard settings (e.g., number of hidden layers, activation functions, dropout, learning rate, optimizer) as well as Transformer-specific parameters (e.g., d_model, n_heads, dff) and problem-specific parameters like composite loss weighting, batch size, and Gaussian noise standard deviation.

4.3 Inference in CARLA Simulation

This section outlines the methodology employed to deploy and evaluate the trained Virtual Trajectory Generation (VTG) model within the CARLA simulation environment, a platform designed to emulate real-world driving scenarios. The implemented inference pipeline is illustrated in Figure 4.3. The highlighted VTG Module represents the contributions of this thesis, seamlessly integrated into the overall framework. This pipeline structured to process real-time data, integrating various stages such as data acquisition, model execution, and reference path computation. It begins with the collection of sequential lane data, followed by pre-processing and structuring of the inputs to ensure compatibility with the VTG model. The model, leveraging its trained capabilities, predicts future lane positions based on past observations. The predicted lane data is then processed to compute a reference path, which serves as the desired trajectory for the vehicle. This reference path is subsequently fed into a control system for path-following and motion planning.

The ultimate objective of this pipeline is to evaluate the VTG model's effectiveness in generating robust trajectories under varying conditions, including scenarios with missing lanes, noise, or sensor inaccuracies. By embedding this workflow in CARLA's simulation environment, it becomes possible to rigorously test the model's performance in controlled yet realistic conditions.

4.3.1 Data Generating Process for Real-Time Inference

The process begins with the data acquisition pipeline. CARLA's simulation environment is configured to load specific maps, such as Town04. Sensors, including RGB, depth, and semantic segmentation cameras, along with IMU and GNSS modules, are mounted on a Tesla Model 3 vehicle to simulate a comprehensive perception system. Each frame captured by the Ultra Fast Lane Detection module undergoes preprocessing to extract lane coordinates. The extracted data is cleaned, normalized, and structured into a consistent format using the same data preparation steps employed during training to form input samples for the VTG pipeline. For each frame iteration, nine consecutive frames are aggregated to create a sample of size (1,9,128), representing the batch size, sequence length, and features. This sliding window approach ensures a consistent input size while maintaining temporal dependencies in the data.

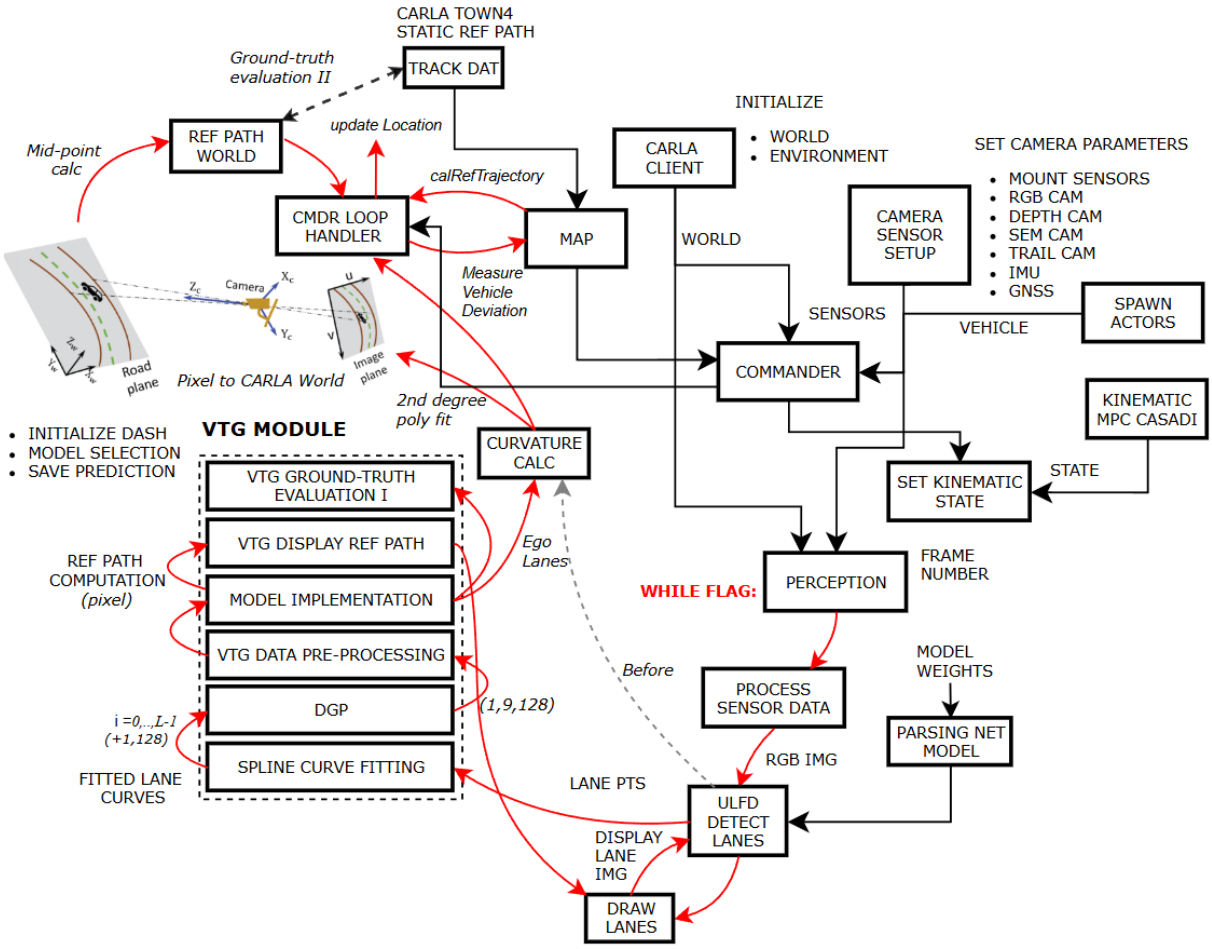


Figure 4.3: Overview of Implemented Pipeline

4.3.2 Deployment of Trained Model

The trained VTG model is integrated into the inference pipeline to predict lane coordinates for the next T frames, where T represents the forecast horizon. The VTG module, embedded within the CARLA environment, seamlessly processes each sample to produce frame-wise predictions. The model leverages a sequence-to-sequence architecture to process temporal data. During deployment, lane-aware masking, as previously discussed, ensures robust handling of missing or incomplete lane sequences. The predicted output retains the same structural format as the input data, facilitating its integration into subsequent steps for reference path computation, ground-truth evaluation, and visualization.

4.3.3 Computation of Reference Path from Predicted Output

The reference path is computed in both pixel coordinates and CARLA world coordinates. In the existing framework from [1], the reference path is generated by extracting the ego lanes from the detected lane coordinates and then calculating the mean of the left and right lane points. A second-degree polynomial is fitted over this center line, which, along with its curvature, is directly fed into the controller as input for path-following.

The implementation pipeline in this research adopts a similar approach, replacing ego lanes derived from real-time detection with those predicted by the VTG model. The

VTG model's output maintains the same shape as the input sample (1, 9, 128). From this output, the last frame (9, 128), representing the latest predicted frame $Z(t+T)$), is extracted. To compute the reference path in world coordinates, the fitted mid-trajectory in pixel coordinates is transformed into CARLA world coordinates using the camera transform from rgb_cam mounted on the vehicle. This trajectory in world coordinates is then provided to the controller for lane-keeping. For visualizing the reference path in pixel coordinates, polynomial curve fitting is applied to the midpoints and the fitted curve is displayed in the Open CV window.

4.4 Evaluation of Methods

Ground truth evaluation involves quantifying and visualizing the deviation between the model's predicted outputs and the actual observed values (ground truth). For each iteration, the prediction for the current frame $X(t)$ is saved and after T frames it is compared with the recent ongoing frame $X(t+T)$, where T is the forecast horizon. For each comparison, the predictions and ground truth are aligned, and their discrepancies are analyzed using a set of metrics. The evaluation is focuses on three primary objectives:

- **Quantification of Errors:** Metrics like Mean Absolute Error (MAE) and Euclidean Distance measure point-wise discrepancies.
- **Curve Characteristics:** Metrics such as Curvature Loss and Fréchet Distance evaluate the shape and path similarity of the predicted and actual curves.
- **Visualization:** A comprehensive visual representation, including bar charts and annotated plots, is employed to assess errors at each point and overall similarity.

4.4.1 Evaluation Metrics

Trajectory datasets are a specialized form of multivariate time series data. To effectively quantify errors in trajectories, this study bases the selection of evaluation metrics on insights from [51], which provides a comparative analysis of trajectory similarity measures. The key challenge lies in choosing a metric that can assess the similarity between two curves by considering both their absolute magnitude and overall shape. For spatiotemporal datasets, the Fréchet distance stands out as it accounts for both these aspects, making it the primary criterion for evaluating similarity in this study.

4.4.1.1 Fréchet Distance

The Fréchet Distance evaluates the alignment between two trajectories by considering continuous correspondences, ensuring that all points along both trajectories are accounted for in the measurement [51]. The Fréchet Distance between two trajectories A and B is defined as:

$$F(A,B) = \inf_{\sigma} \max_{t\in[s_1,s_n]} \text{dist}(A(t), B(\sigma(t))), \tag{4.7}$$

where the infimum is computed over all continuous, strictly monotonic functions $\sigma : [s_1, s_n] \to [t_1, t_m]$. This formulation ensures a mapping that preserves the order and continuity of points along both trajectories, effectively capturing their alignment in both shape and scale.

4.4.1.2 Hausdorff Distance

The Hausdorff distance between two curves A and B is defined as the smallest δ such that every point on A lies within a δ-neighborhood of B, and vice versa [52]. It is given by:

$$H(A, B) = \max \left\{ \sup_{a \in A} \inf_{b \in B} d(a, b), \sup_{b \in B} \inf_{a \in A} d(a, b) \right\}, \tag{4.8}$$

where:

- $d(a, b)$ is the distance between the points $a \in A$ and $b \in B$,
- sup represents the supremum (maximum value),
- inf represents the infimum (minimum value).

While this metric is simple and intuitive for measuring the proximity of compact sets, it has notable drawbacks. Specifically, it does not account for the continuity or shape of the curves [52]. Consequently, there are cases where the Hausdorff distance is small even though the curves are significantly different in structure [52]. This limitation arises because the correspondence between points on P and their closest counterparts on Q may lack continuity.

To overcome these shortcomings, the Fréchet distance introduces a more nuanced approach by considering the continuous mapping of points along the curves. A widely used analogy compares this to a man walking his dog: the man moves along one curve while the dog follows the other [52]. Both can adjust their speeds independently but cannot retrace their steps. The Fréchet distance is then the shortest leash length required to keep the man and dog connected as they traverse their respective paths [52].

4.4.1.3 Similarity Score

The similarity score translates the Fréchet Distance into a percentage score, where lower Fréchet values correspond to higher similarity. The ranges used are based on experimental analysis of the average pixel deviation per frame, ensuring that the similarity score provides an accurate and intuitive representation of prediction accuracy across varying trajectories. The frechet distance ranges and corresponding similarity scores are defined as follows:

- $0 \leq d_F \leq 13$: $100 - \frac{10}{13} \cdot d_F$ (Similarity decreases linearly from 100% to 90%).
- $13 < d_F \leq 25$: $90 - \frac{10}{12} \cdot (d_F - 13)$ (Similarity decreases linearly from 90% to 80%).
- $25 < d_F \leq 50$: $80 - \frac{20}{25} \cdot (d_F - 25)$ (Similarity decreases linearly from 80% to 60%).
- $50 < d_F \leq 70$: $60 - \frac{30}{20} \cdot (d_F - 50)$ (Similarity decreases linearly from 60% to 30%).
- $70 < d_F \leq 100$: $30 - \frac{20}{30} \cdot (d_F - 70)$ (Similarity decreases linearly from 30% to 10%).
- $100 < d_F \leq 250$: $10 - \frac{0.1}{1} \cdot (d_F - 100)$ (Similarity decreases linearly from 10% to 0%).
- $d_F > 250$: 0% (No similarity).

In addition to these metrics, the Mean Absolute Difference (MAD) is calculated between corresponding points on the curves. X-Errors and Y-Errors capture deviations along the individual coordinate axes, offering insights into directional discrepancies. To quantify variability in the errors, the Standard Deviation is employed, highlighting inconsistencies in the predictions.

Furthermore, Curvature Loss is computed using the parametric equation for second-order derivatives as defined in 4.5, providing a measure of the curvature accuracy. Lastly, the Euclidean Distance consolidates point-wise errors into a single metric, representing the overall spatial deviation between the trajectories.

