TU WIEN Informatics

Cycle Safely

Ein Kollisionsvorhersagesystem

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Simon Pointner, BSc
Matrikelnummer 01612401

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Projektass. Mag.rer.soc.oec. Stefan Ohrhallinger, PhD

Wien, 30. November 2025

Simon Pointner	Michael Wimmer

TU WIEN Informatics

Cycle Safely

A collision prediction system

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Simon Pointner, BSc
Registration Number 01612401

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Projektass. Mag.rer.soc.oec. Stefan Ohrhallinger, PhD

Vienna, November 30, 2025

Simon Pointner	Michael Wimmer

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Simon Pointner, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 30. November 2025

Simon Pointner

Danksagung

Ich möchte meinem Betreuer, Projektass. Mag.rer.soc.oec. Stefan Ohrhallinger, PhD, für seine Geduld, Unterstützung und bedingungslose Ermutigung während der Vorbereitung und des Verfassens dieser Arbeit danken. Seine Ratschläge und Inspirationen waren sehr wertvoll, und er war immer verfügbar und ansprechbar, wenn ich ihn brauchte.

Mein aufrichtiger Dank gilt auch meiner Partnerin Annika, deren Unterstützung, Toleranz und Verständnis während des Studiums und des Schreibens der Abschlussarbeit diese Zeit wesentlich erleichtert haben. Ich bin auch meinen Eltern sehr dankbar für ihre Unterstützung und Ermutigung, nicht nur bei dieser Abschlussarbeit, sondern während meiner gesamten akademischen Laufbahn und auch im Leben.

Ebenfalls ein besonderer Dank gilt meinen Freunden und Kollegen, insbesondere Thorsten und David, mit denen ich sowohl die Herausforderungen als auch die Freuden des Masterstudiums geteilt habe. Die gemeinsame Bewältigung der Aufgaben machte diese nicht nur leichter, sondern auch viel angenehmer.

Abschließend möchte ich die Open-Source-Community erwähnen, deren Engagement für die freie Verbreitung von Datensätzen und die Veröffentlichung von Source-Code die Grundlage für dieses Projekt geschaffen hat. Ihre Initiative verkörpert den Geist der Zusammenarbeit, der den wissenschaftlichen Fortschritt vorantreibt.

Acknowledgements

I would like to express my gratitude to my advisor, Projektass. Mag.rer.soc.oec. Stefan Ohrhallinger, PhD, for his patience, support, and unconditioned encouragement while preparing and writing this thesis. His advice and inspiration have been very valuable, and he was always available and responsive when needed.

I want to extend my sincerest gratitude to Annika, my partner, whose persistent motivation, tolerance, and understanding made the long years of study and writing of the thesis into difficulties much easier and worthwhile. I am also grateful to my parents for the support and encouragement not only in this thesis but throughout the entire course of my academic career and in life as well.

I would also like to give special thanks to my friends and colleagues, particularly Thorsten and David, with whom I shared both the challenges and joys of the master's program. Sharing the burden only made the task more feasible as well as much more enjoyable.

Finally, I would like to cite the open-source world, whose commitment to distributing datasets freely and releasing research code has set the foundation for this project. Their initiative embodies the spirit of collaboration that drives scientific progress.

Kurzfassung

Diese Arbeit präsentiert den Entwurf und die Implementierung einer End-to-End-Pipeline zur Kollisionsvorhersage und -erkennung, die speziell auf Radfahrer zugeschnitten ist. Das primäre Ziel besteht darin, potenzielle Kollisionen zwischen dem Radfahrer (Ego-Agent) und anderen Verkehrsteilnehmern, insbesondere motorisierten Fahrzeugen, vorherzusagen, die aufgrund ihrer höheren Bewegungsenergie und Geschwindigkeit ein höheres Risiko darstellen.

Die vorgeschlagene Pipeline integriert konventionelle Techniken für die Lösung der Teilprobleme wie Objekterkennung, Objektverfolgung und Trajektorienvorhersage. Konkret wird SFA3D für die Erkennung von 3D-Objekten verwendet, ein Kalman-Filter-basierter Multi-Objekt-Tracker für die zeitliche Zuordnung und ein auf maschinellem Lernen basierendes Modell (PRECOG) werden für die Vorhersage zukünftiger Trajektorien angepasst und trainiert. Um das Training und die Entwicklung des Vorhersagemodells zu unterstützen, wird ein synthetischer Datensatz zum Radfahren erstellt, indem die Bewegung von Radfahrern und die Interaktion mit anderen Verkehrsteilnehmern mit dem CARLA-Fahrsimulator simuliert werden. Das System wird anhand des synthetischen Datensatzes und auch anhand des realen KITTI-Datensatzes evaluiert, und zusätzliche Ablationsstudien untersuchen den Beitrag jeder Pipeline-Stufe.

Experimente zeigen, dass der vorgeschlagene Ansatz in der Lage ist, zuverlässige Leistungen bei der Objekterkennung und -verfolgung zu erzielen. Dies bestätigt, dass eine solche Pipeline auch unter begrenzten Sensormöglichkeiten mit Zugang nur zu LIDAR- und GPS/IMU-Messungen möglich ist. Die Trajektorienvorhersage bleibt jedoch eine schwierige und rechenintensive Aufgabe, vor allem aufgrund des Mangels an dokumentierten und leicht einsetzbaren Open-Source-Modellen. Der Beitrag enthält auch ein Visualisierungsframework, das auf dem Rerun-Tool basiert und die interaktive Überprüfung der Zwischen- und Endergebnisse der Pipeline ermöglicht.

Insgesamt bietet diese Arbeit ein praxisnahes Framework für die Erkennung von Fahrradunfällen, gibt Einblicke in die Kombination konventioneller und maschineller Lernmethoden in solchen Pipelines und ermittelt wichtige Einschränkungen und Ansatzpunkte für zukünftige Arbeiten zur Verbesserung der Trajektorienvorhersage in schwierigen Verkehrssituationen.

Abstract

This thesis presents a design and implementation of an end-to-end collision prediction and detection pipeline tailored to cyclists. The primary goal is to predict potential collisions between the cyclist (ego agent) and other road users, particularly motorised vehicles, which pose a higher risk due to higher momentum and speed.

The proposed pipeline integrates conventional techniques for solving the sub-tasks of object detection, object tracking, and trajectory forecasting. Specifically, Super Fast and Accurate 3D Object Detection (SFA3D) is used for the detection, a Kalman filter-based multi-object tracker for temporal association of these detections, and a machine learning-based model (prediction conditioned on goals in visual multi-agent settings) is adapted and trained for the prediction of future trajectories. CARLA driving simulator facilitates the training and development of the prediction model by creating a synthetic dataset of cycling and the interaction with other road users. The system is evaluated on the synthetic dataset and also on the real-world KITTI dataset, and additional ablation studies examine the contribution of each pipeline stage.

Experiments demonstrate that the proposed approach is capable of achieving reliable performance in object detection and tracking tasks. This confirms the feasibility of such a pipeline under limited sensing capabilities, such as LIDAR and GPS/IMU measurements. However, trajectory prediction remains a difficult and computationally expensive task, primarily due to the lack of documented and easily deployable open-source models. The implementation comes with a visualisation framework, built from the Rerun tool, for interactive inspection of the pipeline's intermediate and final results.

The contribution of this thesis can be summarised by the implementation of a framework for cyclist collision detection. It offers insights into how conventional and machine learning methods can be combined into a pipeline, and key limitations and points of future work for the creation of better trajectory prediction in adverse traffic contexts are outlined.

Contents

CHAPTER 1

Introduction

The evolution of automotive safety has been a significant achievement over the past few decades. Advances such as adaptive cruise control, collision avoidance systems and lane-keeping assistance have continuously improved the safety of motorists. Recently, the emergence of self-driving technology promises to take this progress even further. Previous implementations focus on autonomous driving and motorised vehicles, but one group of road users has been consistently overlooked: cyclists. Due to their particular vulnerability, many safety systems stemming from motorised vehicles can not be applied to cycling. For cyclists, the emphasis must shift from damage control to damage prevention, aiming to predict and prevent dangerous situations before they occur. Developing such systems for bicycles presents unique challenges. Space, weight and cost constraints are far more restrictive than in cars or trucks, limiting the range of sensors and computational hardware that can be deployed. Under these conditions, this work proposes a collision prediction and prevention pipeline for cyclists based solely on position data and LIDAR point cloud input. To overcome the scarcity of real-world data containing collisions, a simulation-based approach is adopted, enabling the generation of diverse training and evaluation scenarios.

But before detailing the implementation, this Chapter introduces the motivation, problem statement, and key contributions of the work, followed by an overview of related research and the methodological approach taken in this thesis.

1.1 Motivation

As previously mentioned, cyclists are arguably the most vulnerable traffic participants and are at a disadvantage when it comes to safety innovations. While cars are equipped with sophisticated systems that can predict and prevent collisions, cyclists are still reliant on rudimentary safety devices such as helmets, reflective tapes or vests, protective vests and or knee pads, all of which offer limited protection. The disparity in safety gadgets

between motorists and cyclists is a call to action, especially with an increase in the number of people choosing bicycles as a mode of commuting and recreation. Urban areas promote cycling as an environmentally friendly means of transportation, calling for improved safety features for cyclists. A deficiency in innovative cyclist protection technology is not just a technological gap but a matter of public health [1].

A revealing case of the challenges cyclists face is the development of the *Open Bike Sensor* [2], a crowd-funded initiative aimed at measuring the distance at which powered traffic passes cyclists. The endeavour is both a symptom of the problem and the potential of bottom-up innovation: it provides valuable data on overtaking behaviour, yet also reflects the underlying asymmetry of power in favour of motorised transport and against cyclists. Drivers can rely on protective infrastructure as well as advanced in-car systems, while cyclists are forced to use external monitoring projects to report and campaign for their safety.

Machine learning and related technologies have opened new possibilities for improving road safety. Learning-based methods that analyse traffic flow and predict potential hazards have been developed primarily for motor vehicles. These systems utilise inputs from several sensors to anticipate and avoid risks, which in turn helps prevent many crashes from occurring. These technologies have mainly evaded cyclists, who have particular issues on the road. Such an imbalance only highlights the need for a dedicated collision detection system for cyclists. Bicycles do not offer a lot of physical protection in the event of a crash. Therefore, any cyclist safety system must be prevention-oriented and instead of performing damage control.

Separation of space between cyclists and motor cars is perhaps the optimal way to improve cyclist safety. Segregated bicycle lanes are well known to reduce accidents significantly [3]. However, in the majority of urban environments, constraints of space as well as infrastructure make such measures hard to implement. Hence, a programmatic approach to safety, utilising advanced sensors and machine learning, could fill this gap. To build an effective collision prediction system for cyclists, there are a number of considerations that are required. Firstly, the sensors themselves are the top priority. While RGB cameras are commonplace in automotive safety use cases, they may not be the optimal solution for cycling due to their limitations in low-light and adverse weather conditions. LIDAR (Light Detection and Ranging) technology provides a more resilient alternative. Unlike cameras, LIDAR sensors reliably detect obstacles and measure distance in the dark, in rain, or in fog, with stable performance where vision-only systems fail.

Also, sensor fusion techniques that combine LIDAR with other sensory inputs (see Figure 1.1 for commonly used sensors in driving-related tasks) are even more resilient because they can cover each other's blind spots and provide redundancy. The trade-off, however, is that such systems are more expensive, not only in sensor acquisition but also in processing time. Despite this, their insensitivity to adverse conditions makes them particularly valuable for safety-critical systems. Furthermore, unlike camera-based solutions, LIDAR sensors do not capture identifiable visual information, thereby avoiding privacy concerns associated with image data collection in public spaces.

Furthermore, the limitations of camera-only solutions are well illustrated by the issue of phantom braking in Tesla cars. As pointed out in Phantom Attacks on Driver-Assistance Systems [4], vision-based models are susceptible to adversarial patterns such as manipulated billboards or projected images, leading to unwarranted and even dangerous emergency braking. Relying only on RGB cameras can expose vehicles to safety risks that could be addressed by multi-modal sensor setups.

With the rise of machine learning and better sensors, it should be possible to make a safer environment for cyclists, such that cycling can become a sustainable and accessible form of transportation for a larger number of people. A valuable side effect of such systems is that the data they collect on cycling activity and near-miss incidents can provide officials with evidence-based insights to improve the safety and design of roads and bike paths.

1.2 Problem Statement

The primary problem addressed in this work is the prediction of potential collisions between the cyclists, from here on referred to as ego vehicle, and other road users. These other road users include pedestrians, other cyclists, and particularly motorised vehicles. The latter are of significant concern due to the higher potential for severe injury or damage caused by higher speed and momentum.

Predicting collisions involving a cyclist presents unique challenges because the dynamics of interactions between the cyclist and various classes of road users differ significantly. Pedestrians, for example, move at relatively slow speeds and have the agility to change direction quickly. In contrast, motorised vehicles, such as cars, trucks and motorcycles, move faster and have higher momentum. These vehicles typically follow more predictable paths dictated by the road layout, which simplifies their trajectory modelling compared to the more unpredictable behaviour and movement of pedestrians.

For this reason, the focus of this work is on motorised vehicles. This allows for a more straightforward approach to predicting their movements based primarily on their past trajectories and the road layout. Trajectory prediction for pedestrians and the related challenges will also be discussed in the related work section of this thesis.

Key Challenges of this problem include:

- Sensor selection: Identifying the appropriate sensors to detect and monitor surrounding vehicles. The selection of sensors impacts the accuracy and reliability of the data collected, which is vital for effective collision prediction. (Commonly used sensors can be seen in Figure 1.1)
- Vehicle detection and tracking: Once the sensors are in place, the system must be capable of accurately identifying other vehicles on the road and determining their positions. This requires robust algorithms that can handle various environmental conditions and partial occlusions.

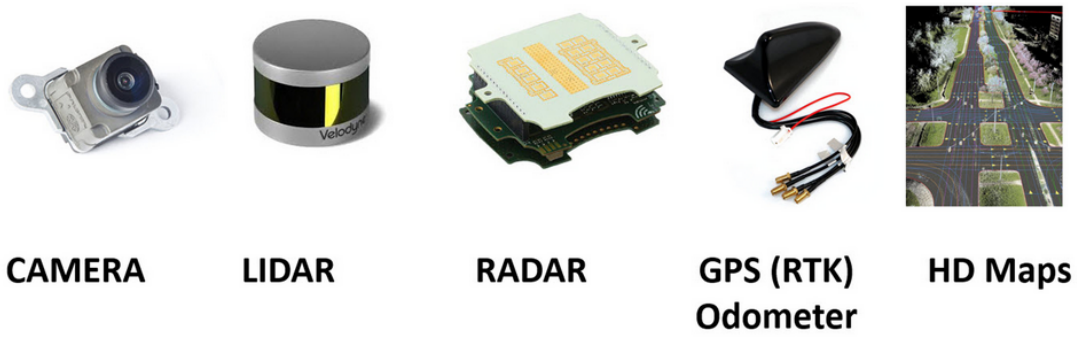


Figure 1.1: Commonly used sensors/inputs in computer vision for driving-related tasks. [5]

- Trajectory prediction: After detecting and tracking the vehicles, the next challenge is to predict their future trajectories based on the surrounding area (scene context) and their past trajectories.
- Collision prediction: Based on the predicted trajectories, the system must assess the likelihood of collisions between the cyclist and other road users.

1.3 Contributions

The contributions of this work include the design and implementation of an end-to-end collision prediction pipeline tailored for cyclists, combining existing methods and models for solving sub-problems such as object detection, object tracking and trajectory prediction. This includes the generation of a synthetic dataset for the prediction of the cyclist trajectory using the CARLA driving simulator and the training of a machine learning model for the prediction of the path based on the synthetic dataset. In addition, the model and the pipeline were evaluated in both the synthetic dataset and the KITTI dataset. All of this is achieved using a limited set of sensors, namely LIDAR and IMU/GPS for relative positioning. In addition, a tool named rerun [6] is utilised to better visualise and demonstrate intermediate results and the output of the pipeline in an interactive 3D viewer.

1.4 Structure of the Work

The thesis is structured in the following way: First, in Chapter 2, related work is reviewed and discussed, structured by topics and sub-problems relevant to the overarching task of predicting potential collisions. In addition, end-to-end frameworks are presented, and publicly available datasets for driving-related tasks are listed and compared.

Subsequently, in Chapters 3 and 4, the methodology and implementation are described in detail. Since the main contribution of this thesis lies in the development of a pipeline that

integrates existing methods, each pipeline stage is first explained individually. This is followed by a description of the data structures used for training the trajectory prediction model, which was implemented as part of this work. The visualisation of pipeline stages, including the generation of images used throughout this thesis, is then outlined, along with the details of the training procedure. Finally, the process of generating the dataset for training the trajectory prediction model is described.

After the implementation section, the results are presented in Chapter 5, focusing on the evaluation of the pipeline. The results provide a statistical analysis of the pipeline outputs, complemented by ablation studies for individual pipeline components. Lastly, in Chapters 6 and 7, the findings are discussed, and the limitations of this work as well as directions for future research are addressed.

CHAPTER 2

Related work

In this section, the key advances in the field of autonomous vehicle perception and motion prediction relevant to this thesis will be outlined, reviewed. The discussion is organised in the main sub-tasks addressed by this thesis: Object detection, object tracking and trajectory prediction.

But first, end-to-end methods which produce high-level outputs directly from sensor inputs, such as trajectory predictions or collision warnings, are presented. Afterwards, the historical development from classical methods to learning based and finally to deep learning based methods is outlined for the aforementioned sub-tasks of this thesis. The focus of this section is to provide an understanding of the underlying algorithms and core ideas.

In addition to methods and research papers, finally, an overview of widely used datasets in autonomous driving research, such as KITTI, nuScenes, Argoverse or Waymo Open, is presented, examining sensor configurations and differences between the datasets, to provide context for the choice of algorithms and methods used in this thesis.

2.1 End-to-end Frameworks

In modern driver-assistance systems, end-to-end frameworks refer to networks that accomplish everything from parsing input data (camera images, LIDAR, radar, HD-maps, etc.) and produce directly high-level outputs like computed vehicle trajectories or alerts/warnings of potential collisions or steering instructions or braking/acceleration.

One very early example of such a system was already introduced in 1989; ALVINN [8] demonstrated that a simple neural network can learn to map camera inputs to steering instructions directly. It learned online from human driving examples to produce instructions to keep driving on the lane. While crude by today's standards, ALVINN

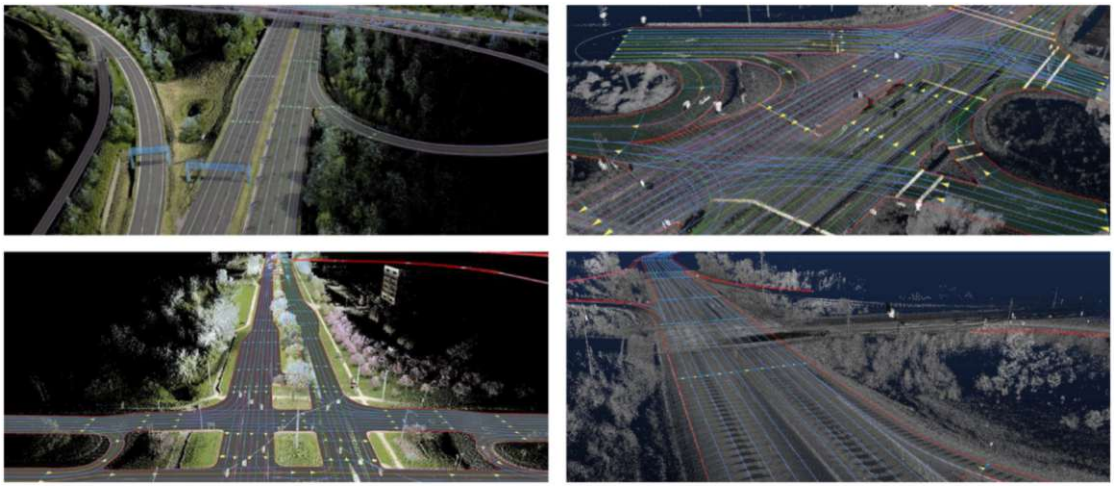


Figure 2.1: Examples of HD maps [7]

proved that a single network can learn driving from raw images without hand-designed steps.

