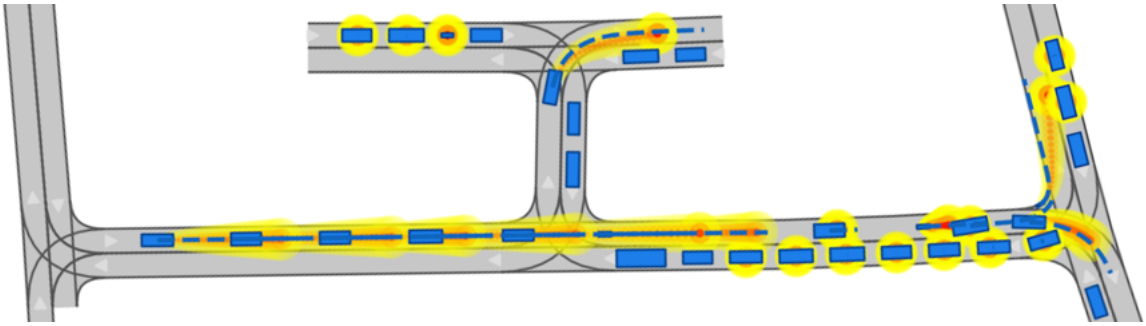
Robust and Computational-Efficient Vehicle Trajectory Prediction for Autonomous Driving through Deep Learning

Robuste und recheneffiziente Trajektorienprädiktion für autonomes Fahren mittels Deep Learning



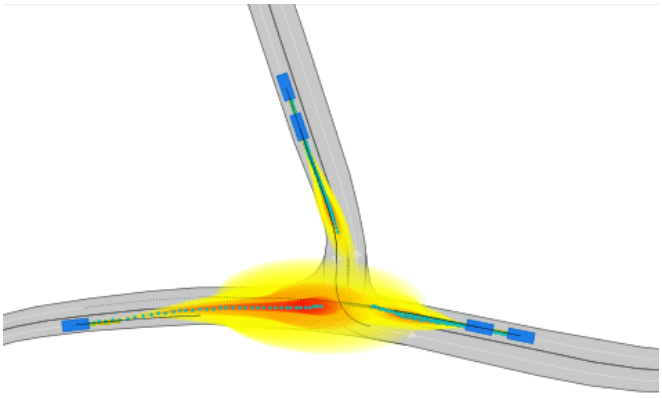
Scientific work for obtaining the academic degree
Bachelor of Science (B.Sc.)
at the Department Mobility Systems Engineering
of the TUM School of Engineering and Design
at the Technical University of Munich

Supervised by	Prof. Dr.-Ing. Markus Lienkamp Nico Uhlemann, Dipl.-Ing. Chair of Automotive Technology
Submitted by	Jakob Gollreiter Veterinärstraße 9. 80539 München
Submitted on	August 20, 2023

Lehrstuhl für Fahrzeugtechnik
Fakultät für Maschinenwesen
Technische Universität München

Bachelorarbeit

Robuste und recheneffiziente Trajektorienprädiktion für autonomes Fahren mittels Deep Learning



Prädiktion mit Unsicherheiten in CommonRoad*

Am Lehrstuhl für Fahrzeugtechnik haben wir das Ziel, eine gesamtheitliche Software für autonomes Fahren zu entwickeln, die am Ende im Realfahrzeug eingesetzt werden soll. Im Rahmen des Projekts „EDGAR“ wird dabei aktuell ein Fahrzeug entwickelt, welches sowohl auf Autobahnen, in der Innenstadt und schlussendlich auch rund um die Wiesn autonom fahren soll (Level 3+). Hierfür ist es wichtig, die Intentionen anderer Fahrzeuge präzise vorherzusagen, um sowohl kooperatives Verhalten zu gewährleisten als auch Unfälle zu vermeiden.

Im Rahmen dieser Arbeit soll deshalb unter Nutzung Neuronaler Netze ein Modell entwickelt werden, welches zuverlässig Trajektorien anderer Verkehrsteilnehmer prädiziert. Neben öffentlich verfügbaren Datensätzen wie HighD oder inD für das Training bietet die Softwareumgebung CommonRoad der TUM eine Simulationsumgebung für die Zusammenführung aller nötigen Softwarekomponenten.

Folgende Arbeitspakete umfasst die zu vergebende Studienarbeit:

- Literaturrecherche und Aufzeigen des Stands der Technik
- Anforderungsanalyse und Auswahl eines geeigneten Ansatzes zur Trajektorienprädiktion
- Implementierung eines ausgewählten Machine Learning Verfahrens und Training mittels einem geeigneten Datensatz
- Qualitativer und Quantitativer Vergleich mit vorhandenen Methoden
- Integration und Auswertung in CommonRoad
- Evaluation und Dokumentation der Ergebnisse

Die Ausarbeitung soll die einzelnen Arbeitsschritte in übersichtlicher Form dokumentieren. Der Kandidat/Die Kandidatin verpflichtet sich, die Studienarbeit selbständig durchzuführen und die von ihm verwendeten wissenschaftlichen Hilfsmittel anzugeben.
Die eingereichte Arbeit verbleibt als Prüfungsunterlage im Eigentum des Lehrstuhls.

Prof. Dr.-Ing. M. Lienkamp

Betreuer: Dipl.-Ing. Nico Uhlemann

*Source: https://github.com/TUMFTM/Wale-Net

Lehrstuhl für Fahrzeugtechnik
Fakultät für Maschinenwesen
Technische Universität München

Ausgabe:_____ Abgabe: _____

*Source: https://github.com/TUMFTM/Wale-Net

Chair of Automotive Technology
TUM School of Engineering and Design
Technical University of Munich

Geheimhaltungsverpflichtung

Herr: **Gollreiter, Jakob**

Gegenstand der Geheimhaltungsverpflichtung sind alle mündlichen, schriftlichen und digitalen Informationen und Materialien die der Unterzeichner vom Lehrstuhl oder von Dritten im Rahmen seiner Tätigkeit am Lehrstuhl erhält. Dazu zählen vor allem Daten, Simulationswerkzeuge und Programmcode sowie Informationen zu Projekten, Prototypen und Produkten.

Der Unterzeichner verpflichtet sich, alle derartigen Informationen und Unterlagen, die ihm während seiner Tätigkeit am Lehrstuhl für Fahrzeugtechnik zugänglich werden, strikt vertraulich zu behandeln.

Er verpflichtet sich insbesondere:

- derartige Informationen betriebsintern zum Zwecke der Diskussion nur dann zu verwenden, wenn ein ihm erteilter Auftrag dies erfordert,
- keine derartigen Informationen ohne die vorherige schriftliche Zustimmung des Betreuers an Dritte weiterzuleiten,
- ohne Zustimmung eines Mitarbeiters keine Fotografien, Zeichnungen oder sonstige Darstellungen von Prototypen oder technischen Unterlagen hierzu anzufertigen,
- auf Anforderung des Lehrstuhls für Fahrzeugtechnik oder unaufgefordert spätestens bei seinem Ausscheiden aus dem Lehrstuhl für Fahrzeugtechnik alle Dokumente und Datenträger, die derartige Informationen enthalten, an den Lehrstuhl für Fahrzeugtechnik zurückzugeben.

Eine besondere Sorgfalt gilt im Umgang mit digitalen Daten:

- Für den Dateiaustausch dürfen keine Dienste verwendet werden, bei denen die Daten über einen Server im Ausland geleitet oder gespeichert werden (Es dürfen nur Dienste des LRZ genutzt werden (Lehrstuhllaufwerke, Sync&Share, GigaMove).
- Vertrauliche Informationen dürfen nur in verschlüsselter Form per E-Mail versendet werden.
- Nachrichten des geschäftlichen E-Mail Kontos, die vertrauliche Informationen enthalten, dürfen nicht an einen externen E-Mail Anbieter weitergeleitet werden.
- Die Kommunikation sollte nach Möglichkeit über die (my)TUM-Mailadresse erfolgen.

Die Verpflichtung zur Geheimhaltung endet nicht mit dem Ausscheiden aus dem Lehrstuhl für Fahrzeugtechnik, sondern bleibt 5 Jahre nach dem Zeitpunkt des Ausscheidens in vollem Umfang bestehen. Die eingereichte schriftliche Ausarbeitung darf der Unterzeichner nach Bekanntgabe der Note frei veröffentlichen.

Der Unterzeichner willigt ein, dass die Inhalte seiner Studienarbeit in darauf aufbauenden Studienarbeiten und Dissertationen mit der nötigen Kennzeichnung verwendet werden dürfen.

Datum: 20. August 2023

Unterschrift: 

Chair of Automotive Technology
TUM School of Engineering and Design
Technical University of Munich

Erklärung

Ich versichere hiermit, dass ich die von mir eingereichte Abschlussarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Garching, den 20. August 2023

Jakob Gollreiter, B.Sc.

Chair of Automotive Technology
TUM School of Engineering and Design
Technical University of Munich

Declaration of Consent, Open Source

Hereby I, Gollreiter, Jakob, born on October 1, 1999, make the software I developed during my Semester Thesis available to the Institute of Automotive Technology under the terms of the license below.

Garching, August 20, 2023

Jakob Gollreiter, B.Sc.

Contents

Acknowledgments

After spending five months of interesting work in the Autonomous Vehicle Lab at the TUM Institute of Automotive Technology, I want to thank my supervisor Nico Uhlemann for his great guidance and the valuable insights in conducting research.

1 Abstract

Predicting the future behavior of vehicles in complex traffic scenarios is an essential task for self-driving cars, which requires the integration of multiple data sources and dynamic constraints. Autonomous vehicles navigating intricate road environments rely on accurate multi-agent behavior prediction, leveraging a synthesis of perceptual inputs and map information to anticipate diverse trajectories and ensure safe interactions. This bachelor thesis presents a novel and computationally efficient neural network architecture designed to enable multi-agent trajectory prediction by considering various input features. These include high definition map data which is encoded to highlight the important features for each prediction task. The prediction process also considers both the inherent physical characteristics of the vehicle as well as the dynamic influence exerted by external vehicles on the expected trajectory of the subject vehicle.

We conduct training and performance evaluation using numerous recorded real-world scenarios sourced from the TUM-CommonRoad raw data [1]. To enable the training of the prediction network, a sophisticated pre-processing data pipeline has been developed for extracting the required training features out of numerous high-level scenarios. This pipeline efficiently manages the processes of generating and extracting training data (input features and ground truth), which subsequently serves as the foundation for training the proposed neural network architecture. Our approach achieves robust prediction results with competitive inference times while employing a significantly streamlined network architecture.

2 Introduction and Motivation

In recent years a lot of effort has been made from academia and industry to develop verifiable safe autonomous driving systems [2]. Addressing the fundamental challenge of modeling and predicting the future actions of human agents holds immense significance across for this task. In real-world scenarios, such as autonomous driving, accurate anticipation of the future behaviors of entities like vehicles, cyclists, and pedestrians plays an important role in achieving safe, smooth, and human-like driving [3]. It is important to capture the complex interactions between traffic participants. On top of that this has to be done in real time and with varying sensor data quality. Accurate trajectory prediction of all vehicles involved has a large impact on the decision-making process, as it is essential in detecting possible dangerous situations ahead. After observing a situation, the planning algorithms can generate a save trajectory for the own vehicle [2].

The perception of environmental information is gathered through various hardware sensors installed at different positions of the vehicle. These include active sensors like LIDAR, radar and ultrasonic and passive sensors like camera [4]. After the perception sensors obtained the data and it is grouped into relevant features, algorithms are used to build a representation of the current surrounding scene. This includes the understanding of the road network and the detection of moving objects. The road detection is obtained though vision based convolutional neural networks and the fusion of LIDAR based information to construct a 3D environmental representation [5]. The advances of computer vision algorithms for example object recognition, play an important role in enabling fully autonomous vehicles [6]. With the rise of deep learning, more accurate trajectory predictions have become feasible (see figure 2.1) by taking into account the vehicle's interaction dynamics and incorporating high-definition maps [7]. The representation of the driving environment encompasses a dynamic mix of static and dynamic inputs. These inputs consist of information about the road network, including lane configurations, connectivity, crosswalks, and stop lines, as well as data on traffic light states and the historical motion patterns of agents. Driving scenarios often involve intricate interactions and multiple agents simultaneously. A successful model not only captures the interplay among agents within the scene but also encapsulate the intricate relationships between road elements and agents' behaviors within the context of the road layout [3].

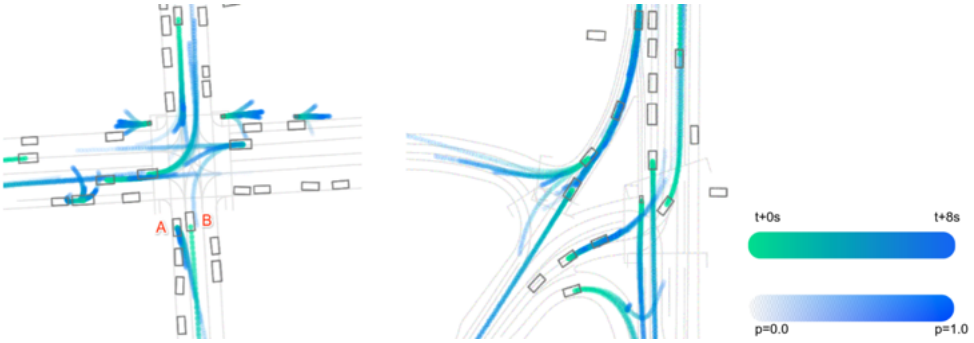


Figure 2.1: Trajectory prediction example Multipath++. The color hue corresponds to the time horizon, while the level of transparency reflects the anticipated probability [3].

Deep Learning algorithms as used for this task require a large amount of training data and are computational expensive to train [6]. The acquisition of a significant volume of real-life data is essential for effectively training the neural networks utilized in autonomous cars. This data collection process must encompass various challenging scenarios, such as adverse weather conditions, low-light environments and complex urban settings [4]. The validation of software quality, reliability, privacy, safety and security in the context of autonomous cars necessitates the development of novel methodologies, which often involve substantial time and effort [4].

The structure of this thesis continues as follows. Chapter 3 introduces three State of the Art trajectory prediction methods used in research and industry, to derive the structure of the concepts and software algorithms presented in this thesis. The methodology is detailed in Chapter 4 by presenting the data generation pipeline followed by our network architecture design used for training and evaluation. We continue by presenting and outlining the results of each aspect of the project in Chapter 5. Chapter 6 continues with a detailed discussion of our findings. Finally, in Chapter 7, we conclude by summarising our results and outlining possible areas for future work.

3 State of the Art of vehicle trajectory prediction

3.1 Problem formulation and terminology

Trajectory prediction requires past and present observations of a traffic participant to estimate its future states. In order to perform a trajectory prediction task, a lot of real-time data from sensors like LIDAR, radar and camera as well as map information is needed. This information is further processed into the following attributes for each traffic participant: position, heading, yaw rate, velocity and acceleration. Also, the relevant map information, like location, lane direction, crosswalk or traffic light is needed. Hence, all this data is transformed into car-centric coordinates [8].

To navigate an autonomous vehicle safely, it should be capable of predicting the future trajectories of its surrounding vehicles. Over the last two decades multiple different approaches have been developed to solve this problem [2]. The first attempts were physics-based methods [2, 8]. These methods try to model the social interactions between traffic participants, like a clustering of vehicles or their general tendency for collision avoidance, with physical quantities. However they lack in representing the complex interactions necessary for predicting human-like trajectories. More recently, machine and deep learning based approaches [9] and even reinforcement learning approaches [2] gained popularity.

3.2 Trajectory prediction categories

Prediction methods can be categorized in four general classes: physics-based, classic machine learning-based, deep learning-based and reinforcement learning-based method, see figure 3.1 [2].

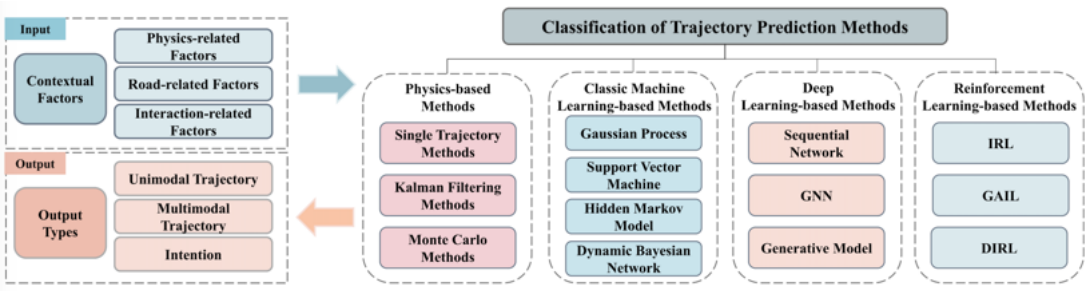


Figure 3.1: Categories of trajectory prediction [2]

3.2.1 Input contextual factors

In order for each model to make a prediction it needs contextual information of the current scenario (see Figure 3.1). In a driving environment this is a combination of static and dynamic input like the road network and the physical motion and interaction of the agents involved [3]. This includes the physics-related, road-related and interaction-related factors. The physics-related factors depict the dynamic and kinematic properties of the surrounding vehicles (acceleration, steering angle...). The road-related factors contain the local map information as well as the underlying traffic rules. The interaction-related factors describe the inter-dependencies and social regulations between the vehicle's maneuvers. [2]

In this thesis the input contextual factors are defined as "input features" (see section 4.4).

3.2.2 Output types

Figure 3.1 shows the three different categories of representing the predicted trajectories: The unimodal trajectory prediction outputs a single future trajectory for every traffic participant. A multimodal trajectory prediction generates multiple possible future trajectories, with their specific likelihood, for every traffic participant. An intention prediction tries to model the intended behaviour of each traffic participant. The latter can also be utilised as an interstep for both unimodal and multimodal prediction [2]. The multimodal prediction truly capture the complex stochastic environment and is best suited as it can handle different possible outcomes (vehicle could turn left or right at intersection) [3].

In this thesis the output type is unimodal and is referred to as "ground truth".

3.2.3 Prediction methods

In the following section four different prediction methods (see Figure 3.1) will be briefly introduced:

I) Physics-based methods
In trajectory prediction physics-based methods are the earliest implemented approaches. Although these methods are mostly computational inexpensive, they lack in accuracy. These models work best in describing short term kinematics and dynamics of a vehicle but do not incorporate the intention of the traffic participants. Therefore they are mainly used for short prediction times (less than 1 second). Usually they are implemented as unimodal prediction [2].

II) Classic machine learning-based methods
The classical machine learning-based methods work by determining the probability distribution based on the characteristics of the data, the so called features. In contrast to physics-based techniques, these models rely on a greater number of factors to make their predictions. Typically, they forecast trajectories by considering a maneuver-based approach [2].