4.5 Results

This section examines the experiments conducted during model training, emphasizing the evaluation of model performance, validation against ground truth in offline conditions, and subsequent testing in real-time simulations using the CARLA environment. The evaluation process begins with identifying the best-performing models based on training and validation loss trends, guided by extensive experimentation with hyperparameter tuning, forecast horizons, batch sizes, and noise augmentation. Models exhibiting promising performance are selected for further testing in both offline and real-time simulation scenarios.

The offline evaluation is conducted over test data from the seven datasets discussed earlier, offering a controlled environment to measure prediction accuracy. Following this, the two best-performing models are deployed in the CARLA simulation for real-time testing. Predictions are generated for each frame and compared against ground truth data using the evaluation metrics outlined in 4.4.1. These results are visualized in real-time through a dashboard, providing insights into the models' behavior under challenging conditions, such as missing data, noise, and outliers. From the predicted results, the reference path for ego lanes is computed and validated against the ideal trajectory or map from CARLA Town4. The findings from these experiments demonstrate the practical viability and robustness of the proposed models in generating accurate local reference paths under challenging conditions.

4.5.1 Overview of Experiments from TensorBoard

The analysis of training and validation losses is conducted for various model architectures, ensuring a fair comparison by limiting the evaluation to the first 100 epochs. This approach provides a reliable benchmark to compare model performance under different configurations, as highlighted in the attached graphs 4.4. These graphs focus on a subset of experiments with the lowest loss values, showcasing models that excel in convergence, generalization, and robustness. The experiments span architectures such as Seq2Seq Transformers, GRU, LSTM, TCN, and TFT, each tested with variations in forecast horizons, activation functions, batch size and hyperparameter configurations.

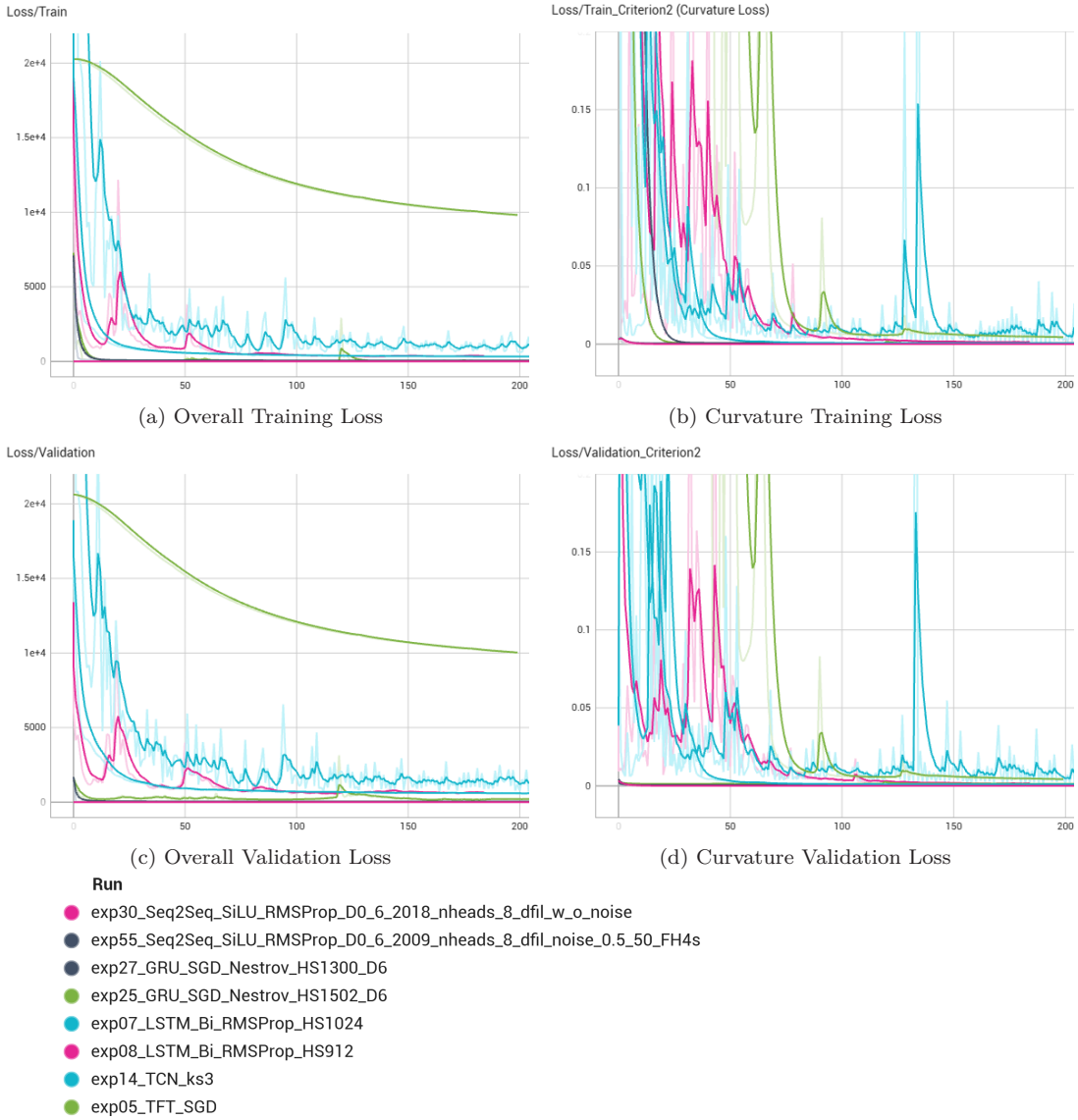
Table 4.3: Model Training Results (Sorted by Minimum Overall Validation Loss)

Sr. No.	Experiments	Step	Min Train (overall)	Min↓ Val (overall)	Min Train (curvature)	Min Val (curvature)	Δ_{val} (overall) (%)
1	exp30-Seq2Seq-SiLU-RMSProp-D0-6-2018-nheads-8-dfil-w-o-noise	99	0.0004	0.0006	0	0.0000	99.99%
2	exp46-Seq2Seq-SiLU-RMSProp-D0-6-nheads-8-dfil-noise-0.8-40	99	0.1578	0.0289	0.0006	0.0001	95.05%
3	exp49-Seq2Seq-SiLU-RMSProp-D0-6-2018-nheads-8-dfil-noise-0.5-50	99	0.0918	0.0305	0.0004	0.0002	94.69%
4	exp52-Seq2Seq-SiLU-RMSProp-D0-6-2018-nheads-8-dfil-w-o-noise-FH4s	99	0.0048	0.0308	0.0001	0.0001	94.79%
5	**exp55-Seq2Seq-SiLU-RMSProp-D0-6-2009-nheads-8-dfil-noise-0.5-50-FH4s**	99	**0.0921**	**0.0622**	**0.0004**	**0.0002**	**93.54%**
6	exp35-Seq2Seq-SiLU-RMSProp-D0-6-nheads-8-dfil-noise-1-30	99	0.1776	1.2524	0.0005	0.0019	86.23%
7	exp27-GRU-SGD-Nestrov-HS1300-D6	99	69.7026	46.0293	0.0008	0.0006	81.12%
8	exp25-GRU-SGD-Nestrov-HS1502-D6	99	34.0857	179.3245	0.0007	0.0014	64.56%
9	exp08-LSTM-Bi-RMSProp-HS912	99	443.2938	652.3025	0.0037	0.0039	53.21%
10	exp26-GRU-SGD-Nestrov-HS1696-D0-1-6	99	524.4475	661.9496	0.0008	0.0007	47.68%
11	exp07-LSTM-Bi-RMSProp-HS1024	99	420.7885	709.3739	0.0008	0.0012	35.39%
12	exp31-GRU-SGD-Nestrov-1300-D0-6-dfil	99	334.2537	735.3599	0.0005	0.0008	28.76%
13	exp20-GRU-SGD-Nestrov-HS1502-D0-1-6	99	244.828	762.9416	0.0006	0.0009	20.87%
14	exp10-LSTM-Bi-RMSProp-HS1068	99	350.0713	906.5952	0.0011	0.0018	18.65%
15	exp11-LSTM-Bi-RMSProp-HS1068-except-D5	99	340.1356	939.5213	0.0008	0.0018	14.72%
16	exp13-LSTM-Bi-RMSProp-HS1496-prelu	99	381.2461	939.6773	0.0043	0.003	12.55%
17	exp12-LSTM-Bi-RMSProp-HS1024-except-D5	99	393.9151	985.6178	0.005	0.0048	8.85%
18	exp17-LSTM-Bi-RMSProp-HS2000-D0-1-6	99	310.7946	1,018.1259	0.0008	0.0062	6.34%
19	exp14-TCN-ks3	99	766.8837	1,062.0108	0.0059	0.0063	4.76%
20	exp32-LSTM-Bi-RMSProp-781-D0-6-dfil	99	518.4156	1,369.5801	0.001	0.0013	3.12%
21	exp21-GRU-SGD-Nestrov-HS1502-D6-7	99	549.1744	1,460.2063	0.0005	0.0007	1.43%
22	exp03-Seq2Seq-SiLU	99	2,677.9498	1,767.7174	0.0007	0.0006	1.00%
23	exp19-TCN-ks7-D0-1-6	99	1,276.4829	2,792.9639	0.0095	0.0087	0.89%
24	exp02-GRU-HS1024	99	1,474.0953	3,663.2009	0.0019	0.002	0.75%
25	exp-01-LSTM-Bi-HS583	99	1,301.1844	5,125.5542	0.0018	0.0015	0.58%
26	exp05-TFT-SGD	99	11,953.3038	12,141.125	0.0087	0.0084	0.47%
27	exp04-TFT-ReLU	99	20,224.008	21,334.4711	0.306	0.3062	0.12%

The graphs 4.4 reveal a significant reduction in loss for most experiments during the initial epochs. This indicates successful convergence across various configurations. The graphs presented have been smoothed using a magnitude of 0.7 in TensorBoard, helping to reduce noise and better visualize trends. Models with higher complexity (e.g., Seq2Seq Transformers or GRU Models with larger hidden size) generally outperform simpler architectures like LSTMs and other RNN variants. Experiments with filtered data using feature engineering (named with dfil) consistently demonstrate faster convergence and lower final losses compared to those without filtering. Models with noise augmentation perform slightly worse compared to their counterparts trained without noise but are expected to be more robust in real-world scenarios.

The Seq2Seq Transformer models, characterized by their encoder-decoder structure and multi-head attention mechanisms, consistently outperform other architectures in terms of training and validation losses. This superior performance is evident in experiments such as Exp55, Exp30, Exp52, and Exp49, which achieve some of the lowest minimum loss values. The inclusion of advanced activation functions such as SiLU, MisH and PReLU further enhances the performance of these models by improving gradient flow and non-linearity during optimization.

One of the critical factors influencing model performance is the forecast horizon (FH). Most experiments utilize a default forecast horizon of 1 second, which simplifies the



(a) Overall Training Loss

(b) Curvature Training Loss

(c) Overall Validation Loss

(d) Curvature Validation Loss

Figure 4.4: Training and Validation loss

prediction task and contributes to lower losses. However, certain experiments, such as Exp55, stand out for their ability to maintain competitive loss values with a forecast horizon of 4 seconds. The rationale for selecting Exp55 as the best-performing model becomes clear when compared to similar experiments such as Exp30, Exp52, and Exp49. While these models exhibit lower losses than Exp55, they are limited by either their shorter forecast horizons or the absence of noise augmentation. Exp55, on the other hand, achieves exceptional accuracy even with a forecast horizon of 4 seconds and Gaussian noise added to 50% of the data.

In summary, architectural differences significantly influence model performance. Seq2Seq Transformers dominate the results due to their ability to effectively capture complex temporal relationships through their attention mechanisms and encoder-decoder design. In contrast, simpler architectures like GRU and LSTM struggle to match the performance of Transformers, particularly under noisy conditions or extended forecast horizons.

4.5.2 Best Model Results

In this thesis, Seq2Seq Transformer model 4.5 with an encoder-decoder structure, proposed in [9], was found to deliver the best performance with the highest accuracy. This foundational design was adopted and further refined to address specific challenges such as target data leakage [7] and over-fitting. Unlike standard Seq2Seq architectures, which often rely on the teacher forcing mechanism [53], this model intentionally avoids using target data as input during training to prevent data leakage. Teacher forcing [53], while beneficial in many scenarios, introduces a risk of target leakage, where future information might influence the model predictions. This approach is avoided as it compromises generalization, especially in time series tasks, where predictions must rely strictly on past and present data [7].