A more modern example of an end-to-end framework is Fast and Furious (2018) [9]. The goal of this network, unlike ALVINN, is not to produce steering commands, but to unify detection, tracking and motion forecasting in one trainable network. The architecture can be outlined as follows. The LIDAR point cloud is projected into a BEV representation such that the network can process the point cloud using a 2D CNN, which captures spatial-temporal features about the scene around the ego vehicle. This backbone is shared by all three tasks solved by this framework (detection, tracking, forecasting). Then, multiple heads (task-specific output modules) are used to output 3D bounding boxes for objects, tracking results and motion forecasts. The training optimises for a joint loss function, which optimises for detection loss, tracking loss and L1 or L2 trajectory prediction loss. FaF achieved real-time performance with around 33 frames per second on the tested hardware.

Following this development, RobustTP [10] in 2019 was published as another end-to-end motion forecasting network. The main improvement compared to FaF was made by using LSTMs (long-short term memory) to encode each agent's past trajectory. For encoding the spatial information (the LIDAR point cloud or, in this case, camera images), again a CNN is used to learn how agents influence each other. Fusing the output of the LSTMs and the CNN produces a context-aware embedding for each agent.

In 2020, LiraNet [11] fused LIDAR, radar and HD-maps in one method to achieve better trajectory prediction results. Fusing LIDAR and radar is challenging because of the relatively low angular resolution of the radar and the sparsity of the produced data compared to LIDAR. This problem is tackled by a novel spatio-temporal radar feature extraction scheme. The output of this spatio-temporal network is then fused in a multi-scale fusion backbone with the LIDAR and HD-map inputs. These fused features

are then fed into a joint perception-prediction network, which produces detections and their future trajectories. Similarly, PnPNet [12] proposed a similar multi-modal network based on LIDAR and HD-maps, using LSTMs for the trajectory estimation module.

More recently, end-to-end prediction models were still more advanced with richer representations. A network called MVFuseNet [13] was proposed, which fuses BEV view and range-view of the input LIDAR data in one network to jointly detect objects and predict their motion.

Another regime of end-to-end trajectory prediction frameworks is based on sampling multiple possible future trajectories per agent based on multiple predicted target locations an agent might steer towards, like TnT [14] or the later version, DenseTnT [15]. The goal of the network is to first predict target locations per agent and then condition the full trajectory prediction on those goals.

RNN and LSTMs have been standard tools for time-series forecasting in the past decade, but more recently, following the advances in natural language processing using transformer-based networks (like GPT), these transformers were also applied to end-to-end trajectory prediction modelling. For example, ViP3D [16] uses a transformer-based query attention network as a backbone and uses concepts like cross-attention, replacing classical tracking association methods with learned attention. Another modern example, CATPlan [17], builds upon existing end-to-end trajectory prediction models and uses a transformer decoder to output probabilities of potential collisions in addition to trajectories.

Although this list of end-to-end frameworks is far from complete, it gives an overview of existing methods and the basic solution proposals, from classical methods to different machine learning models.

2.2 Sensors

The basis of all autonomous driving challenges and benchmarks is defined by the available recorded data from different sensors. The most widely used modalities are RGB cameras, LIDAR, radar, GPS/IMU, and HD maps with their respective strengths and trade-offs. Strictly speaking, HD-maps are not a direct sensory output, but can be extracted using complex, multi-stage processes that involve specialised sensors equipped on mapping vehicles to create lane-level geometry.

Cameras are utilised most extensively among computer vision research sensing modalities. RGB cameras provide high-definition semantic data required for object classification, scene understanding, and human-centric trajectory prediction (e.g., Social-LSTM [18], Social-GAN [19]). Their low form factor, low cost and simplicity of use render them attractive for large-scale deployment, as evidenced by datasets like KITTI [20] (see sensor setup in Figure 2.2) and WAYMO Open [21]. But cameras lack good depth perception and do not handle poor weather or low light well, so they are limited in their application to safety-critical systems.

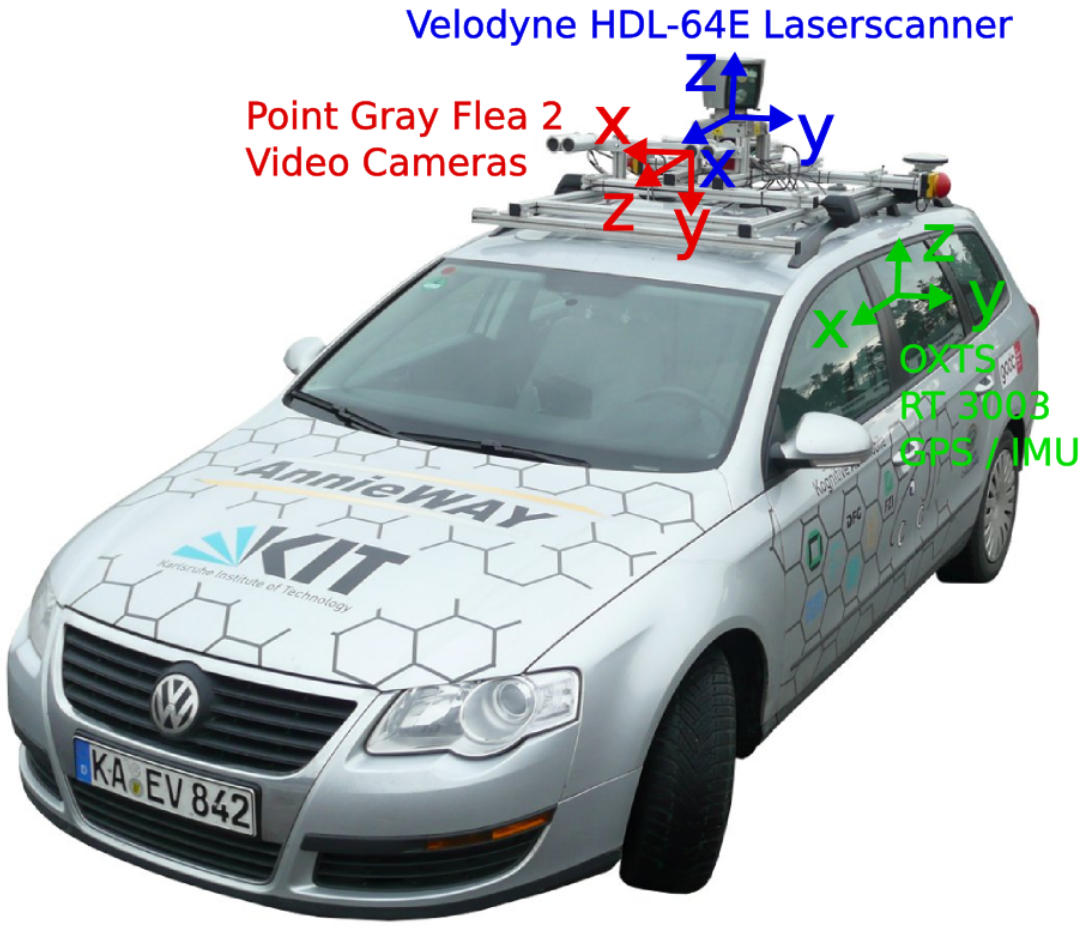


Figure 2.2: Sensor setup used for recording the KITTI dataset [20]

LIDAR sensors measure the environment around them with accurate 3D geometry by direct measurement of depth in terms of capturing a laser's reflections and measuring the time it takes for the laser beam to return, the so-called "Time of Flight" (ToF). LIDAR excels at sensing for obstacles and measuring distance with extreme precision. But the technology has drawbacks: sensors are expensive, make infrequent measurements at greater ranges and can also deteriorate in rain or fog.

The combination with radar, though, can eliminate the problems of deteriorating performance of LIDAR in bad weather conditions and therefore increase performance under rain/fog or weak illumination. Radar sensors appear in datasets such as nuScenes [22], but find less usage in the academic prediction model. Their lower spatial resolution compared to LIDAR and cameras makes them less valuable for granular object classification, but this robustness makes them an excellent second sensor for multi-modal sensor fusion.

GPS/IMU systems are used in the creation of datasets (e.g., WAYMO [21], KITTI [20])

to provide precise ego-vehicle localisation. While not typically used for object detection or prediction, they enable precise ground truth annotation and multi-sensor data alignment.

Finally, HD maps provide static, formal definitions of lane topology, traffic rules and drivable space (see Figure 2.1 for examples of HD maps), such as in ARGOVERSE [23] and WAYMO Open. Providing map data significantly improves the accuracy of trajectory forecasting by constraining possible agent paths and destinations. The only downside is that HD maps are expensive to construct and maintain, making it hard to scale to unseen areas.

2.3 Object Detection

Now, after discussing end-to-end frameworks and commonly used sensors in computer vision and driving-related tasks, let's discuss related work for the three main sub-problems emerging from the problem stated. The first one, object detection, is probably the most important task of perception in driving-related tasks, as it enables vehicles to see the agents in their vicinity or static objects. The accuracy of the object detection is not only essential to understanding scenes captured by sensors but also serves as the foundation for downstream operations such as trajectory prediction and collision detection. Over the past decade, methods have been evolving at a quick pace, from two-stage proposal-based architectures to one-shot detectors and advanced multi-modal fusion models, each tuned for trading off speed, accuracy and robustness across multiple sensing modalities. While two-stage detectors are very powerful and useful for fields where high accuracy is more important than real-time capabilities (e.g. medical applications), one-stage detectors are more widely used for real-time applications.

2.3.1 One-Stage Detectors

The most notable work in the field of object detection and one-stage detectors is probably YOLO [24], which showed that one-stage detectors strike a good balance between accuracy and speed. To illustrate the idea of the YOLO one-stage detector, let us consider the simpler case of detecting 2D bounding boxes of objects in a 2D image retrieved from a camera. The principle idea is the following: First, the input image is divided into grid cells and the networks task is to predict a bounding box (offsets of the box relative to it's corresponding grid cell) and a confidence score as well as a class label (e.g. "dog" or "bicycle" as shown in Figure 2.3). A CNN architecture is used to parse the input, followed by two fully connected layers. Then, based on the confidence score, grid cells that likely contain no valid object detection can be discarded, leaving only the predicted bounding boxes of detected objects. One downside worth mentioning of this approach is that the granularity of the grid defined limits the maximum number of detectable objects in a given image. But depending on the size of the grid, in practice, this is not an issue.

Similarly, SSD (Single Shot Multi-Box Detector) [25], skips the object proposals defined by the grid layout used in YOLO, but instead features maps from multiple layers of the

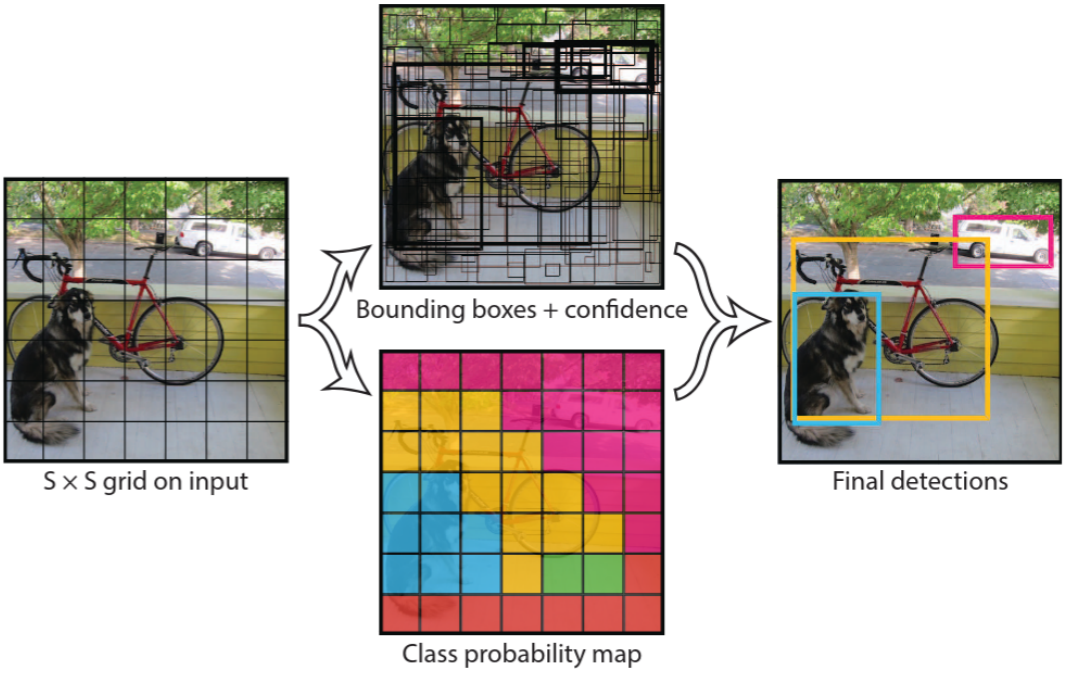


Figure 2.3: Principle idea and outputs of the YOLO model for object detection [24].

CNN are used to predict several default boxes (anchors) of different sizes and aspect ratios. The model again predicts offsets, confidence scores and class probabilities for each box. The approach of taking the feature maps on different levels of the CNN allows the network to better capture very small or very large objects, in comparison to YOLO.

But one-stage detectors have one major drawback, they treat all detections equally, favouring larger, easily detectable objects while suppressing harder to detect smaller object classes. This problem was tackled by RetinaNet [26], which uses a ResNet backbone to extract low-level features like edges and shapes. A ResNet is a residual network that uses residual connections to solve the problem of diminishing gradients, which results in performance degradation as the network becomes very deep. Then those features are combined at different scales using a feature pyramid network (FPN), which allows each layer to capture objects of different sizes. The main improvement of RetinaNet is that it uses a focal loss function instead of cross-entropy, which means during training of the network, a higher importance is given to harder samples. In other words, if a sample is already easy to detect, it contributes less to the loss, reducing its influence during training. This allows RetinaNet to compete with two-stage detectors on datasets such as COCO [27] with the same speed as one-stage detectors.

2.3.2 Two-Stage Detectors

Contrary to one-stage detectors, the idea of two-stage detectors is to solve the object detection problem by firstly finding possible object regions (region proposals) and then classifying and refining those regions. A prominent architecture of two-stage detectors was introduced in 2014, R-CNNs [28]. The main idea of R-CNNs is to use features learned by a CNN instead of using hand-crafted features like HOG (histogram of gradients) or SIFT (scale-invariant feature transform). But before extracting features using CNNs, the category-independent region proposal stage relies on a method called selective search, which is a classical computer vision approach. It starts off with small regions that are then grouped by similarity, as for example colour or texture. As these small regions are merged, each set of combined regions forms a region proposal. This process is repeated at multiple scales to create differently scaled region proposals. The CNN, as mentioned, consequently extracts features per region proposal, which are processed by a support vector machine (SVM) to classify the proposed region. A separate SVM is used per object class. The next step is using a bounding box regressor to refine the sometimes not perfectly aligned region proposal to the classified object. Finally, since many proposals may overlap the same object, they are filtered using non-max suppression (for example, to keep the proposal with the highest score).

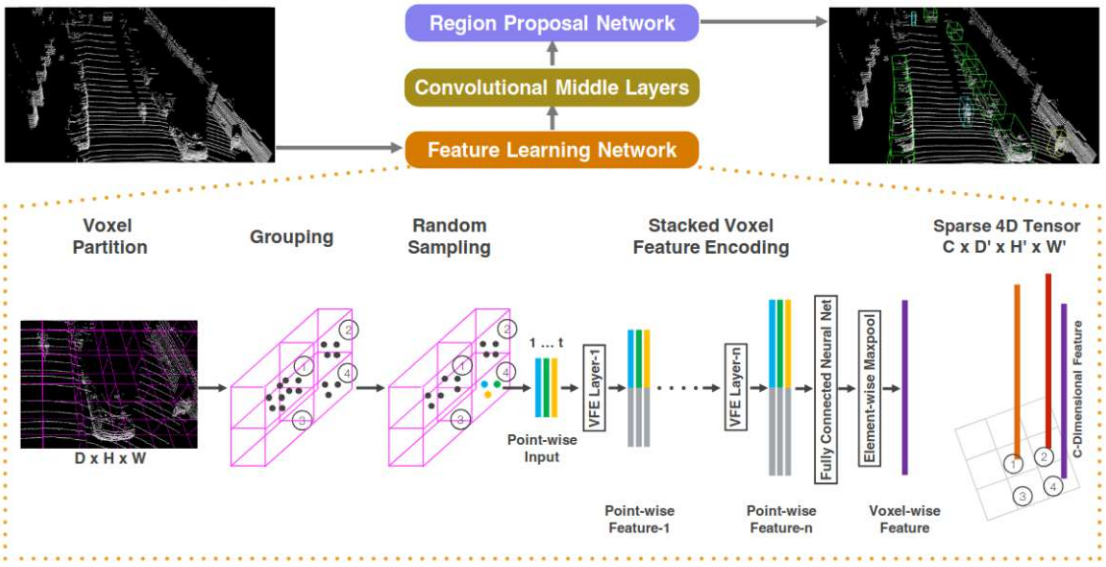


Figure 2.4: VoxelNet architecture, which processes raw point cloud input by voxelization of space and per voxel encoding of features, which allows processing sparse input data like a 2D image using a CNN [29].

2.3.3 LIDAR-Based 3D Detectors

Now that the basics of object detection based on machine learning methods have been discussed, these ideas must be applied and adjusted from functioning on 2D image inputs

to detecting 3D bounding boxes based on LIDAR input data. LIDAR point clouds are, contrary to images, sparse and irregular, meaning they cannot be directly fed into a CNN. A major advancement in this field was VoxelNet [29], which, as the name suggests, divides the space into a grid of voxels. Each voxel may contain a variable amount of input points from the LIDAR. VoxelNet encodes this variable amount of input points into a fixed-length feature vector, by first centring the points inside each voxel by the mean of all LIDAR points inside the current cell (centring). Then, all positions are concatenated and fed into a fully connected neural network to transform the positions into fixed-length features. A convolutional backbone then produced a 3D feature map that captures object shapes and locations. The 3D feature map is then collapsed into a BEV. Now, the input is transformed into a 2D image that can be conveniently transformed using a CNN and object detection can be achieved similarly to the previous methods, but now, instead of trying to predict offsets for 2D bounding boxes, 3D bounding boxes and their class labels, as well as confidence scores are predicted. The process is illustrated in Figure 2.4.

A modification of this idea was presented by PointPillars [30], where instead of encoding the LIDAR point cloud in grid cells, the space is divided into pillars. This is faster than first encoding the points per voxel, but at a loss of accuracy. But it works reasonably well for LIDAR point clouds, which were recorded with a sensor with a low vertical field of view (FOV).

2.3.4 Multi-Modal Detectors

Based on the developments of object detectors using either image or point cloud input data, to further improve detection quality, multi-modal object detectors have been proposed. These combine input data from multiple sensors (modalities) to achieve better results, especially when the sensors' weaknesses and strengths complement each other. Which, for example, is the case for cameras, which are great for capturing colour information, and LIDAR, which capture depth, and therefore are less prone to optical illusions. A pioneering work in multi-modal detectors was proposed by the name MV3D [31]. In simple terms, it combines the ideas from the previously explained detectors, taking several individual neural network architectures, and then fuses the outputs by taking the element-wise mean. There are several strategies to combine the inputs or outputs: early, deep or late fusion. In early fusion, the inputs are simply concatenated before parsing them into the network, while deep fusion combines outputs of different layers of the CNN architecture. Late fusion refers to the element-wise mean combination of the individual high-level outputs. The deep fusion method achieved the best results on the KITTI dataset.

Similarly, AVOD [32] combines LIDAR and camera inputs, but contrary to MV3D, the input is combined before creating bounding box proposals, improving performance on detecting smaller objects. The training strategy of AVOD also differs; instead of training each stage individually as performed in MV3D, the network is trained end-to-end.