III) Deep learning-based methods
Deep learning refers to a machine learning concept grounded in the utilization of artificial neural networks [10]. Nowadays deep learning-based methods reach State of the Art results, with especially the transformers producing extraordinary results [11]. The transformer architecture consists of an encoder-decoder framework. The encoder takes the input sequence and transforms it into a series of abstract representations. The decoder then generates the output sequence based on these representations (see Figure 3.2). The attention mechanism is a essential component of a transformer model. It allows the model to focus on relevant parts of the input sequence while processing each element of the output sequence. This mechanism enables the model to capture long-range dependencies in the data effectively. Compared to physics-based methods and

machine learning-based ones these kind of networks can perform reliable trajectory prediction for a longer prediction horizon [2].

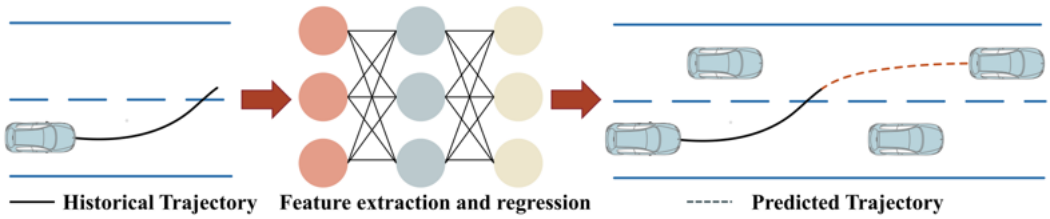


Figure 3.2: Deep learning network, extracting features from current situation and historical trajectory to decode a predicted trajectory [2]

IV) Reinforcement learning-based methods

Reinforcement learning agents learn to make decisions by interacting with an environment. They receive feedback in form of rewards or penalties and adjusts their actions to maximize the cumulative reward over time. In the trajectory prediction application, many methods leverage the Markov Decision Process to maximize the expected cumulative reward. Reinforcement learning-based trajectory prediction methods have seen significant advancements, offering a novel approach to comprehend complex policies in high-dimensional environments. Up to now reinforcement learning-based approaches are still very computationally expensive and hard to train [2].

3.3 Selected trajectory prediction methods

In this section, three State of the Art methods of trajectory prediction will be presented, that are used in research and industry. Each of these three implementations exhibits a unique and innovative architecture, highlighting important methods for trajectory prediction.

3.3.1 Trajectron++

I) Problem formulation of trajectory prediction

The objective is to create realistic trajectories for a time-varying number of agents. This means to predict the future states of all agents for the next timesteps (e.g. time horizon). Each individual agent is associated to a specific semantic class, such as car, bus or pedestrian. Geometric semantic maps are available providing the contextual information of the surrounding of the specific vehicle. These maps include details like road boundaries, sidewalks and crosswalks [12].

II) The approach of Trajectron++

For each scene a spatiotemporal graph, which represents the whole scene is created. This graph is fed into a similar structured deep learning architecture that predicts the evolution of specific node attributes and produces future trajectories for each agent [12].

III) Scene representation

A scene is represented using a spatiotemporal graph, where the agents are depicted as nodes and their interactions as edges. Each node in the graph is assigned a semantic class (e.g. car, bus, pedestrian). The presence of an edge e_{ij} between two agents, a_i and a_j, indicates the influence a_i has on a_j . This influence is quantified using the L2 distance metric. The resulting influence graph is directed, reflecting the asymmetric nature of interactions caused by varying perception ranges among agents. For instance, a car can perceive objects much further ahead compared to a pedestrian.

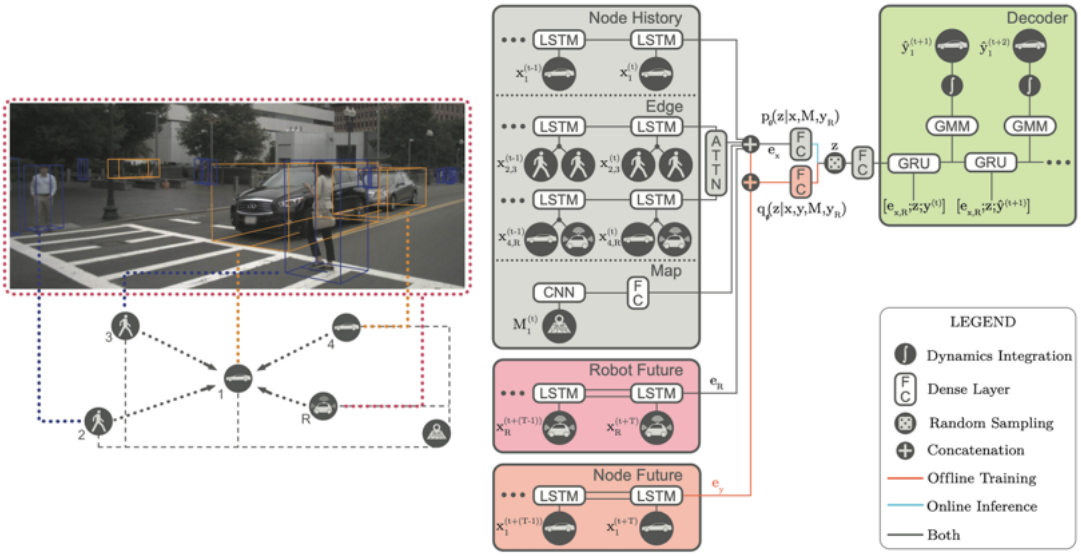


Figure 3.3: Trajectron++ approach. $Left$: Represents a scene as a directed spatiotemporal graph. Nodes and edges represent agents and their interactions, respectively. $Right$: The corresponding network architecture for Node 1 [12].

IV) Incorporating heterogeneous data

To enhance the model's understanding of the surrounding road environment, a high definition map is utilized, which contains detailed information about different semantic types in various map areas, such as driveable roads, road blocks, walkways, and pedestrian crossings. For each agent, a local map is processed by rotating and cropping it to match the agent's orientation. This transformation is performed using a convolutional neural network (CNN) with four layers. The CNN extracts relevant features from the rotated and cropped map, allowing the model to comprehend the current road context.

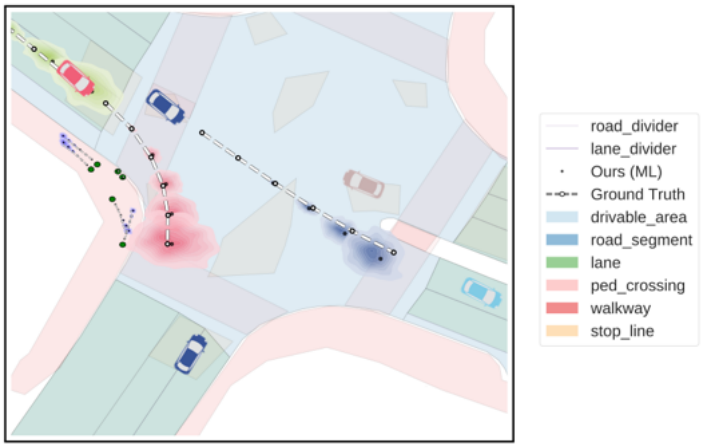


Figure 3.4: Trajectron++ high definition map representation. Regions within the map are assigned distinct labels, such as drivable_area, pedestrian_crossings, or road_segments etc. [12].

V) Modeling agent history

To capture the dynamics of a scene, the model employs a graph representation that encodes both the current state and the historical observed states of each node. This graph representation is then input into a Long Short-Term Memory (LSTM) Network with 32 hidden dimensions. The LSTM network allows to effectively model and learn the temporal dependencies, enabling it to comprehend the evolving dynamics of the scene

over time.

VI) Encoding agent interactions
To consider the influences of neighboring agents on the modeled agent, two steps are undertaken. Firstly, information from agents belonging to the same semantic class is gathered and aggregated. Subsequently, these accumulated states are passed through an LSTM network with 8 hidden dimensions. Consequently, all pedestrian-car edges have the same weights. Secondly, the encoding of all distinct semantic classes is once again aggregated to obtain an Influence Representation Vector. This vector incorporates all the encoded information from neighboring nodes and is achieved using an additive attention module. In a last step, the encoding of the node's historical data and the influence representation of other nodes are fused into a single backbone vector representation e_x, which serves as the final output. This allows for a comprehensive representation that captures both the agent's history and the impact of neighboring agents on its behavior.

VII) Multimodality
Trajectron++ addresses the challenge of multimodality by using the conditional variational autoencoder latent variable framework. This approach enables the model to generate diverse and probabilistic target distributions $p(y|x)$ by introducing a discrete categorical latent variable $z \in Z$. This latent variable z captures high-level behaviors, like the intended driving direction etc. The target distribution is defined as:

$$p(y|x) = \sum_{z \in Z} p_\psi(y|x,z)\, p_\theta(z|x) \tag{3.1}$$

ψ and θ refer to the weights of deep neural networks that are used to parameterize their corresponding distributions [12].

The use of a discrete latent variable z contributes to interpretability, as it enables visualization of high-level behaviors associated with each z through trajectory sampling. This resulting encoding process helps in learning the underlying structure of high-level behaviors and aids in generating diverse and realistic future trajectories for different contexts.

VIII) Producing dynamically-feasible trajectories
The information obtained up to this processing step, includes the agent's history, influence representation from neighboring nodes, and the features extracted from the high definition map, combined into a comprehensive representation vector e_x. Once the latent variable z is obtained, both z and the backbone representation vector ex are used as inputs to the decoder, which consists of a 128-dimensional Gated Recurrent Unit (GRU). The GRU is responsible for generating the parameters of a bivariate Gaussian distribution over control actions such as acceleration and steering rate for the agent. This output $u(t)$ can be integrate to obtain the predicted trajectory y.

3.3.2 VectorNet

I) Representing trajectories and maps
The VectorNet implementation establishes a uniform framework for representing both trajectories and high definition maps. The used maps are in the form of splines for lanes, closed shapes for regions of interest and points (e.g. traffic lights) with additional annotation of the semantic label (e.g. traffic light, speed limit). The trajectories for agents are directed splines with respect to time. All this information can be approximated as a sequence of vectors. To get a good approximation of the original map and trajectories small enough spatiotemporal sampling intervals are chosen. This one-to-one vector mapping allows to further process the data with a graph neural network [13].

All geographic entities can be closely approximated as polylines defined by multiple control points, along with their attributes. Each polyline P_j is made up by vectors v_i with the node features: $v_i = [ds_i, de_i, a_i, j]$. ds_i and de_i are the start and end point coordinates of the vector. a_i are to attribute features, like object type, timestamps for trajectories, or road feature type or speed limit for lanes. j is the integer id of P_j, indicating $v_i \in P_j$. In order to make the input features invariant the target agents location the coordinates are normalized around the center of the target agent [13].

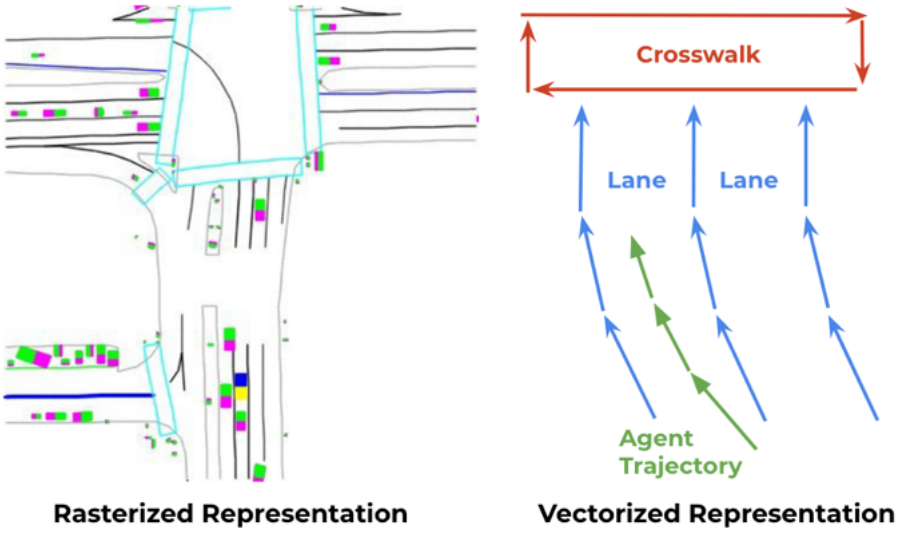


Figure 3.5: Illustration of the rasterized rendering (left) and vectorized approach (right) representing a high-definition map and agent trajectories [13].

II) Constructing the polyline subgraphs

Nodes with spatial and temporal locality can be grouped in a hierarchical approach. A single layer subgraph propagation operation for a polyline P with the following nodes $v_1, v_2, ..., v_p$ can be defined as following:

$$v_i^{(l+1)} = \phi_{rel}(g_{enc}(v_i^{(l)}), \phi_{agg}(\{g_{enc}(v_j^{(l)})\})) \tag{3.2}$$

where v_i^l is the node feature for l-th layer of the subgraph and v_i^0 is the input features v_i. The function $g_{enc}()$ transforms the node features and is implemented as an multi-layer perceptron (MLP). $\phi_{agg}()$ combines data from all adjacent nodes [13], which is the structured by a relational operator $\varphi_{rel}()$ with respect to a node v_i. This method is well suited for encoding structured map annotations and agent trajectories. It is done by embedding the ordering information into vectors and constraining the connectivity between subgraphs [13].

III) Global graph for high-order interactions

To model the higher-order interactions the relations of the polyline node features $\{p_1, p_2, ..., p_P\}$ are considered. This is done with a global interaction graph (see Figure 3.6):

$$p_i^{(l+1)} = GNN(\{p_i^{(l)}\}, A) \tag{3.3}$$

The $GNN()$ is a single layer of a graph neural network and $\{p_i^{(l)}\}$ is the set of polyline features. A is an adjacency matrix for the set of polyline nodes, that can be given a heuristic like spatial distances [13].

IV) Prediction

After the hierarchical graph is constructed, the network is trained on this multi-task objective:

$$loss = loss_{trj} + \alpha * loss_{node} \tag{3.4}$$

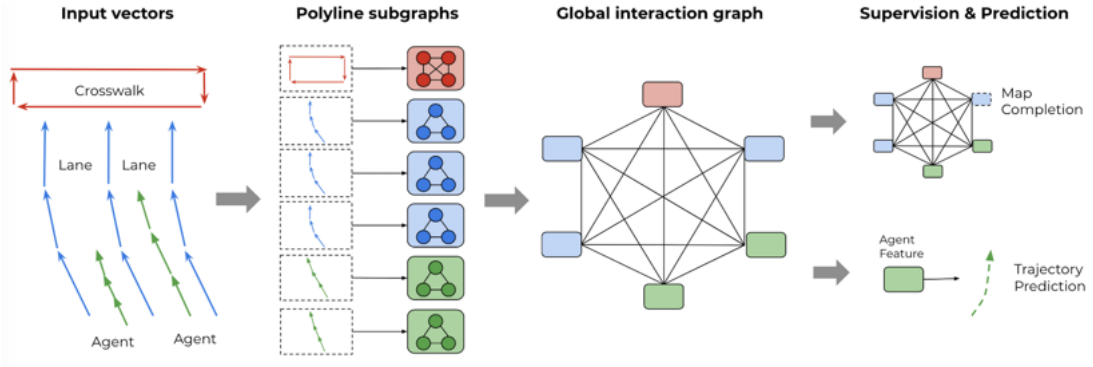


Figure 3.6: VectorNet overview: Observed agent trajectories and map features are represented as a sequence of vectors, and passed to a local graph network to obtain polyline-level features. The features are then passed to a fully-connected graph to model the higher-order interactions. The loss function is composed of two losses. $loss_{trj}$ predicting future trajectories from the node features corresponding to the moving agents and $loss_{node}$ predicting the node features when part of the features are masked out [13].

where $loss_{trj}$ is the negative Gaussian log-likelihood for the ground truth trajectories and $loss_{node}$ is the Huber loss [14] between the predicted node features and the ground truth masked features. These two loss terms are balanced with the scaling factor α. The predicted trajectories are parameterized as per-step offset and then rotated based on the heading of the target vehicle [13].

3.3.3 Watch-and-Learn-Net

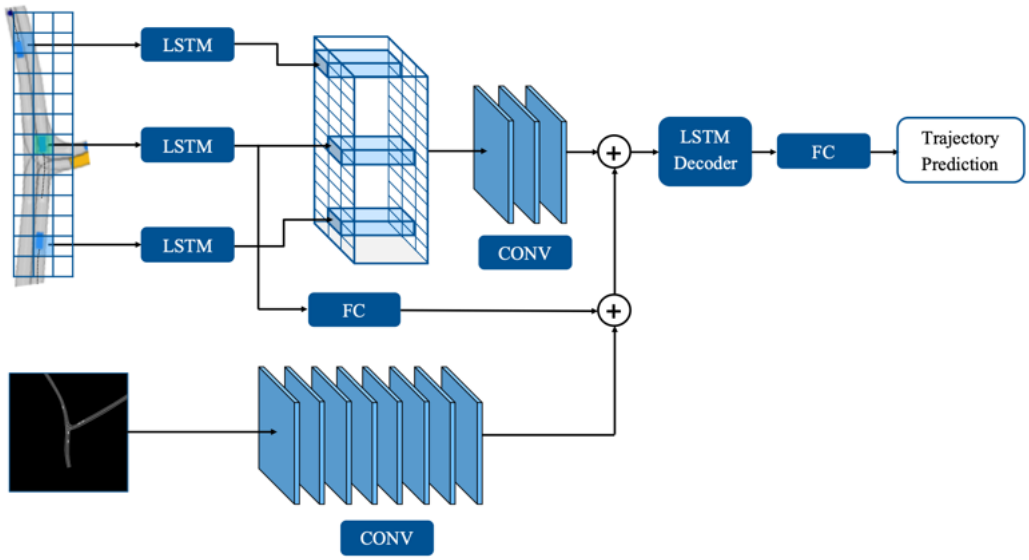


Figure 3.7: Network architecture of the base model building up on the work of Deo et al. [9]. The neural network consists of long-short term memory (LSTM), convolutional (CONV) and fully-connected (FC) layers. Semantic information is provided by bird's-eye-view images [15].

I) Overview

The Watch-and-Learn-Net (WaleNet) presents a new learning strategy that utilizes the concept of online learning during inference. This approach takes observed data during inference to dynamically optimize the underlying neural network in real time. Specifically, this means that during inference the network's weights are fine-tuned based on new observations. This can be useful if the vehicle has to dynamically adapt to new patterns in the data [15]. This method enables improvements beyond offline training. While offline training aims for generalized pattern recognition, this approach allows for targeted adaptation to specific behaviors of individual vehicles. The deep neural network is exclusively employed for predicting the behavior of a particular vehicle by overfitting to its observed behavior [15].