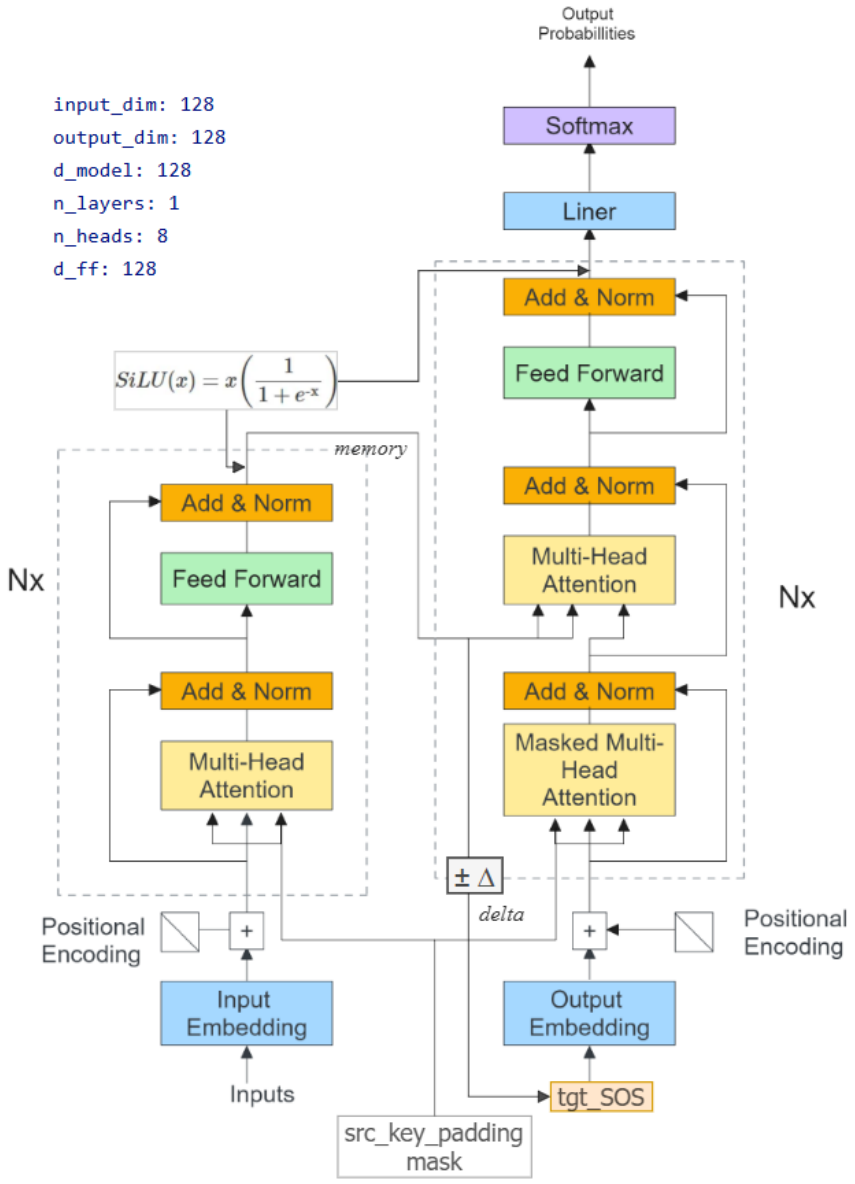


Figure 4.5: Seq-2-Seq Transformer Model with custom modifications

As an alternative, the target input to the decoder is replaced with a delta-based mechanism.

The encoder processes sequential input, which is then adjusted using a delta value to initialize the decoder tgt_SOS. Since x-coordinate values range from -640 to 640, the delta is dynamically added or subtracted based on the sign of the input coordinates. This ensures that the tgt_SOS is initialized in the correct direction.

Both the encoder and decoder layers integrate Scaled Dot-Product Attention and Multi-Head Attention to capture contextual dependencies effectively [9]. The SiLU (Sigmoid Linear Unit) offers smoother gradient flow and improved stability compared to traditional ReLU [54]. The encoder layers comprise "multi-head self-attention, residual connections, layer normalization, and feed-forward networks". The decoder mirrors this structure while including multi-head attention mechanism twice.

4.5.2.1 Model Parameters and Training Configuration

The table below 4.4 presents the optimal configuration and parameters for the custom Seq-2-seq Trasformer Model:

Table 4.4: Model Training Parameters

Parameter	**Value**
Batch Size Percentages	'label_0313_1': 40% 'label_0313_2': 40% 'label_trippstadthin': 100% 'label_trippstadtruck': 100% 'label_trippstadterstrhin': 100% 'label_driver23_30frame': 5% 'label_way_pts_carla_town04': 50
Number of Epochs	100
Input Dimension	128
Output Dimension	128
Model Dimension (d_model)	128
Number of Layers	1
Number of Heads	8
Feed-Forward Dimension (d_ff)	128
Dropout Rate	0
Learning Rate	0.006747233616890091
Optimizer Parameters	RMSprop (Parameter Group 0 - alpha: 0.99 - centered: False - eps: 0.005774495743944666 - lr: 0.006747233616890091 - momentum: 0.5 - weight_decay: 1×10^{-35})
Loss Criterion 1	MSELoss()

Loss Criterion 2	MAE Curvature Loss
Loss Criterion 3	Polynomial Coefficients Loss (FALSE)
Loss function Weighting factors (W1, W2, W3)	0.5, 0.5, 0
Device	CUDA
Model Architecture	VTG_SegmentPredictionModel Encoder: - MultiheadAttention - LayerNorm - Feed-forward with SiLU Decoder: - MultiheadAttention (Self and Src) - LayerNorm - Feed-forward with SiLU
Min Validation Loss	0.061983734369277954
Max Validation Loss	6.167443752288818
Min Training Loss	0.0920151025056839
Max Training Loss	6.291810989379883
Min Training Loss at Epoch	100
Min Validation Loss at Epoch	100
Total Elapsed Time	1767.33 seconds
Total Number of Model Parameters	265,472

4.5.3 Ground-Truth Evaluation Results and Summary

The evaluation of ground truth data involves two primary tasks:

- **Model Performance Evaluation:** Assessing the performance of the best models from model training against the ground truth data. This includes both offline and online evaluations of the model.
- **Reference Path Validation:** Validating the generated reference path. This includes converting it into CARLA world coordinates, and comparing it with the ideal static trajectory.

4.5.3.1 Model Performance Evaluation

Offline evaluation refers to validating the trained model against the test data from each of the seven datasets before simulation or implementation in CARLA. During training, in most experiments, the models were trained on less than 50% of the available data due to hardware limitations, architectural restrictions, or the large size of the data resulting from time delay embedding. Consequently, at least 50% of the data was still available for testing purposes, ensuring comprehensive evaluation. This complete data was utilized during the offline evaluation. After training, de-normalization was performed on both the predictions and the target data using the origin values saved during the data preparation phase. The predicted values were then compared against the target data for each dataset.

Examples from the evaluation of two of the seven datasets are shown in the figures 4.6 and 4.7. These include two types of plots: the first type depicts the mean error values for each frame, plotted over the number of frames. As each frame contains a maximum of four lanes, the error values of these lanes were averaged for each frame and plotted against the frame numbers. The four lines on the plots represent different forecast horizons (FH). The second type of plot shows lane-wise errors, where error metrics are presented individually for each lane to enable detailed comparison.

The table provides an overview of the results over other datasets.

Table 4.5: Comparison of datasets across forecast horizons (FH1-FH4)

Dataset	Error Metric	FH1	FH2	FH3	FH4
TU Simple 0313.1	Mean Similarity	98.6702	98.4996	98.3943	98.3532
	Mean MAE	0.7161	0.7028	0.6954	0.6841
	Mean Frechet	1.7287	1.9505	2.0874	2.1408
TU Simple 0313.2	Mean Similarity	98.6710	98.6628	98.6672	98.6710
	Mean MAE	0.6910	0.6940	0.6919	0.6954
	Mean Frechet	1.7277	1.7384	1.8326	1.8326
CARLA_waypts_Town4	Mean Similarity	98.2339	97.6994	97.7422	97.4882
	Mean MAE	0.7492	0.6329	0.8813	1.0617
	Mean Frechet	2.2959	3.2533	2.3351	3.2641
CULane_driver23_30frame	Mean Similarity	96.7901	96.7973	96.8648	96.9232
	Mean MAE	1.3219	1.3205	1.3005	1.2767
	Mean Frechet	4.1728	4.1634	4.0757	3.9997
RPTU_tripstadterstrthin	Mean Similarity	95.7975	95.7974	95.7974	95.7973
	Mean MAE	1.7177	1.7177	1.7177	1.7177
	Mean Frechet	5.4632	5.4632	5.4632	5.4632
RPTU_tripstadthin	Mean Similarity	96.0081	96.0081	96.0081	96.0081
	Mean MAE	1.7161	1.7161	1.7161	1.7161
	Mean Frechet	5.1893	5.1893	5.1893	5.1893
RPTU_tripstadtruck	Mean Similarity	96.0060	96.0060	96.0036	96.4180
	Mean MAE	1.7129	1.7131	1.7136	1.7136
	Mean Frechet	5.1921	5.1921	5.1952	5.1945
Average across all datasets	Mean Similarity	96.8824	96.8109	96.8254	96.9513
	Mean MAE	1.2321	1.1852	1.2165	1.2663
	Mean Frechet	3.5385	3.7744	3.5970	3.7673

In most cases, the similarity scores remain consistently high (>95%) across all frames, indicating strong alignment with the ground truth, and with frechet distances less than 6. From the clean datasets with less noise such as TUSimple and CARLA Town4, it is evident that accuracy decreases as the forecast horizon increases. However, this reduction in accuracy is relatively small, particularly when considering the 4-second forecast horizon. Frechet Distance trends confirm that trajectory similarity decreases with longer horizons, while curvature loss reveals that the model preserves trajectory smoothness better for

shorter horizons. Notably, the curvature loss remains consistently low, demonstrating the model's strong ability to capture curvatures. In both datasets, ego lanes (specifically lane 3 in the first dataset and lane 2 in the second) exhibit better similarity results compared to non-ego lanes.

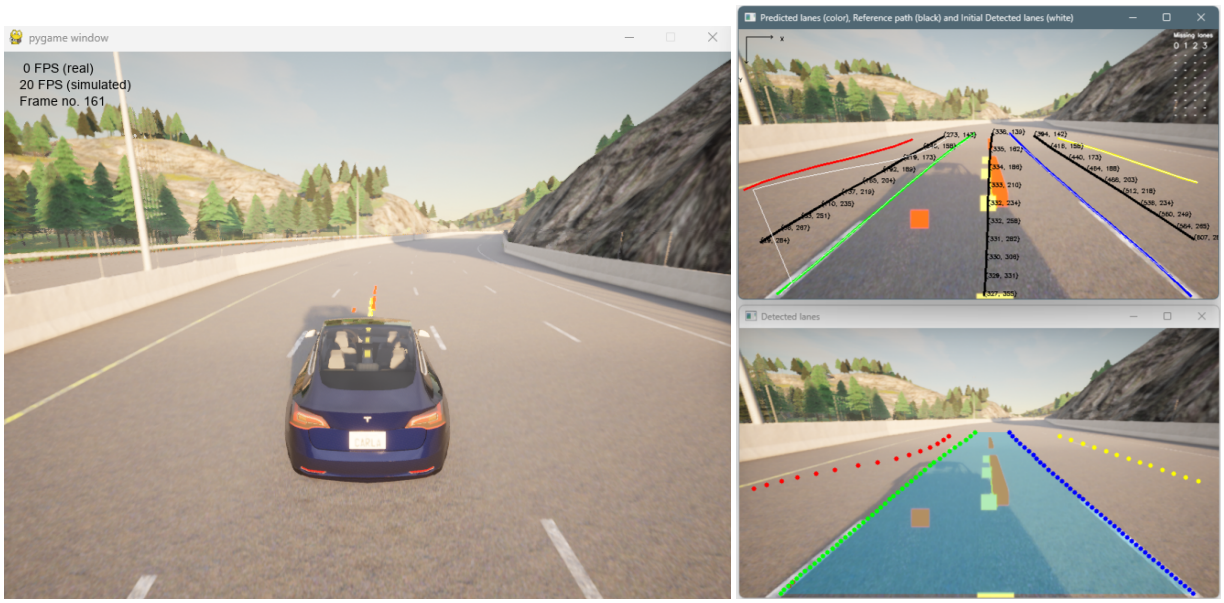
4.5.3.2 Reference Path Validation

The validation of the generated reference path entails a systematic comparison with the ideal static trajectory in the CARLA simulation environment. To accomplish this, the reference path data, originally output image pixels coordinates, is first converted into CARLA world coordinates using rgb_cam trasform, as shown in 4.3. This transformation ensures compatibility with the predefined ideal trajectory, which serves as the benchmark for validation. Following the conversion, each segment of the reference path is then analyzed against the ideal trajectory. To facilitate this comparison, a KDTree [55] from scikit-learn library is employed, offering an efficient way to identify the nearest points on the ideal trajectory for each point in the reference path. This nearest-neighbor search ensures that each segment of the reference path is directly matched to the corresponding nearest segment of the ideal trajectory. Figure 4.8 presents the results of the generated reference paths from the first 3000 runs of the CARLA simulation on the Town 4 map.