2.4 Object Tracking

Object tracking refers to the task of assigning unique IDs to objects in the scene over time. A common approach for object tracking is tracking-by-detection, which refers to searching for an arbitrary number of objects in each frame, not assuming any knowledge of objects from the previous frame, and then associating these objects with previous detections. Further, methods can be distinguished into offline and online methods. In offline methods, the whole sequence can be used for better association results, while in online scenarios, the frames are processed sequentially. Further, trackers differ by their capability of either tracking a single object over time (SOT) or multiple objects (MOT). The latter are most relevant for this work.

Object tracking in autonomous vehicles is the problem of maintaining persistent object identities, such as vehicles or pedestrians, over time, ideally from 2D images, 3D bounding boxes, or point clouds. The dominant solution is tracking-by-detection, where frame-by-frame detections are linked to trajectories through motion modelling and data association. It is a challenging problem due to occlusion, detection failures, and dynamic motion, and thus efficient prediction and robust association are required.

A prominent example for object tracking-by-detection is SORT [33], which, receiving the detected bounding boxes every frame, uses an algorithm called Hungarian matching to associate detections from different frames. But before matching, the next state of all tracked objects is predicted using a Kalman filter. In simple terms, a Kalman filter estimated the state of a system (in this case, the positions of the tracked objects) over time from noisy observations. It can be thought of as a smart moving average that predicts the next state while correcting for noisy input. The Hungarian algorithm then uses the intersection over union (IoU) to match the new detections to the predicted positions of the already tracked objects. This results in a cost matrix, where each detection can be greedily matched with the lowest cost associated with it.

While SORT was implemented to track 2D objects, the 3D variant of it, AB3DMOT [34], matches 3D detections. Although this Kalman+Hungarian-based approach is very light-weight, it is surprisingly competitive and yields stable results. But both approaches fail when the detection quality is poor or under heavy occlusion or more sophisticated motion (for example, pedestrian motion).

These shortcomings are addressed by learning-based approaches. An early example, DeepSORT [35], builds upon SORT, but incorporates deep appearance descriptors to improve the object tracking. The idea: in addition to the bounding box outputs, DeepSORT uses a pre-trained CNN to extract appearance embeddings from each detected bounding box, which capture how the object looks (e.g. colour, texture, shape). Intuitively, this information helps to not confusing tracked objects of, for example, different colours. The appearance features are used similarly to the cost matrix produced by the IoU based matching, but the cosine distance between the feature vectors is used as the cost between a detection and a prediction from the Kalman filter.

Another example for object-tracking is FairMOT [36], which jointly performs object detection and tracking, training a network to perform both tasks. Compared to DeepSORT, this approach improves the frame per second processable from 6.4 to 25.9, including both the detection and association time. Another more recent example of a network putting both detection and tracking into one network is RetinaTrack [37]. The core idea is to use an anchor-based one-stage detector and extend it with tracking capabilities.

End-to-end methods have made the distinction between detection and tracking even more difficult recently. Methods based on transformers such as TrackFormer [38], CenterPoint [39], P2B [40] and MOTR [41] use extended object queries that learn simultaneously continuous detection and temporal association end-to-end. In the 3D matching problem, SimTrack [42] and S2-Track [43] further extend this idea by predicting track states directly from LIDAR point clouds with uncertainty-aware methods that can reason about occlusions and missed detections.

2.5 Trajectory Prediction

Trajectory prediction refers to the task of forecasting the future positions of the traffic agents based on their past movement and the behaviour of neighbouring agents, as well as based on an understanding of the scene around the agents. Similarly to many other computer vision problems, new advances solve this problem using deep learning. Before diving into deep learning based methods, though, the following outlines classical approaches and the historical development of trajectory prediction.

Prior to learning-based methods, physics-based methods were used for trajectory prediction. The idea is to use kinematic models to predict the motion of a vehicle. Simple examples are constant velocity or constant acceleration models [44]. These models are computationally efficient but cannot handle complex interactions between vehicles or uncertainties produced by noisy sensor inputs in real-world applications. Probabilistic approaches like Monte Carlo or Kalman filters were employed to measure these uncertainties and improve trajectory predictions.

The first learning based trajectory prediction methods were based on support vector machines (SVM) [45] or Gaussian processes. But these methods did not capture dynamic interactions between multiple traffic agents.

This problem was addressed with deep learning approaches, DESIRE [46], for example, produces distant future predictions in dynamic scenes using a conditional variational autoencoder (CVAE) to generate multiple trajectory prediction candidates for each agent. In simpler terms, a CVAE learn to map input data, such as the past trajectory of the detected and tracked objects, and maps it to a latent distribution. From this distribution, samples are then taken and decoded to retrieve multiple hypothetical future trajectories. But since the agents don't move independently, DESIRE models interactions between agents using a RNN to encode the past trajectories and a scene context module to encode the map layout (e.g. the LIDAR point cloud processed by a CNN). Trajectron

[47], on the other hand, uses a graph-structured RNN to produce future trajectories in dynamic scenes. The idea, each tracked object is represented as a node in a graph. The features of a node include the past trajectory and its velocity. Edges in the graph connect nodes which can influence each other (for example, are in each other's proximity). A graph neural network (GNN) is used to propagate information along the edges of the graph. Trajectron++ [48], an improvement to Trajectron published in 2020, maintains the graph-based, multi-agent interaction modelling presented in Trajectron, but adds significant improvements in representation and scalability and multi-modality of the model. A key feature of Trajectron++ is that it can, compared to the original paper, predict motion continuously and not only at concrete time-steps. Further improvements were made by TnT [14] (Target-driving trajectory prediction). The idea of the paper: instead of predicting the future trajectory of the tracked objects directly, instead predict potential target locations where the tracked objects might want to move, and later figure out how the objects might get to those locations. So in contrast to other methods, TnT does not rely on latent variables and sampling to generate diverse trajectories, but instead encodes the agent's environment using a CNN, the past trajectories using LSTMs (long-short term memory) and nearby agents using an interaction encoder. These features are combined into a context encoding to represent the agent's current state. The target prediction model then predicts several target offsets and target scores. These are sorted based on the target scores, and a motion estimator module generates the possible trajectories to reach the target offsets. Another incarnation of the idea of predicting target goals instead of sampling trajectories was introduced by PRECOG [49], the model used in this thesis.

2.6 Datasets

Datasets are an important basis for developing algorithms for diverse computer vision problems, because they allow comparing different methods with each other, form a baseline and alleviate the need for researchers to go out and record custom datasets first before being able to work on a problem. For autonomous driving tasks, multiple datasets have been proposed, which contain different data but also ground truth annotations depending on the challenges that can be solved with those datasets. Since most of the sub-tasks listed above, in an attempt to crack this thesis problem are complex in themselves, it is useful to get an overview over available datasets, compare them with the requirements of each step, check if the inputs and outputs given are according to the problem definition and finally select a method, preferably with open-source code, to solve the sub-task. One excellent place to seek open source code for solving the sub-problems is Papers with Code, where one can filter or search by the problem being tackled. In the topic of object detection, there are many datasets, but very few have to do with the detection of other road users in LIDAR point clouds.

In autonomous perception and driving-related tasks, a number of large datasets have been created in the recent past to enable research on object detection, semantic segmentation, tracking and motion prediction tasks. The datasets are quite different in terms of sensor

configuration, size, annotation richness and diversity of driving scenarios. Most modern autonomous driving datasets are multi-modal, incorporating the joint use of RGB imagery and other sensors such as LiDAR, radar, and inertial measurement units (IMU), for capturing the 3D geometry and trajectory of surrounding road users.

Some of the most widely used datasets available include **KITTI**, **nuScenes**, **Argoverse**, and the **Waymo Open Dataset**, which are all based on real-world sensor captures from instrumented vehicles. There is also the **CARLA** simulator, where a pseudo-dataset based on simulated scenes allows accurate ground-truth labelling and controlled variation of weather and lighting. These datasets together provide the basis for developing and testing perception algorithms for autonomous driving research.

The following sections provide a description of these datasets, their sensor configurations and the type of annotations they offer, followed by a comparison of their most significant aspects relative to this work.

2.6.1 KITTI

KITTI Vision Benchmark (2012) is a pioneering on-road dataset captured in Karlsruhe, Germany. The setup consists of a rooftop arrangement with four synchronised video cameras (two colour and grayscale stereo pairs per side) and a Velodyne HDL-64E 3D LIDAR. The cameras record 1392x512 pixel frames (90°x35° FOV) at 10 Hz and the LIDAR scans at 10 Hz generating 100 k points per sweep (64 beams, ±13° vertical range). GPS/IMU information (OXTS RT3003) is employed for complete 6-DOF localisation. Highway, residential and urban scenes (most daytime, decent weather) have been recorded (see Figure 2.5). Annotations involve richly annotated 2D bounding boxes over the images and correlated 3D bounding boxes in LIDAR space for pedestrians, cars and cyclists (and also tracking across frames) [20, 50].

2.6.2 nuScenes

This dataset was published in 2020 and recorded in Singapore and Boston (USA) and serves as a large-scale multimodal driving dataset. It was recorded on a sensor-packed test car with the following configuration: six ring-mounted cameras (1600x900 pixel per camera, 360° horizontal; front and side cameras 70° FOV, rear camera 110°), one rotating LIDAR (Velodyne VLP-32C, 32 beams, –30° to +10° vertical FOV, 360° horizontal, 20 Hz) and five automotive radars (77 GHz, FMCW, each 13 Hz, up to 250 m range). IMU and GPS provide robust ego-motion. The dataset contains 1,000 scenes (20 s each) under different conditions (day, night, rain, different traffic densities). A "keyframe" is labelled every 0.5 s with 3D bounding box annotations over 23 object classes (cars, trucks, pedestrians, etc.) and rich attributes (object pose, visibility, etc.). High-definition semantic maps of the recorded areas are also provided within the dataset [22, 51].

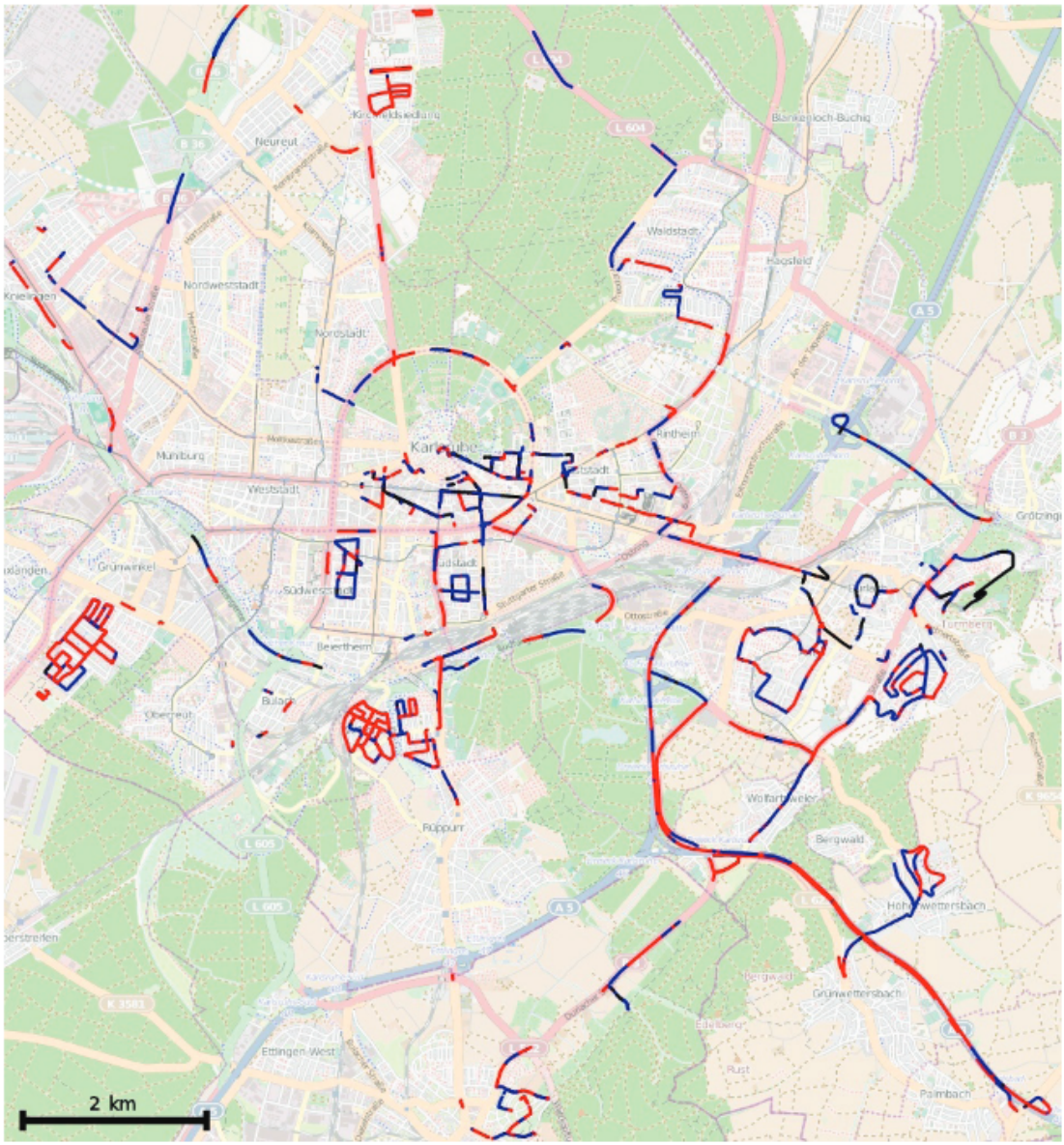


Figure 2.5: GPS Traces of the sequences in the KITTI Dataset (red tracks have a higher GPS precision than the blue tracks due to the usage of RTK corrections, black tracks did not have any GPS signal, so they are not contained in the dataset) [20]

2.6.3 Argoverse

Argoverse has two versions of data released (2019 and 2022). Argoverse-1 was the first dataset collected in Miami and Pittsburgh using Argo AI test vehicles, which were equipped with a sensor rig that has 7 ring cameras of high resolution (1920×1200 pixels each, 30 Hz, together giving 360° surrounding view) and two front-facing stereo cameras

(each 2056×2464, 5 Hz). Two overlapping 40° vertical FOV roof-mounted Velodyne HDL-64Es offer 200 m range and 107 k points per frame at 10 Hz. Complete 6-DOF GPS/IMU localisation and high-accuracy HD maps (lane centerlines, driveable area) are available. The recorded drives cover diverse urban environments (city downtowns, suburbs, different weather conditions and times of day). Argoverse-1 3D Tracking subset has 113 labelled sequences (17 k frames) with 3D bounding-box tracks (vehicles, pedestrians, cyclists). Argoverse-2 is significantly larger in scale: it includes 1000 labelled 20 s sequences from multiple U.S. cities (seven ring cameras + two stereo + LIDAR same as before), with 10 Hz 3D bounding box annotations for 30 object classes, and dense ground-truth semantic labels and height maps [23, 52].

2.6.4 Waymo Open Dataset

The Waymo Open Dataset (2020–2024) is one of the largest autonomous driving datasets and benchmarks. The data was recorded using Waymo's self-driving car fleet in the San Francisco Bay Area, Phoenix and Mountain View. Five high-resolution cameras capture the front and the sides of the vehicle (three are 1920x1280 pixels for front/forward-left/forward-right angles and two are 1920x886 pixels for left/right side angles) at around 20–30 Hz, with full 360° coverage. It also has five LIDAR sensors (one high-res rotating LIDAR, along with four short-range LIDAR units) spinning at 10 Hz. All five LIDARs send synchronised point clouds (with the mid-range glspl lidar sweeping 75 m range, side/short-range LIDARs 20–25 m). Complete localisation is provided by an IMU/GPS/INS. The data comprises urban and suburban scenes, daytime and nighttime, under a wide range of weather (but largely clear weather) and is geographically highly heterogeneous, meaning there is not much terrain diversity in the dataset. The perception dataset release has 2,030 driving segments (390 k total frames, 20 s each). It includes detailed annotations: 3D bounding boxes with track IDs for pedestrians, cyclists, vehicles and signs on the LIDAR data and corresponding 2D boxes on the camera images (more than 12 million labels per set). There are additional 2D/3D semantic segmentation, panoptic segmentation (pixel-wise segmentation) and keypoint labels on some of the data [21, 53].

2.6.5 CARLA (pseudo dataset)

CARLA is a Unity-based open-source driving simulator used typically to generate synthetic perception datasets. In CARLA, an arbitrary number of sensors can be simulated on an ego-vehicle: e.g. multiple RGB cameras (default 800x600 pixel, 90° FOV, up to 100 Hz, configurable intrinsics), depth and semantic segmentation cameras, a spinning LIDAR (usually 32 or 64 beams, default ±10° vertical FOV, 360° horizontal, at 10 Hz) or a radar (default 77 GHz, 30° FOV, 1500 points/sec). GPS and IMU readings can be synthesised as well. The worlds (urban scenes, traffic, pedestrians) are completely synthetic but support varied lighting (day, dusk, night) and weather (sunny, rainy, foggy) conditions. CARLA can supply perfect ground-truth annotations for every visible object (instance segmentation masks, 2D and 3D bounding boxes for cars/pedestrians/cyclists,

road geometry, etc.). Pseudo-datasets generated by CARLA typically contain ground-truth labels that are perfect for training/testing perception algorithms in controlled environments [54, 55].

A brief comparison of the used sensors and included data of the individual datasets can be seen in Table 2.1.

Table 2.1: Comparison of sensor availability and the number of sensors across major autonomous driving datasets.

Dataset	**Cameras**	**LiDAR**	**Radar**	**GPS/IMU**	**HD Maps**
KITTI (2012)	✓(4)	✓(1)	✗	✓	✗
nuScenes (2019)	✓(6)	✓(1)	✓	✓	✓
Argoverse 2 (2022)	✓(9)	✓(2)	✗	✓	✓
Waymo Open (2020)	✓(5)	✓(5)	✗	✓	✓
CARLA (2017)	✓(∞)	✓(∞)	✓	✓	✓

This concludes the related work for this thesis, listing end-to-end frameworks that perform similar tasks to the pipeline that will be presented, as well as highlighting different methods for solving the sub-problems present in this thesis. In the next section, the methods used for solving the object detection, object tracking, and trajectory prediction will be discussed and explained in more detail, as well as the structure and usage of the datasets used or generated.

CHAPTER 3

Methodology

In this Chapter, the methodologies used in the implementation of this work, as well as other methods that were considered or tested, will be described. The focus is on understanding how data is processed to understand strengths and weaknesses, and why they have been selected to be used in this thesis implementation.

3.1 Super fast and accurate 3D object detection

While there are many published works on object detection, open-source, well-documented implementations for detecting 3D bounding boxes on LIDAR point clouds are way less common. The primary sources for comparing sources on this topic were papers with code, which is unfortunately no longer active, and the Benchmark results listing on the KITTI dataset webpage. The starting point for finding a suitable method was finding a YOLO-based implementation for 3D objects, which led to the open-source implementation of "Complex YOLOv4" [56].

While the implementation meets the input and output requirements, it is relatively slow. But the implementation references a newer, faster 3D bounding box method called "Super fast and accurate 3D object detection", which forms the first stage of the pipeline performing the object detection task [57]. It is a real-time 3D detection framework based on RTM3D [58]. SFA3D operates on BEV image representations of the input LIDAR data. From this view, the model extracts keypoints, such as the centre of objects, their physical dimensions and their orientation. By predicting these properties directly, SFA3D does not require post-processing of the detected object bounding box like non-max suppression, which is often required in other detection pipelines. The model predicts the object centres using a heatmap per object class it is trained to detect. This method builds upon a ResNet backbone and a keypoint feature pyramid network, and outputs objects with 7 degrees of freedom. Figure 3.1 demonstrates the outputs of SFA3D. The method was picked because it is MIT licensed, works solely on LIDAR input without

any need for prior segmentation or other forms of data preprocessing, contains good execution instructions and performed well on the KITTI benchmark.

Figure 3.1: Demonstration of the outputs of SFA3D by visualising both the RGB camera input image as well as a top-down view of the LIDAR point cloud annotated with the detected objects. Note that although visualised on the RGB camera input, the detection does not utilise the camera image.