II) Encoding agent interactions

For the encoding of the agent interactions, the social dynamics between different road users are represented using a grid-based approach, illustrated in Figure 3.7. This grid, referred to as a "social tensor," is encoded through a convolutional neural network. Notably, the vehicle that is the subject of prediction is incorporated into this tensor, resulting in an enhancement of the approach's capabilities [15].

III) Scene representation

The utilized maps encompass various elements such as road geometry, drivable areas, lane configuration, and lane-specific motion directions. Additionally, the positions of sidewalks and crosswalks are taken into account [16]. To depict the street environment, a bird's-eye view of the road network is rasterized. These rasterized images undergo convolution (max-pooling) and are combined to extract higher-level map features before being input into the LSTM Decoder [15].

IV) Output representation

The network generates predictions in the form of time-dependent positions, accompanied by covariance matrices that quantify uncertainty. These predictions can then be utilized to compute collision probabilities for dynamic trajectory planning [15]. The prediction results for a specific vehicle v over the prediction time span N can be represented as following:

$$prediction_v = \{'pos\_list' : [\mathbf{P}]_{n\times 2}, \quad 'cov\_list' : [\mathbf{\Sigma}]_{n\times 2\times 2}\} \tag{3.5}$$

with

$$\mathbf{P} = \begin{bmatrix} (x_1, y_1) \\ (x_2, y_2) \\ \vdots \\ (x_n, y_n) \end{bmatrix} \quad \text{for } i = 1, 2, \ldots, N. \quad \text{Where } (x_i, y_i) \text{ are positions.} \tag{3.6}$$

and

$$\mathbf{\Sigma}_i = \begin{bmatrix} \sigma^2_{x_i} & \sigma_{x_i y_i} \\ \sigma_{x_i y_i} & \sigma^2_{y_i} \end{bmatrix}, \quad i = 1, 2, \ldots, N. \quad \text{Where } \mathbf{\Sigma}_i \text{ is the respective covariance matrix.} \tag{3.7}$$

V) Overfitting during online learning

After the initial pre-training of the model using the mean squared error loss function, a subsequent training phase takes place using negative log likelihood. During this process, the network's weights are fine-tuned based on the observations specific to each vehicle. This intentional overfitting to individual vehicle observations is performed independently for every vehicle. As it would be highly memory inefficient to load a unique network for each vehicle that should be predicted, only one single layer is switched (see Figure 3.8), which specifically is overfitted to this unique vehicle [15].

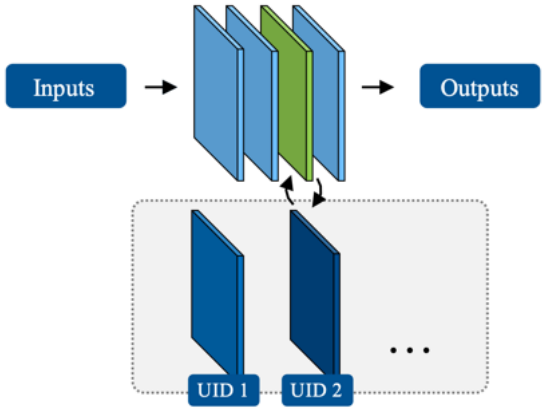


Figure 3.8: Layer switching: Each vehicle has a unique ID (UID) that references to its specific layer that is used for prediction and optimisation [15]

4 Methodology

In the following chapter the methodology of our trajectory prediction prototype is presented.

4.1 Objective of this thesis

The main objective of this thesis is to create an input data pipeline and to train two neural networks for trajectory prediction. The goal is to predict the trajectories of each traffic participant in a given scenario as accurately as possible, matching the actual ground truth trajectories. This is achieved by iterating through all traffic participants in the given scenario at each timestep and make a robust trajectory prediction for each of them using the previously trained neural networks.

To assess the results, the correctness of the predictions is evaluated at each timestep. This evaluation involves calculating two standard evaluation metrics (see 4.8) which are then averaged across all vehicles present. Additionally, a later-created video showcases both the ground truth and the predicted trajectories, displaying the displacement errors (evaluation metric) to quantify the accuracy and visually confirm the reliability of the predictions.

4.2 Key concepts derived from State of the Art

In this paragraph we explore the key concepts from the previous selection of State of the Art models, which are combined and partially integrated into our new architecture. Taking inspiration from the Trajectron++ implementation by Salzmann et al. [12], we've adopted a similar approach to handle semantic maps. We create a local semantic map that is positioned and oriented to the current agent. Then this cropped local map is further encoded using a multilayer convolutional neural network (CNN). This helps the model to extract and understand important information about the road context more effectively.
A key conceptual idea that is derived from the VectorNet [13] implementation is to train the model on a multi-task objective. In our approach, the neural network is designed to optimize two tasks simultaneously. As usual, the error between the ground truth and the predicted trajectory is calculated by a standard metric. The second goal of the network is to optimize the trajectory to only predict feasible trajectories. A feasible trajectory is defined as a trajectory that lies within a road segment, ensuring it is both possible and permissible to be traversed from the vehicle's current position.
Motivated by the WaleNet [15] implementation, we have structured our output representation similarly. Our predicted trajectory is organized as a list of the next 30 positions, mirroring their approach. In the same manner, we have integrated covariance matrices for each of these positions. These matrices serve to express the uncertainty linked to that specific part of the predicted trajectory.

4.3 Training environment

An **agent** in a trajectory prediction task is defined as a road user with cognitive abilities, such as a pedestrian, driver, or cyclist [9]. Over the course of this thesis cars, cyclists, motorcyclists and buses are abstracted and depicted as a vehicle object. In output visualisation they are displayed as a proportional scaled blue square.

Each agent has a **trajectory** represented as a sequence of two-dimensional coordinates. This trajectory is split into two parts: the observed trajectory and the future ground truth trajectory. The observed trajectory contains historical trajectory coordinates of the agent up to a specific time point, while the future ground truth trajectory comprises forthcoming coordinates of the agent's trajectory. The objective is to predict the future expected trajectory based on the observed trajectory for each agent [9].

The goal of **trajectory prediction** is to develop a model that can accurately forecast future trajectories based solely on the observed information available up to the present moment. This observed information typically includes the trajectory data recorded up to the current time, and the objective is to use this data to make robust predictions about the future movement of the vehicle [9].

The TUM-CommonRoad platform [1] delivers raw data in the form of **scenarios** that are used for the development, training, and testing of our trajectory prediction model. The platform offers many diverse scenarios captured and digitalized from a bird's-eye-view perspective, including cross-sections, highways, and urban environments. These scenarios are represented in the XML file format, capturing essential information about the road network and the static and dynamical obstacles present within them as depicted in figure 4.1. Each training scenario lasts between 5 to 15 seconds and has a specific sampling rate, typically set to 0.1 seconds [17].

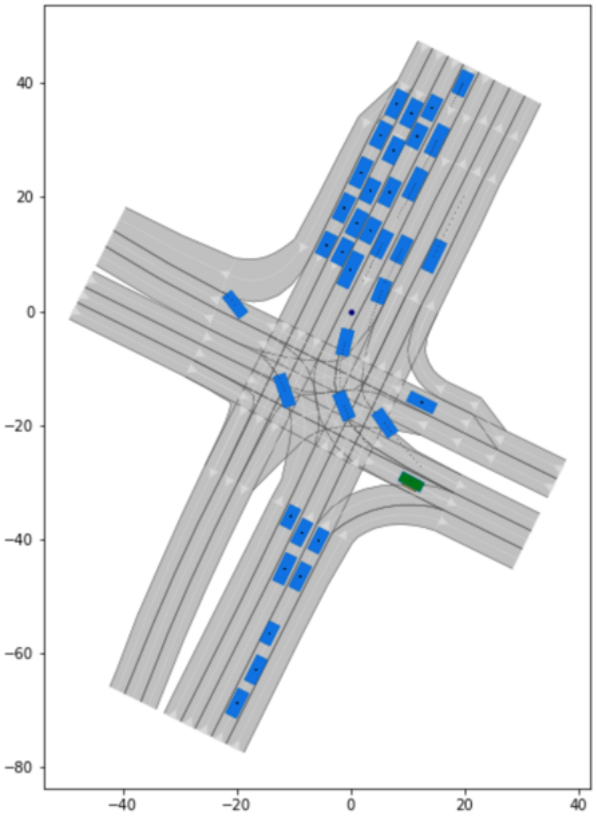


Figure 4.1: Bird's eye view scenario from TUM-CommonRoad used as raw data source [18].

A driveable road segment is defined as a **lanelet** element (see Figure 4.2), which is composed of left and right boundaries represented as polylines. Each boundary element holds four attributes: predecessor, successor, adjacentLeft, and adjacentRight. Due to the existence of road forks, lanelets can have multiple predecessors and successors (see Figure 4.2). Every lanelet encompasses a distinct driving direction [17].

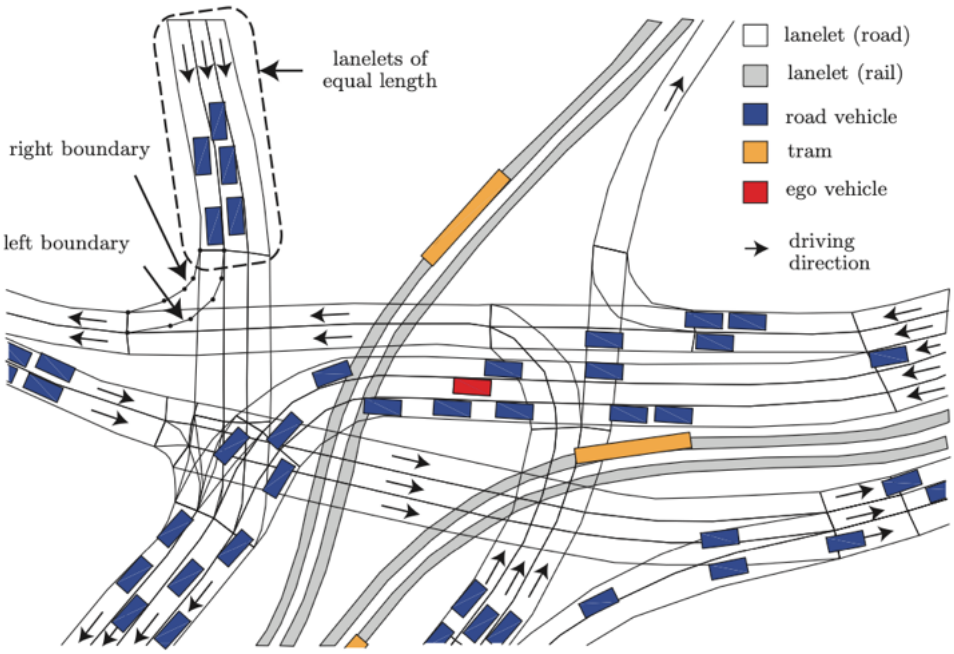


Figure 4.2: A TUM commonRoad scenario higlighting the underlying lanelet network, with each lanelet having multiple predecessors and successors and a distinct driving direction [17]

4.4 Input features and ground truth

During the data generation for each vehicle at every timestep, a data point is created, which contains two elements of the so called "ego vehicle": the input features and the ground truth. The input features are further divided into three features: the current velocity of the vehicle, a local map of the vehicle surroundings and the physical information of vehicles that are close to the ego. The ground truth is solely employed for training the network and evaluating the prediction accuracy. The three input features are utilized during training, evaluation and inference. The subsequent paragraphs will provide a comprehensive definition of the input feature composition and ground truth.

4.4.1 Ground truth trajectory (*groundTruthTrajectory*)

The ground truth of each single vehicle is represented by a two-dimensional array with dimensions $30x2$. The internal trajectory representation consists of 30 tuples (dx, dy) corresponding to a spatial displacement that, when concatenated, forms the entire trajectory. These displacements are sampled at intervals of $1/10$ seconds. In the following figures, the ground truth trajectory is visualized as a blue dotted line. Internally these sequential displacements are always in respect to the previous position. To reconstruct the actual trajectory, the angles between the consecutive displacements need to be summed up.

Both for training the neural network and evaluating the performance of the predicted trajectories it is necessary to consider a ground truth as reference. Hence, the ground truth for each vehicle is computed by iterating

through the entire time duration of a scenario. This ground truth contains, for every time step, the succeeding 30 local displacements. Furthermore, it includes both the global velocity and the local velocity. The local velocity is determined by the ego vehicle's orientation. The consideration of the local velocity is important to facilitate the contextual transformation of ground truth trajectory into an ego-centric view.

Conceptually, the calculation of global velocity involves the subtraction of the current two-dimensional position from the position at a future timestep. To convert this velocity from a global to an ego-centric (local) reference frame, a multiplication operation is employed. This multiplication utilizes a rotation matrix predicated on the prevailing orientation of the ego vehicle at the respective timestep.

$$\text{Global velocity vector at timestep } n: \quad \mathbf{v}^n_{\text{global}} = (v^n_{\text{global},x}, v^n_{\text{global},y}) \tag{4.1}$$

$$\text{Rotation matrix for timestep } n: \quad \mathbf{R}^n = \begin{bmatrix} \cos(\theta^n) & -\sin(\theta^n) \\ \sin(\theta^n) & \cos(\theta^n) \end{bmatrix} \tag{4.2}$$

$$\text{Local velocity vector at timestep } n: \quad \mathbf{v}^n_{\text{local}} = \mathbf{R}^n \cdot \mathbf{v}^n_{\text{global}} \tag{4.3}$$

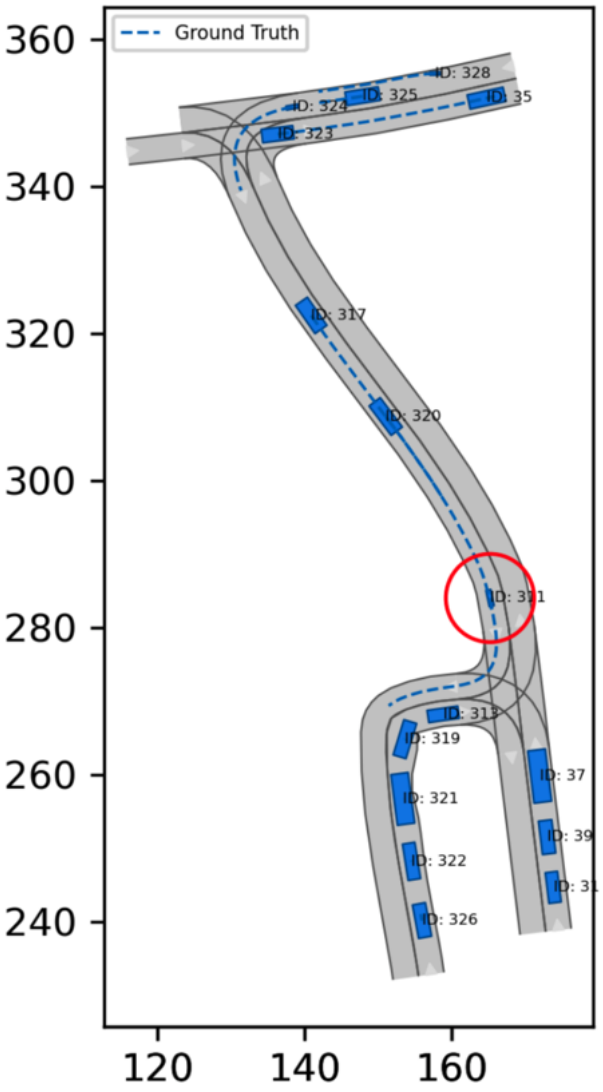
4.4.2 Ego velocity (*egoVelocity*)

The egoVelocity represents the current velocity of the vehicle at the timestep and is stored as a floating-point number in the unit of m/s. This feature plays an essential role in determining the vehicle's trajectory, making it a key physical property for the neural network to understand. The value of egoVelocity significantly influences the predicted trajectory.

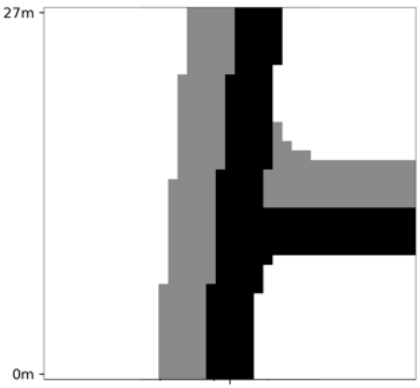
4.4.3 Local map in view (*mapInView*)

For each single vehicle the surrounding environment at the current timestep is stored in the local map (mapInView feature). It is oriented and cropped in the direction of the vehicle and has a viewing scope of 27 meter ahead. The information is stored in a tensor of dimensions $39x39$. The vehicle is centered at the bottom of the map at $[0, 20]$. Each pixel within the *mapInView* input feature is categorized into one of three distinct values:

- 0.0 indicates that there is no road at all in that specific location (represented as white, see Figure 4.3b).
- 0.5 indicates that there exists a road in this area, but this particular lanelet is either not reachable by the ego vehicle or is the opposite track (grey).
- 1.0 indicates that there exists a road, which could be a potential successor from the current lanelet (black).



(a) Scenario flensburg 29 with red circle highlighting the ego vehicle



(b) Centered mapInView of ego vehicle 311. This 39x39 tensor is a representation of the analysed Lanelet Network with black representing a possible road the agent can drive and grey representing the opposite track

Figure 4.3: mapInView input feature of specific car depiceted in global scenario

The process for the computation of the mapInView feature tensor involves iterating over the 39x39 tensor and calculating the global x and y positions for each specific pixel $[ix, iy]$ in the mapInView tensor (see Algorithm 1, line 2 and 3). After that, we determine whether the current pixel with its transformed global coordinates x and y lies on a driveable lanelet, non-driveavle lanelet or off road. This is done by comparing it with the different successor options (see Algorithm 1, line 4) which are returned after analysing the lanelet network at the current position and orientation. After that step the mapInView tensor is filled with one of three possible values (0.0, 0.5, 1.0) as described in section 4.4.3.

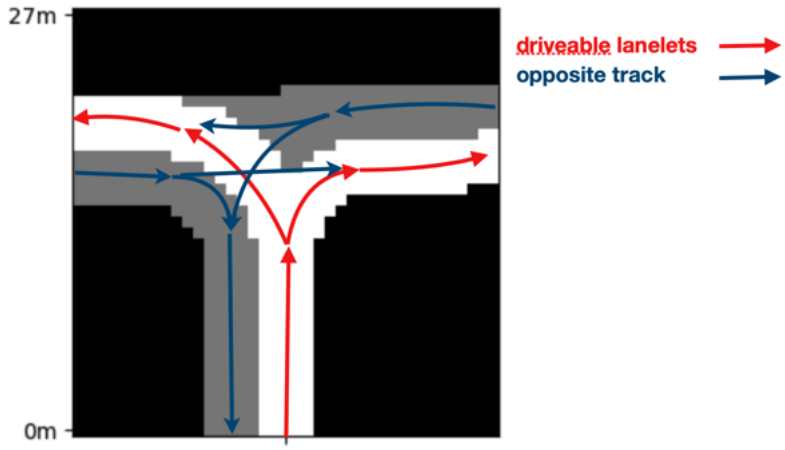


Figure 4.4: Illustration of the lanelet network analysis process involved in constructing the mapInView feature. The lanelet network is analysed to understand the underlying street situation and the possible successor paths. The creation of the mapInView feature involves determining whether each position (ix, iy) corresponds to the opposite track (blue) or a driveable lanelet (red) from the ego vehicle's current position.