The figure 4.8 analyzes the deviation of the generated reference paths from the ideal trajectory using Mean Absolute Error (MAE), Frechet Distance, and Standard Deviation (STD). The low average MAE (0.57 m) and STD (0.13 m) indicate consistent and accurate alignment of the reference paths with the ideal trajectory.The average Frechet Distance of 0.87 m demonstrates good similarity, maintaining over 90% alignment between the reference paths and the ideal trajectory.

4.5.4 Simulation Results from CARLA

The CARLA simulation pipeline allows for real-time visualization of the model's performance during online simulations. In Figure 4.9, the bottom image displays input lane points from the ULFD model, while the top image shows predicted lane points from the VTG model. Both use the same color scheme: red for lane 1, green for lane 2, blue for lane 3, and yellow for lane 4. The top window also includes detected lane points from the ULFD model in white for reference. This figure specifically shows that the outliers in lane 1 are effectively mitigated.



(a) pygame window: Vehicle Model Display (b) Predicted (top) vs Detected (bottom) Lanes

Figure 4.9: Simulation over CARLA Simulator

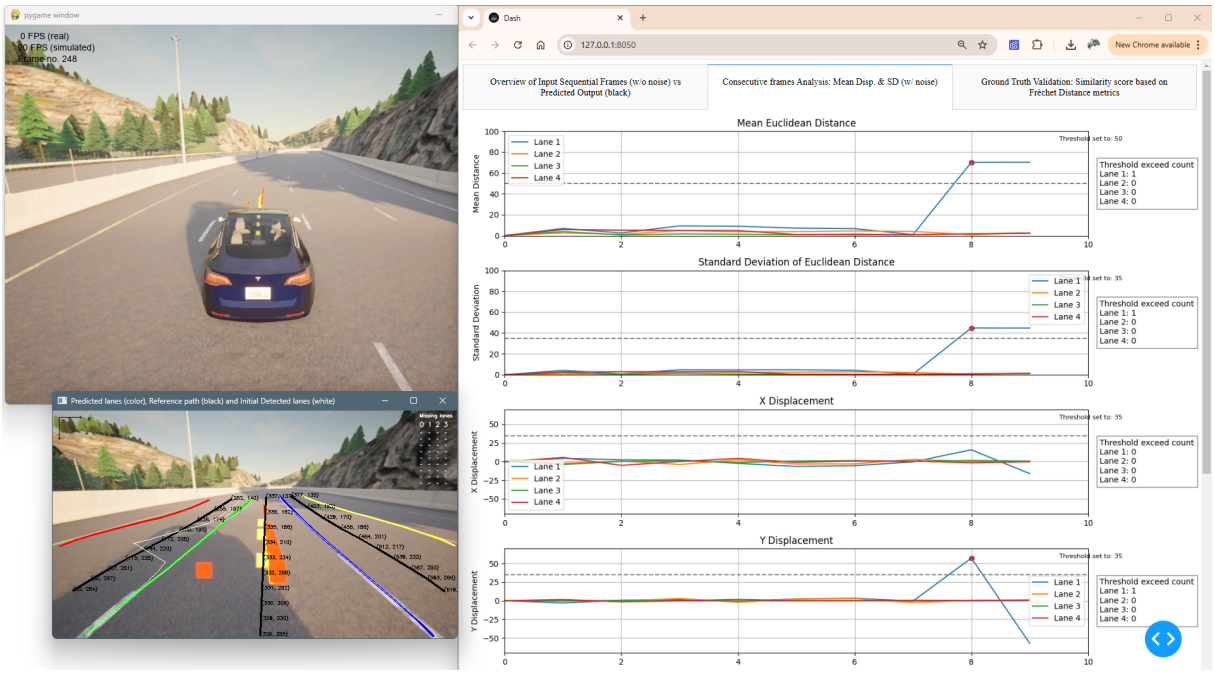


Figure 4.10: Dashboard: Lane-wise Mean Displacement between Consecutive Frames

Figure 4.10 illustrates the lane-wise mean displacement between consecutive frames, quantifying the impact of outliers and missing lanes on predictions and the overall reference path. Figures 4.11 and 4.12 present additional statistical data and ground truth evaluations, performed in real-time and plotted on the dashboard.

Figure 4.6: Results: RPTU_xy_train_tripstadtruck (Batch Size 41,9,128)

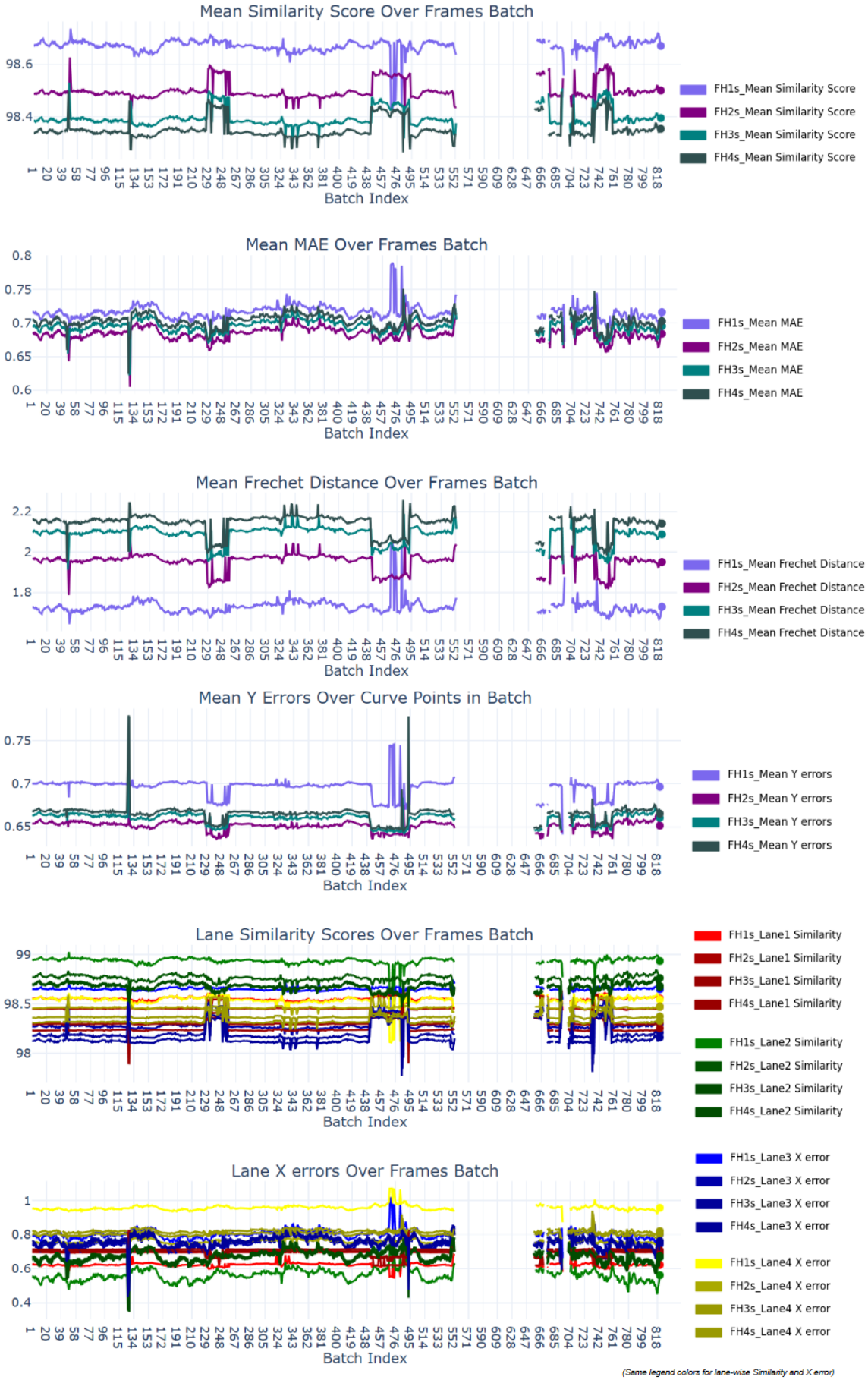


Figure 4.7: Results: TUSimple_0313_1 (Batch Size 820,9,128)