3.2 3D Multi-object-tracker

The first considerations for tackling the problem of object tracking were the baseline object tracking method SORT [33] or its deep learning variant DeepSORT [35]. The open source implementation for SORT [59] and DeepSORT [60] offer examples and demos for the tracking capabilities, but focus on 2D bounding box tracking. Although the methods are extendable to track 3D bounding boxes, these implementations struggle with occlusions of objects over several frames as they only implicitly keep track of the object's trajectory internally through a Kalman Filter, but don't actively forecast future positions.

For this reason, a more mature method was picked. The 3D multi-object tracker used to assign IDs to the object detections and track them over multiple frames is a reproduced and simplified implementation of the paper "3D Multi-Object Tracking Based on Uncertainty-Guided Data Association" [61]. The method also uses a Kalman filter to predict the next positions of the tracked objects and associates detections using greedy matching,

but compared to SORT and DeepSORT, a larger state vector for the 3D tracking and a different motion model are used. Furthermore, the method was chosen because it is very fast, and the implementation is based on 3D bounding boxes with 7 degrees of freedom and supports input of bounding boxes in the format provided by the KITTI dataset annotations.

3.3 Predictions conditioned on goals in visual multi-agent scenarios

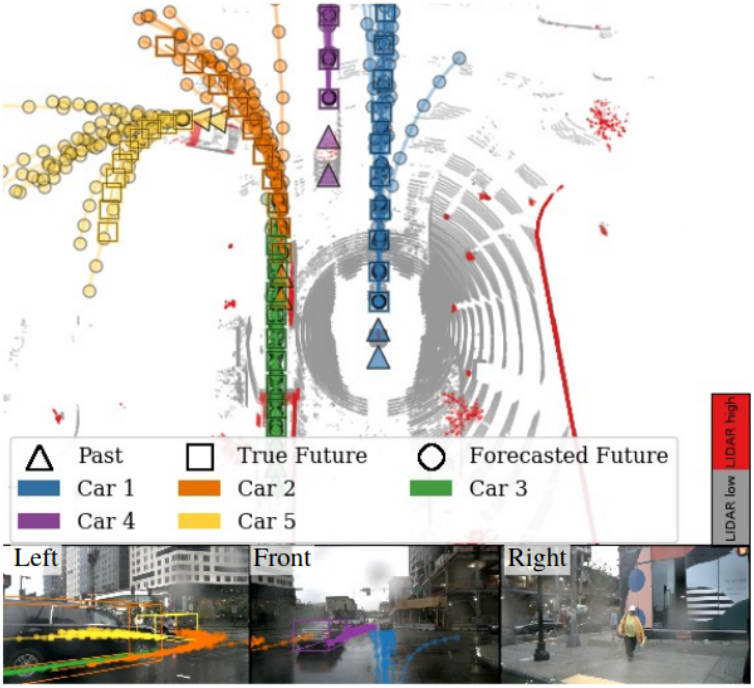


Figure 3.2: Predicted trajectories from the PRECOG model on the nuScenes dataset. The model receives a LIDAR point cloud and the past trajectories of several road agents as inputs and outputs multiple plausible future trajectories per agent. The image illustrates the predictions and the ground truth for the agents' trajectories.

Finding a suitable method for trajectory prediction turned out to be very challenging. This is due to less published work on the topic compared to the tasks of object detection or tracking, but also due to a wider range of possible input types and output formats. For example, most methods for trajectory prediction are designed for autonomous driving applications, which often use additional inputs like camera, radar or HD-maps. As for this work, the requirements on the inputs are more restrictive, and many methods are not open-source. At first, it seemed necessary to implement a custom solution, but a promising method, MANTRA [62], fulfilled most of these requirements, as it is open-source and requires only LIDAR and the past trajectories of the agents as inputs, and

outputs multiple plausible future trajectories. Though pre-trained models are available, the problem with those is that they were trained using complete BEV maps of the whole sequence, which effectively means the method is for an offline but not online use-case. But since the input of the scene context is an image, the idea was to train the model for an online use-case by simply only providing a BEV representation of the LIDAR point cloud of the current frame during training. However, this turned out not to achieve any reasonable results during training, so this approach was discarded. The implementation of the method also does not provide any format specification for the training data, making it hard to train on other training data than the KITTI dataset, for example, synthetic data produced by a driving simulator.

Ultimately, the method used for the trajectory prediction stage of the pipeline implementation is a deep learning model based on the idea to predict goals for the objects of which to predict the trajectories instead of directly predicting the trajectories. The model incorporates two encoders, one for encoding the past trajectories of the objects and one for encoding the scene context represented by the LIDAR point cloud. These encodings allow the model to learn interaction patterns. As an intermediate output, the model predicts several goals for each object, while the actual trajectories are then produced by a recurrent generator. These generated trajectories are then conditioned on the goals produced by an intermediate layer of the model. The paper's implementation [63] was chosen because the author provided a separate implementation [64] to train the model using data generated from the CARLA driving simulator, which encodes information about the scene from a LIDAR point cloud, and the model can output trajectories of an arbitrary number of agents. Although the paper describes a flexible count implementation, the provided model is limited by a fixed count of nearby objects. An illustration of the model's inputs and inputs for five road users can be seen in Figure 3.2, which compares the ground truth to several predictions made per tracked object.

CHAPTER 4

Implementation

The pipeline for detecting, tracking and predicting the future paths of vehicles was implemented using Python 3.8.19 and was initially implemented and executed in a Windows environment. But the usage of PRECOG's CARLA driving simulator integration required using Linux because the CARLA driving simulator version 0.8.4 required for the dataset generation for training the PRECOG model did not support Windows (or more specifically, the Windows download for this release was no longer available). Although the recommended Python versions of the used implementations of the pipeline would have required different Python versions, they were upgraded to version 3.8 in order to allow the usage of the rerun package with this minimum required version. Rerun enabled the use of enhanced visualisation and debugging of the pipeline. For training and inference of machine learning models, on the one hand, PyTorch 1.5.0 was used for object detection (SFA3D), while TensorFlow 1.15.0 was used for the trajectory prediction (PRECOG). Although mixing TensorFlow and PyTorch versions is generally not considered a good practice, in this case, due to the limited choice in open-source trajectory prediction models which meet the requirements, the downsides of mixing both were accepted. In the following Chapter, the design and structure of the pipeline will be described in more detail.

4.1 Pipeline

The implemented end-to-end pipeline requires a stream of input LIDAR point clouds and a camera-to-world transformation matrix. Example input data are visualised in Figure 4.2. The requirements for how the LIDAR point cloud is structured and how it was recorded for the pipeline to properly function are dictated by the pretrained object detection model. The utilised SFA3D model was trained on the KITTI dataset, where the LIDAR sensor was mounted on top of the car at a height of 1.73 m from ground level and can process 120 k points per second. With a field of view of 360° horizontally

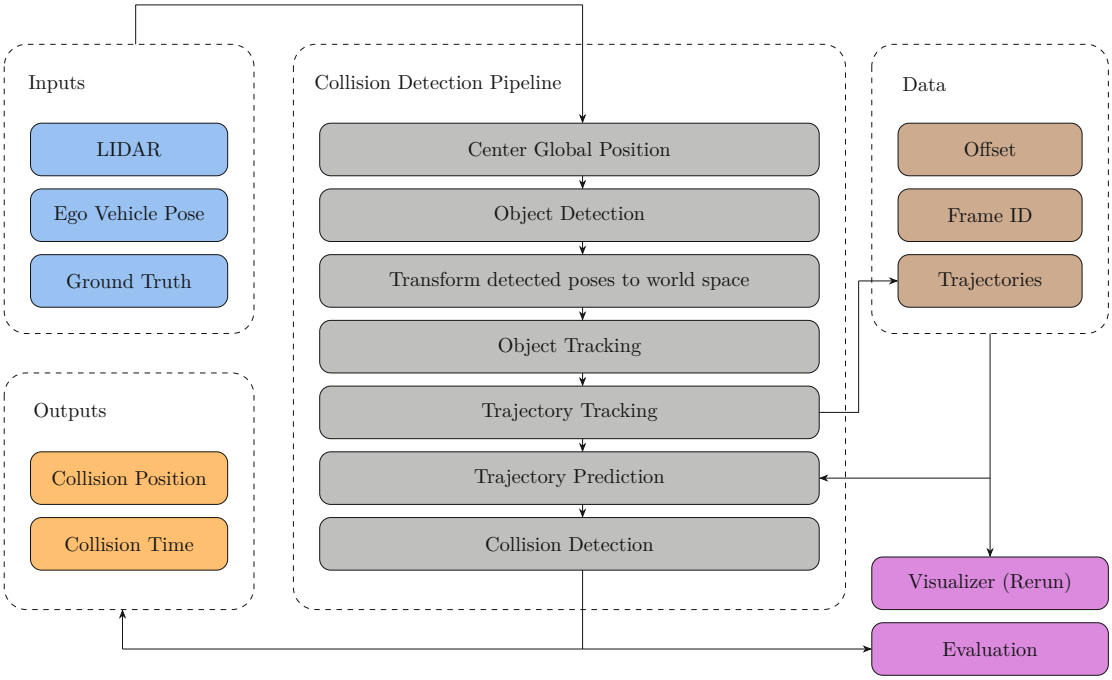


Figure 4.1: Pipeline structure overview

and 26.8° vertically, the sensor provides the coverage necessary for detecting most road users within a distance that is close enough that they could interfere or collide with the ego vehicle. The resulting point cloud is transformed into a bird-eye view map of the environment in both the object detection and the trajectory prediction models. Although the preprocessing of the point cloud is very different in the two models, unfortunately, it is not possible to perform the preprocessing step only once and use it for both models. Figure 4.1 illustrates the inputs, outputs, processing nodes and intermediate data stored in the pipeline, as well as the side modules for visualisation and evaluation, which will be explained in more detail in the following subsections.

4.1.1 Object detection

The object detection implementation used, as mentioned before, SFA3D, is based on the RTM3D paper, which works only on monocular RGB images. SFA3D applies the key concepts of RTM3D to LIDAR point clouds. The core innovation of RTM3D lies in the prediction of nine keypoints that correspond to the 3D bounding box of a road user. Eight keypoints correspond to the corners of the bounding box projected onto the image, including the occluded corners, and the ninth keypoint corresponds to the centre of the bounding box. Using these keypoints, the 3D dimensions, location, and orientation can then be recovered by leveraging geometric constraints between the 2D projections and the 3D points. In order to detect objects with varying sizes, RTM3D employs a ResNet-based keypoint feature pyramid network (KFPN) to handle multiscale

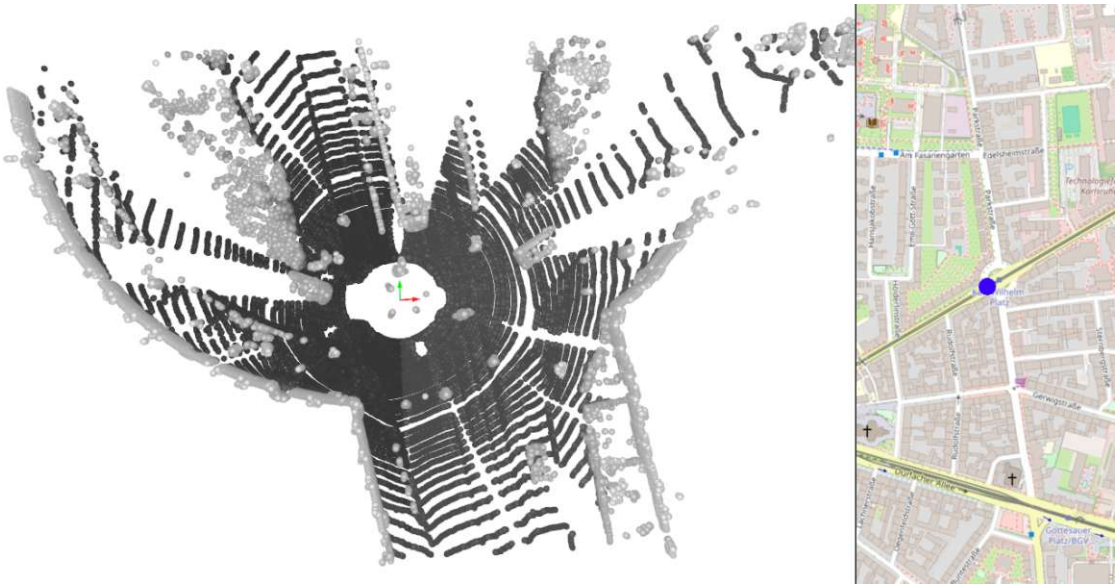


Figure 4.2: Pipeline input - left: LIDAR input (z-coordinate is encoded using a grayscale transition from dark for the street to light grey for obstacles, right: GPS input (Map data from OpenStreetMap)

feature extraction. Relying on and processing of only one input allows the RTM3D implementation to achieve real-time performance on the KITTI benchmark. Now, to apply the KFPN, which takes an image as input, to LIDAR point clouds, the point cloud is transformed into a bird's-eye view representation. In more detail, each point of the point cloud is projected into the image, and the three colour channels encode the height, intensity, and density of the point cloud. The point cloud is divided into a front part and a back part. In principle, this could also be done in a single pass, using only one BEV image of the whole point cloud. But splitting the point cloud into the front and back view of the vehicle allows the design to adapt to, for example, LIDAR sensors that have a limited field of view, usually focusing more on the front of the vehicle. By separating the BEV maps, the system can better handle differences in the density or distribution of the point-cloud data. Other objects in the front and rear of the vehicle may have different characteristics due to a difference in perspective or occlusion. In summary, object detection consists of the following steps:

1. Split LIDAR point cloud into front and back facing parts
2. Create BEV images of both sides, encoding height, intensity and density
3. Detect vehicles on the front BEV image (by inference of the described model)
4. Detect vehicles on the back BEV image

$$
\begin{aligned}
R &= 6378137 \quad \text{Mean Earth radius, WGS84 standard} \\
s &= \cos(\text{latitude}_0) \quad \text{Latitude of first observation (in radians)} \\
x &= s \cdot R \cdot \text{latitude} \quad \text{Latitude (in radians)} \\
y &= s \cdot R \cdot \ln\left(\tan\left(\frac{\text{longitude}}{2} + \frac{\pi}{4}\right)\right) \quad \text{Longitude (in radians)}
\end{aligned}
\tag{4.1}
$$

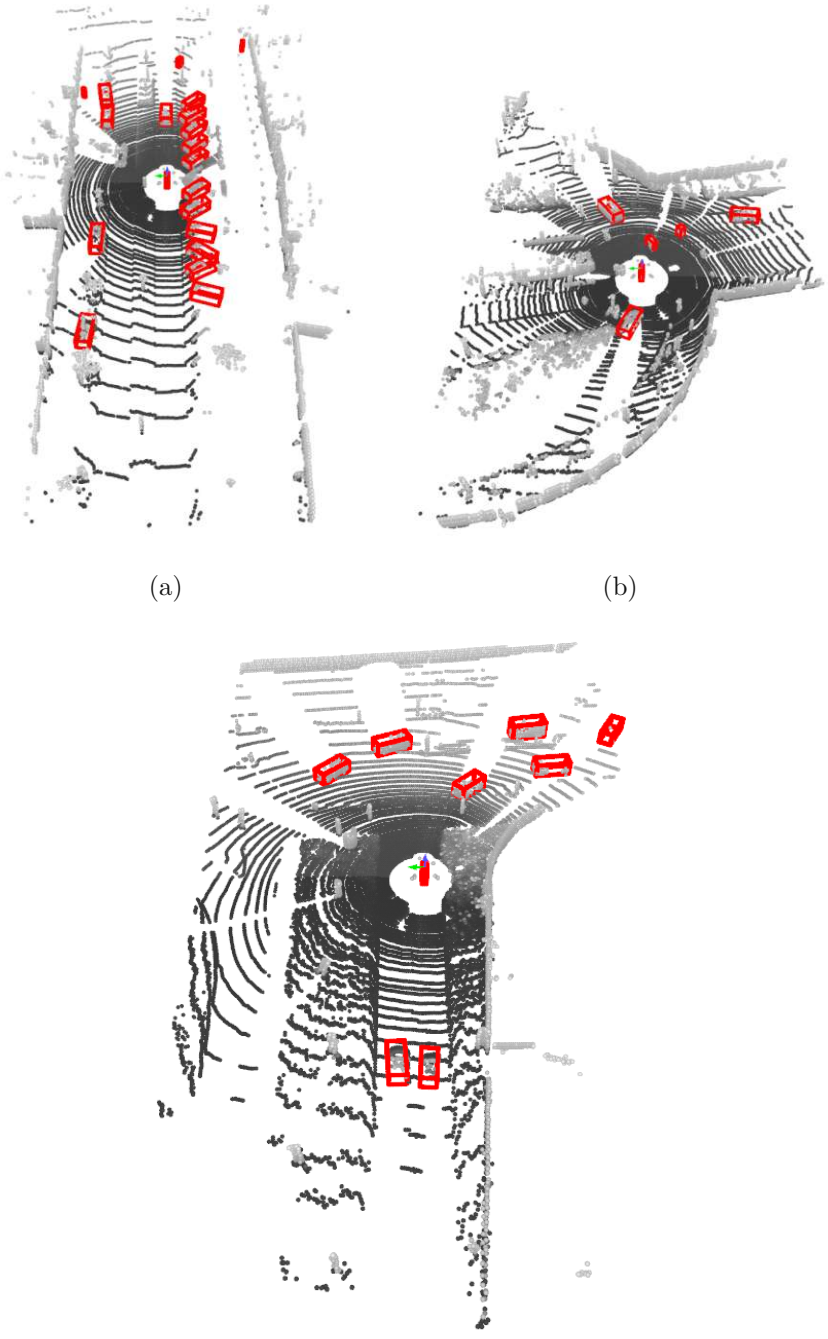
The output of SFA3D is the 3D bounding boxes in the format (score, x, y, z, height, width, length, yaw), where the position (x, y, z) and the heading angle (yaw) are relative to the ego vehicle. To retrieve the trajectory of a vehicle relative to the ego vehicle, the position must be transformed into a global frame of reference. The obvious solution here might be to use the GPS location of the ego vehicle to transform the detected bounding boxes into world coordinates. While this procedure works, it has an issue with precision, which leads to very inaccurate trajectories. The following example illustrates the problem: The unit used for detections is meters, while GPS positions are given in degrees for the longitude and latitude. To compare the detections to the GPS location, the Mercator projection and the mean earth radius (according to the WGS84 standard) are used according to Equation 4.1, resulting in the ego vehicle's position in meters. When converting geographic coordinates from latitude and longitude (in degrees) to metric coordinates using the Web Mercator projection, the resulting values are typically in the range of 10^6 meters. Incorporating additional small-scale measurements, such as object detections within ± 50 meters, into these large coordinate values can lead to floating-point precision errors, which were observed to introduce positional inaccuracies of up to approximately 30 centimetres. These inaccuracies are further compounded when applying additional transformations, such as rotation and translation relative to the ego vehicle's local reference frame to a global world frame. Experimental results indicated that the total accumulated error could reach up to one meter. To mitigate this issue, the initial GPS coordinate was used as a fixed spatial anchor, and all subsequent GPS positions were expressed relative to this reference point. This anchoring approach effectively re-centres all computations around the origin, significantly reducing floating-point error accumulation and improving overall positional accuracy. Pseudo-code of the transformation from local to global coordinates originating around the first observation of the ego vehicle can be found in Algorithm 4.1.

Algorithm 4.1: Transform detections from local to global coordinates originating around the first observation to enable trajectory tracing

Input: metadata, metadata_0, detections (Metadata of first and current frame, consisting of GPS position and IMU measurements, and front and back-facing object detections)

Output: $\text{detections}_{global}$ (Detections transformed from local to global coordinates)

1. Compute camera-to-world matrix using Equation 4.1
$\mathbf{T}_{cw} \leftarrow \text{camera\_to\_world\_matrix}(\text{metadata}, \text{metadata}_0)$
if $\mathbf{T}_{cw,offset}$ *not initialized* **then**
    Store translation of $\mathbf{T}_{cw}$ as $\mathbf{T}_{cw,\text{offset}}$ (origin reference)
end
2. Centre transformation around offset
Subtract $\mathbf{T}_{cw,\text{offset}}$ translation from translation components of $\mathbf{T}_{cw}$
3. For each detected object (front and back detections)
foreach *detection in detections* **do**
    Transform detection point from local coordinates to ego vehicle frame (involving an 180° rotation for back-facing detections and a scaling factor to retrieve detection position in meters)
    Apply centered $\mathbf{T}_{cw}$ to obtain global coordinates
    Adjust orientation of detection by ego vehicle heading angle
    Append transformed position and orientation to the global detection list
end



(a) (b)

(c)

Figure 4.3: Output of the object detection: detections are shown with red bounding boxes, while the axis in the centre shows the ego vehicle position.