The following pseudo code, shows how the mapInView feature tensor is computed by analysing the underlying lanelet network at the position of the ego vehicle:

Algorithm 1 Pseudo Code for getting mapInView feature

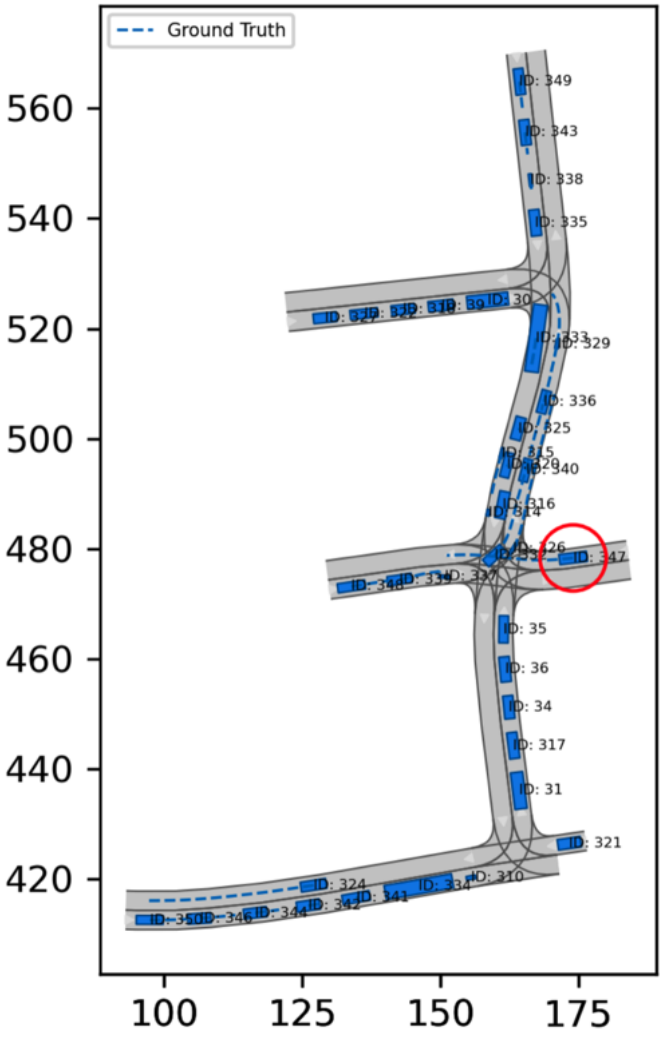
1: $mapInView \leftarrow initialiseMapInView()$ ▷ 39x39 Array
2: **for** iy *in* $len(mapInView$ **do**
3: **for** ix *in* $len(mapInView$ **do**
4: $reachableRoad \leftarrow LaneOps.fillReachable(succLanelets, ix, iy)$ ▷ Area reachable road?
5: $allRoad \leftarrow LaneOps.fillAll(allLanelets, ix, iy)$ ▷ Area road?
6: **end for**
7: **end for**
8: $mapInView \leftarrow np.where(reachableRoad != 0.0, reachableRoad, allRoad)$
9: **return** $mapInView$

4.4.4 Social information of cars in view (*socialInformationOfCarsInView*))

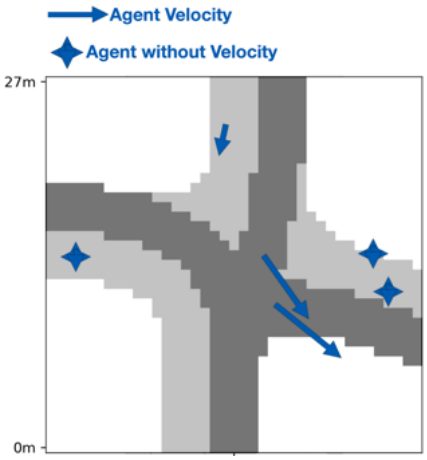
The social information provides important context about the presence and dynamics of neighboring vehicles, which is essential for accurately predicting the trajectory of the current vehicle. The *socialInformationOfCarsInView* is a tensor that contains the information of vehicles that are in close proximity of the ego vehicle. For each of these vehicles, their relative position and relative velocity with respect to the current vehicle are stored. To prioritize the significance of nearby vehicles in predicting the trajectory, the tensor is ordered by distance in an ascending manner. Hence, the closest vehicle occupies the first position in the tensor. The tensor has a maximum capacity of 20 vehicles, implying that up to 20 traffic participants are considered for

predicting a single trajectory. Nonetheless, this capacity can be adjusted as a hyperparameter, allowing for an increase in the number of vehicles considered.

In Figure 4.5b the blue arrows indicate the directed velocity of a vehicle in its field of view (mapInView dimensions). A blue star indicates a vehicle at this position which is not moving at the moment. The



(a) Scenario flensburg 69 with red circle highlighting the ego vehicle 342



(b) Centered socialInformationOfCarsInView of ego vehicle 342. The velocity of agents is depicted with a blue arrow. A blue star stands for an agent with no velocity at the moment

Figure 4.5: socialInformationOfCarsInView of specific car and global scenario. Remark: There are also motorcycles present in this scenario, which are barely visible (414, 326, 337)

socialInforamtionOfCarsInView feature is computed in two steps:

Grouping: In order to gather the interacting vehicles around the ego vehicle, all vehicles with their respective physical properties (position, velocity) are grouped based on a defined proximity radius r, as illustrated in the later Figure 4.6 where the proximity region is depicted in grey. Only vehicle agents a_n that satisfy the condition of having a distance $d_n <= r$ are considered to be in close proximity to the ego vehicle and are taken into account in the transformation step.

Transformation: In order to acquire the physical attributes of all other vehicles in relation to the ego vehicle, it is necessary to convert all the gathered properties into the perspective of the ego vehicle's global coordinate system. This ensures that we have the physical properties (position and velocity) of all vehicles with respect

to the ego vehicle. Therefore for each vehicle in proximity the following two two specific transformations are performed:

1. The relative position is calculated, with respect to the ego vehicles orientation, drawn as Dx in Figure 4.6.
2. The relative velocity is calculated, with respect to the ego vehicles orientation.

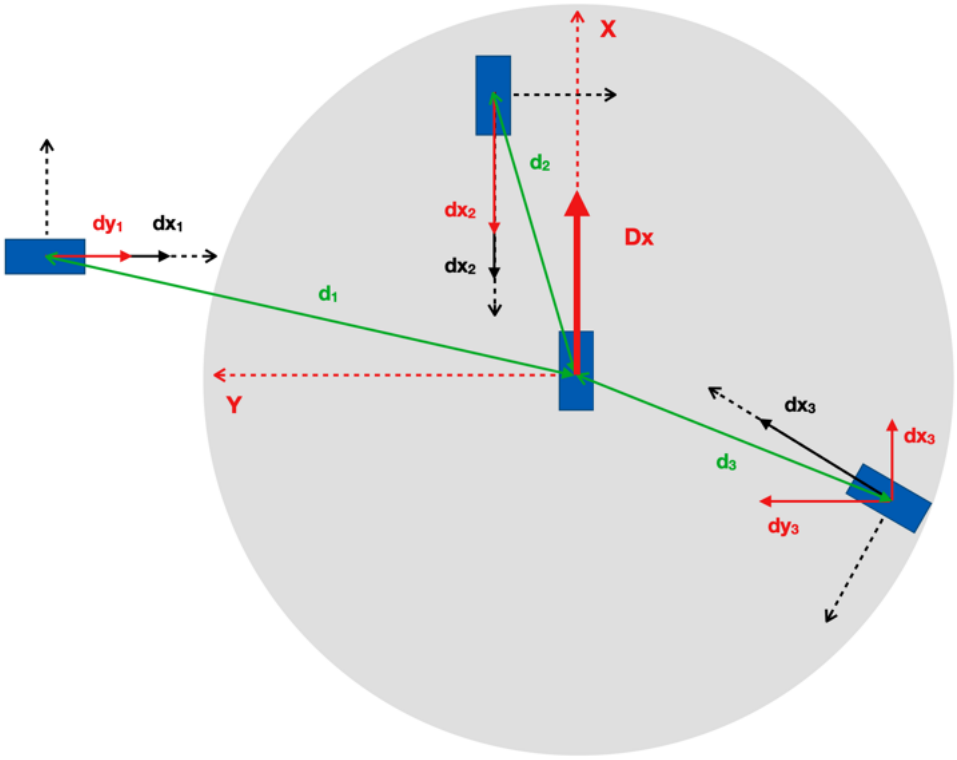


Figure 4.6: The diagram depicts four vehicles. The egoVehicle is in the center of the grey circle. The grey area illustrates the proximity radius, that defines the distance of surrounding vehicles that are considered in the socialInformationOfCarsInView feature. Vehicle 1 is not considered as an agent interacting with the ego vehicle. The whole process involves two steps: (1) Spatial grouping, where vehicles within a specified proximity radius are aggregated for analysis, and (2) Coordinate transformation, ensuring that all position and velocity attributes are aligned and transformed into the ego vehicle's global coordinate system.

The following pseudo code describes how the socialInformationOfCarsInView feature is created. This process involves first grouping vehicles that are nearby and subsequently adjusting their properties to align with the perspective of the ego vehicle (see Algorithm 2, line 5, line 6):

Algorithm 2 Pseudo Code for computing socialInforamtionOfCarsInView feature

1: $socInfOfCarInV \leftarrow initialiseSocInfOfCarInV()$ ▷ 20x4 Array
2: **for** $vehicle\ in\ vehiclesInProximity$ **do**
3: $velocity \leftarrow scenario.getVelocity(vehicle, timestep)$
4: $globalOrientation \leftarrow scenario.getOrientation(vehicle, timestep)$
5: $relativePosition \leftarrow calcRelPosition(position, egoPosition, egoOrientation)$
6: $relativeVelocity \leftarrow calRelVelocity(velocity, globalOrientation, egoVelocity, egoOrientation)$
7: $socInfOfCarInV.append(relativePosition, relativeVelocity)$
8: **end for**
9: $socInfOfCarInV \leftarrow sortSocInfOfCarInV(socInfOfCarInV)$
10: **return** $socInfOfCarInV$

After the physical properties (position, velocity) of all vehicles in proximity are transformed into the ego-centric perspective, up to 20 vehicles are stored in this tensor. A sorting algorithm is then employed based on the distance d_n of each vehicle to the ego vehicle (see Algorithm 2, line 9). The algorithm used for this purpose is merge-sort, a general-purpose method that arranges the vehicles in ascending order of proximity, with the closest vehicle being assigned the first position. This sorting step ensures data consistency.

4.5 Extracting the input features and ground truth from a scenario

Figure 4.7 illustrates a scenario unfolding across multiple time steps. For every vehicle within the scenario, the three input features and ground truth are computed and aggregated across all time steps. The figure highlights this process specifically for the three vehicles, identifiable by the red circles. This exemplifies the process of extracting input features and ground truth for an entire scenario, a procedure undertaken during the data generation phase. Recall, that this is done for all vehicles present in a scenario during run time of the data generation phase. Later, during the inference phase, a similar extraction process is done without collecting the ground truth as it is not available.

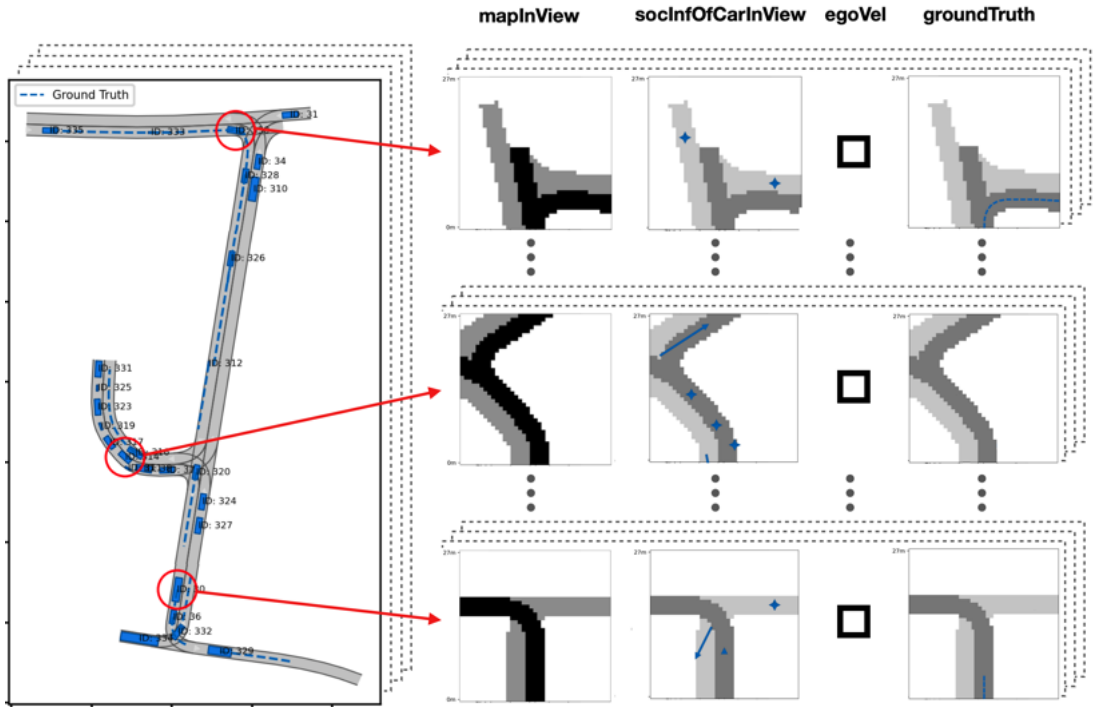


Figure 4.7: Feature extraction process across multiple time steps for a scenario for three vehicles (red circles). Input features and ground truth are computed and aggregated for each vehicle.

In the pseudo code of Algorithm 3, a high level view of the input feature and ground truth extraction is presented. A dual nested loop is employed to calculate the three input features for every vehicle agent a_n, $n = 1 \ldots N$ and at each time-step t_i, $i = 1 \ldots T$, within a given scenario.

As the first input feature the egoVelocity is determined (Algorithm 3, line 8). After that the orientation and position of the current vehicle is used to get a cropped and turned portion of the global map, the so called mapInView. This mapInView feature contains essential information about potential future lanelets the agent can drive as well as the opposite track. This is achieved by extrapolating multiple possible lanelet combinations from the current vehicle's position. Therefore, the underlying lanelet network is analyzed

Algorithm 3 Pseudo Code for extracting input features and ground truth from Lanelet Network

```
1: LaneletNetwork = scenario.getLaneletNetwork()
2: for timestep in scenarioTimesteps do                          ▷ iterate through all timesteps
3:     for vehicle in vehiclesPresentAtTimeStep do               ▷ iterate through all vehicles
4:         orientation = scenario.getGlobalOrientation(vehicle, time-step)
5:         position = scenario.getPosition(vehicle, time-step)
6:         succLanelets, curLanelet = calculateSuccessorOptions(position, orientation, timestep)
7:         allLanelets = calculateAllLanelets(curLanelet, timestep)

8:         egoVelocity = scenario.getEgoVelocity(vehicle, time-step)
9:         mapInView = getMapInView(succLanelet, allLanelets, vehicle, time-step)
10:        socialInformationOfCarsInView = getSocInfOfCarsInV(position, orientation, vehicle, time-step)

11:        inputFeatures = combineFeatures(egoVelocity, mapInView, socialInformationOfCarsInView)
12:        if outOfMapInView(groundTruth) == TRUE then
13:            groundTruth = approximateGroundTruth(groundTruth)
14:        end if
15:    end for
16: end for
17: return combineInputFeaturesWithGroundTruth(inputFeatures, groundTruth)
```

by exploring every possible successor option from the current vehicle position (Algorithm 3, line 6). By considering the position p_n and global orientation o_n of the current vehicle, we extrapolate multiple potential lanelet combinations, which allow us to determine the future lanelets and the opposite track in the mapInView object. The socialInformationOfCarsInView is computed by examining the context surrounding the particular vehicle agent a_n (see Algorithm 3, line 10). This involves utilizing the agent's current position p_n global orientation o_n for the computation .

Training data preparation involves concatenating input features (egoVelocity, mapInView, socialInformationOf-CarsInView) with their corresponding ground truth (Algorithm 3, line 17). Each data entry contains all of theses four elements. These are grouped in a dataset, which is used for training the subsequent neural networks. During the data generation process, every dataset entry is stored in a CSV file, which serves as a comprehensive source of input features and ground truth for the training of the neural networks.

4.6 Input feature compression

An encoder is used to compress the mapInView input feature tensor. The input feature mapInView has a dimension of $39x39$, yielding a combined count of 1521 data points, each encompassing a floating-point value. The mapInView feature can be regarded as a grayscale image, characterized by three discrete pixel values of 0.0, 0.5, and 1.0. As it is more efficient to use a reduced representation of the mapInView input feature, the encoder is used to compress the data. The encoder reduces the total size in latent space into the dimension $64x1$ (see Figure 4.8). This approach not only mitigates computational overhead but also utilizes on the inherent capacity of the encoder to fractionate spatial information into a compact form. Two different encoder architectures are presented in this section.

Autoencoder overview: An autoencoder is a classical deep learning architecture which consists of the encoder, the latent space and the decoder. It is capable of reconstructing the input data from the previous encoded latent space. An autoencoder is a type of unsupervised learning algorithm designed to reconstruct the input data with minimal distortion. Its primary purpose is data compression, which helps reduce storage

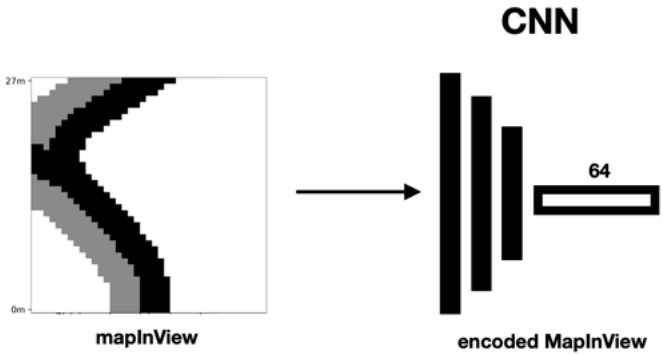


Figure 4.8: Encoder for mapInView input feature compression

requirements and facilitates faster computations. By removing redundant information and simultaneously reducing noise from the input data, an autoencoder effectively compresses the data. This deep learning architecture is particularly well-suited for image-related tasks, as it excels in learning efficient representations from images [19].