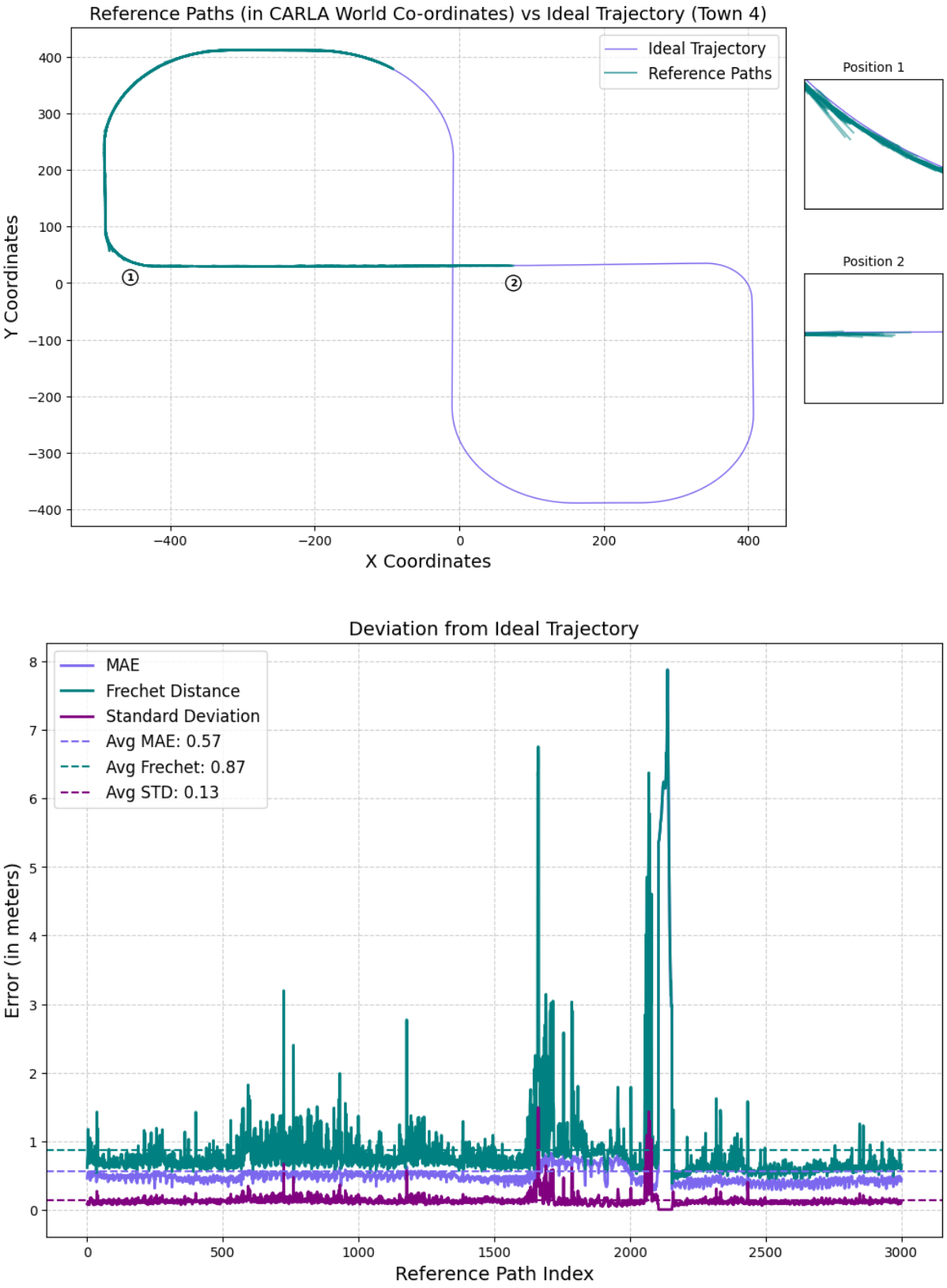


Figure 4.8: Reference Path Validation

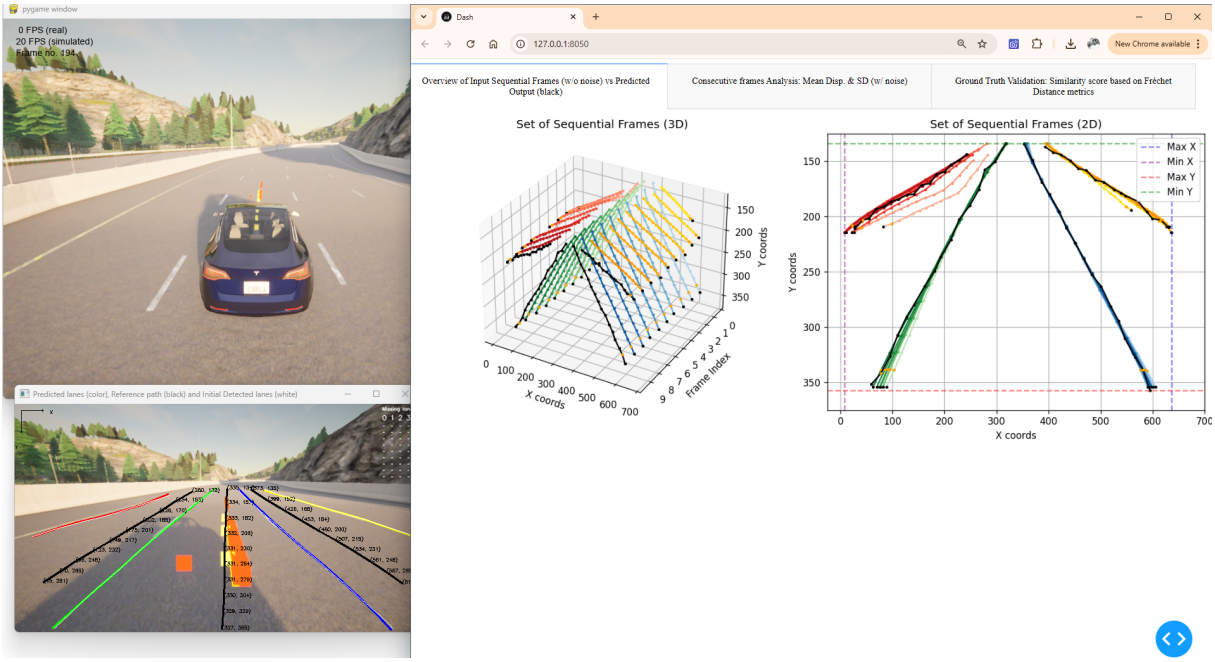


Figure 4.11: Dashboard: Sequential Frames Visualization

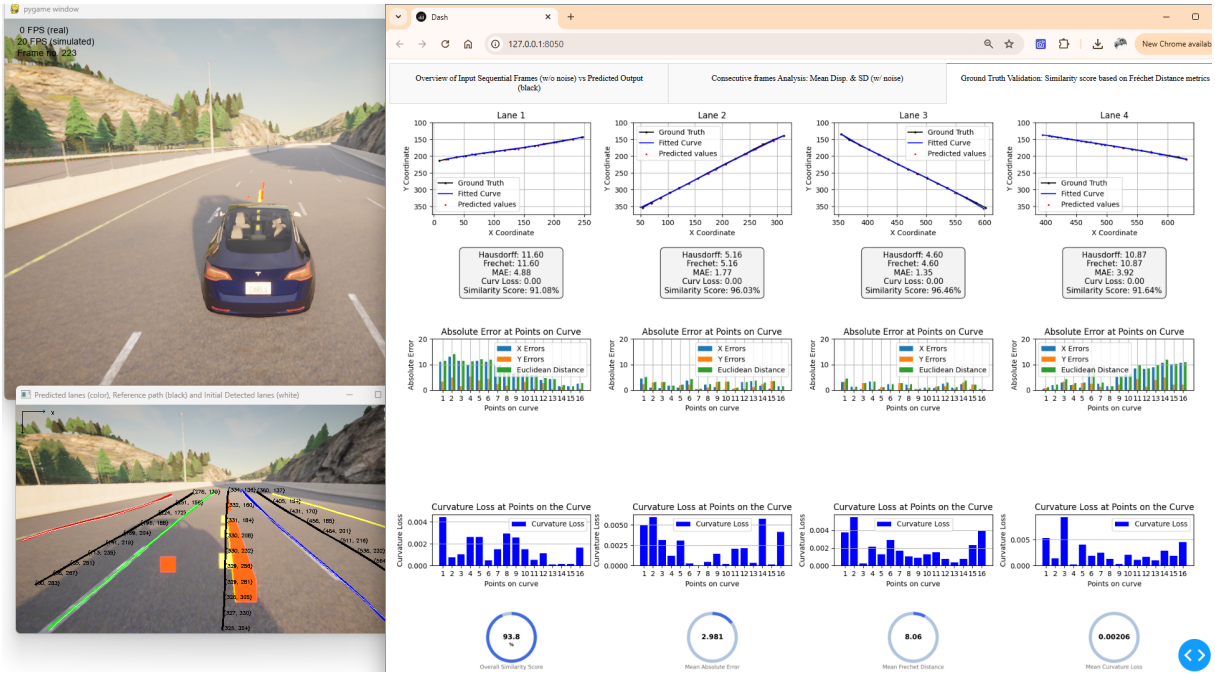


Figure 4.12: Dashboard: Real-time Ground truth evaluation

5. Conclusion and Future Scope

This thesis presents a novel and efficient approach for generating local reference paths in autonomous driving scenarios, addressing challenges of robustness, accuracy, and computational efficiency. The introduction of a temporal regression-based approach ensures accurate and reliable reference paths, even under challenging conditions such as missing lane markers, outliers, and noise. A detailed analysis and comparison of various state-of-the-art DNN architectures were conducted to address the multivariate time-series problem. Among the tested architectures, the Seq2Seq Transformer model outperformed the others, achieving the lowest validation loss (0.062) with a forecast horizon of 4 seconds, demonstrating superior capability in handling temporal dependencies. During offline evaluation, the model consistently maintained an MAE below 1.25 across datasets and forecast horizons, with a Fréchet distance below 6. When integrated into an image-based Model Predictive Control (iMPC) framework, the predicted reference path for ego lanes demonstrated precise and smooth lateral motion control during CARLA simulation, reducing trajectory deviation to an average MAE of 0.57 m and a Fréchet distance of 0.87 m from the ideal trajectory. While accuracy decreased slightly for longer forecast horizons (e.g., 4 seconds), the impact was minimal, with only a 2–3% reduction in the similarity score.

Future work could explore integrating vehicle dynamics, such as velocity and yaw rate, alongside lane detection to enhance the model's robustness in dynamic driving scenarios. Incorporating multimodal sensor data, including LiDAR and radar, could further improve prediction accuracy under adverse weather and lighting conditions. Extending the prediction horizon beyond 4 seconds, while maintaining high accuracy, could also be a focus to address scenarios requiring longer-term planning. Additionally, real-world validation through hardware-in-the-loop simulations or deployment in test vehicles would provide deeper insights into the model's performance and scalability.

As the reference path operates at a local level, it does not consider obstacles or surrounding vehicle conditions, presenting another area for future improvement. Moreover, while lane-aware masking effectively handles missing lanes, excessive distortions and outliers in ego lanes could hinder predictions and, consequently, the lane-keeping system. To address this, further experiments could involve extending the forecast horizon or increasing the

sequence length in the sample size to mitigate noise in scenarios with heavy outliers. The model could also be retrained on a larger dataset while adhering to the same processing pipeline to enhance its robustness. Expanding the dataset would enable the model to generalize better across diverse driving scenarios, including varying road conditions, lane geometries, and dynamic environments, ultimately enhancing its resilience and accuracy in real-world applications.

References

[1] S. A. Hiremath, P. K. Gummadi, and N. Bajcinca, "Image based model predictive controller for autonomous driving," *IEEE*, p. 150–157, Jun. 2023. [Online]. Available: http://dx.doi.org/10.1109/MED59994.2023.10185808

[2] A. Mancuso, "Study and implementation of lane detection and lane keeping for autonomous driving vehicles," https://webthesis.biblio.polito.it/9514/1/tesi.pdf.

[3] J. Ho Yang, W. Y. Choi, and C. Choo Chung, "Strikenet: Deep convolutional lstm-based road lane reconstruction with spatiotemporal inference for lane keeping control," *IEEE Access*, vol. 12, p. 97500–97514, 2024. [Online]. Available: http://dx.doi.org/10.1109/ACCESS.2024.3424944

[4] Z. Qin, H. Wang, and X. Li, *Ultra Fast Structure-Aware Deep Lane Detection*. Springer International Publishing, 2020, p. 276–291. [Online]. Available: http://dx.doi.org/10.1007/978-3-030-58586-0_17

[5] T. Weiskircher, Q. Wang, and B. Ayalew, "Predictive guidance and control framework for (semi-)autonomous vehicles in public traffic," *IEEE Transactions on Control Systems Technology*, vol. 25, no. 6, p. 2034–2046, Nov. 2017. [Online]. Available: http://dx.doi.org/10.1109/TCST.2016.2642164

[6] Z. D. S. Kusuma, K. Indriawati, B. L. Widjiantoro, A. I. Hija, and H. Nurhadi, "Optimal trajectory planning generation for autonomous vehicle using frenet reference path," *IEEE*, p. 480–484, Dec. 2022. [Online]. Available: http://dx.doi.org/10.1109/ISRITI56927.2022.10052833

[7] M. Joseph and J. Tackes, *Modern time series forecasting with python: Industry-ready machine learning and deep learning time series analysis with pytorch and Pandas*. Packt Publishing Ltd, 2024.

[8] J. L. Elman, "Finding structure in time," *Cognitive Science*, vol. 14, no. 2, p. 179–211, Mar. 1990. [Online]. Available: http://dx.doi.org/10.1207/s15516709cog1402_1

[9] C. Subakan, M. Ravanelli, S. Cornell, M. Bronzi, and J. Zhong, "Attention is all you need in speech separation," *IEEE*, p. 21–25, Jun. 2021. [Online]. Available: http://dx.doi.org/10.1109/ICASSP39728.2021.9413901

[10] R. A. de Lemos, O. Garcia, and J. V. Ferreira, "Local and global path generation for autonomous vehicles using splines," *IEEE*, p. 1–6, Oct. 2015. [Online]. Available: http://dx.doi.org/10.1109/WEA.2015.7370124

[11] A. M. Musolf, E. R. Holzinger, J. D. Malley, and J. E. Bailey-Wilson, "What makes a good prediction? feature importance and beginning to open the black box of machine learning in genetics," *Human Genetics*, vol. 141, no. 9, p. 1515–1528, Dec. 2021. [Online]. Available: http://dx.doi.org/10.1007/s00439-021-02402-z

[12] T. Getahun and A. Karimoddini, "An integrated vision-based perception and control for lane keeping of autonomous vehicles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 25, no. 8, p. 9001–9015, Aug. 2024. [Online]. Available: http://dx.doi.org/10.1109/TITS.2024.3376516

[13] Y. S. Son, W. Kim, S.-H. Lee, and C. C. Chung, "Robust multirate control scheme with predictive virtual lanes for lane-keeping system of autonomous highway driving," *IEEE Transactions on Vehicular Technology*, vol. 64, no. 8, p. 3378–3391, Aug. 2015. [Online]. Available: http://dx.doi.org/10.1109/TVT.2014.2356204

[14] C. Kang, S.-H. Lee, and C. C. Chung, "On-road vehicle localization with gps under long term failure of a vision sensor," *IEEE*, p. 1638–1643, Sep. 2015. [Online]. Available: http://dx.doi.org/10.1109/ITSC.2015.266

[15] S.-H. Lee and C. C. Chung, "Robust multirate on-road vehicle localization for autonomous highway driving vehicles," *IEEE Transactions on Control Systems Technology*, vol. 25, no. 2, p. 577–589, Mar. 2017. [Online]. Available: http://dx.doi.org/10.1109/TCST.2016.2562607

[16] C. M. Kang, S.-H. Lee, S.-C. Kee, and C. C. Chung, "Kinematics-based fault-tolerant techniques: Lane prediction for an autonomous lane keeping system," *International Journal of Control, Automation and Systems*, vol. 16, no. 3, p. 1293–1302, May 2018. [Online]. Available: http://dx.doi.org/10.1007/s12555-017-0449-8

[17] D. A. Pomerleau, "Alvinn: An autonomous land vehicle in a neural network," *NIPS*, vol. 1, 1988. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/1988/file/812b4ba287f5ee0bc9d43bbf5bbe87fb-Paper.pdf

[18] D. Pomerleau, "Ralph: rapidly adapting lateral position handler," *IEEE*, p. 506–511, 1995. [Online]. Available: http://dx.doi.org/10.1109/IVS.1995.528333