4.1.2 Object tracking

The 3D multi-object tracker receives an input, a timestamp, in this case the frame index, as the timestamps of the datasets are very stable with a very low variance (for KITTI below 10^{-4} seconds), for which reason the timestamp can be omitted as an input to the pipeline. Further, the current pose, which is a 4x4 matrix of the ego vehicle pose (shifted by the origin) and a detection score for each bounding box. And last but not least, the actual detections. Internally, the tracker keeps track of the detections by storing them in a list of dead and active trajectories. Dead trajectories are trajectories that have not been updated for a while. For each active trajectory, a prediction for its next state is made in order to match new detections with the existing trajectories. To predict the next state (positions) of the trajectories, the tracker uses a Kalman filter internally in the following manner:

1. Retrieve the previous positions of the tracked objects.
2. Determine which positions to use: If a previous position was not updated, use the predicted position.
3. Calculate a prediction score depending on whether the position was updated or not (meaning the object was missed during detection in the last state). Penalise if the position was not updated in the previous state (score decay).
4. Predict the next positions using the standard Kalman filter prediction process, applying the state transition matrix and the state covariance matrix. See [65] for more details about the Kalman filter formulation and [66] for the implementation.

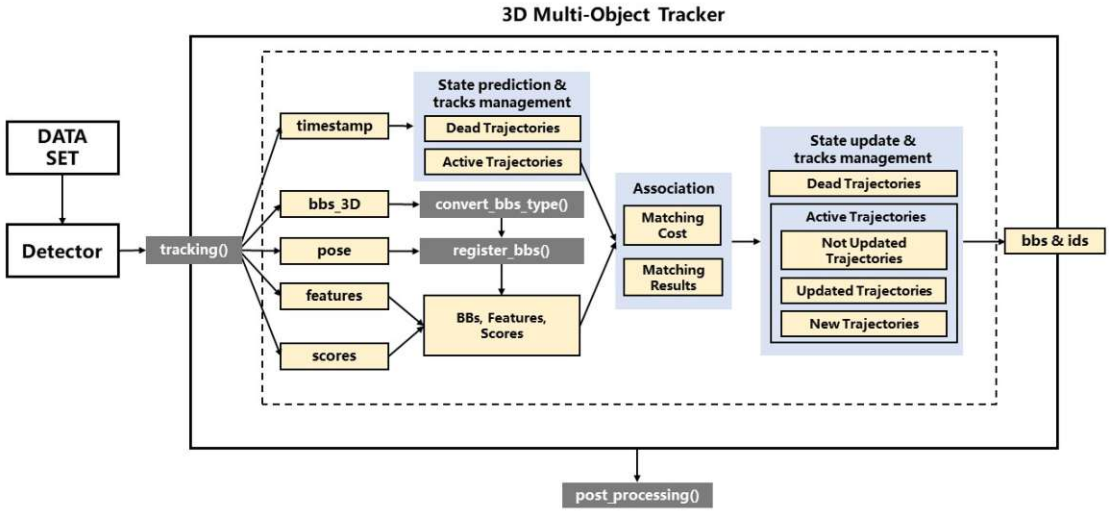


Figure 4.4: Multi-object tracker framework used to perform the bounding box association across multiple frames [66].

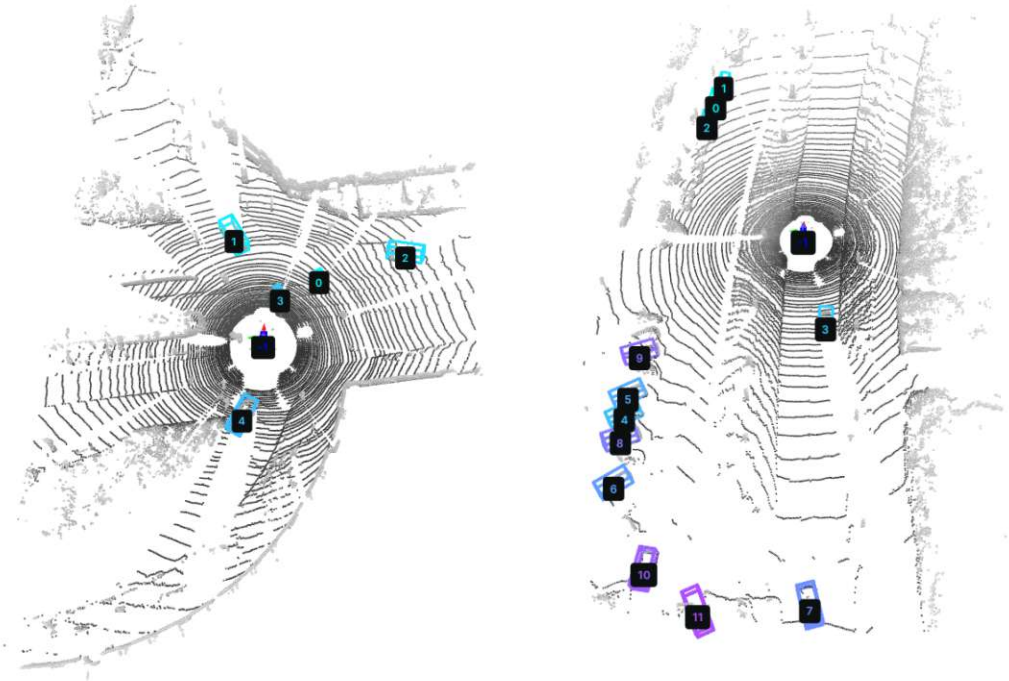


Figure 4.5: Example of labels assigned by the object tracker to each detection.

Now, in order to assign the new detection with existing trajectories, or in case no existing trajectory fits, a new object ID and a new trajectory are created, and a cost map between all detections and the predicted states is computed. The cost map contains all Euclidean distances between each pair of state predictions and detection. Using the cost map, a greedy data association between the predictions and detections is performed, assigning each detection to an existing tracked object. If the minimum cost for a detection is smaller than a given threshold (in this case, a threshold of 2.0 meters is used), the detection is matched. Otherwise, the detection is treated as a new object. Figure 4.5 showcases the greedy label assignment. If a trajectory failed to be updated for a specified number of frames, the trajectory is removed from the list of active trajectories, which are considered for the tracking, and added to the list of dead trajectories. Figure 4.4 shows the structure of the object tracker. Note that keeping the dead trajectories has no direct purpose but is used for post-processing purposes only. Since the tracking is not perfect and errors occur or propagate from the detector, the resulting trajectories may contain errors. Figure 4.6 visualises all tracked paths in a sequence from the generated CARLA dataset presented later on in this thesis, illustrating the expected length and accuracy of the tracking over a longer sequence.

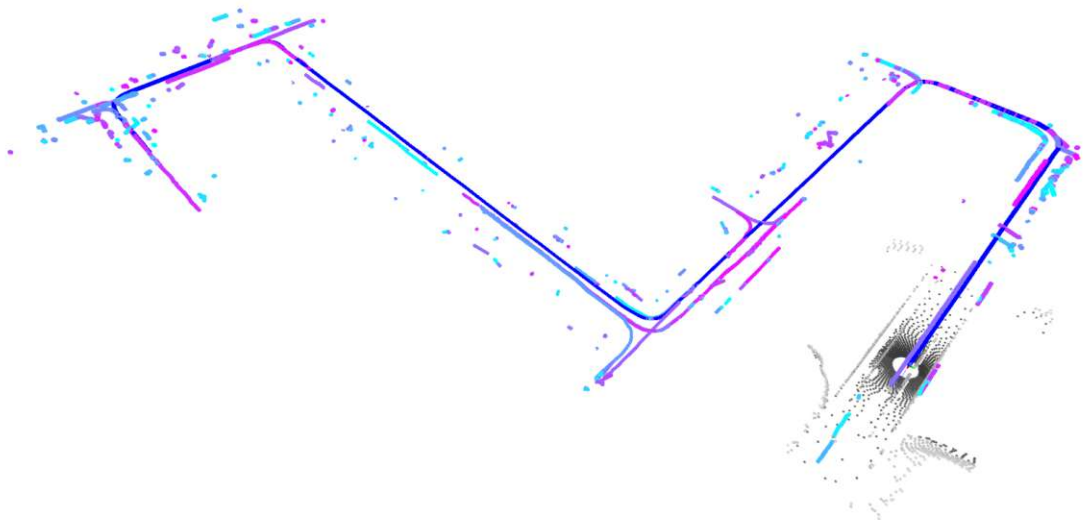


Figure 4.6: Tracked paths resulting from the object tracking (blue to purple trajectories). The latest available LIDAR data is also visualised.

4.1.3 Trajectory prediction

This section will talk about the challenges faced while testing different approaches to trajectory prediction in practice and the integration of the PRECOG model. The goal at this stage is to predict future trajectories for all currently tracked agents, using all information that is available. That includes the scene as a LIDAR point cloud. It also includes past trajectories of the tracked agents, which can vary in length and quality. But as mentioned already, the PRECOG model implementation uses a fixed-size model, so in case too many agents are tracked simultaneously, only the nearest-K agents are considered for prediction. But before that, a discussion of considered and tested methods:

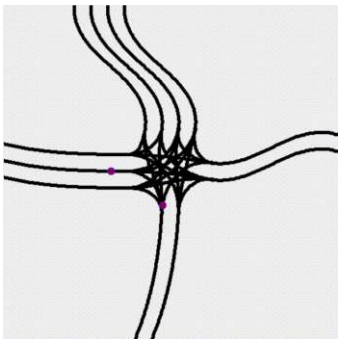


Figure 4.7: Implementation of a simple trajectory generator using clothoids with a varied number of input lanes at an intersection and vehicles driving through it.

A basic way to predict future motion is a constant velocity model, which assumes each vehicle keeps moving with the same velocity and direction from the past. This means to estimate positions at future times with simple kinematic equations, simply by extrapolating the current state. This method does perform well for very short-term predictions and works in simple traffic situations, like vehicles on a straight road at constant speed. It is easy to implement and computationally efficient, for which reason, it is appealing for real-time systems; still, the big drawback is that it ignores the road layout and restrictions given by the surrounding area, which limit movements or may affect the most probable future trajectory. It also misses out on interactions with other road users. Meaning that a constant velocity model cannot predict turns, lane changes or braking behaviours, which happen a lot in urban areas. Predictions get inaccurate fast as the length of the horizon of the prediction increases.

To improve trajectory prediction and to create more training data for learning based models, one attempt was made in generating synthetic trajectories, using a simple self-developed 2D driving simulator based on clothoids (see Figure 4.7). Clothoids are a type of curve, also known as the Euler spiral, which are often used for modelling vehicle trajectories or road layouts, as when traversing clothoids, the rate of change in steering is constant. The simulator implementation made vehicle paths through intersections modelled with clothoids. Those had different numbers of incoming and outgoing lanes. Vehicles travelled at various speeds. By tweaking curvature and speed parameters, it

produced smooth, realistic paths in a wide range. But we ended up dropping this approach. Modelling real intersections and vehicle interactions was complicated, even in two dimensions. One would need many extra rules for behaviours like stopping or yielding, or merging. Plus, tools like CARLA [67] already handle this well. They offer realistic environments, accurate physics and agent behaviours. Those generate high-quality trajectory data more efficiently, so the custom simulator was not needed.

Simple physical models have limits, though; learning-based methods offer a more flexible option, as already mentioned in the related work section of this thesis. These can use motion history, but also context from the scene and other agents. Trajectory data is sequential, meaning models are required that can handle temporal dependencies, similar to natural language processing, speech recognition or, in this case, this problem is also known as time series forecasting. Common approaches to solve this task that have evolved are RNNs, LSTMs, GRUs, Transformers and Hidden Markov Models.

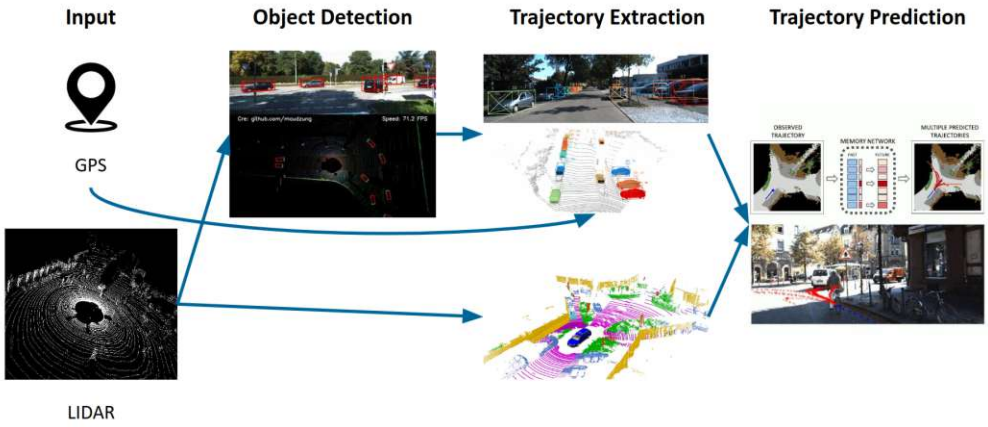


Figure 4.8: Initial proposed pipeline using MANTRA [62], which required a separate model for semantic segmentation on the LIDAR data.

The first proposal of the pipeline (see Figure 4.8) was based on the open source model named MANTRA. It was originally trained on top-view semantic maps of the whole scene (as shown in Figure 4.9). Though the model encodes not just the current lidar frame, it requires the full sequence of frames to capture the whole scene. This suits offline applications fine, but not online operation as required here, where future LIDAR frames are not available yet. To adapt it for real-time, retraining using only the current lidar frame for the top-view encoder was required. This approach was tested, but unfortunately, the results were not satisfying, most likely because the original MANTRA model relies on semantic info from datasets like KITTI, which is missing in this online setup. One fix for this could have been a separate model to add semantic labels to each lidar frame first, then pass the labelled LIDAR to MANTRA. We did not follow through on that, though, because the approach already seemed unlikely to be successful and the extra processing time per frame would exceed the real-time limit, which is set by the lidar sensor at 10 Hz. In the end, as already presented in Chapter 3, we picked the target-driven trajectory

prediction model PRECOG, which does not require semantic labels for the lidar points and works in an online setting, unlike MANTRA, which was trained on a complete BEV map. Like MANTRA, though, a drawback of PRECOG was that the pre-trained model was trained solely on ego vehicle trajectory input, although the model is capable of considering and predicting the trajectories of a fixed count of road users.

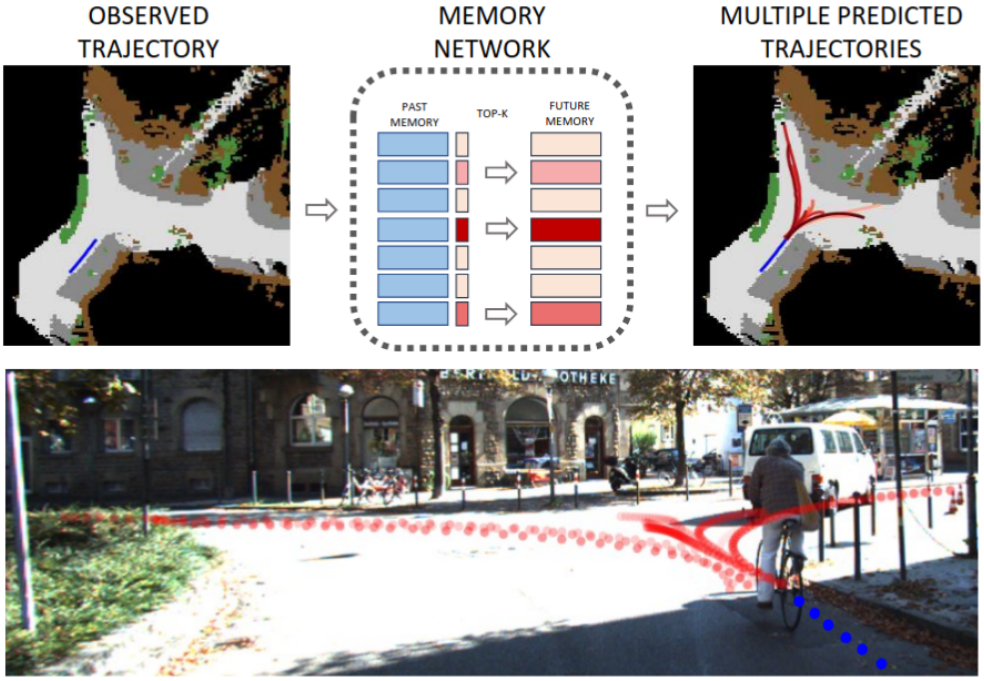


Figure 4.9: MANTRA model: input (left), output (right)

4.1.4 Collision detection

As a last step, the goal is to detect potential collisions using the trajectory predictions. Now, one simple method would be to just search for intersections in the trajectory predictions. But this does not consider the fact that two road users may have intersecting trajectories but do not collide because they passed the intersection point at different times. For this reason, the bounding box of each vehicle is determined by the extent of the bounding box from the first detection of this road user. Or more specifically, the first time a road user with label x got assigned that label by the object tracking step. Then the bounding box is swept over the trajectory and checked for intersections with other bounding boxes at each point in time. If an intersection is found, a potential collision is detected.

4.2 Data Structures

This section mainly describes the structure of the data used for training the trajectory prediction model, but also briefly explains what data transfer objects were used inside the pipeline.

The training input parser parses a series of JSON files storing the current state of the scene as well as past and future trajectories. The JSON files are directly produced by invoking the CARLA driving simulator script based on the scripts provided along with the PRECOG model, but adapted for this thesis. These output a JSON in this format every n-th frame. The LIDAR is not saved directly but parsed to a BEV representation before being saved to save disk space and speed up training of the model. The agents' past and future trajectories are sorted by distance to the ego vehicle. All heading angles (yaw) are stored relative to the ego vehicle. The structure of the training data format is as follows:

- `player_past` - 2D past trajectory of the ego agent, shape $[2, T_{\text{past}}]$
- `player_future` - 2D future trajectory of the ego agent, shape $[2, T_{\text{future}}]$
- `player_yaw` - Heading angle (yaw) of the ego agent, shape $[1, 1]$
- `agent_pasts` - Past trajectories of other agents, shape $[2, T_{\text{past}}, N]$
- `agent_futures` - Future trajectories of other agents, shape $[2, T_{\text{future}}, N]$
- `agent_yaws` - Heading angles of other agents, shape $[1, N]$
- `overhead_features` - Bird's-eye view LIDAR features, shape $[H, W]$
- `LIDAR_params` - Dictionary with LIDAR metadata:
  - `pixels_per_meter` - Resolution of BEV image
  - `val_obstacle` - Value representing obstacles in BEV
  - `hist_max_per_pixel` - Max history count per pixel
  - `meters_max` - Maximum LIDAR range in meters

As shown in Chart 4.1, the pipeline maintains a list of detected and tracked vehicles, their past trajectories and the current future predictions in memory. The detections are stored in the format provided by SFA3D [57], but the output of the two-phase detection (front and back of the ego vehicle) is parsed into one list. The detections are stored relative to the ego vehicle position but also relative to the global coordinate system. This corresponds to the first GPS location of the ego vehicle, which is done to avoid numerical issues during conversion from GPS coordinates to meters, as previously mentioned. The heading angle is also saved relative to the ego vehicle as well as globally. The relatively

stored detection results are used upon invoking the tracker, which outputs a list of bounding boxes and a list of IDs, stored as tuples, before being appended to the list of all tracked vehicles and their trajectories. The trajectory data structure consists of:

- `positions`: all past positions of this tracked object
- `yaws`: all heading angles of this tracked object (with the same size as the positions field)
- `sample_indices`: a list of indices, corresponding to the frame indices where the observations (positions/yaws) of this vehicle were detected/tracked. This field is used to check if the trajectory is outdated or if several detections were missed, and the intermediate trajectory needs to be interpolated
- `prediction`: If enough past positions are available, this field contains the current best prediction (out of a set of 12 provided by the trajectory prediction model)
- `all_predictions`: All 12 predictions

All data structures utilise the vector/matrix representations provided by the Python version of the OpenGL Mathematics (GLM) library [68].