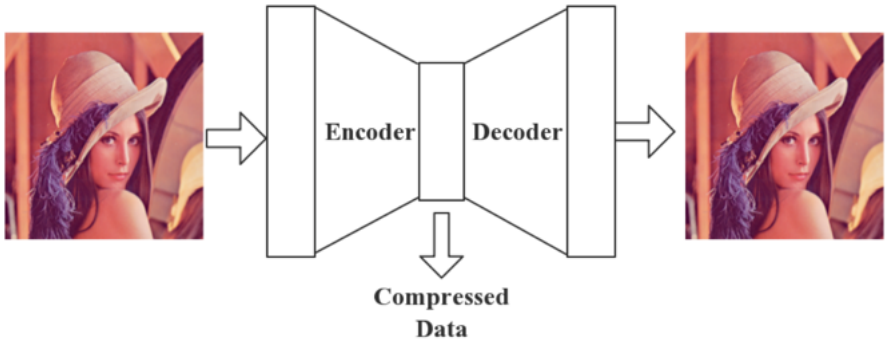


Figure 4.9: Autoencoder: Encoding input image in latent space(compressed data). Decoding latent space to output image and comparing output to input image [19].

In order to do an image compression, the size of the hidden layer (latent space) is strictly smaller than the output size. To train an autoencoder the reconstructed output is compared to the original input image. Subsequently, backpropagation is used to facilitate the learning of a lower-dimensional representation of the data. [19]. Once the autoencoder has been trained, one can just employ the forward pass (encoder) of the autoencoder to encode the data.

The encoder is the first part of an autoencoder and acts as a feature extractor. This part reduces the dimensionality of the input picture to a smaller representation in the latent space, which contains the important features for reconstructing the image later on with the decoder.
The latent Space has the compressed information of the image saved in a smaller vector representation. This vector contains the complex mappings from which the image can be reconstructed again.
The decoder is the last part of an autoencoder. This section tries to make sense of the information from the latent space and decompresses the data back to the original size of the image [20]. In the next two paragraphs, an architecture comparison is presented between two distinct autoencoders: one employing a linear methodology, while the other adopts a convolutional framework.

I) Building a linear autoencoder A linear autoencoder is a feed-forward network with 3 layers. The layers are fully connected [19]. In our setup the dimensionality of the hidden layer is $64x1$. This leads to an overall

compression ratio of $64/1521$, equating to less than 5%. This linear model consists of 3 layers in both the encoder and decoder sections. Each subsequent layer within the encoder reduces the node count by half.

Table 4.1: Linear encoder architecture

Layer	Type	Input Features	Output Features
1	Lin	1521	256
2	Lin	256	128
3	Lin	128	64

II) Building a convolutional autoencoder A convolutional autoencoder [21] extends the idea of a linear autoencoder by using convolutional layer instead of fully connected layers [19]. In our setup, a total of 6 convolutional layers are employed, each characterized by specific parameters including stride, padding, and kernel size. The convolutional layers are followed by a flattening operation and two subsequent linear layers. The hidden layer maintains a latent dimension of $64x1$, preserving the same compression ratio as in the linear counterpart. This ensures comparability in the performance of both architectures.

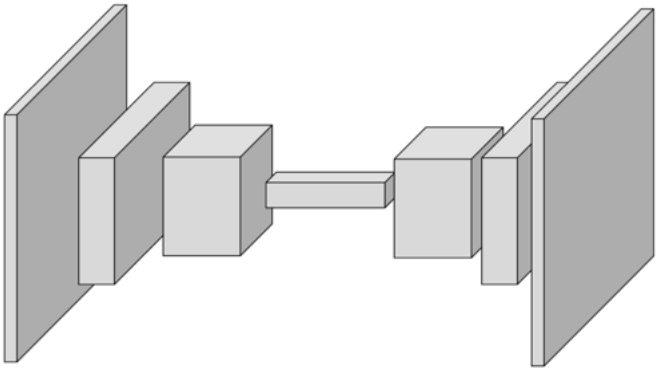


Figure 4.10: Convolutional autoencoder architecture with $39x39$ input dimensions and $64x1$ latent space dimension. For simplification only half of the layers are displayed [19]

Table 4.2: Convolutional encoder convolutional section architecture

Layer	Type	Kernel	Stride	Padding
1	Conv	3	1	1
2	Conv	3	1	1
3	Conv	3	2	1
4	Conv	3	2	1
5	Conv	3	2	1
6	Conv	3	2	1

Table 4.3: Convolutional encoder linear section architecture

Layer	Type	Input Features	Output Features
1	Flat	64x3x3	576
2	Lin	576	128
3	Lin	128	64

4.7 Trajectory prediction with a Multi-Layer Perceptron

The entire network architecture is a combination of two seperate networks: An encoder to compressed the mapInView input feature and a Multi-Layer Perceptron (MLP) with four sequential layers for the trajectory prediction. As a pre-processing step the mapInView feature is compressed with an encoder before it is passed on to the MLP input layer. Previously the encoder is trained on its own using an autoencoder structure as described in section 4.6.

The input layer of the MLP consists of 145 nodes that include the essential data such as the encoded mapInView (64), egoVelocity (1), and the flattened socialInformationOfCarsInView (80). After the input layer, the information is passed using fully connected layers with the dimensions: 512, 128, 64 and 60 as depicted in figure 4.11. Prior to entering the perceptron, the input features undergo normalization, as detailed in section 4.7.1.

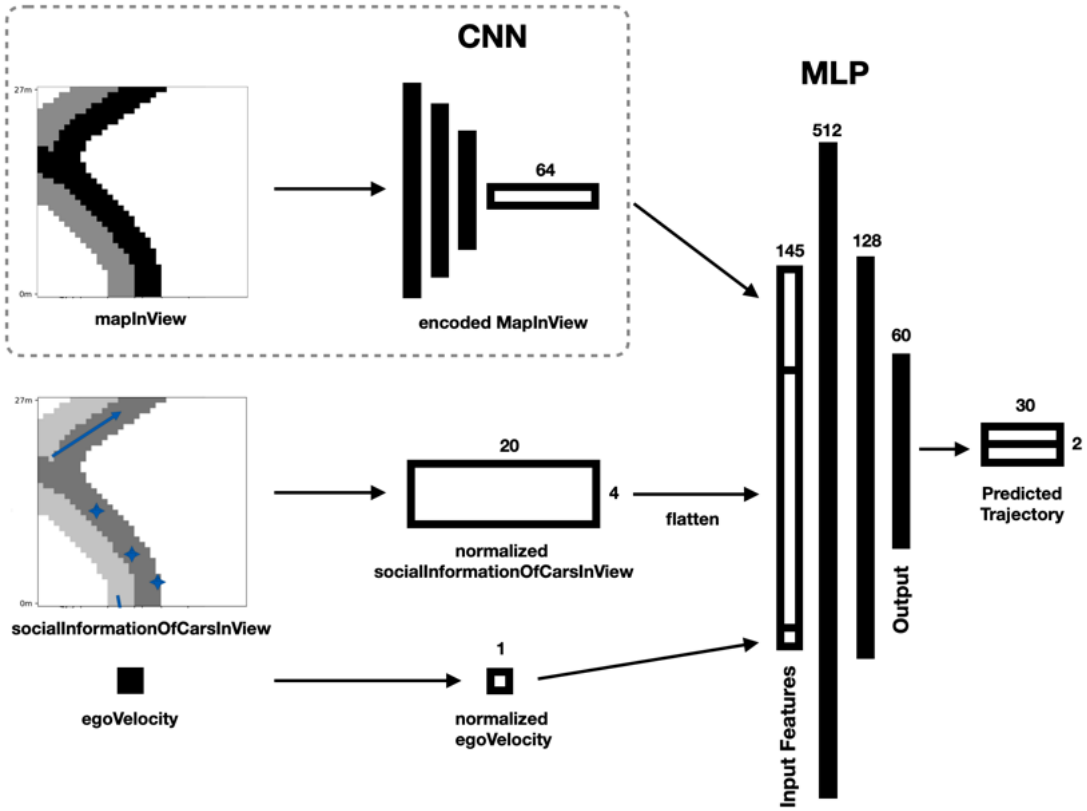


Figure 4.11: Shows the structure of the prediction forward pass of the whole network architecture (CNN encoder + MLP). The mapInView is encoded trough the CNN (defined in section 4.6) The other input features (socialInformationOfCarsInView, egoVelocity) are normalized before passed to the input Layer of the MLP. After the MLP the predicted trajectory is in the form of 30x2 (dx,dy) displacements

The following table (4.4) provides details about the structure of the MLP, including the type of each layer, the number of input features it receives, and the number of output features it generates.

Table 4.4: MLP layer architecture

Layer	Type	Input Features	Output Features
1	Lin	145	512
2	Lin	512	256
3	Lin	256	128
4	Lin	128	60

4.7.1 Normalizing the input features

It is difficult to train deep neural networks with non-linearities without normalizing the input of the layers. By utilising the power of normalization one can dramatically accelerate the training and improve the accuracy of deep networks [22].

Input normalization is a widely used technique in machine learning, by effectively eliminating the varying magnitudes between different features. This results in improved learning [23]. In this study, the normalization process is done on three of the four features: egoVelocity, socialInteractionsBetweenCars, and groundTruth-Trajectory. The mapInView is already normalized between 0 and 1. The normalization procedure consists of calculating the average values for each vector and subsequently dividing them by the representative norm. Following this step, all entries within the vectors are divided by their corresponding norm values.

4.7.2 Internal output representation

The resulting output of the MLP has the dimensions $30x2$ (see Figure 4.11), mirroring the structure of a ground truth trajectory. To ensure accurate interpretation and proper visualization, this output is subjected to a process of "unnormalization". Specifically, each element of the output is multiplied element-wise with its trajectory norm. These 30 displacements (dx, dy) then can be transformed into 30 points $\boldsymbol{P}$ with each of them having a covariance matrix $\boldsymbol{\Sigma}$, defining the uncertainty related to that specific position.

After the output of the MLP is unnormalized and converted to point representation a predicted trajectory for a specific vehicle v over the prediction time span n can be represented as following:

$$prediction_v = \{'pos\_list' : [\mathbf{P}]_{n\times 2}, \quad 'cov\_list' : [\boldsymbol{\Sigma}]_{n\times 2\times 2}\} \tag{4.4}$$

with:

$$\mathbf{P} = \begin{bmatrix} (x_1, y_1) \\ (x_2, y_2) \\ \vdots \\ (x_n, y_n) \end{bmatrix} \quad \text{for } i = 1, 2, \ldots, n. \quad \text{Where } (x_i, y_i) \text{ are positions.} \tag{4.5}$$

and

$$\boldsymbol{\Sigma}_i = \begin{bmatrix} \sigma^2_{x_i} & \sigma_{x_i y_i} \\ \sigma_{x_i y_i} & \sigma^2_{y_i} \end{bmatrix}, \quad i = 1, 2, \ldots, n. \quad \text{Where } \boldsymbol{\Sigma}_i \text{ is the respective covariance matrix.} \tag{4.6}$$

4.7.3 Training objective and loss functions

Our approach of the training is to minimize the mean squared error between the predicted and the ground truth trajectory by ensuring that the predicted trajectory adheres to the rule to stay on a road that it possible to drive. Effectively, this is a so called multi-task objective [13]. The first objective $loss_{trj}$ is for the predicted trajectory to match the ground truth. The second objective $loss_{road}$ is for the predicted trajectory, to follow a road that it can possibly drive as defined in 4.12. The overall loss function with its scaling factor α is defined as:

$$loss = loss_{trj} + \alpha * loss_{road} \tag{4.7}$$

The L1-Loss is used, because it is less sensitive to outliers (e.g. invalid data) [24]. Both the predicted and the ground truth trajectories are normalized by their mean values. $loss_{trj}$ is computed element-wise between the vectorized predicted and ground truth trajectory. We have developed a method to compute the

$loss_{road}$, which involves assessing the validity of a given trajectory and producing a specific scalar value called trajectoryOnRoad. This value is associated with the trajectory's adherence to a drivable road. In the visualization depicted in Figure 4.12, two predicted trajectories are displayed. If a trajectory remains exclusively on the drivable road, its corresponding trajectoryOnRoad value is 0. However, if the trajectory deviates from the road, the trajectoryOnRoad value becomes a positive floating-point number, wherein a higher value indicates a more unfavorable trajectory. This numeric value effectively measures the proximity of individual points within the predicted trajectory to the drivable road based on the information provided by the mapInView feature (as detailed in section 4.4.3). The function's operational logic is demonstrated with pseudocode Algorithm 4. Our implementation of this loss function is designed to be differentiable, ensuring the feasibility of training it using gradient descent [25]. To compute the trajectoryOnRoad value the vectorized output trajectory is transformed in a point representation (Algorithm 4, line 6). After that the distance between each point of the trajectory and the closest point on a road of the mapInView is determined (Algorithm 4, line 12). This computation uses the Chamfer distance. The Chamfer distance is computed by summing the squared distances between nearest neighbor correspondences of two point clouds. In our context, these point clouds correspond to the two dimensional predicted trajectory points and the points outlining the driveable road on the map. Further implementation details of this computation can be found in the Appendix, where comprehensive code is provided.

Algorithm 4 Pseudo Code for computing TRAJECTORYONROAD

1: **function** TRAJECTORYONROAD(trajectory, mapInView)
2: viewLength $\leftarrow 27$ ▷ mapInView scope
3: arraySubcornerLength $\leftarrow \frac{\text{viewLength}}{39-1}$ ▷ distance between two pixels
4: mapInView $\leftarrow$ flip(mapInView, $[0]$)
5: unNormedTrajectory $\leftarrow$ where(trajectory $\neq 0$, trajectory $\times$ norm.view(60), zeros_like(trajectory))
6: trajectory $\leftarrow$ convertRelativeDxDyTensorAbsolutePointListTorch(unNormedTrajectory) ▷ converting vectorised trajectory to absolute points
7: lengthInsideMap $\leftarrow$ lengthInsideMapInView(trajectory) ▷ length of trajectory inside the mapInView
8: roadPoints $\leftarrow$ where(mapInView $= 1$) ▷ get set of points in the mapInView
9: x, y $\leftarrow$ split(roadPoints, axis $= 1$)
10: points $\leftarrow$ stack($[$x $\times$ arraySubcornerLength, $\frac{\text{viewLength}}{2} -$ y $\times$ arraySubcornerLength$]$, axis $= 1$)
11: trajectory $\leftarrow$ view(trajectory, shape $= (30, 2)$)$[:$ lengthInsideMap$]$ ▷ converting pixel points to mapInView system
12: dists $\leftarrow$ chamfLoss(trajectory, points) ▷ chamfer dist (trajectory points -> mapInView points)
13: dists $\leftarrow$ where(dists < 0.6, dists $\times 0.0001$, dists) ▷ distance is smaller certain boundary?
14: dist $\leftarrow \frac{\text{sum(dists)}}{\text{lengthInsideMap}}$ ▷ calculating average distance
15: **return** dist, lengthInsideMap
16: **end function**

4.8 Evaluation Metrics

For the evaluation of the predicted trajectories the widely used standard metrics Average Displacement Error (ADE) and Final Displacement Error (FDE) are used [13] [26]. Both of these errors are calculated for each timestep and each vehicle and are than averaged to get a metric for the over all performance. The error calculation is done with the predicted positions and the ground truth positions.

The **Average Displacement Error** (ADE) value provides an overall measure of how well the predictions align with the ground truth data, with lower values indicating better prediction accuracy. For each vehicle $n = 1 \ldots N$ and each timestep $t = 1 \ldots T$ the root mean squared error (RSME) gets calculated, between the

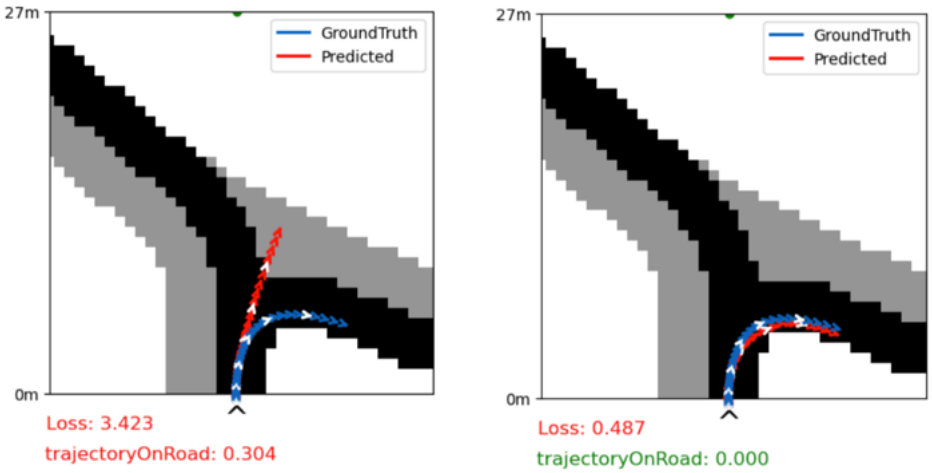


Figure 4.12: trajectorOnRoad value indicating the validity of the predicted trajectory. The trajectory on the left is predicted of road and therefore has a large trajectoryOnRoad value. The trajectory on the right is perfectly on the road and therefore has a value of 0.0.

predicted x_{ni} and the ground truth $\hat{x}$ positions ni. [26] This is done for the whole predicitonHorizon I. These RSME of all predicions for that timestep are than aggregated and then averaged.

$$\text{ADE} = \frac{1}{N * T} \sum_{n=1}^{N} \sum_{t=1}^{T} \sqrt{((x_{nt} - \hat{x}_{nt})^2 + (y_{nt} - \hat{y}_{nt})^2)} \tag{4.8}$$

The **Final Displacement Error** (FDE) measures the accuracy of the last predicted position compared to the last ground truth position for each obstacle. To compute the FDE, the RSME between the last predicted position and the corresponding ground truth position is determined [26]. Similar to the ADE it is then averaged over all vehicles in the scenario. Lower FDE values indicate higher accuracy in predicting the final positions, whereas higher FDE values indicate greater deviations from the ground truth.

$$\text{FDE} = \frac{1}{N} \sum_{n=1}^{N} \sqrt{(x_n - \hat{x}_n)^2 + (y_n - \hat{y}_n)^2} \tag{4.9}$$

4.9 Visualization

This section summarizes the visualization process of the predicted trajectories and their respective ground truth, as this is essential for visually evaluating the predicted trajectories

The visualisation of the predicted trajectory works as following: Each of the 30 predicted positions that are in the viewing scope of mapInView is displayed in the evaluation. These positions are showcased using a gradient shading ranging from yellow to red, where a reddish hue signifies (see figure 4.13) higher prediction certainty in that region. The width of the color transition, ranging from red to yellow, is determined by the covariance matrix Σ, as defined in section 4.7.2.