[19] D.-H. Lee and J.-L. Liu, "End-to-end deep learning of lane detection and path prediction for real-time autonomous driving," *Signal, Image and Video Processing*, vol. 17, no. 1, p. 199–205, Apr. 2022. [Online]. Available: http://dx.doi.org/10.1007/s11760-022-02222-2

[20] U. M. Gidado, H. Chiroma, N. Aljojo, S. Abubakar, S. I. Popoola, and M. A. Al-Garadi, "A survey on deep learning for steering angle prediction in autonomous vehicles," *IEEE Access*, vol. 8, p. 163797–163817, 2020. [Online]. Available: http://dx.doi.org/10.1109/ACCESS.2020.3017883

[21] V. Rausch, A. Hansen, E. Solowjow, C. Liu, E. Kreuzer, and J. K. Hedrick, "Learning a deep neural net policy for end-to-end control of autonomous vehicles," *IEEE*, p. 4914–4919, May 2017. [Online]. Available: http://dx.doi.org/10.23919/ACC.2017.7963716

[22] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars," 2016. [Online]. Available: https://arxiv.org/abs/1604.07316

[23] J. H. Yang, W. Y. Choi, and C. C. Chung, "Recurrent end-to-end neural network design with temporal dependencies for model-free lane keeping systems," *IEEE*, p. 551–556, Oct. 2019. [Online]. Available: http://dx.doi.org/10.23919/ICCAS47443.2019.8971744

[24] T. Wu, A. Luo, R. Huang, H. Cheng, and Y. Zhao, "End-to-end driving model for steering control of autonomous vehicles with future spatiotemporal features," *IEEE*, Nov. 2019. [Online]. Available: http://dx.doi.org/10.1109/IROS40897.2019.8968453

[25] M. M. Madebo, C. M. Abdissa, L. N. Lemma, and D. S. Negash, "Robust tracking control for quadrotor uav with external disturbances and uncertainties using neural network based mrac," *IEEE Access*, vol. 12, p. 36183–36201, 2024. [Online]. Available: http://dx.doi.org/10.1109/ACCESS.2024.3374894

[26] S. C. Yogi, V. K. Tripathi, and L. Behera, "Adaptive integral sliding mode control using fully connected recurrent neural network for position and attitude control of quadrotor," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 12, p. 5595–5609, Dec. 2021. [Online]. Available: http://dx.doi.org/10.1109/TNNLS.2021.3071020

[27] A. Gurghian, T. Koduri, S. V. Bailur, K. J. Carey, and V. N. Murali, "Deeplanes: End-to-end lane position estimation using deep neural networks," *IEEE*, p. 38–45, Jun. 2016. [Online]. Available: http://dx.doi.org/10.1109/CVPRW.2016.12

[28] A. Amini, W. Schwarting, G. Rosman, B. Araki, S. Karaman, and D. Rus, "Variational autoencoder for end-to-end control of autonomous driving with novelty detection and training de-biasing," *IEEE*, p. 568–575, Oct. 2018. [Online]. Available: http://dx.doi.org/10.1109/IROS.2018.8594386

[29] Y. Chai, S. Wang, and Z. Zhang, "A fast and accurate lane detection method based on row anchor and transformer structure," *Sensors*, vol. 24, no. 7, p. 2116, Mar. 2024. [Online]. Available: http://dx.doi.org/10.3390/s24072116

[30] J. H. Yoo, S.-W. Lee, S.-K. Park, and D. H. Kim, "A robust lane detection method based on vanishing point estimation using the relevance of line segments," *IEEE Transactions on Intelligent Transportation Systems*, vol. 18, no. 12, p. 3254–3266, Dec. 2017. [Online]. Available: http://dx.doi.org/10.1109/TITS.2017.2679222

[31] S. Jung, J. Youn, and S. Sull, "Efficient lane detection based on spatiotemporal images," *IEEE Transactions on Intelligent Transportation Systems*, vol. 17, no. 1, p. 289–295, Jan. 2016. [Online]. Available: http://dx.doi.org/10.1109/TITS.2015.2464253

[32] M. Werling, J. Ziegler, S. Kammel, and S. Thrun, "Optimal trajectory generation for dynamic street scenarios in a frenamp;x00e9;t frame," *IEEE*, May 2010. [Online]. Available: http://dx.doi.org/10.1109/ROBOT.2010.5509799

[33] M. Wang, L. Zhang, Z. Wang, Y. Sai, and Y. Chu, "A real-time dynamic trajectory planning for autonomous driving vehicles," *IEEE*, p. 1–6, Sep. 2019. [Online]. Available: http://dx.doi.org/10.1109/CVCI47823.2019.8951735

[34] M. Díaz-Zapata, J. M. Correa-Sandoval, J. Perafán-Villota, and V. Romero-Cano, *Lane Detection and Trajectory Generation System.* Pontificia Universidad Javeriana Cali, Dec. 2021, p. 97–110. [Online]. Available: http://dx.doi.org/10.2307/j.ctv2d6jrr4.8

[35] S. Wang, X. Wang, and S. Wang, *Lane Change Decision and Trajectory Planning for Intelligent Cars in Curved Road Scenarios.* IOS Press, Dec. 2022. [Online]. Available: http://dx.doi.org/10.3233/ATDE221112

[36] F. Pucher, "Trajectory Planning in the Frenet Space — fjp.at," 2018, https://fjp.at/posts/optimal-frenet/#frenet-coordinates.

[37] A. E. Sallab, M. Abdou, E. Perot, and S. Yogamani, "End-to-end deep reinforcement learning for lane keeping assist," 2016. [Online]. Available: https://arxiv.org/abs/1612.04340

[38] F. Codevilla, M. Muller, A. Lopez, V. Koltun, and A. Dosovitskiy, "End-to-end driving via conditional imitation learning," *IEEE*, p. 4693–4700, May 2018. [Online]. Available: http://dx.doi.org/10.1109/ICRA.2018.8460487

[39] M. Bojarski, P. Yeres, A. Choromanska, K. Choromanski, B. Firner, L. Jackel, and U. Muller, "Explaining how a deep neural network trained with end-to-end learning steers a car," 2017. [Online]. Available: https://arxiv.org/abs/1704.07911

[40] W. Li and K. L. E. Law, "Deep learning models for time series forecasting: A review," *IEEE Access*, vol. 12, p. 92306–92327, 2024. [Online]. Available: http://dx.doi.org/10.1109/ACCESS.2024.3422528

[41] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "Carla: An open urban driving simulator," 2017. [Online]. Available: https://arxiv.org/abs/1711.03938

[42] I. Gridin, *Time series forecasting using deep learning explore the infinite possibilities offered by Artificial Intelligence and neural networks.* BPB Publications, 2021.

[43] S. Bai, J. Z. Kolter, and V. Koltun, "An empirical evaluation of generic convolutional and recurrent networks for sequence modeling," 2018. [Online]. Available: https://arxiv.org/abs/1803.01271

[44] TuSimple, "Tusimple-benchmark: https://github.com/tusimple/tusimple-benchmark/issues/3," 2017. [Online]. Available: https://github.com/TuSimple/tusimple-benchmark

[45] X. Pan, J. Shi, P. Luo, X. Wang, and X. Tang, "Spatial as deep: Spatial cnn for traffic scene understanding," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, Apr. 2018. [Online]. Available: http://dx.doi.org/10.1609/aaai.v32i1.12301

[46] P. Cai, Y. Sun, H. Wang, and M. Liu, "Vtgnet: A vision-based trajectory generation network for autonomous vehicles in urban environments," *IEEE Transactions on Intelligent Vehicles*, vol. 6, no. 3, p. 419–429, Sep. 2021. [Online]. Available: http://dx.doi.org/10.1109/TIV.2020.3033878

[47] D. Singh and B. Singh, "Feature wise normalization: An effective way of normalizing data," *Pattern Recognition*, vol. 122, p. 108307, Feb. 2022. [Online]. Available: http://dx.doi.org/10.1016/j.patcog.2021.108307

[48] A. Jadon, A. Patil, and S. Jadon, *A Comprehensive Survey of Regression-Based Loss Functions for Time Series Forecasting*. Springer Nature Singapore, 2024, p. 117–147. [Online]. Available: http://dx.doi.org/10.1007/978-981-97-3245-6_9

[49] "PyTorch 2.5 documentation- pytorch.org," 2020, https://pytorch.org/docs/stable/generated/torch.nn.Transformer.html.

[50] S. Schmidl, P. Wenig, and T. Papenbrock, "Hypex: Hyperparameter optimization in time series anomaly detection," *Gesellschaft für Informatik e.V.*, 2023. [Online]. Available: http://dl.gi.de/handle/20.500.12116/40327

[51] Y. Tao, A. Both, R. I. Silveira, K. Buchin, S. Sijben, R. S. Purves, P. Laube, D. Peng, K. Toohey, and M. Duckham, "A comparative analysis of trajectory similarity measures," *GIScience amp; Remote Sensing*, vol. 58, no. 5, p. 643–669, Jun. 2021. [Online]. Available: http://dx.doi.org/10.1080/15481603.2021.1908927

[52] H. Alt, C. Knauer, and C. Wenk, "Comparison of distance measures for planar curves," *Algorithmica*, vol. 38, no. 1, p. 45–58, Oct. 2003. [Online]. Available: http://dx.doi.org/10.1007/s00453-003-1042-5

[53] R. Li and X. Zhang, "All you need is transformer: Rtt prediction for tcp based on deep learning approach," *IEEE*, p. 348–351, Dec. 2021. [Online]. Available: http://dx.doi.org/10.1109/DSInS54396.2021.9670591

[54] S. Elfwing, E. Uchibe, and K. Doya, "Sigmoid-weighted linear units for neural network function approximation in reinforcement learning," *Neural Networks*, vol. 107, p. 3–11, Nov. 2018. [Online]. Available: http://dx.doi.org/10.1016/j.neunet.2017.12.012

[55] scikit learn, "KDTree - documentation: scikit-learn.org," 2015, https://scikit-learn.org/1.5/modules/generated/sklearn.neighbors.KDTree.html.

A. Appendix I

A.1 Model Architecture

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

class EncoderLayer(nn.Module):
    def __init__(self, d_model, n_heads, d_ff, dropout=0):
        super(EncoderLayer, self).__init__()
        self.self_attn = nn.MultiheadAttention(d_model, n_heads, dropout=dropout)
        self.linear1 = nn.Linear(d_model, d_ff)
        self.linear2 = nn.Linear(d_ff, d_model)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)
        self.silu = nn.SiLU()

    def forward(self, src, src_key_padding_mask=None):
        src2 = self.self_attn(src, src, src, key_padding_mask=src_key_padding_mask)[0]
        src = src + self.dropout(self.norm1(src2))
        src2 = self.linear2(self.silu(self.linear1(src)))
        src = src + self.dropout(self.norm2(src2))
        return src

class Encoder(nn.Module):
    def __init__(self, input_dim, d_model, n_layers, n_heads, d_ff, dropout=0.3):
        super(Encoder, self).__init__()
        self.layers = nn.ModuleList([EncoderLayer(d_model, n_heads, d_ff, dropout) for _ in range(n_layers)])
```

```
28
29      def forward(self, src, src_key_padding_mask=None):
30          for layer in self.layers:
31              src = layer(src, src_key_padding_mask)
32          return src
33
34  class DecoderLayer(nn.Module):
35      def __init__(self, d_model, n_heads, d_ff, dropout=0):
36          super(DecoderLayer, self).__init__()
37          self.self_attn = nn.MultiheadAttention(d_model, n_heads,
    dropout=dropout)
38          self.src_attn = nn.MultiheadAttention(d_model, n_heads,
    dropout=dropout)
39          self.linear1 = nn.Linear(d_model, d_ff)
40          self.linear2 = nn.Linear(d_ff, d_model)
41          self.norm1 = nn.LayerNorm(d_model)
42          self.norm2 = nn.LayerNorm(d_model)
43          self.norm3 = nn.LayerNorm(d_model)
44          self.dropout = nn.Dropout(dropout)
45          self.silu = nn.SiLU()
46
47      def forward(self, tgt, memory, tgt_key_padding_mask=None,
    memory_key_padding_mask=None):
48          tgt2 = self.self_attn(tgt, tgt, tgt, key_padding_mask=
    tgt_key_padding_mask)[0]
49          tgt = tgt + self.dropout(self.norm1(tgt2))
50          tgt2 = self.src_attn(tgt, memory, memory,
    key_padding_mask=memory_key_padding_mask)[0]
51          tgt = tgt + self.dropout(self.norm2(tgt2))
52          tgt2 = self.linear2(self.silu(self.linear1(tgt)))
53          tgt = tgt + self.dropout(self.norm3(tgt2))
54          return tgt
55
56  class Decoder(nn.Module):
57      def __init__(self, output_dim, d_model, n_layers, n_heads,
    d_ff, dropout=0):
58          super(Decoder, self).__init__()
59          self.layers = nn.ModuleList([DecoderLayer(d_model,
    n_heads, d_ff, dropout) for _ in range(n_layers)])
60
61      def forward(self, tgt, memory, tgt_key_padding_mask=None,
    memory_key_padding_mask=None):
62          for layer in self.layers:
63              tgt = layer(tgt, memory, tgt_key_padding_mask,
    memory_key_padding_mask)
64          return tgt
65
66  class SegmentPredictionModel_Map(nn.Module):
67      def __init__(self, input_dim, output_dim, d_model, n_layers,
    n_heads, d_ff, dropout=0):
```