4.3 Visualization

Visualisation was a big part of this work as visualising the intermediate steps and the outcome of the pipeline is essential to gain a better understanding of the processes and potential issues/bottlenecks, and verify the correctness of the implementations. While at first, standard Python tools like matplotlib for displaying images and point clouds, or tkinter for creating an interactive interface, were used, creating implementations that were scalable and interactive was time-consuming and hard. For this reason, the visualisation was finally done using rerun [6]. Rerun is a fairly new visualisation tool that relies on a server-client-based setup, where the client is the user-side code, in this case, the collision detection pipeline, and the server is a separate application running on the local host. Although not used in this work, this can be especially useful when running this pipeline on a separate device, but streaming the data to a desktop machine to visualise it. So, in comparison to writing your own visualisation code, the procedure is as follows:

1. Connect to rerun-viewer (or spawn a viewer if none is present)
2. Notify the rerun-viewer about the current frame ID
3. Stream data converted to rerun visualisation data-types during program execution (for example, LIDAR input, detected bounding boxes, etc.)

4. Disconnect from server (program execution stops, but the streamed data remains visible in the rerun-viewer)

After the execution, the streamed data can then still be observed in the viewer and stored on the disk as rerun recording files. This has the further benefit that the recordings can be shared with other people without the need to set up the pipeline and install all Python packages, but rather just install the rerun viewer. The rerun viewer itself organises all streamed data in timelines and uses a hierarchy of data streams. In this work, the timelines represent different KITTI and CARLA sequences recorded. The hierarchy of data streams is used to organise the data into everything as follows:

- **Inputs** LIDAR point cloud, GPS location, ego vehicle bounding box
- **Ground truth** If available, the full trajectory of the ego vehicle, the future and past paths of all other road users
- **Outputs** Outputs of the pipeline as well as intermediate outputs, which include all tracked detections, all tracked past paths and all predicted paths.

See Figure 4.10, which shows how the data can be organised in the rerun viewer using a 3D view to visualise the current state of the pipeline, and a map view (based on OpenStreetMap) to visualise the GPS location. The slider at the bottom allows scrolling through the recording in a video-recorder-like fashion and visualises the pipeline state at any given frame.

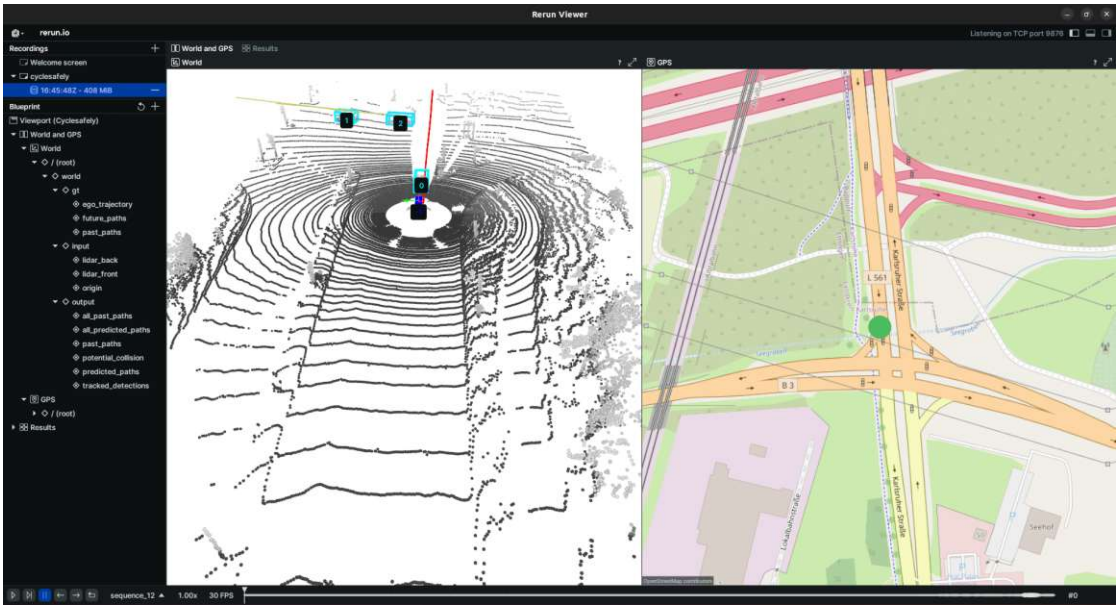


Figure 4.10: Visualisation of the pipeline inputs and outputs using the rerun-viewer.

On the left side of the viewer, the data hierarchy that is being visualised can be seen. The logged data is organised into inputs, outputs, ground truth and evaluation results, visible in the side panel. As inputs, the LIDAR point cloud is being logged (but for every frame except the first one, only a subset of the point cloud is logged to save data and increase performance), the GPS location and a 3D axis representing the scene origin (this is a verbose log and not really required). The outputs consist of the detected objects represented by a set of bounding boxes with labels, as well as different set of paths. Furthermore, the output contains the past paths of the currently tracked objects, as well as the past paths of all objects tracked so far but which are not tracked anymore. Additionally, the predicted paths of the currently tracked objects (12 per object), as well as only the most likely path prediction, are also logged to the visualizer. Further, in case a potential collision is detected, a red sphere is logged to indicate where the potential collision might occur.

The logged data is organised into timelines, which can be used to switch between the logged data from different recorded sequences. This allows us to inspect a collection of sequences after executing the collision detection pipeline on them. Note that, especially depending on how densely sampled the point cloud is logged, the streamed data can quickly add up to gigabytes of data that needs to be stored in memory. In the recording shown in Figure 4.10, the 17 KITTI sequences require 408 megabytes of data. Depending on the hardware and especially the GPU, a large rerun recording can significantly reduce the performance of the viewer and stop the viewer from visualising the data in real time.

4.4 Training

The published model from the PRECOG trajectory prediction framework was trained on the ego vehicle only. This means that only a single trajectory could be predicted, which did not enable the estimation of potential collisions between multiple vehicles. The authors, however, released the CARLA dataset described in their work, which includes the past and future trajectories of the ego vehicle and four other vehicles. In total, the dataset consists of 60,701 training, 7,586 validation, and 7,567 test samples, most of which were recorded in Town01 of the CARLA simulator.

A drawback for reuse in combination with the object detection model SFA3D was that a differently configured LiDAR sensor was employed for the released dataset. The sensor recorded approximately 100 k points per second, resulting in a less densely populated BEV (bird's-eye view) histogram compared to the 120 k points per second contained in the KITTI dataset. Another crucial difference is the height at which the LiDAR was mounted. At 2.5 meters above the ground, the sensor was positioned higher than the roof of most cars and above the typical helmet height of cyclists, whereas the KITTI dataset used a mounting position closer to the top of the vehicle.

PRECOG does not provide a pretrained multi-agent model; for this reason, the trajectory prediction model was trained from scratch using data generated with CARLA. For each training cycle, the system generated 50 episodes with 2000 frames per episode for training.

After each cycle, the dataset was discarded and regenerated (this process will be described in more detail in the next section), ensuring diverse samples across episodes. The network was trained in an effectively endless-loop regime, where new synthetic data continuously replaced the previously used dataset. The training procedure is outlined as pseudo-code in Algorithm 4.2, which shows the order in which the dataset is regenerated and how the model is trained. The architecture was based on a SocialConvRNN configuration with convolutional, recurrent, and MLP components. The training uses a stochastic gradient descent (SGD) with Forward KL divergence as the optimisation objective.

Table 4.1 and Table 4.2 summarise the key parameters of the training setup and the major building blocks of the network, also illustrated in Figure 4.11.

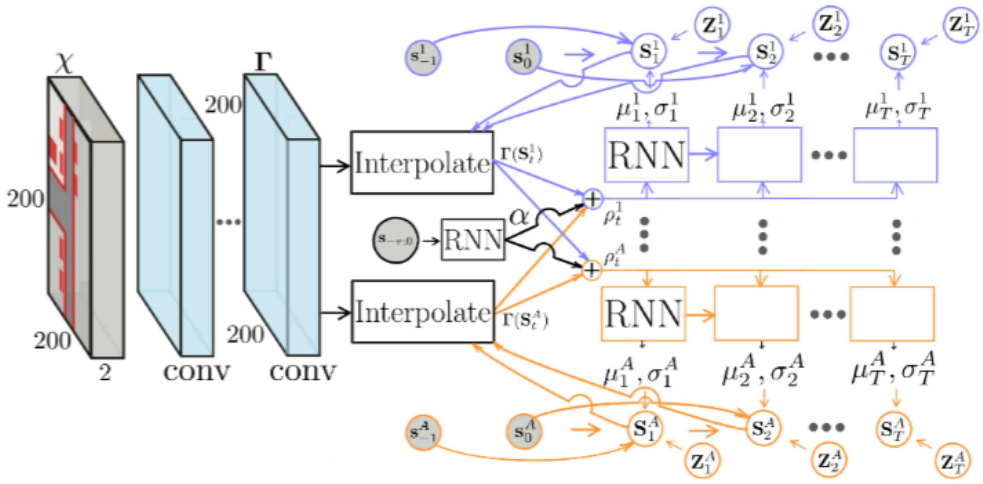


Figure 4.11: PRECOG model architecture, BEV image of the scene is processed by a CNN-backbone, while the past trajectories are fed directly into the RNN.

Training proceeded for a couple of days on a university server with an Intel Core i7-3770K and a NVIDIA GeForce RTX 2080Ti for 10 million steps before being stopped at epoch 116 as training did not show improvements for a period of time. The latest best model resulted in the following evaluation results: log-likelihood values $H(p, q)$ of -600 to -700 and error estimate $\hat{e}$ ranging from 0.3 to 1.1. Evaluation runs were executed every 5000 steps, since they were computationally expensive, often taking over 10 minutes per validation cycle. Intermediate outputs from the validation cycles can be seen in Figure 4.12.

The loss term provided $H(p, q)$ is the cross-entropy between the ground-truth trajectory distribution p and the predicted distribution q by the model. That is, it approximates how much likelihood the model assigns to the actual observed paths. The more negative the value is, the more probability density the model allocates to the correct results, which is a metric for improved training.

Table 4.1: Model architecture and optimiser configuration for PRECOG.

Component	**Configuration**
CNN backbone	4 layers, 32 filters, kernel size 3
RNN (past)	GRU, 128 units, pre-conv horizon = 3
RNN (future)	GRU, 200 units
MLP layers	2 pre-RNN + 2 post-RNN layers, 200 units
Social features	Map-based features enabled, MLP = 200 units
Light features	Representation size = 5, integrated post-RNN
Trajectory samples	$K = 12$, perturbed samples $K_{\text{perturb}} = 12$
Optimizer	SGD learning rate = 0.0001, epochs = 10,000 (but stopped at 116 epochs)
Objective	Forward KL divergence
Batch size	$B = 1$
Seed	42

Table 4.2: Dataset generation settings for PRECOG training.

Parameter	**Value**
Episodes per split	50 (train, validation, test)
Frames per episode	2000
Past horizon T_{past}	10 timesteps
Prediction horizon T	20 timesteps
Max agents per scene	6 (ego vehicle + 5 other road users)
Feature resolution	2.0 pixels/meter
Data format	Serialized JSON, regenerated each cycle

The best-of-K error prediction is described by the error term $\hat{e}$. Since the model makes more than one likely future path prediction for every scene, $\hat{e}$ indicates the ground-truth to best predicted among sampled trajectories. This tests not just the model on a single prediction, but on whether it has at least one of its plausible forecasts close to the truth. Lower values for $\hat{e}$ therefore represent better predictions, while higher values indicate that even the best predicted trajectory is quite far from the ground truth.

Algorithm 4.2: Pseudo-code of training procedure for the trajectory prediction model

```
Input: epochs (Number of epochs)
Output: model (Trained model for trajectory prediction)
for epoch ← 0 to epochs do
    while True do
        minibatch ← get_minibatch(epoch)
        Perform gradient update using minibatch
        if plot condition met then
            Plot current training status
        end
        if evaluation condition met then
            Evaluate on validation set
            if validation score improved then
                Save model checkpoint
            end
        end
    end
end
Function get_minibatch(epoch)
    if epoch mod 25 = 0 and no data left in current dataset then
        generate_new_data()
    end
    Fetch and return minibatch data from storage
Function generate_new_data()
    Discard the previous dataset Generate a new dataset using the CARLA
     simulator (invoked as a subprocess)
    Shuffle and store newly generated data
```

4.5 Dataset generation

As already discussed, for training the trajectory prediction model, we decided to generate a dataset to train the model instead of using the ground truth from the KITTI dataset, as the amount of annotated data is very small for the KITTI dataset, which likely would cause overfitting. Another benefit of generating a dataset is that the model can later be retrained with different sensor settings. For example, the location of the LIDAR (how high up it is mounted on the bicycle) or the field of view of the LIDAR can be adjusted easily. In a first attempt, as already mentioned previously, a hand-made trajectory simulation was tested, which simulated only the paths of vehicles but not any information about the surroundings. Although this simulation was very fast to generate paths, it obviously lacked the capabilities to generate information that also encodes the environment, and it also lacked the capability to model interactions between road users and did not include

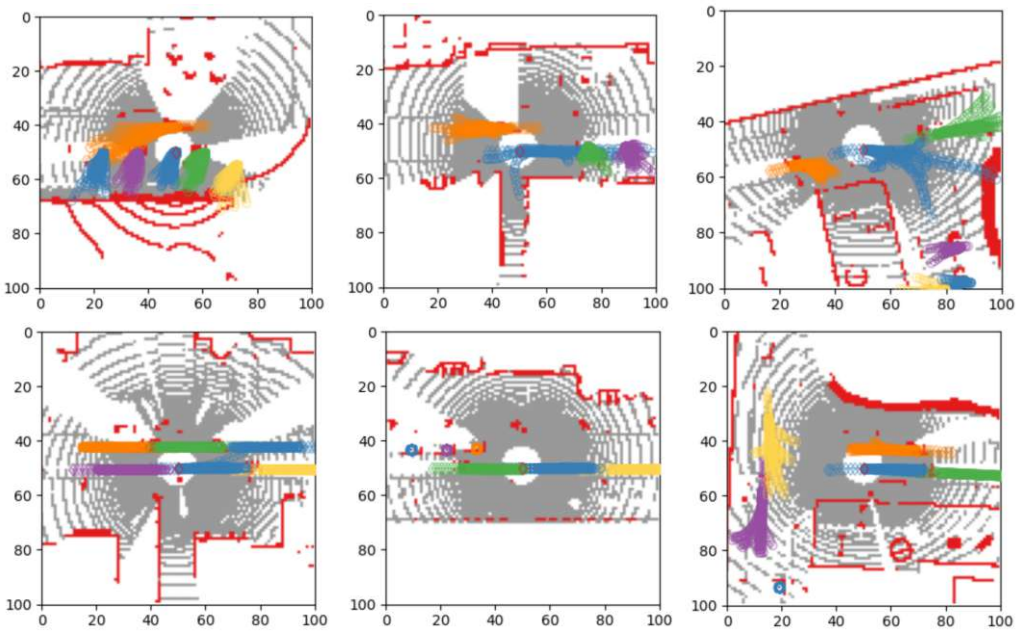


Figure 4.12: Intermediate training snapshots showing the convergence of the predictions. The LIDAR data is separated into obstacle (red) and drivable areas (grey). The top row shows outputs from the first 10 epochs of training, while the bottom row shows outputs from epochs > 90. The predictions in the first 10 epochs are still noisy, colliding with obstacles and not following the road layout, while the outputs from later epochs converge to a single trajectory for cases where no intersection is nearby, while diverging into the different lane directions for intersections.

any pedestrians. For this reason, we switched to the Unity-based open source driving simulator, CARLA. This driving simulator is based on a server-client architecture. The server performs the physics and traffic simulation, the rendering of the scene, and the generation of the sensory output information (most importantly, the simulation of LIDAR data). The client, which is controlled by a user-side script, sends commands to the server. For example, which map to load, how many and which road users to spawn, which sensor to mount on the ego vehicle, properties about the traffic simulation and road-user behaviour. Also, a major benefit of using a simulator over training with an annotated dataset such as KITTI is that the simulation can contain data samples with collision or other infrequently observable traffic situations. A downside of the simulation is that it represents idealised conditions, which are not present in the real data. For example, the simulated data are smoother, the recorded LIDAR point cloud is more regular and not affected by weather conditions.

In the following, the setup in CARLA is described, which generates the data samples required to train the precog trajectory prediction model:

First, the CARLA server is launched, depending on configurations, either in head-

Figure 4.13: Town01 map from CARLA driving simulator representing a suburban area

less/offscreen mode or with the viewport enabled. Then the map is selected, in this example, Town01 (see Figure 4.13) or Town02 (see Figure 4.14). In total CARLA features 12 built-in maps, ranging from urban area maps to maps with mostly highways, rural areas with narrow roads or very large maps. For the generation of the dataset, only the first two maps were used, as they contain many intersections and have smaller memory footprints than other maps. Then the ego vehicle is spawned, which is, in our case, a bicycle with a head-mounted LIDAR sensor and a GPS sensor. The same LIDAR sensor properties as the sensor used in the car that recorded the KITTI data set, namely a Velodyne HDL-64E sensor. The simulated sensor records data with 10 Hz, generating 130 k points per frame (see [67] with a vertical field of view of -24.8° to +2°. The horizontal field of view is 360°. In other words, the sensor is sensing down on the street and straight ahead, but not up any walls/buildings. So, naturally, bridges and overhangs are not visible in the sensor output.

Then the simulation is started, using the built-in autopilot from CARLA, and the traffic manager, which controls the other vehicles and agents. The client is collecting sensor data for every frame and storing the GPS location of the ego vehicle, the relative positions of nearby road users using simply a threshold on the Euclidean distance between them and the ego vehicle. This means that information about nearby road users is collected, even when the line of sight is blocked. As soon as more frames than the desired prediction length have been collected, data samples for the training can be saved to the file system as JSON files. But instead of creating a new data sample for every upcoming frame, only every n-th frame is saved, to get more variation in the collected data faster. PRECOG used a data sample every 10th frame; in this implementation, the frequency was reduced

Figure 4.14: Town02 map from CARLA driving simulator representing a more urban area with a commercial area

to every 5th frame to speed up the dataset generation. Although the CARLA simulation is, in general, reasonably fast, the computation of the LIDAR sensor occupies a huge chunk of the time spent during dataset generation. As an example, generating the training data for a single episode running for 2000 frames takes approximately 21 minutes, resulting in 400 training samples (as every 5th frame is actually saved). This means that in total it takes around 17 hours to generate one set of training data that is used for 25 epochs during training. And even though the LIDAR frame is only required every time a data sample is generated (in our case, every 5th frame), unfortunately, the used CARLA version 0.8.4 does not support controlling the output frequency of the sensor. This means that the LIDAR point cloud is still computed for every frame, but discarded most of the time. Since CARLA is open-source, this functionality could be added to a fork of the repository, identifying the corresponding code and creating a custom build that allows skipping the computation of unnecessary LIDAR frames.

Now, by default, the traffic manager [55] in CARLA prohibits any collisions and dangerous manoeuvres from occurring, but by changing the settings of the traffic manager, more interesting data samples can be generated. For generating the dataset, the following adjustments to the manager were made: At the start of each episode, for each vehicle, randomise the following properties:

- **Ignore lights percentage**: Controls the likelihood that a vehicle will ignore a red traffic light. Range used: [0 - 10]%

- **Vehicle percentage speed difference** Relative difference to the current speed limit, a value of 20 corresponds to the car driving 20% faster than the speed limit. Range used: [-25 - 25]%
- **Distance to leading vehicle** Minimum distance to any vehicle in front of this vehicle. Range used: [0-10]m
- **Ignore signs percentage** Same as ignore lights percentage, but for stop signs. Range used: [0 - 10]%
- **Ignore vehicles percentage** During the collision detection stage of the traffic manager, this property controls the likelihood that another vehicle is ignored (basically like the driver of one vehicle did not notice another vehicle and might collide with it). Range used: [0-10]%

The CARLA traffic manager offers many more ways to manipulate the behaviour of vehicles in the simulation, which can be exploited to generate even more realistic data.