The core purpose of the ground truth visualization is to depict a visual portrayal of the forthcoming actual trajectory of all vehicles. This involves filtering the ground truth positions for the subsequent 30 time steps,

which are displayed in the evaluation image for every vehicle of that scenario. The ground truth trajectory is visualized as a blue dotted line (see figure 4.13).

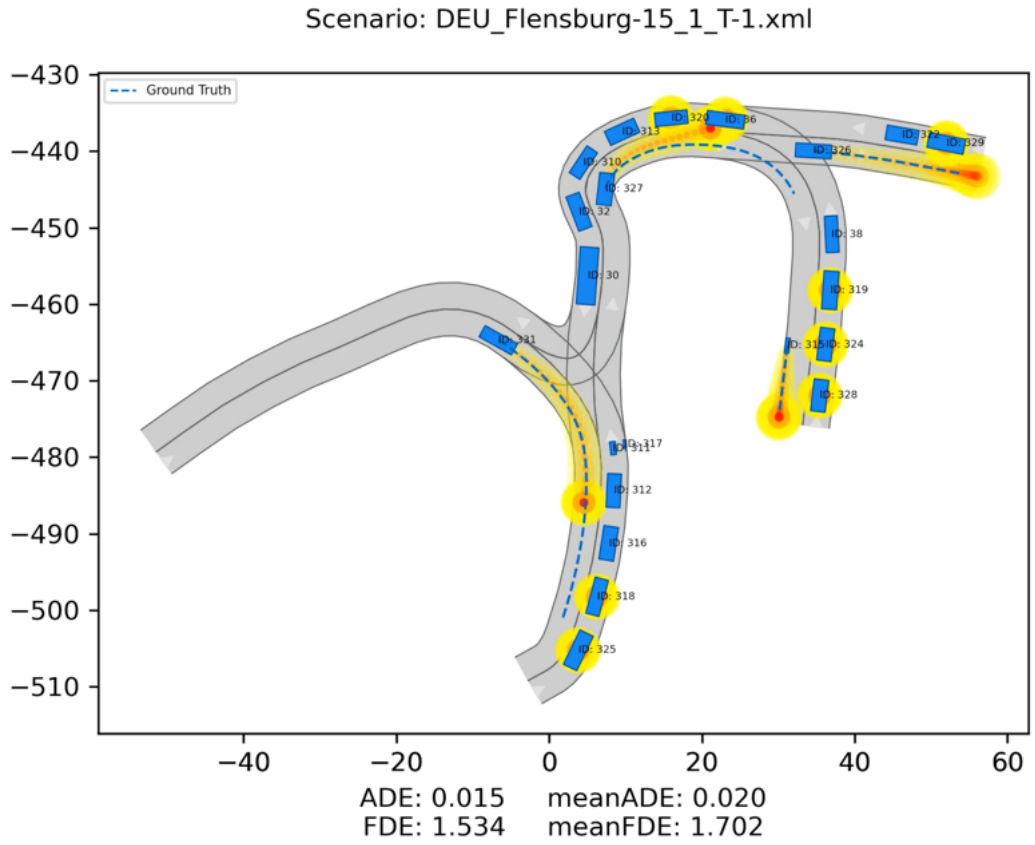


Figure 4.13: Visualisation of a trajectory prediction for multiple vehicles of a scenario in Flensburg. Predicted trajectories are visualised with a colour gradient between red and yellow. Read indicating areas with increased prediction, probability. The ground truth for each is depicted as a blue dotted line.

4.10 Data Generation, Evaluation and Inference

This section delves into the comprehensive assessment of the data pipeline, with specific emphasis on the pivotal stages of data generation, evaluation, and inference. A consistent and standardized structure is adopted throughout these processes to ensure simplicity and error reduction. The following paragraphs aim to show the differences between those stages

Generating the training data: The data generation starts with the loading of a commonRoad scenario.

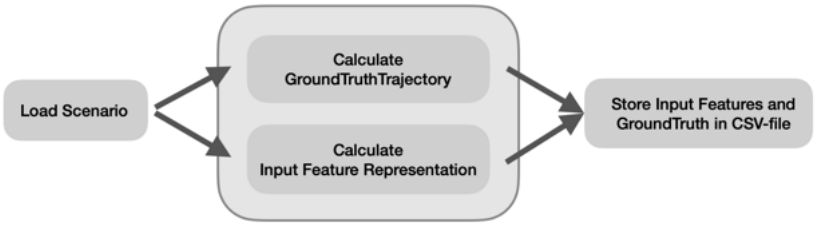


Figure 4.14: Data Generation

In the further steps the input features and groundTruth gets collected over all timesteps (e.g.100 steps)

and all vehicles that are present at this time-step. The number of vehicles varies, as vehicles can drive out of the scope of the scenario.The input features are the combination of the three previously defined (mapInView, egoVelocity and socialInformationOfCarsInView). After all input features and their respective groundTruth are calculated, this information is saved to a CSV-file, which than can be further used for training the prediction network. This file is structured by filename, carID, timestep, egoVelocity, mapInView, socialInformationOfCarsInView and groundTruth. The other values are not important for training but necessary to enable a unique access key for each data-point.

Evaluation: For the evaluation each data point gets loaded and processed in a similar fashion like for the training. The only difference is that the ground truth trajectory is masked for the trained prediction network. After defining and loading the previous networks (convolutional encoder and linear model) the prediction is made with the 3 visible elements of each data-point(egoVelocity, mapInView, soicalInformationOfCarsInView). Each prediction trajectory contains a position and covariance list, which both are later on utilised to further process the data for motion planning and also visualisation.

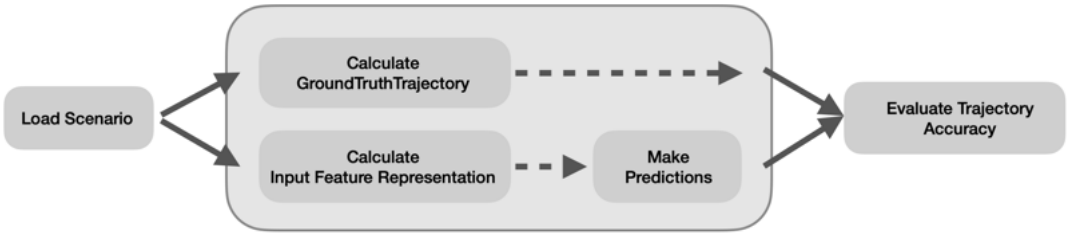


Figure 4.15: Evaluation

Inference: During the inference, various sensors such as LIDAR, cameras, and other internal vehicle sensors, along with the navigation system, collect data. This data is then transformed into an input feature representation (egoVelocity, mapInView, socialInformationOfCarsInView) This transformation occurs within the internal computer of the vehicle. Once the data is pre-processed, the prediction is made using pre-trained neural networks.

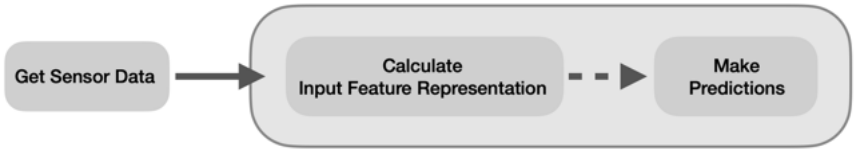


Figure 4.16: Inference

5 Results

In this chapter the results of the Autoencoder and Multi-Layer Perceptron are presented.

5.1 Autoencoder

An encoder is utilized to reduce the input dimensionality of the mapInView feature for each vehicle. This encoded representation not only accelerates inference speed across the entire network due to the reduced input dimensionality but also contributes to enhancing network generalization capabilities. To compare the results of the two different autoencoder architectures (see section 4.6), the mean squared error is calculated between the original and the reconstructed mapInView (see 4.4.3).
The two autoencoders are both trained on 28 scenarios with roughly $80,000$ grey-scale images (see 4.4.3). The goal is to compress the original image with a resolution of $39x39$ pixels through an encoder to downsample the input feature size to 64 pixels. The compressed result of the encoder (forward pass) is used as an input feature for the Multi-Layer Perceptron (see 4.11). The images are fed batch-wise into the encoder. After the encoder, the image is represented in to the latent space and then then further decode up to the original resolution of $39x39$.

5.1.1 Quantitative and qualitative comparison between linear and convolutional architecture

Two different network architectures are compared and challenged to figure out the best architecture. The results of both autoencoders are averaged over a test set of $11,600$ images to get a more general comparison.

Quantitative: Table 5.1 displays a comparison between the linear and convolutional autoencoders, as outlined in section 4.6. This comparison is based on the validation loss, which is computed using the mean squared error, and the time taken for inference during each forward pass of the autoencoders. To ensure a meaningful comparison, both autoencoder architectures maintain a latent space dimension of 64.
Although the validation loss of the linear autoencoder is reasonable good, it falls short of the performance achieved by the convolutional autoencoder. The straightforward design of the linear model offers the advantage of faster computation during inference, with a speed approximately 40% faster than the convolutional model. Noteworthy is the linear model's superior performance in trivial situation, such as a simple straight road segment without intersections.
In comparison to the linear autoencoder, the convolutional autoencoder achieves a 25% reduction in test loss. This suggests that the convolutional approach excels in creating a compressed representation within the latent space from the original image. However, it lags in terms of inference speed, being around 40% slower than the linear variant. The convolutional architecture seems to perform even better with more complex street scenarios ahead, like e.g. complex street crossings. Particularly in rare and unconventional situations the convolutional architecture surpasses the linear architecture's capabilities (see Figure 5.1).

Table 5.1: Comparison between Linear and Convolutional Autoencoders

Comparison Method	Linear Autoencoder	Convolutional Autoencoder
Mean Squared Error	0.0032431	0.002434
Inference Time [s/image]	0.0032185	0.0045662
Latent Space Dimension	64	64

Qualitative: In figure 5.1 four visual comparisons between an original mapInView input feature and the reconstructed images generated by the respective autoencoder is shown. The corresponding validation loss values are illustrated bellow each mapInView image, where the color green indicates the better accuracy of one architecture over the other. Evidently, in more complex scenarios, the convolutional neural network (CNN) architecture demonstrates superior performance in the process of map encoding.The results from both autoencoders provide robust output, and in both cases the original road scenario is clearly recognizable. It is noticeable that with the linear architecture several edges are washed out and there are not so sharp dividing lines between the roads and opposite track. The convolutional architecture on the other hand offers much sharper reconstructed images and does not show the washed out edges. These screenshots encapsulate the essence of the analysis and serve as tangible representations of the comparison between the linear and convolutional autoencoder architectures.

5.1.2 Architecture selection

The convolutional architecture is used for the entire network as the encoder for the mapInView input feature, due to its exceptional ability to capture and compress complex scenarios effectively. While the linear autoencoder demonstrated respectable validation Loss and faster inference times, it was surpassed by the CNN with a substantial 25% lower loss. This emphasizes the CNN's proficiency in compactly representing the original image information within the latent space. The preference for the CNN architecture stems from its adeptness in managing less common and intricate scenarios, prioritizing robustness over marginally quicker inference times and slightly better performance in trivial cases. This choice ensures that the model can handle a wider range of real-world scenarios and provides a more robust solution for the task at hand. The visual comparison between both architectures further emphasizes the CNN's superiority, reinforcing its performance metrics. The selection of the convolutional architecture is grounded in its capacity to effectively manage diverse scenarios, showcasing resilience and dependability.

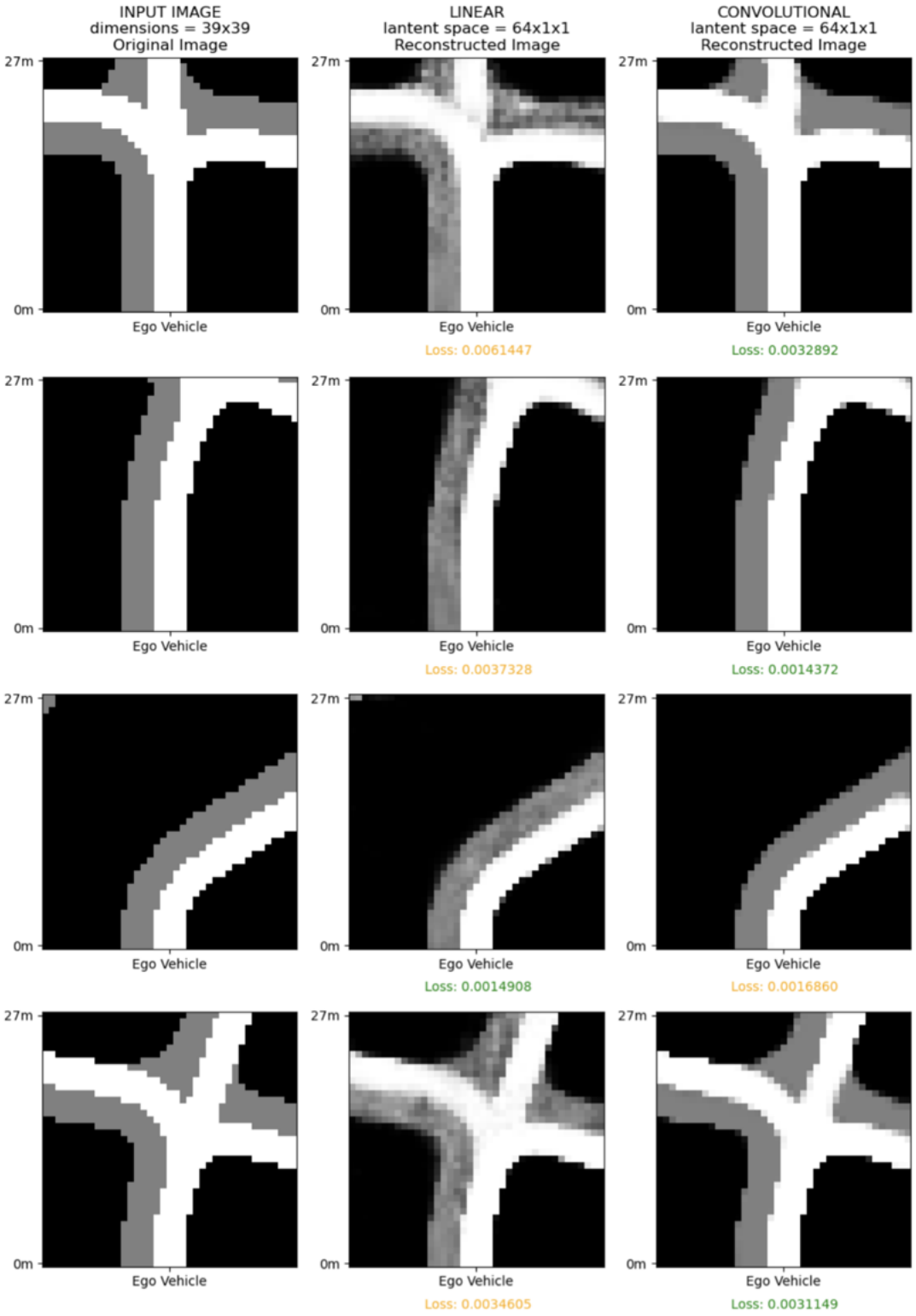


Figure 5.1: Visual comparison between the original images (left column) and the reconstructed images using a linear (center column) and a convolutional (right column) autoencoder. The validation loss (Mean Squared Error) values are compared. The convolutional autoencoder excelling in complex scenarios (indicated in green). The decision to use the convolutional architecture is based on its ability to handle a wide range of situations, demonstrating robustness and reliability.

5.2 Multi-Layer Perceptron

We present our results of the training process, the inference runtime and a quantitative and qualitative evaluation.

5.2.1 Training process

The Multi-Layer Perceptron, shown in figure 4.4 was trained on a dataset containing 131 scenarios. Looping over the scenario time steps and all vehicles present in the scenarios, this leads to roughly $230,000$ data entries (egoVelocity, mapInView, socialInformationOfCarsInView, groundTruth). This was trained on a $Nvidia\ Tesla\ V100$ GPU with a batchsize of 128 over 25 epochs using the $ADAM$ optimizer and roughly took 30 hours (see figure 5.2). For training a train dataset and a validation dataset is used. A fixed group of 3 chosen validation images was printed out after each epoch to visually report the training process.