```
68          super(SegmentPredictionModel_Map, self).__init__()
69          self.encoder = Encoder(input_dim, d_model, n_layers,
    n_heads, d_ff, dropout)
70          self.decoder = Decoder(output_dim, d_model, n_layers,
    n_heads, d_ff, dropout)
71
72      def forward(self, src, src_key_padding_mask=None,
    tgt_key_padding_mask=None):
73          memory = self.encoder(src, src_key_padding_mask)
74          delta = torch.tensor(0.1, dtype=memory.dtype, device=
    memory.device)
75          tgt_SOS = torch.where(src >= 0, memory + delta, memory -
    delta)
76          output = self.decoder(tgt_SOS, memory,
    tgt_key_padding_mask, src_key_padding_mask)
77          return output
```

Listing A.1: PyTorch Implementation of Seq-2-Seq Transformer Model

A.2 Bayesian Optimization Tuner and Training Loop

```
1
2  import torch
3  import torch.nn as nn
4  import torch.optim as optim
5  from skopt import gp_minimize
6  from skopt.space import Real, Categorical
7  from skopt.utils import use_named_args
8  import time
9  import statistics as st
10 import data_loader  # Assuming data_loader is a module
11
12 # Dataset and Batch Size Percentages
13 batch_size_percentages_map = {1: 40, 2: 40, 3: 50, 4: 30, 5: 20,
    6: 80, 7: 10}
14
15 # Loss Functions
16 criterion1_options = [nn.MSELoss(), nn.HuberLoss()]
17 criterion2_enabled = True  # Enable curvature MAE
18 criterion3_enabled = False  # Enable polynomial coefficients loss
19
20 # Define the function to minimize (validation loss)
21 @use_named_args([
22     Real(0.4, 0.99, name='alpha'),  # RMSprop alpha
23     Real(0.4, 0.99, name='momentum'),  # Momentum for optimizers
24     Categorical([40, 80, 120, 160], name='d_ff'),  # Feedforward
    dim
25     Categorical([8, 16, 24], name='n_heads'),  # Number of
    attention heads
26     Real(1e-5, 1e-2, name='learning_rate'),  # Learning rate
```

```
    Categorical([0.1, 0.2, 0.3], name='dropout'),  # Dropout rate
    Categorical(['adam', 'sgd', 'rmsprop'], name='optimizer'),  # Optimizer choice
    Real(1e-10, 1e-5, name='eps'),  # Optimizer epsilon
    Real(1e-5, 1e-2, name='weight_decay'),  # Weight decay
    Real(0.0, 1.0, name='gaussian_noise_sd'),  # Gaussian noise standard deviation
    Real(0.0, 1.0, name='gaussian_noise_data_percentage'),  # Percentage of noisy data
    Categorical([1, 2, 3, 4, 5, 6, 7], name='selected_datasets'),  # Dataset selection index
    Real(0.0, 1.0, name='W1'),  # Weight for criterion1
    Real(0.0, 1.0, name='W2'),  # Weight for criterion2
    Real(0.0, 1.0, name='W3'),  # Weight for criterion3
    Real(0.0, 1.0, name='delta')  # Delta parameter for the model
])
def train_model(alpha, momentum, d_ff, n_heads, learning_rate, dropout, optimizer, eps, weight_decay,
                gaussian_noise_sd, gaussian_noise_data_percentage, selected_datasets, W1, W2, W3, delta):
    # Resolve batch size percentage from the selected dataset
    batch_size_percentage = batch_size_percentages_map[selected_datasets]

    # Model Parameters
    input_dim = 128
    output_dim = 128
    d_model = 128
    n_layers = 1

    # Load Data
    inputs, targets, val_inputs, val_targets = data_loader.data_loader(
        gaussian_noise_sd, gaussian_noise_data_percentage, batch_size_percentage, selected_datasets
    )

    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    inputs, targets, val_inputs, val_targets = inputs.to(device), targets.to(device), val_inputs.to(device), val_targets.to(device)

    src_mask1 = (inputs >= 1280).any(dim=2).T
    src_mask_v1 = (val_inputs >= 1280).any(dim=2).T

    mask = (targets < 1280).to(device)
    mask_v = (val_targets < 1280).to(device)

    # Model Initialization
```

```
65      model = VTG_model13.SegmentPredictionModel_Map(input_dim,
    output_dim, d_model, n_layers, n_heads, d_ff, dropout).to(
    device)
66
67      # Optimizer Selection
68      optimizer_dict = {
69          'adam': optim.Adam(model.parameters(), lr=learning_rate,
    eps=eps, weight_decay=weight_decay),
70          'sgd': optim.SGD(model.parameters(), lr=learning_rate,
    momentum=momentum, weight_decay=weight_decay, nesterov=True),
71          'rmsprop': optim.RMSprop(model.parameters(), lr=
    learning_rate, alpha=alpha, eps=eps, weight_decay=weight_decay,
     momentum=momentum)
72      }
73      optimizer = optimizer_dict[optimizer]
74
75      # Training Loop
76      num_epochs = 5
77      train_losses, val_losses = [], []
78      global ref_call, min_train_loss, min_val_loss, min_train_ref,
     min_val_ref
79      ref_call += 1
80
81      start_time = time.time()
82
83      for epoch in range(num_epochs):
84          model.train()
85          optimizer.zero_grad()
86
87          # Forward Pass
88          outputs = model(inputs, src_key_padding_mask=src_mask1,
    delta=delta)
89
90          masked_targets = targets[mask]
91          masked_outputs = outputs[mask]
92
93          # Loss Calculation
94          loss_criterion1 = criterion1_options[0](masked_outputs,
    masked_targets)
95          train_loss = loss_criterion1.item()
96
97          if criterion2_enabled:
98              cv_masked_targets, cv_masks_list = cv.
    MaskingForCurvatureLoss(targets)
99              cv_masked_outputs, _ = cv.MaskingForCurvatureLoss(
    outputs, cv_masks_list=cv_masks_list)
100             curvature_mae = torch.mean(torch.abs(cv.
    CurvatureEvaluationForTensor(cv_masked_targets) - cv.
    CurvatureEvaluationForTensor(cv_masked_outputs)))
101             loss = W1 * loss_criterion1 + W2 * curvature_mae
```

```
102         else:
103             loss = W1 * loss_criterion1
104 
105         if criterion3_enabled:
106             coeff_diff = torch.mean(torch.abs(cff.
    PolynomialCoefficientsForTensor(masked_targets) - cff.
    PolynomialCoefficientsForTensor(masked_outputs)) ** 2)
107             loss += W3 * coeff_diff
108 
109         # Backward Pass
110         loss.backward()
111         optimizer.step()
112         train_losses.append(loss.item())
113 
114         # Validation Pass
115         model.eval()
116         with torch.no_grad():
117             val_outputs = model(val_inputs, src_key_padding_mask=
    src_mask_v1, delta=delta)
118             masked_val_targets = val_targets[mask_v]
119             masked_val_outputs = val_outputs[mask_v]
120 
121             val_loss = criterion1_options[0](masked_val_outputs,
    masked_val_targets).item()
122             val_losses.append(val_loss)
123 
124     total_elapsed_time = time.time() - start_time
125     current_min_train_loss = min(train_losses)
126     current_min_val_loss = min(val_losses)
127 
128     if current_min_train_loss < min_train_loss:
129         min_train_loss = current_min_train_loss
130         min_train_ref = ref_call
131 
132     if current_min_val_loss < min_val_loss:
133         min_val_loss = current_min_val_loss
134         min_val_ref = ref_call
135 
136     return min(val_losses)
137 
138 
139 # Initialization
140 ref_call = 0
141 min_train_loss = float('inf')
142 min_val_loss = float('inf')
143 min_train_ref = None
144 min_val_ref = None
145 
146 # Bayesian Optimization
147 result = gp_minimize(
```

```
    func=train_model,
    dimensions=[
        Real(0.4, 0.99), Real(0.4, 0.99),
        Categorical([40, 80, 120, 160]), Categorical([8, 16, 24]),
        Real(1e-5, 1e-2), Categorical([0.1, 0.2, 0.3]),
        Categorical(['adam', 'sgd', 'rmsprop']), Real(1e-10, 1e-5), Real(1e-5, 1e-2),
        Real(0.0, 1.0), Real(0.0, 1.0), Categorical([1, 2, 3, 4, 5, 6, 7]),
        Real(0.0, 1.0), Real(0.0, 1.0), Real(0.0, 1.0), Real(0.0, 1.0)
    ],
    n_calls=30,
    random_state=42
)