This concludes the pipeline implementation section, leading to the results showing visuals of the pipeline outputs, examples of some interesting traffic situations, as well as quantitative results about the pipeline performance.

CHAPTER 5

Results

This chapter presents a comprehensive evaluation of the proposed collision detection pipeline on two datasets. The analysis focuses on assessing the effectiveness of the trajectory prediction model, while also evaluating the object detection and tracking, as well as the contribution of individual pipeline steps through an ablation study. But before diving into the evaluation results, the evaluation setup and datasets used were as follows:

- **KITTI Sequences with Ground Truth:** Since not all KITTI sequences are annotated, approximately 20 sequences were selected, each containing between 100 and 500 frames.
- **CARLA-Generated Sequences:** A set of 10 sequences was used, each consisting of 2000 frames.

For each sequence and dataset, the evaluation focused on two key metrics. First, the prediction accuracy was measured in terms of average and median error over different time horizons, ranging from 0.1 to 2 seconds. Second, the tracking consistency was analysed by computing the percentage of frames in which each nearby vehicle remained tracked. For instance, a value of 0.9 indicated that a particular vehicle was tracked for 90% of the frames in which it was in the cyclist's vicinity.

Furthermore, an ablation study was conducted to assess the impact of different components in the pipeline. The system was evaluated under the following conditions:

1. **Without Object Detection:** The object detection step (SFA3D) was removed and replaced with ground truth data. The evaluation was then performed using only the tracker and the trajectory prediction module.

2. **Without Object Detection and Tracking:** Both the object detection and tracking components were removed. Ground truth data was used directly, and only the trajectory prediction module was evaluated.

This structured evaluation provided information on the effectiveness of the trajectory prediction pipeline and the individual contributions of object detection and tracking. The results of the analysis are presented in the following section.

The performance analysis of the CARLA-generated and KITTI dataset shows prominent trends regarding tracking performance, prediction accuracy of the trajectory, and object tracking and detection contributions.

5.1 Tracking performance

On the CARLA dataset (see Figure 5.3), the overall pipeline exhibits a mid-range tracking consistency of 30%–50%, indicating that tracked vehicles are appropriately maintained across roughly one-third to half of the frames in which they appear. On the contrary, if the object detection stage is eliminated and ground truth detections are used, tracking consistency reaches an optimal 100% across all tested sequences. This confirms that detection stage failures are the prime cause for limiting tracking stability in CARLA.

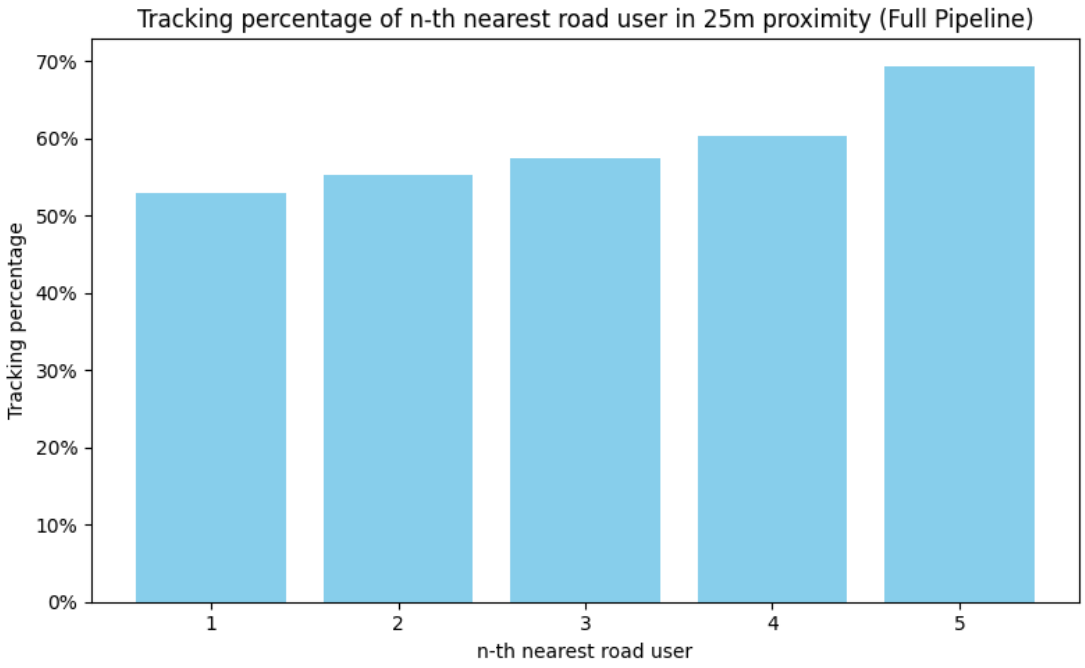
The same trend is seen on KITTI (see Figure 5.1). Using the entire pipeline, consistency of tracking is 50%–70%, demonstrating better robustness than CARLA, but this is explainable with the fact that the CARLA sequences contain annotations for obstructed vehicles while KITTI does not. Once more, substituting detections with ground truth gives near-perfect tracking performance of 98%–99%. This consistency on both datasets reiterates that the tracking module itself works when given clean inputs, and that performance is largely bottlenecked by imperfect detections.

5.2 Trajectory prediction accuracy

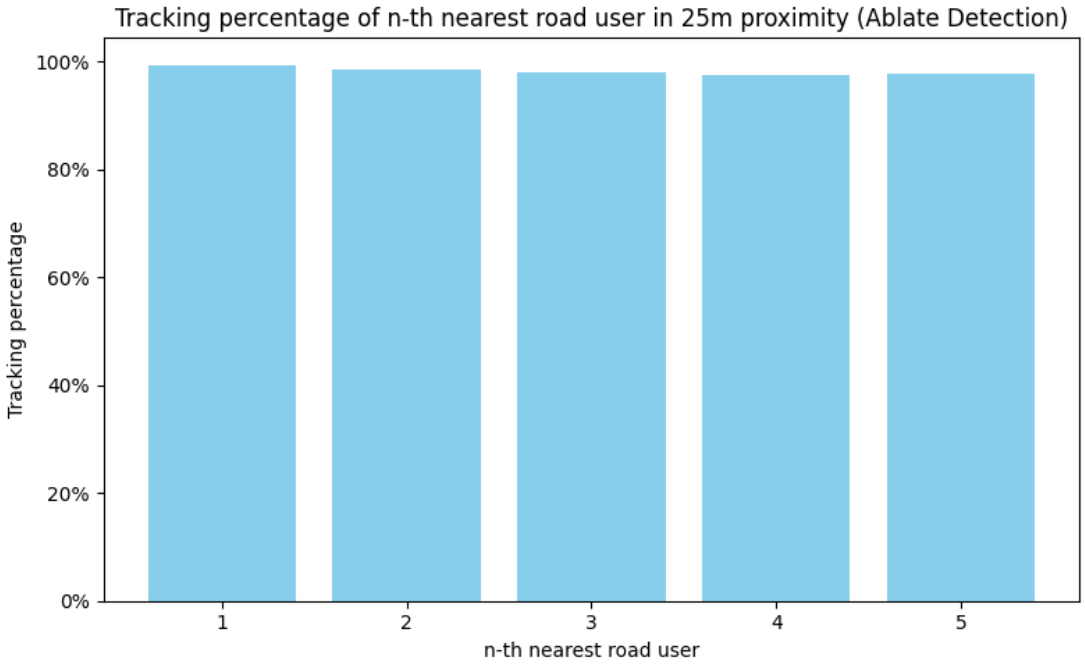
Error in trajectory prediction grows steadily with prediction time horizon in both datasets, as would be expected (see Figure 5.4 as well as Figure 5.2). Average prediction error on CARLA starts at 0.2 m to 1 m (for predicting 0.1 s ahead) and increases nearly linearly to around 4 m (at prediction horizon 2.0 s). While no clear difference between the mean and median values is present, the prediction error on the full pipeline is significantly higher than when ablating the detection and or tracking stage. Meanwhile, there is almost no difference in prediction error between solely ablating the detection stage and also ablating the tracking stage, indicating that while the detection stage induces a lower accuracy in prediction, the tracking stage does not cause any problems.

On KITTI, errors are, as expected, higher and noisier. Mean error begins at 0.6 m and rises to almost 6 m by 2.0 s. But contrary to the evaluation result for the trajectory prediction accuracy on the CARLA dataset, there is no significant difference between the

three modes. In general, the graphs indicate that the KITTI set is significantly harder for prediction than CARLA, likely due to noisier sensor data, more diverse environments, and less structured traffic conditions. The small mean-median error gap in KITTI also means that outliers are common, not due to individual extreme cases.

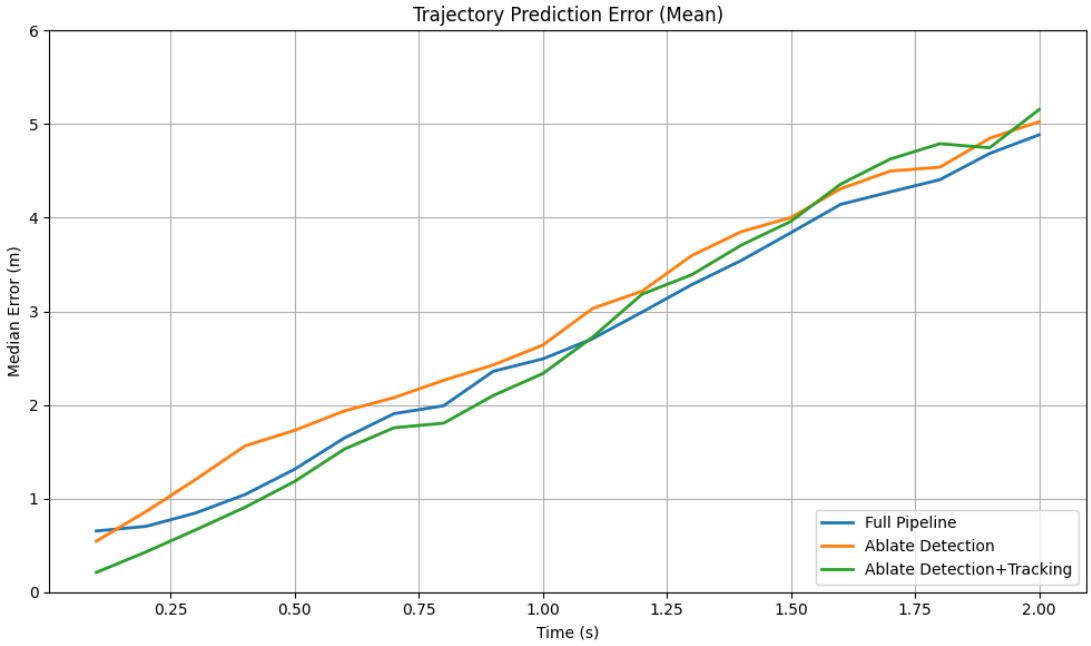


(a) Tracking performance showing an increasing tracking performance with increased distance.

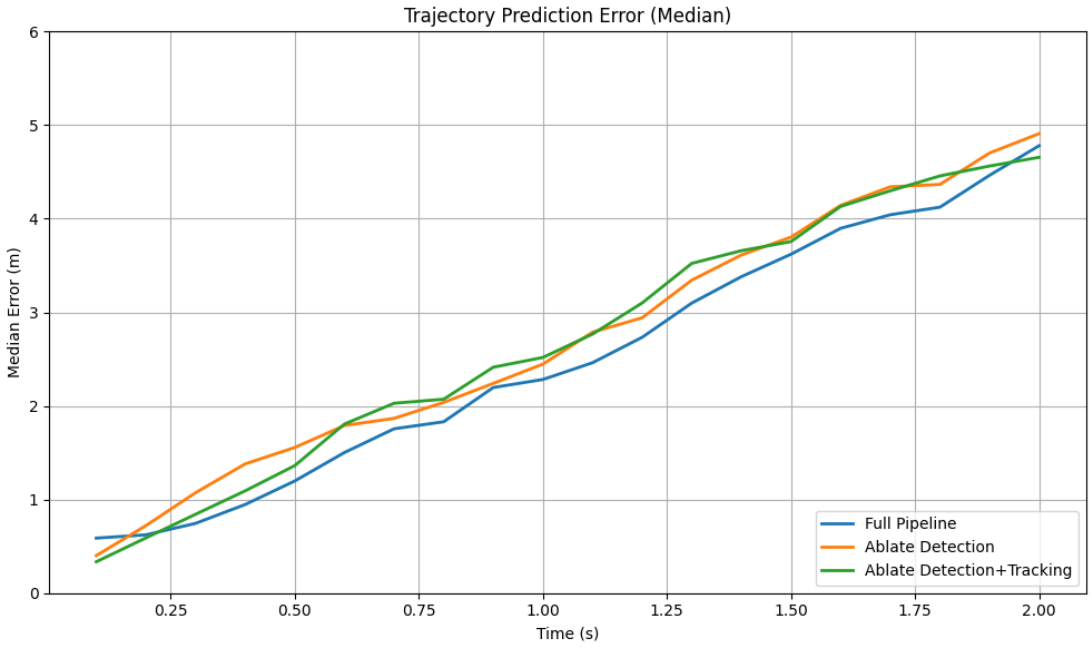


(b) Tracking performance when the detection stage is ablated, indicating that the tracker output is of high quality.

Figure 5.1: Evaluation results of KITTI dataset.

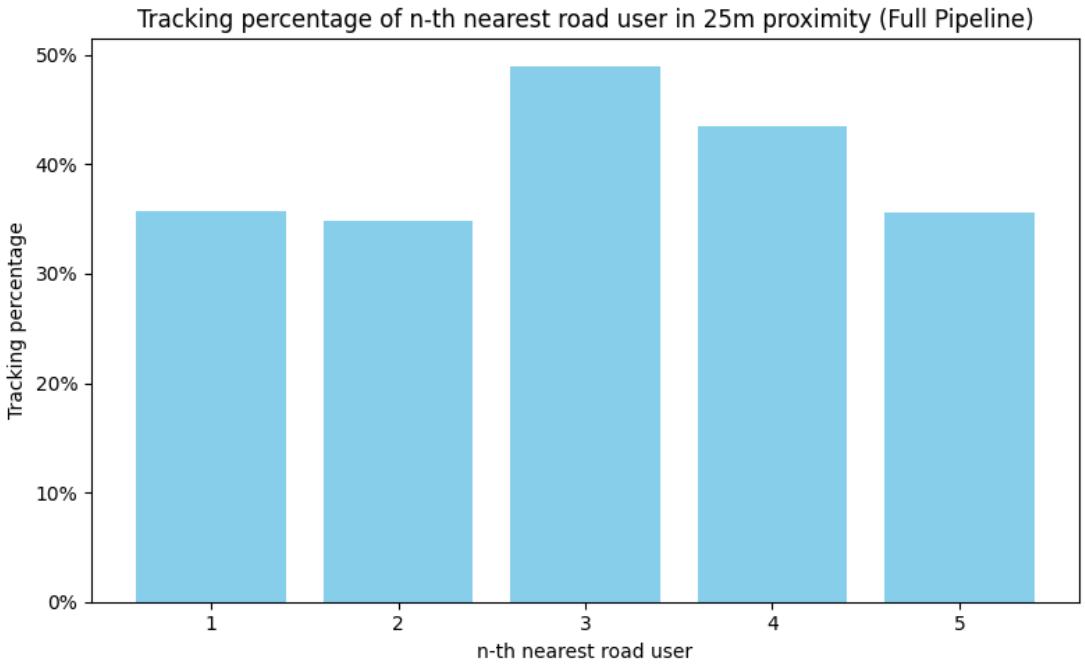


(a) Mean prediction error, showing a clear increase in prediction uncertainty the further into the future the prediction is made.

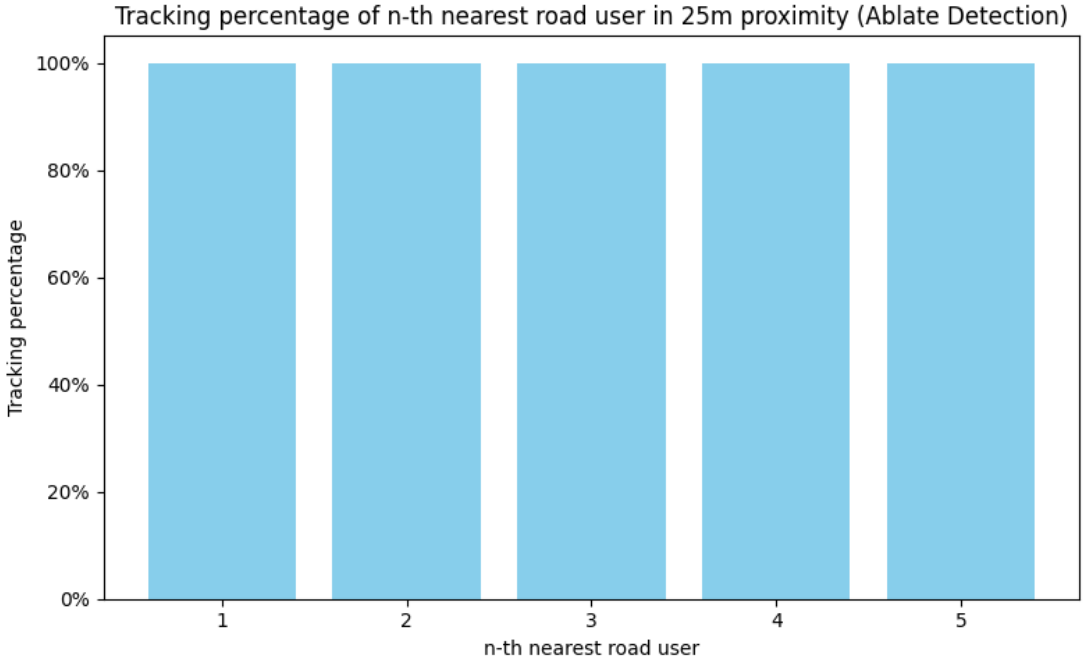


(b) Median prediction error, showing no significant difference between predictions made from road users tracked by the pipeline or ablation of the detection or detection+tracking stages.

Figure 5.2: Evaluation results of KITTI dataset.

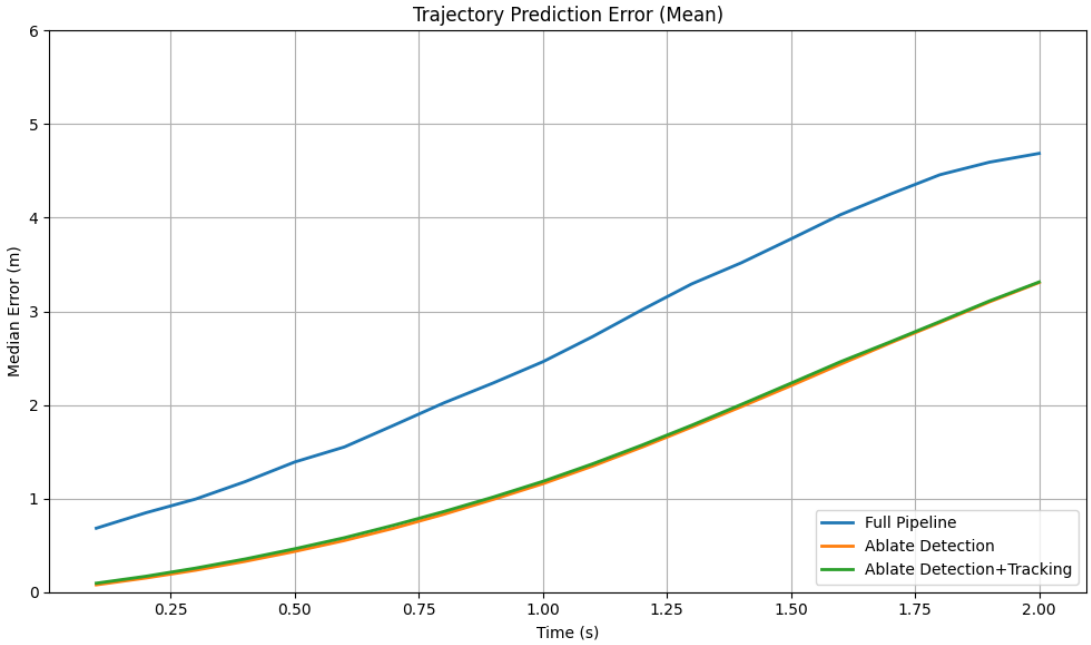


(a) Tracking performance, similar to the performance trend as on the KITTI dataset, but with decreased performance due to the dataset containing annotations for occluded objects.