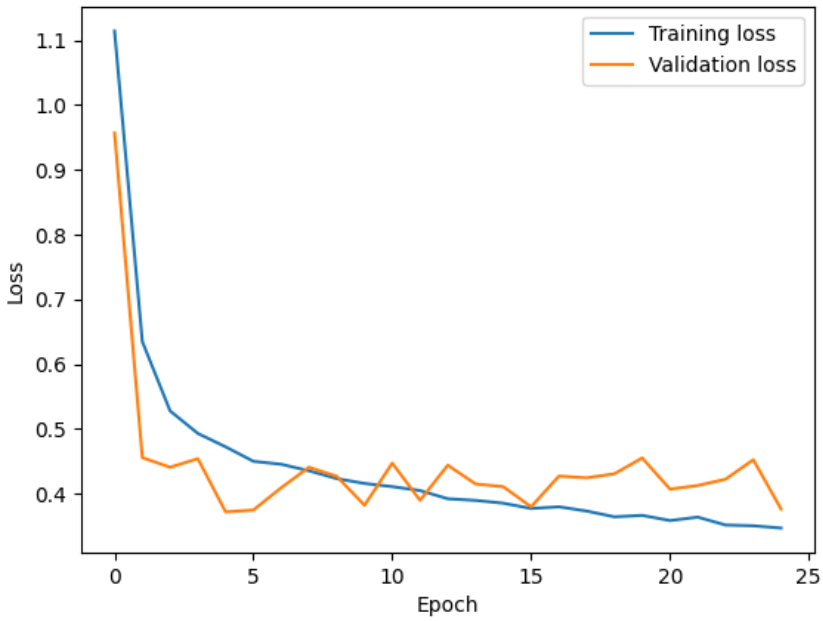


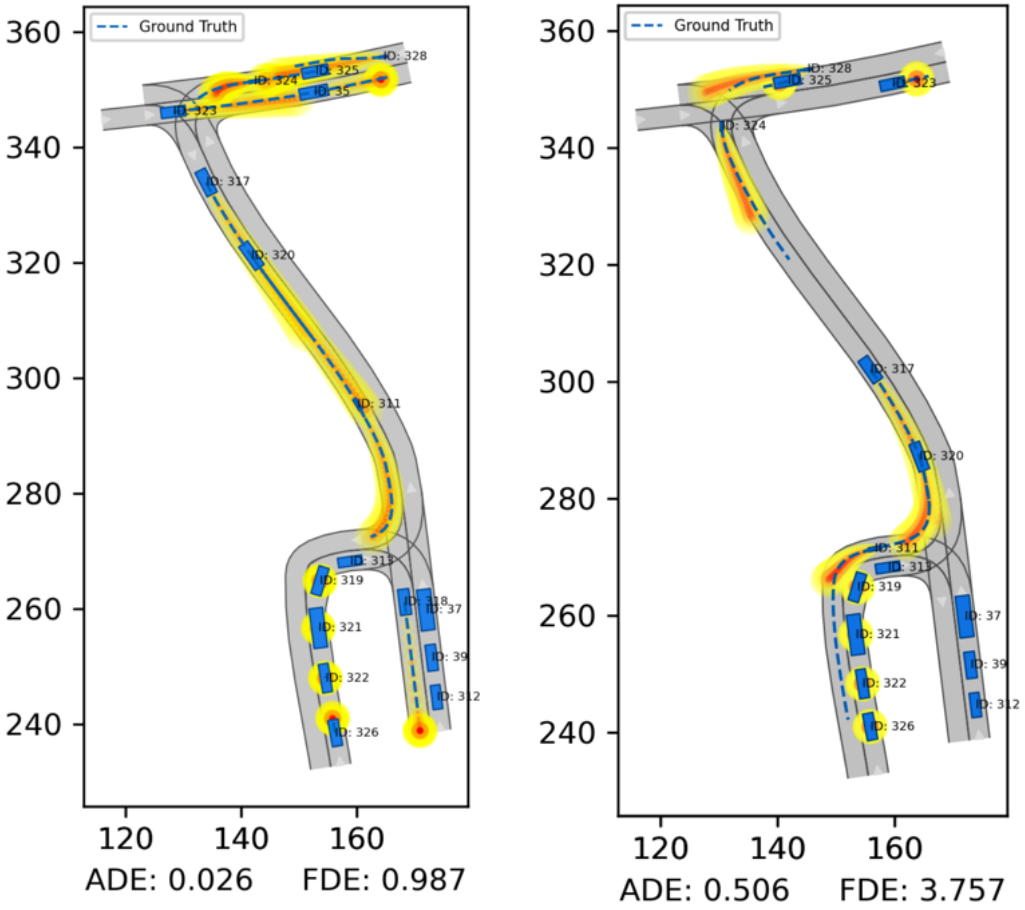
Figure 5.2: Training of the Multi-Layer Perceptron over 25 epochs

5.2.2 Qualitative and quantitative evaluation

We evaluate the performance of the Multi-Layer Perceptron predictions with qualitative and quantitative results.

Qualitative: Following the extensive training of the Multi-Layer Perceptron (MLP) on the large dataset, an assessment of the performance of the two networks was conducted across comprehensive scenarios. The primary objective of this evaluation was to validate the visual coherence of the predictions. The resulting outcomes illustrate the ground truth represented by a blue striped line, consistent with previous representations. The predictions for each vehicle visualized as a gradient spanning from yellow to red, with an increasing intensity of red denoting higher prediction certainty within the corresponding region. The average Final Displacement Error and Average Displacement Error 4.15 are computed for all vehicles within the scenario. Their values are fluctuating across successive timesteps. The subsequent figure 5.3a showcases two distinct timesteps within the same scenario. In both timesteps the sequential multi-agent prediction

output is visualized. This is done on a validation scenario in Flensburg. As visible in figure 5.3a the trajectory of car 311 is accurately predicted. The right turn manoeuvre, which the vehicle will perform several timesteps ahead, is correctly predicted.



(a) Prediction of a validation scenario in Flensburg at timestep 1

(b) Prediction of a validation scenario in Flensburg at timestep 10

Figure 5.3: Multi agent prediction evaluation at two timesteps of the validation scenario Flensburg-29.xml. ADE and FDE (see 4.8) averaged over all vehicles present at the current timestep. A 10 Hz video of this multi-agent prediction is created.

Quantitative: Figure 5.4 shows the evolution of the training results after the 1-st, the 2-nd, the 5-th, the 10-th and the 20-th epoch. The three images plotted after each epoch of training originate from a validation dataset. Each image displays both the ground truth trajectory (depicted in blue) and the predicted trajectory (depicted in red). Additionally, the associated loss and trajectoryOnRoad value are visible for each respective prediction. As the epochs progress, the predicted trajectories gradually converge toward the actual ground truth trajectories. Furthermore, the trajectoryOnRoad value diminishes, indicating that the trajectories become more plausible and are aligned with the drivable road conditions.

Table 5.2 shows the results of the comparison between our method and the WaleNet and Lanelet prediction methods, both integrated in the TUM CommonRoad tool. The comparison is based on 12 different test scenario located in Flensburg, Lohmar and Reutlingen. The 3 seconds ADE and FDE error metrics as described in section 4.8, are computed for each scenario on its own. The values are averaged across the time span of a given scenario, including all vehicles within it. In each line in the table 5.2, the network with the best predicting results is highlighted. This can be different between ADE and FDE measurements. The

comparison demonstrates that our prediction achieves superior performance over the WaleNet and Lanelet prediction. This is evident from the notably lowest average ADE and FDE values.

Table 5.2: Comparison of our method with the WaleNet and Lanelet prediction methods from TUM CommonRoad. ADE and FDE with 3 seconds trajectory prediciton length.

Scenario (XML)	**ADE / FDE in meter**		
	WaleNet	**Lanelet**	**Ours**
Flensburg-1	0.76 / 1.79	1.22 / 2.85	**0.53** / **1.36**
Flensburg-14	1.16 / 2.96	1.66 / 3.93	**0.54** / **1.62**
Flensburg-15	1.25 / 2.98	0.67 / 1.50	**0.50** / **1.40**
Flensburg-36	1.09 / 2.55	2.14 / 4.90	**0.81** / **1.83**
Lohmar-3	**1.12** / **2.67**	2.24 / 5.14	1.34 / 3.90
Lohmar-7	**0.77** / **1.85**	1.13 / 2.80	0.85 / 2.92
Lohmar-11	0.97 / 2.46	1.07 / 2.41	**0.70** / **1.68**
Lohmar-12	**1.26** / **2.99**	2.84 / 6.33	1.37 / 3.72
Lohmar-26	0.70 / **1.57**	1.14 / 2.61	**0.63** / 1.63
Lohmar-32	0.98 / 2.26	1.66 / 3.84	**0.74** / **2.16**
Reutlingen-1	0.82 / 1.97	1.42 / 3.30	**0.57** / **1.55**
Average	0.98 / 2.36	1.56 / 3.60	**0.78** / **2.16**

5.2.3 Inference runtime evaluation

The runtime performance test is done on the CPU. We do our inference computation on a $1,4GHz\ Quad-Core\ Intel\ Core\ i5$ CPU (2019). In Figure 5.5, we present a visual representation of the separate forward passes through the encoder and the MLP, along with the cumulative inference time for the complete prediction process. The aggregate time required for a single prediction (forward pass through encoder and MLP) is approximately $3.6ms$. This observation measurement the pronounced computational efficiency achieved by our network architecture. This calculation excludes the duration required for the extraction and computation of the input features.

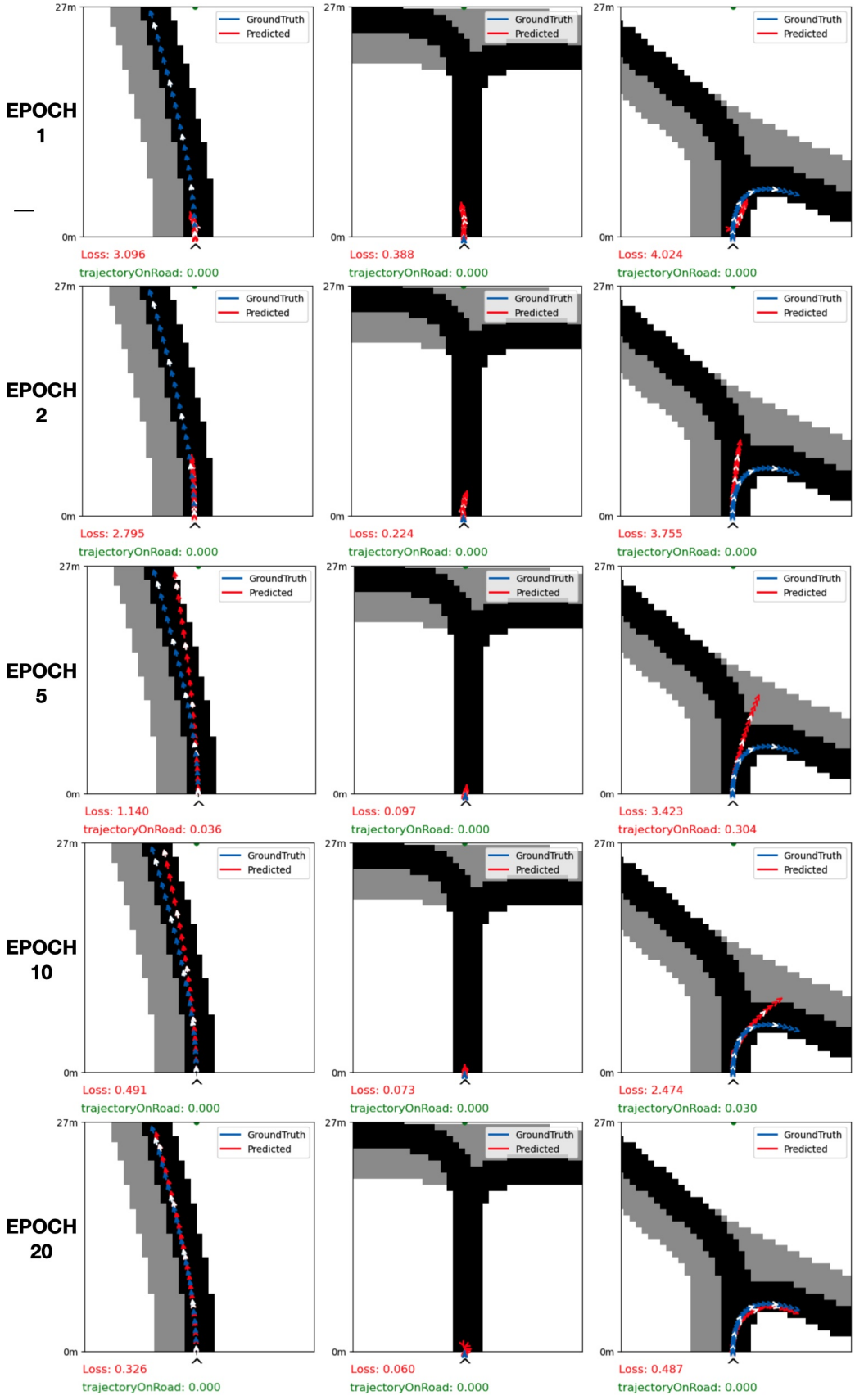


Figure 5.4: Training evaluated on validation dataset. Epoch 1: The predicted trajectory in all three cases has a large offset to the ground truth. The trajectoryOnRoad values are 0, because the trajectories are on the road). Epoch 5: The predicted trajectories in all 3 cases now have a small offset to the ground truth. The trajectoryOnRoad is close to 0 in the first column, because this trajectory is almost perfectly on the road. The trajectoryOnRoad value in the third column is substantially bigger indicating that this trajectory is off road and not feasible. Epoch 20: the predicted trajectory is in all three cases are now almost perfectly aligned with the ground truth. The corresponding trajectory on road values are 0.0.

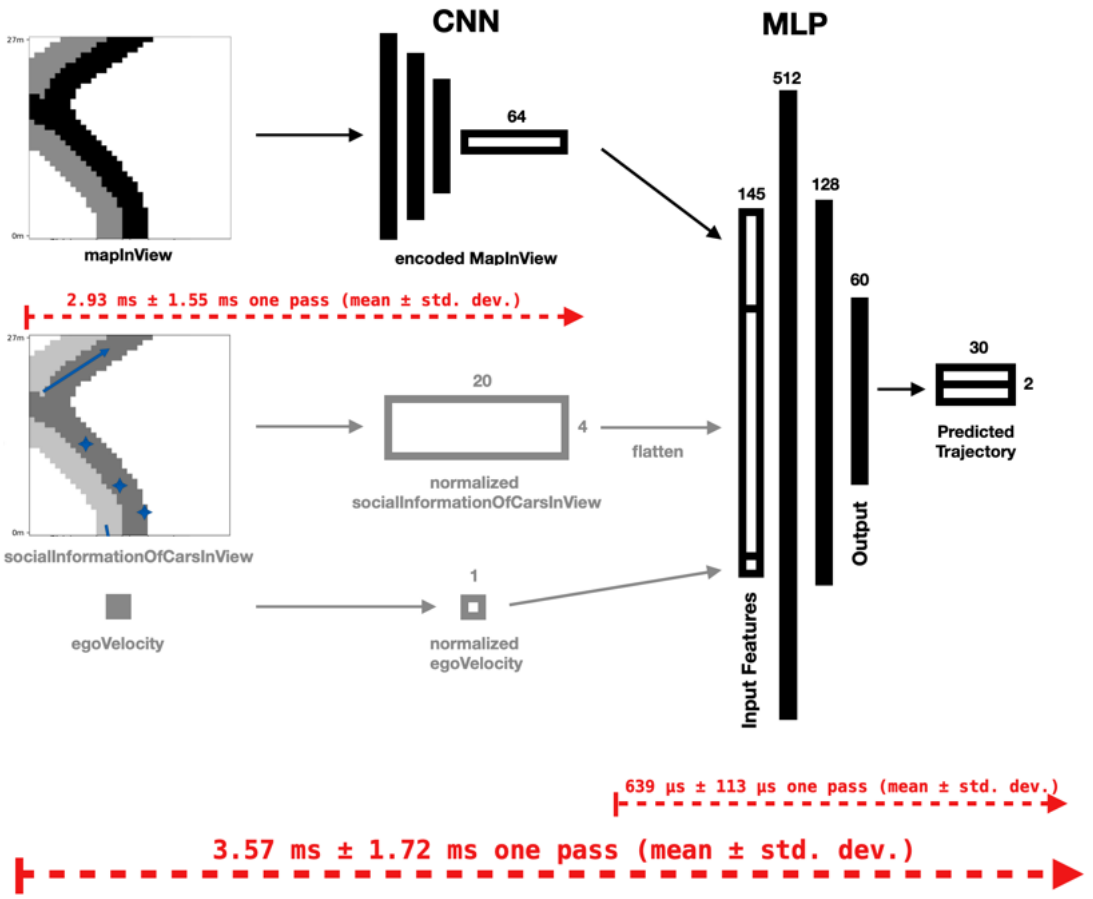


Figure 5.5: Inference time for forward pass in the Proposed Architecture: CNN Encoder Followed by MLP prediction. The overall time taken for a single forward pass is calculated, underscoring the computational efficiency of the architecture. Greyed-out components represent operations related to data loading and setup variations, which are not explicitly timed as part of the process. In the total runtime, the loading of input tensors is negligible and is heavily reliant on the specific setup.

6 Discussion

In this chapter, we introduce five points of discussion based on the insights derived from our results.

Considering multiple possible route options: A fundamental improvement would be enlarge the unimodal ground truth to a multimodal ground truth in the training data. This means that every vehicle which trajectory is predicted has multiple possible routes stored in the ground truth, which it can drive, as depicted in figure 6.1. This could be done by analysing the course of the current lanelet and its successor options and annotate them separately as potential future trajectories. As the picture above perfectly illustrates, there are multiple feasible trajectories in a given situation. As the vehicle is still pretty far away from the crossing there are no indications whether it will go straight or turn left. If the prediction is going straight and the ground truth turning left, the network gets punished in the training process for a wrong prediction because it has a big offset to the ground truth. In an ideal training data configuration this can be avoided, by storing multiple ground truth options until there is only one uniquely defined path that the vehicle will follow.

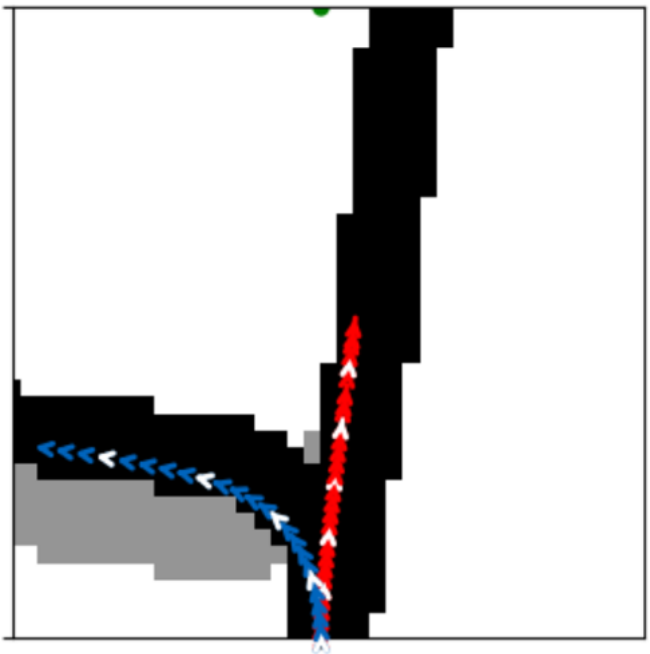


Figure 6.1: Multiple plausible trajectory options can be available if the agent yet not initiated a turn. In this scene, it is imperative to encompass two distinct ground truth trajectories as possible options.

An alternative to this was realised in this thesis, by implementing a method that returns a scalar float value indicating the validity of a predicted trajectory as explained in section 4.7.3. Incorporating this value into the loss function enables more severe penalties for trajectories deviating from the road, yielding a similar, albeit slightly less impactful, result compared to the integration of multimodal ground truth trajectory paths. Without penalizing trajectories off road, the prediction would converge to a mean value between going straight and turning left in the situation depicted in figure 6.1.

Improving the training dataset: We have a suspicion that the training dataset may be unbalanced. A hypotheses for this is that within our right-hand traffic system, there exists a prevalence of right turns as compared to left turns. Consequently, the network acquired a stronger optimization towards learning right turns, thereby introducing challenges in the optimization for left-turn predictions. Nonetheless, this phenomenon was particularly pronounced in the initial stages of training, especially when utilizing smaller datasets, but demonstrated a significant improvement that gradually diminished as training progressed on larger datasets over extended epochs.
During the training phase, we observed that a substantial portion of the vehicles in each scenario remained stationary, particularly in urban settings (see figure 6.2). Given that predicting the behavior of non-moving agents is relatively straightforward, it could be advantageous to develop a data loader specifically tailored for training. This loader could prioritize vehicles that exhibit a certain velocity, focusing on those that are in motion. This approach has the potential to enhance the training process by providing more challenging and relevant instances for learning.