# Results
print("Best parameters found: ", result.x)
print(f'Minimum Training Loss: {min_train_loss} at Ref Call: {min_train_ref}')
print(f'Minimum Validation Loss: {min_val_loss} at Ref Call: {min_val_ref}')
```

Listing A.2: PyTorch Training loop and Bayesian Optimization Tuner

A.3 Additional Graphs

(P.T.O)

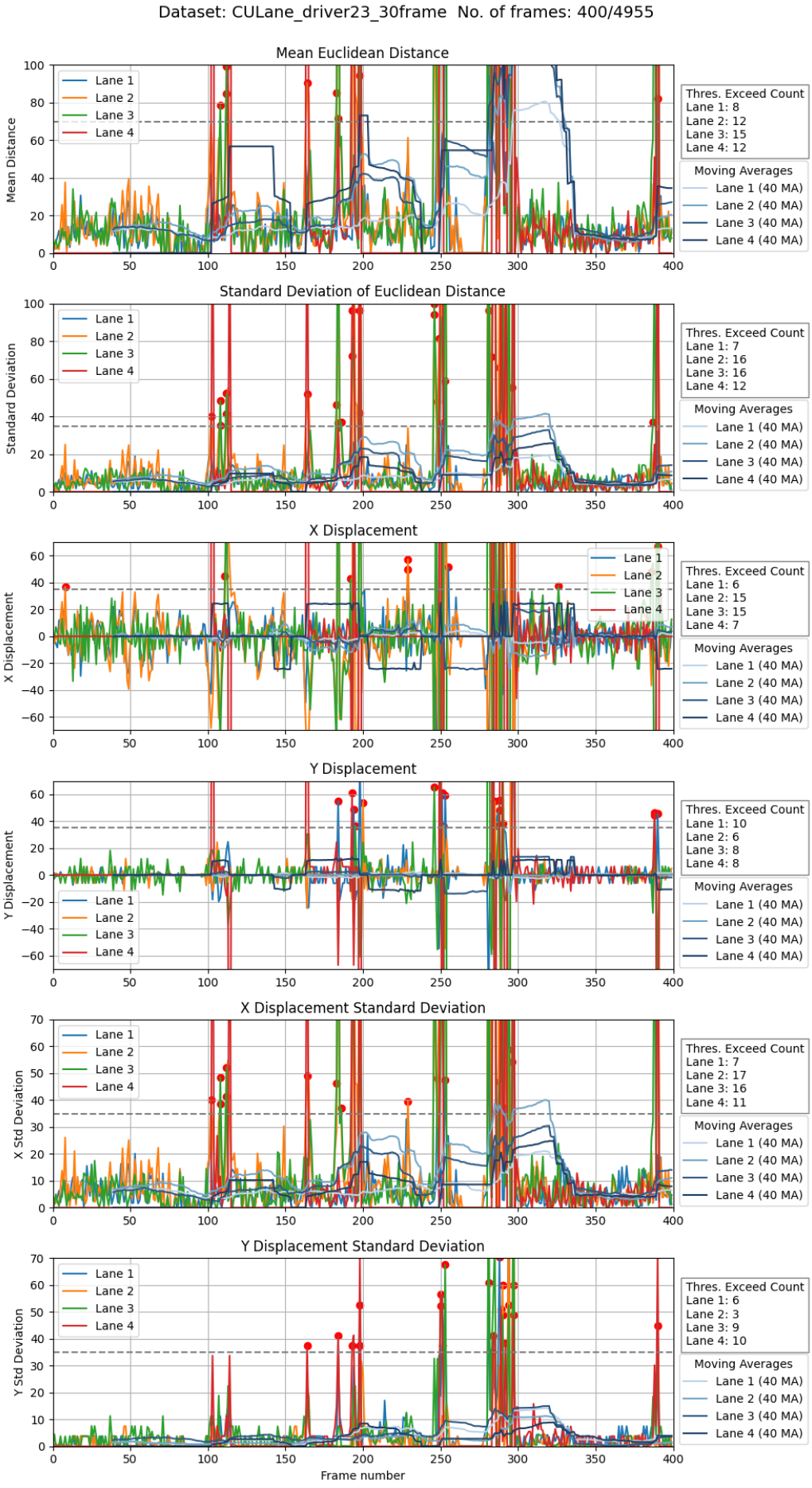


Figure A.1: Time Series Analysis- CULane Dataset.

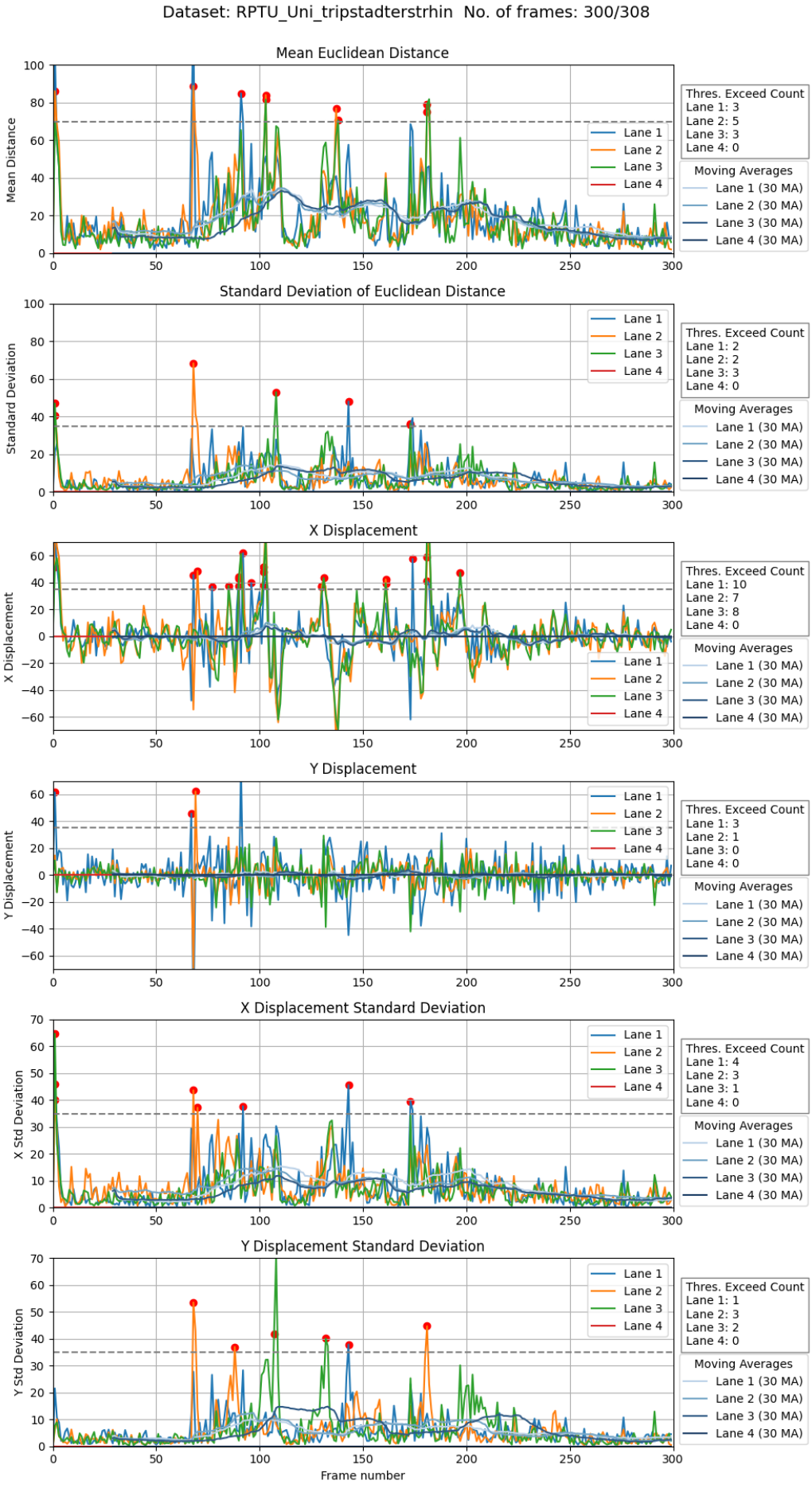


Figure A.2: Time Series Analysis- RPTU-Tripstadtersthin

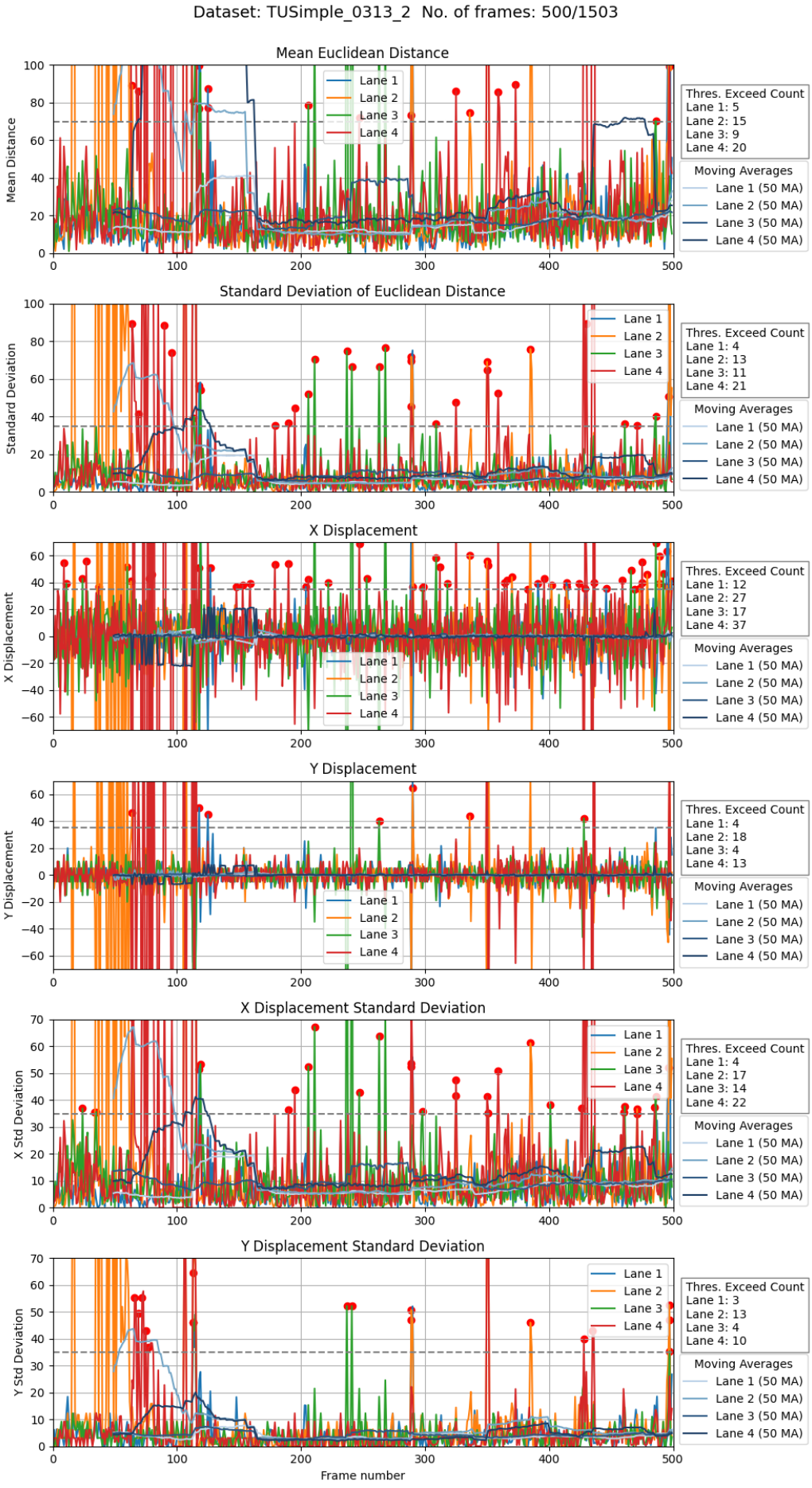


Figure A.3: Time Series Analysis- TUSimple-0313-2

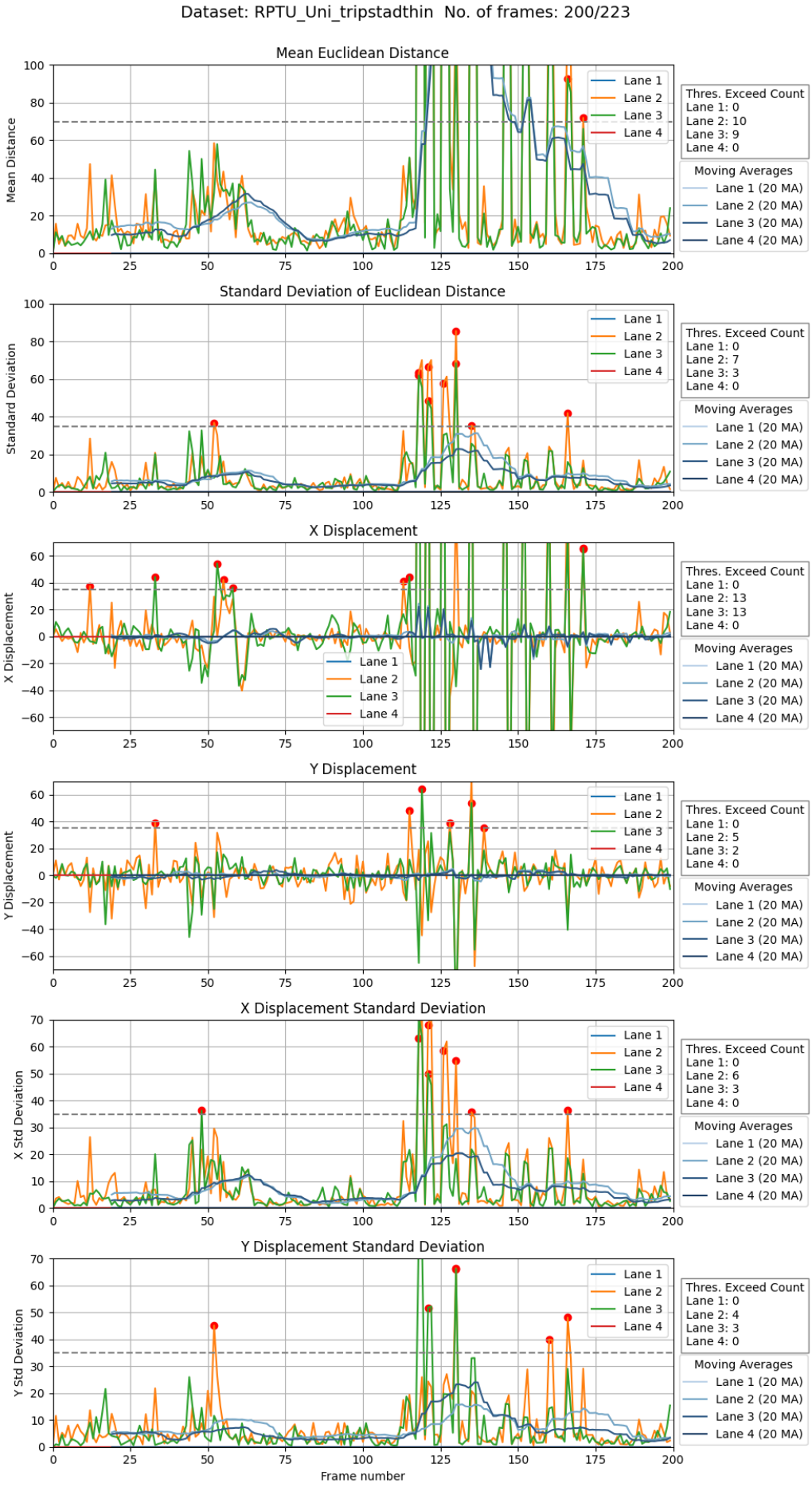


Figure A.4: Time Series Analysis- RPTU-Tripstadtersthin