Figure 6.2: Training scenario with a lot of vehicles that are barely moving.Training could be accelerated by implementing a specific training loader that priorizes vehicles with a certain threshold velocity.

Enlarging the training dataset: Improving the model's prediction performance was largely achieved by using more training data. In this thesis, we used about $130 * 10$ seconds of video material, which roughly accumulates to 20 minutes of recorded street scenarios. Interestingly, a significant number of vehicles in these scenarios remained stationary, offering limited insights into predicting the movement of active agents. By increasing the training dataset substantially, anywhere from $100x$ to $1000x$ times larger, we could

presumably have achieved much better prediction results. However, it is important to note that we are in a prototype phase of experimentation, as the process of training on a much larger dataset would have been considerably more time-consuming and resource-intensive.

Autoencoder: Besides assessing the results through mean squared error (MSE) comparison between the original and reconstructed images, a visual inspection of the output images was also carried out. This visual analysis was particularly insightful when running our algorithms in complex street scenarios. In these challenging cases, the convolutional neural network (CNN) architecture consistently outperformed the linear network. While the linear architecture occasionally showed better performance in straightforward scenarios (like straight roads), prioritizing a more accurate representation of the difficult scenarios became a crucial factor. This consideration played a key role in the selection of the CNN architecture for the final predictive network.

Encode social input feature: As advocated by several research papers, including WaleNet [15] and Trajectron++ [12], the encoding of neighboring vehicle data within a scene holds significant potential. This approach is capable of enhancing generalization and consequently lead to the development of a more robust and accurate prediction. In the further progress of this project, it would be advantageous to consider and implement a similar encoding approach for getting a summarized and compressed input representation of the social interactions between the surrounding vehicles.

7 Conclusion

In this chapter, we make final remarks and conclude the thesis. Section 7.1 summarises our contributions and results. Section 7.2 identifies areas for further exploration.

7.1 Conclusion

This thesis presents a novel approach for the sequential multi-agent trajectory prediction in a scenario. The method is based on the neural network architecture in the form of a Multi-Layer Perceptron combined with a convolutional encoder. The development, training, and validation of this network necessitated significant efforts in creating training datasets and refining input data features. These input features are a computed from the motion data of multiple interacting vehicles.

The results of a comparison between our method and the established methods WaleNet and Lanelet, demonstrate a notable enhancement in the precision of our predictions. We evaluate the performance using the average displacement error (ADE) and final displacement error (FDE) metrics over a timespan of 3 seconds. Our neural network achieves an ADE of 0.78 meters and an FDE of 2.16 meters. Notably, these impressive outcomes are derived from analyzing only approximately 20 minutes video recordings of real-world street scenarios.

The process of generating the dataset involved extracting input features and ground truth information through the implementation of a sophisticated data pipeline. This pipeline facilitated the creation of labeled maps representing the vicinity of each vehicle by analyzing the underlying street network. Furthermore, the dynamic influence of neighboring vehicles was calculated and transformed to align with the respective orientation of the current prediction task. Combined with the physical properties of a vehicle these are the input features which we used to train the neural networks. We designed a multi-objective loss function to assess both the deviation between the predicted and ground truth trajectories as well as to evaluate the validity of the prediction. Our investigations have shown that it is worthwhile to compress the local maps around a vehicle using a convolutional rather than a linear encoder with a compression ratio of about 5%. The convolutional encoder has an excellent performance in compressing the input feature in a lower dimensional representation. This advantage becomes particulary evident in uncommon and complex street situations, where the convolutional architecture outshines the capabilities of the linear one.

Our novel approach's network design drew inspiration from Trajectron++ [12], VectorNet [13] and WaleNet [15]. This resulted in an architecture that is both robust and computationally efficient. One clear benefit of our approach is the relatively shallow structure of our entire architecture, accompanied by a relatively small node count. This specific characteristic strongly facilitates the real-time execution of predictive computations, particularly under real-world conditions.

7.2 Future work

We propose five major improvements for further investigations and possible enhancements of our approach:

1. Including the historical ground truth trajectory in the prediction. This entails not only considering the physical properties (position, velocity) of vehicles in proximity in the present time step, but also including their observed past trajectory progressions.
2. Switching from unimodal to multimodal trajectory prediction. This enables to obtain multiple different potential trajectories, each assigned with a certain probability.
3. Integrating a transformer architecture for a better and more versatile trajectory prediction performance.
4. Implementing multiple plausible route options to be used ground truth for each vehicle. This is necessary to gain the full potential of item 2.
5. Parallelizing the trajectory prediction of all vehicles at a given timestep. This approach enables simultaneous multi-agent prediction, departing from sequential processing. The anticipated outcome is an acceleration in the inference time of the prediction process.

List of Figures

List of Tables

Bibliography

[1] P. D. M. Althoff. „TUM-CommonRoad,“ 2023. [Online]. Available: https://commonroad.in.tum.de [visited on 04/23/2023].

[2] Y. Huang, J. Du, Z. Yang, Z. Zhou, L. Zhang, et al., „A survey on Trajectory-Prediction Methods for Autonomous Driving,“ *IEEE transactions on intelligent vehicles*, vol. 7, no. 3, pp. 652–674, 2022, DOI: 10.1109/tiv.2022.3167103. Available: https://doi.org/10.1109/tiv.2022.3167103.

[3] B. Varadarajan, A. Hefny, A. Srivastava, K. S. Refaat, N. Nayakanti, et al., „MultiPath++: Efficient information fusion and trajectory aggregation for behavior prediction,“ *2022 International Conference on Robotics and Automation (ICRA)*, 2022, DOI: 10.1109/icra46639.2022.9812107. Available: https://doi.org/10.1109/icra46639.2022.9812107.

[4] B. Padmaja, C. V. K. N. S. N. Moorthy, N. Venkateswarulu and M. M. Bala, „Exploration of issues, challenges and latest developments in autonomous cars,“ *Journal of Big Data*, vol. 10, no. 1, 2023, DOI: 10.1186/s40537-023-00701-y. Available: https://doi.org/10.1186/s40537-023-00701-y.

[5] E. Yurtsever, J. Lambert, A. Carballo and K. Takeda, „A Survey of Autonomous Driving: Common Practices and Emerging Technologies,“ *IEEE Access*, vol. 8, pp. 58443–58469, 2020, DOI: 10.1109/access.2020.2983149. Available: https://doi.org/10.1109/access.2020.2983149.

[6] A. Collin, A. Siddiqi, Y. Imanishi, E. Rebentisch, T. Tanimichi, et al., „Autonomous driving systems hardware and software architecture exploration: optimizing latency and cost under safety constraints,“ *Systems Engineering*, vol. 23, no. 3, pp. 327–337, 2019, DOI: 10.1002/sys.21528. Available: https://doi.org/10.1002/sys.21528.

[7] D. Park, H. Ryu, Y. Yang, J. Cho, J. Kim, et al. „*Leveraging Future Relationship Reasoning for Vehicle Trajectory Prediction*,“ 2023. arXiv: 2305.14715 [cs.CV].

[8] F. Leon and M. Gavrilescu, „A review of tracking and trajectory Prediction Methods for Autonomous driving,“ *Mathematics*, vol. 9, no. 6, p. 660, 2021, DOI: 10.3390/math9060660. Available: https://doi.org/10.3390/math9060660.

[9] R. Huang, H. Xue, M. Pagnucco, F. Salim and Y. Song. „*Multimodal Trajectory Prediction: A Survey*,“ 2023. arXiv: 2302.10463 [cs.RO].

[10] C. Janiesch, P. Zschech and K. Heinrich, „Machine learning and deep learning,“ *Electronic Markets*, vol. 31, no. 3, pp. 685–695, 2021, DOI: 10.1007/s12525-021-00475-2. Available: https://doi.org/10.1007/s12525-021-00475-2.

[11] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, et al., „Attention is All you Need,“ *arXiv (Cornell University)*, vol. 30, pp. 5998–6008, 2017. Available: https://arxiv.org/pdf/1706.03762v5.

[12] T. Salzmann, B. Ivanovic, P. Chakravarty and M. Pavone, „Trajectron++: Multi-Agent Generative Trajectory Forecasting With Heterogeneous Data for Control,“ *CoRR*, 2020. Available: http://dblp.uni-trier.de/db/journals/corr/corr2001.html#abs-2001-03093.

[13] J. Gao, C. Sun, H. Zhao, Y. Shen, D. Anguelov, et al., „VectorNet: Encoding HD Maps and Agent Dynamics From Vectorized Representation," *CoRR*, 2020, DOI: 10.1109/cvpr42600.2020.01154. Available: https://doi.org/10.1109/cvpr42600.2020.01154.

[14] K. Gokcesu and H. Gokcesu, „Generalized Huber Loss for Robust Learning and its Efficient Minimization for a Robust Statistics." *arXiv (Cornell University)*, 2021. Available: https://arxiv.org/pdf/2108.12627.pdf.

[15] M. Geisslinger, P. Karle, J. Betz and M. Lienkamp, „Watch-and-Learn-Net: Self-supervised Online learning for Probabilistic vehicle trajectory Prediction," *2021 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2021, DOI: 10.1109/smc52423.2021.9659079. Available: https://doi.org/10.1109/smc52423.2021.9659079.

[16] K. Messaoud, N. Deo, M. M. Trivedi and F. Nashashibi, „Multi-Head Attention with Joint Agent-Map Representation for Trajectory Prediction in Autonomous Driving." *arXiv (Cornell University)*, 2020. Available: https://arxiv.org/abs/2005.02545v1.

[17] M. Althoff, M. Koschi and S. Manzinger, „CommonRoad: Composable benchmarks for motion planning on roads," *2017 IEEE Intelligent Vehicles Symposium (IV)*, 2017, DOI: 10.1109/ivs.2017.7995802. Available: https://doi.org/10.1109/ivs.2017.7995802.

[18] commonRoad. „Solving CommonRoad Planning Problems," 2023. [Online]. Available: https://commonroad.in.tum.de/tutorials/commonroad-search [visited on 06/20/2023].

[19] Y. Zhang, „A Better Autoencoder for Image: Convolutional Autoencoder," 2018.

[20] O. Oreolorun. „Convolutional Autoencoders," 2023. [Online]. Available: https://blog.paperspace.com/convolutional-autoencoder/ [visited on 06/04/2023].

[21] E. Anello. „Convolutional Autoencoder in Pytorch on MNIST dataset." 2023. [Online]. Available: https://medium.com/dataseries/convolutional-autoencoder-in-pytorch-on-mnist-dataset-d65145c132ac [visited on 07/20/2023].

[22] S. Ioffe and C. Szegedy, „Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv (Cornell University)*, 2015. Available: http://export.arxiv.org/pdf/1502.03167.

[23] L. Huang, J. Qin, Y. Zhou, F. Zhu, L. Liu, et al., „Normalization Techniques in Training DNNs: Methodology, analysis and application," *arXiv (Cornell University)*, 2020, DOI: 10.48550/arxiv.2009.12836. Available: http://arxiv.org/abs/2009.12836.

[24] A. Jadon, A. Patil and S. Jadon. „*A Comprehensive Survey of Regression Based Loss Functions for Time Series Forecasting*," 2022. arXiv: 2211.02989 [cs.LG].

[25] Y. Patel. „*Neural Network Training and Non-Differentiable Objective Functions*," 2023. arXiv: 2305.02024 [cs.CV].

[26] R. Chandra, T. Guan, S. Panuganti, T. Mittal, U. Bhattacharya, et al., „Forecasting trajectory and behavior of Road-Agents using spectral clustering in Graph-LSTMs," *arXiv (Cornell University)*, 2019. Available: http://export.arxiv.org/pdf/1912.01118.

A Appendix

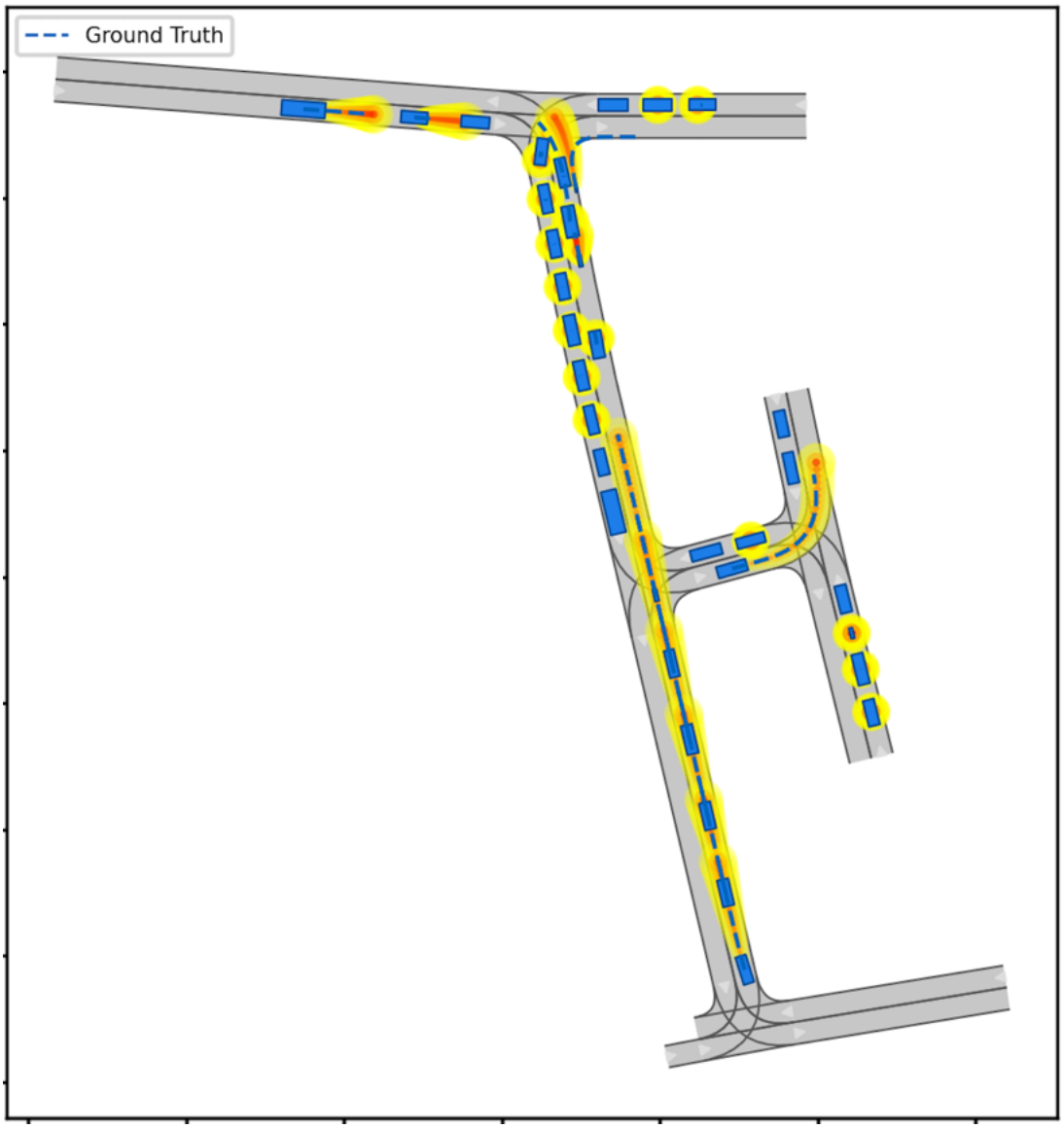


Figure A.1: Multi-Agent trajectory prediction for CommonRoad scenario 'DEU-Lohmar-11-1-T-1.xml'

```python
import torch

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

def orientVectorListTorch(vectorList):
    dx, dy = vectorList[:,0], vectorList[:,1]
    angle_increment = torch.atan2(dy, dx).to(device)

    mask = (torch.abs(dx) < 1e-8) & (torch.abs(dy) < 1e-8)
    angle_increment[mask] = 0
    relativeAngle = torch.cumsum(angle_increment, dim=0)
    relativeAngle[0] = 0
    orientVectorList = torch.stack([
        dx * torch.cos(relativeAngle) - dy * torch.sin(relativeAngle),
        dx * torch.sin(relativeAngle) + dy * torch.cos(relativeAngle)
    ], dim=1)
    return orientVectorList


def convertingToAbsolutePointListTorch(orientVectorList):
    return torch.cumsum(orientVectorList, dim=0)


def convertRelativeDxDyTensorAbsolutePointListTorch(relativeDxDyList):
    relativeDxDyWithOrientetAnglesList = orientVectorListTorch(relativeDxDyList.view(30, 2))
    orientedPointList = convertingToAbsolutePointListTorch(relativeDxDyWithOrientetAnglesList)
    return orientedPointList


def lengthInsideMapInView(trajectory):
    x,y = trajectory[:,0], trajectory[:,1]
    mask = (x < 0) | (x > 27) | (y < -13.5) | (y > 13.5)
    idx = torch.nonzero(mask)
    return 30 if idx.numel() == 0 else idx[0].item() + 1


def chamfLoss(trajectoyPoints, mapPoints):
    # trajectoyPoints (N, 2)
    # mapPoints (M, 2)

    assert torch.is_tensor(trajectoyPoints) and torch.is_tensor(mapPoints)
    dists = (trajectoyPoints[:, None] - mapPoints[None])**2  # (N, M, 2)
    dists = torch.sqrt(torch.sum(dists, axis=-1))  # (N, M)
    dists = torch.min(dists, axis=1).values  # (N,)
    return dists


def trajectoryOnRoad(trajectory, mapInView):
    viewLength = 27 # mapInView view scope
    arraySubcornerLength = viewLength / float(39-1) # distance between two pixels
    mapInView = torch.flip(mapInView, [0]).to(device)
    unNormedTrajectory = torch.where(trajectory != 0, trajectory * norm.view(60), torch.zeros_like(trajectory))

    # converting vectorised trajectory to absolute points
    trajectory = convertRelativeDxDyTensorAbsolutePointListTorch(unNormedTrajectory)

    # length of trajectory inside the mapInView
    lengthInsideMap = lengthInsideMapInView(trajectory)

    # get set of points in the mapInView feature
    roadPoints = torch.where(mapInView == 1)
    x,y = roadPoints[0], roadPoints[1]

    # converting pixel to mapInView system
    points = torch.stack([x * arraySubcornerLength,viewLength/2 - y * arraySubcornerLength], axis=1)
    trajectory = trajectory.view(30,2)[:lengthInsideMap]

    # calculating chamfer distance between every trajectory point and nearest mapInView point
    dists = chamfLoss(trajectory, points)
    dists = torch.where(dists < 0.6, dists * 0.0001, dists)
    dist = torch.sum(dists)/lengthInsideMap
    return dist, lengthInsideMap
```

Figure A.2: Function computing the validity of a predicted trajectory as defined in section 4.7.3. This is part of the multi-objective loss function.