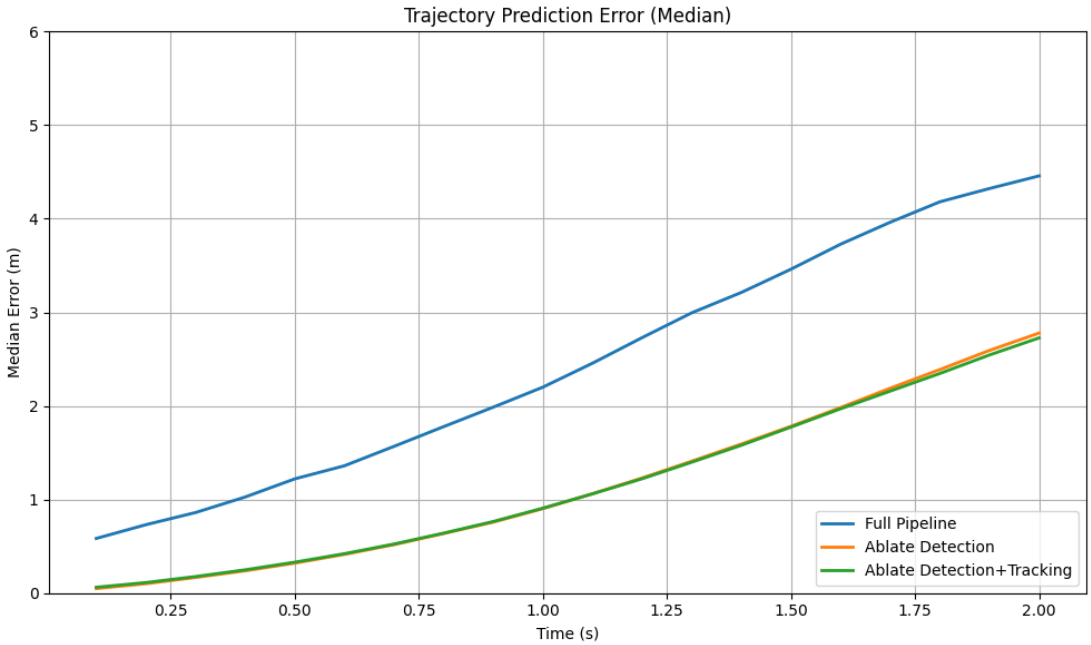


(b) Tracking performance when the detection stage is ablated, indicating that the tracker output is of high quality.

Figure 5.3: Evaluation results of CARLA dataset.



(a) Trajectory prediction mean, showing a clear difference between predictions made on trajectories traced from detected objects compared to when the object detection stage is ablated.



(b) Trajectory prediction median, showing the same trend as for the mean.

Figure 5.4: Evaluation results of CARLA dataset.

5.3 Effect of detection and tracking ablation

Ablation studies uncover the relative contribution of pipeline modules. Removing the detection module eliminates the primary source of tracking inconsistency. When detection and tracking are skipped altogether (using ground truth trajectories as input directly), the results filter out the performance losses of the trajectory prediction module. As the prediction error under ablation is comparable in trend but systematically smaller, it is clear that upstream detection noise carries over to the prediction phase and scales the overall error. But the evaluation showed that the tracking stage is much more reliable than the detection stage, as the performance only significantly increased when ablating the detection stage.

5.4 Visual demonstrations

In addition to quantitative results, the following will show and describe a few snapshots of the pipeline's outputs and how to read the richly annotated lidar point clouds visualised in the rerun-viewer. Figure 5.5 shows an intermediate frame of the KITTI dataset with several tracked vehicles. The predicted paths of the vehicles show the expected movement along the road, while, notably for the vehicle with ID 2, there are two predictions that the vehicle will turn left along with the ego vehicle, potentially crossing paths. The ego vehicle's future trajectory (ground truth) is outlined in red. No future predictions for the ego vehicle exist yet, as it is stationary. In Figure 5.6, an example of a scene can be seen where the ego vehicle is not stationary, for which reason predictions about its future path are made and visualised in blue. The image also shows the difference between the most likely path prediction and the actual ground truth (the hatched lines connecting the path prediction and the yellow line representing the ground truth). Contrary, Figure 5.7 demonstrates a case of a potential collision, where the predicted path of the ego vehicle intersects the predicted path of a pedestrian, although, as also visualised, this is a false positive, as the ego vehicle actually turns right while the pedestrian remains stationary.

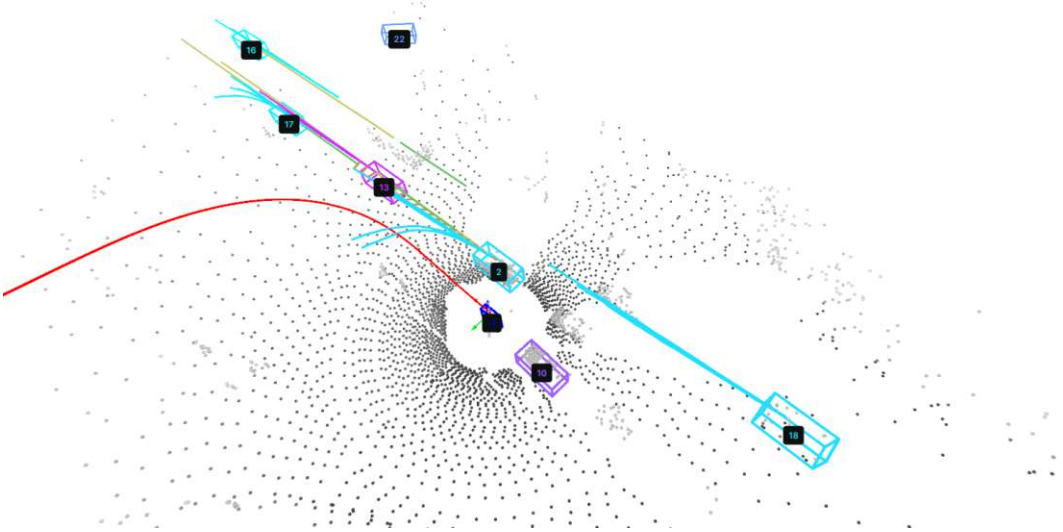


Figure 5.5: Future path predictions and ground truth past and future paths visualised. Note that all trajectory samples are visualised, not only the prediction with the highest predicted probability. Ego vehicle future path ground truth (red), other vehicles past path ground truth (green), other vehicles future path ground truth (yellow), other vehicles future predictions (same colour as the detected vehicle bounding box), the ego vehicle has no future prediction and past path as it is currently stationary.

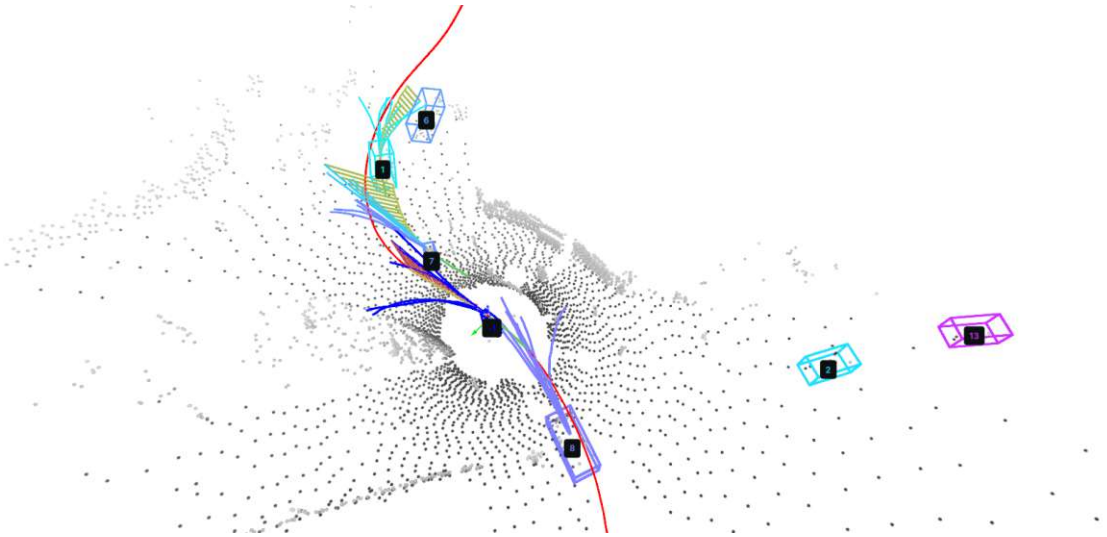


Figure 5.6: Similar to Figure 5.5, but with the ego vehicle in motion and its future trajectory samples visualised (dark blue).

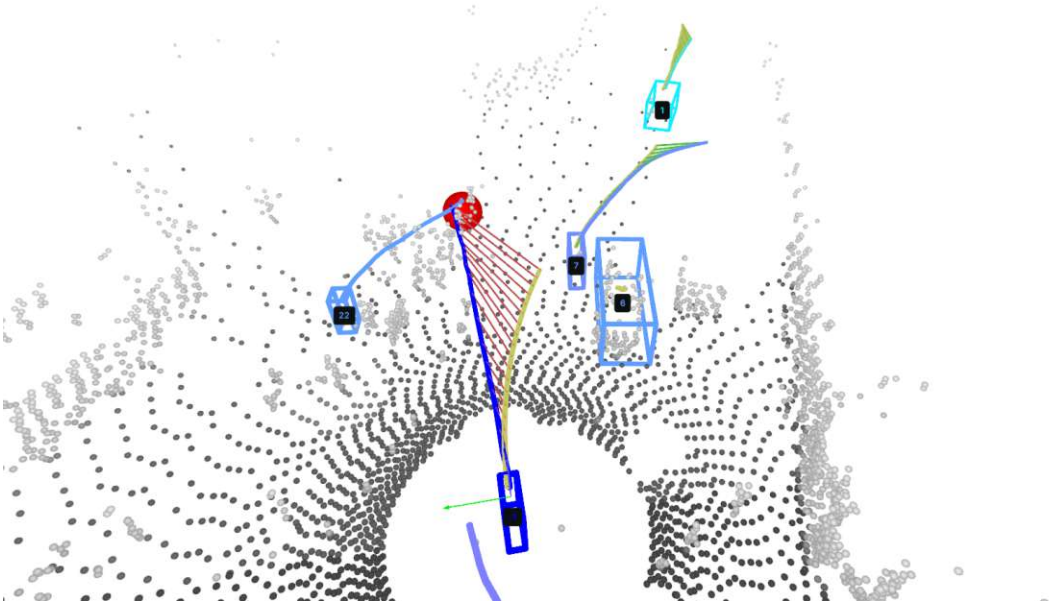


Figure 5.7: Visualisation of a potential collision with a pedestrian in 2 seconds (large red circle). The ego vehicle (dark blue with label -1) and the detected pedestrian with label 22. The true future path of the ego vehicle (yellow), the difference between the true path and the predicted path (red).

5.5 Runtime

Step	Calls	Total Time (s)	Mean per call (s)	FPS
Initialisation/Setup				
Load object detection model	1	0.43	0.43	-
Load trajectory prediction model	1	10.6	10.6	-
Initialize object tracker	17	0.06	0.004	-
Total	1	11.8	11.8	-
Main cycle				
Object detection (inference)	3703	238	0.064	16
Trajectory prediction (inference)	3703	368	0.099	10
Multi object tracker	3703	40	0.011	91
Total	-	646	0.174	6

Table 5.1: Profiling summary of main pipeline steps.

The profiling in table 5.1 summarises the pipeline components' run-times when tested against 17 annotated KITTI sequences (total of 3703 frames).

During the pipeline setup, the object detector, trajectory predictor, and multi-object tracker are initialised. Both the object detector and the trajectory predictor are deep neural network models and therefore require significant one-time initialisation (0.4 s and 10.6 s respectively). Since the models are stateless, they are initialised just once. In contrast, the tracker maintains state across frames of a sequence and is therefore reset 17 times, once per sequence, resulting in many initiations (17 total, but very rapid at 3.6 ms per call). Initialisation time is dominated by loading the neural network model, taking a total of about 23 s.

The main loop of the pipeline is tasked with processing all frames in the sequences. The pipeline processes at 175 ms per frame, slower than the 100 ms per frame available using the 10 Hz LIDAR sampling rate. This shows that the pipeline is close to, but cannot process real-time data under the tested hardware setup (Intel(R) Core(TM) i5-9600KF CPU @ 3.70 GHz, 32 GB RAM, NVIDIA GeForce RTX 4060 GPU).

The largest runtime bottleneck is the trajectory prediction inference (99 ms/call, total 368 s), the most computationally expensive operation. Object detection inference is also significant (64 ms/call, total 238 s). There is further considerable overhead due to redundant preprocessing: converting LIDAR point clouds to bird's-eye view (BEV) images. Both the predictor and detector require BEV inputs, but of different types, leading to repeated conversion and extra data format conversion between modules.

The profiling of the object detection process is broken down into sub-steps:

- Filtering LIDAR: The point cloud is reduced in size by a bounding box filter (side length 50 m). This is light at 1.4 ms per call.
- LIDAR to BEV conversion: Returns BEV images from filtered point clouds, 14.3 ms per call.
- Detection model inference: Performs the actual detection, 16 ms per call.

These sub-steps are invoked 7406 times in total instead of 3703 times because the object detector splits each LIDAR frame into a back and front view and processes them sequentially, combining the detections afterwards.

The multi-object tracker is capable of processing 100 fps, with 10 ms per call (total 38 s for all frames).

In general, pipeline run time is dominated by inference of the neural networks (specifically, trajectory prediction), with further overhead added by preprocessing stages such as LIDAR-to-BEV conversion and inter-module data conversion. Although every non-ML step is very fast, their repeated use per two point clouds per frame totals up to a large fraction of the runtime. Profiling shows that on the hardware available, the system runs at 5.7 Hz rather than the target 10 Hz.

CHAPTER 6

Discussion

The presented work demonstrates the feasibility of operating a complete collision detection pipeline for vulnerable road users from LiDAR data. The integration of SFA3D object detection, a 3D multi-object tracker for ensuring temporal consistency, and PRECOG for trajectory prediction presents a functional proof-of-concept capable of running in real-time on a special computing platform. However, the current implementation remains in a prototypical phase and is not fully ready for use on an actual bicycle. While the simulated environment provided with CARLA data facilitated rapid development and testing, transitioning to real-world implementation comes with some challenges. The system remains without integration with sensors in the real world and a communication setup between a sensor-equipped bicycle and a data-processing device. Testing in real conditions would constitute synchronous operation, data transfer, and calibration among other sensors—factors not sufficiently represented in the simulated environment.

All components of the pipeline—detection, tracking, and prediction—were shown to function independently. Nevertheless, error propagation among these components is still a significant shortcoming. Tracking mismatches, for example, can input faulty object states into the prediction module and cause false collision warnings or at least infeasible future paths. While detection and tracking performance were reasonable, the trajectory prediction module (PRECOG) was unable to provide the anticipated accuracy. Thus, although the system has potential, the predictive reliability is insufficient for real-time collision warnings.

Another issue is the repeated preprocessing of LiDAR data and variable input/output formats across algorithms, resulting in computational inefficiencies. Furthermore, the current pipeline is sensor-specific, and generalisation proves to be difficult. In case LiDAR characteristics change, for example, point density, range, or field of view, the deep learning models would have to be retrained or significantly modified, limiting scalability. Having such a system on a bike in itself also places hardware and computational challenges. Processing onboard would likely be impractical. Hiding computation away into some

external device or cloud system might help, but it introduces additional issues such as communication latency, cost of communicating data, and usability in low-connectivity areas.

Figure 6.1: Redesign project of *Hütteldorfer Straße* in Vienna, realising a clear separation of motorised traffic and cyclists

From a broader perspective, there is an infrastructural vs. technological trade-off for cycling safety. Intelligent perception systems can offer a flexible and inexpensive addition, but infrastructure measures such as improved street design, protected cycling lanes, and physical separation from motorised traffic proved to perform better on collision reduction, albeit at a higher monetary cost. *Praterstraße*, *Alte Donau*, *Mariahilfer Straße*, and recently *Hütteldorfer Straße* (see Figure 6.1) redesign projects are Viennese examples of how infrastructural interventions, such as swapping the parking and cycling lanes, achieve long-term safety benefits compared to reactive technological approaches. But technology-driven safety devices can also make a difference. Existing market solutions, such as bicycle radar systems, brake lights, and turn signal indicators, prove that even simple sensor-based solutions are able to increase cyclist visibility and awareness. The proposed pipeline for collision detection can be a more advanced development of similar systems, potentially adding prediction capabilities when its efficacy and robustness are improved.

To summarise, this thesis illustrates a proof of concept for real-time LIDAR-based cyclist collision prediction and detection. It demonstrates the feasibility of applying cutting-edge deep learning models in an end-to-end processing pipeline but reveals inherent limitations in runtime, accuracy, and adaptability. Future studies would focus on improving data consistency, runtime performance, retraining with real-world data, and addressing the

communication challenges of such systems being integrated into bicycles. In totality, while intelligent sensing solutions may complement existing infrastructure, they will not eliminate the need for safer street design and dedicated cycling infrastructure as the most effective means of avoiding collisions.

CHAPTER 7

Limitations and future work

Despite demonstrating a functioning perception–prediction–planning pipeline based on SFA3D, the 3D Multi-Object Tracker, and PRECOG trajectory prediction, several limitations remain from which the following future work can be derived:

Trajectory prediction constraints: The PRECOG implementation currently available is restricted to a fixed number of predicted trajectories that must be specified during training. This constraint limits its flexibility in real-world scenarios where the number of possible nearby agents varies dynamically. In addition, PRECOG requires the last n-observed positions as input, which reduces its applicability when dealing with incomplete or noisy tracking histories. Improving the generalisation of trajectory prediction remains a central challenge, particularly given potential issues in the CARLA-generated training data (e.g., labelling noise, suboptimal training parameters, or input formatting mismatches).

Tracking and detection limitations: The performance of the 3D Multi-Object Tracker depends heavily on appropriately calibrated parameters such as Kalman filter parameters, detection score thresholds, and latency expectations. Suboptimal tuning diminishes tracking stability and following trajectory prediction. As a further challenge, the detector (SFA3D) and the predictor (PRECOG) were both trained for specific sensor modalities and mounting orientations. Deployments to alternative sensor configurations (e.g., altered LiDAR mounting locations or camera FOV) would require retraining or considerable adaptation. A more liberal multi-modal input pipeline (allowing both LiDAR and camera inputs, with fewer field-of-view limitations) would be more robust.

Data collection and simulation issues: Data generation within CARLA v0.8.4 is limited by its inability to subsample LiDAR frames (e.g., record every n-th frame). This required a hack of only using every 10th frame, which created a large bottleneck. Moreover, occluded objects within the scene were not pruned, leaving behind irrelevant or confounding training instances. Synchronising to more recent CARLA releases and

improved dataset preparation may help alleviate these issues. Beyond simulation, ground truth data in real-world environments is nonetheless restricted. Gathering datasets equipped with sensor-rich bicycles would provide valuable training and test data, and aid in bridging the domain gap between CARLA simulation and real-world deployment of such a system.

System integration issues: There was considerable engineering needed to cross Python version needs and library compatibility between SFA3D, the tracker, and PRECOG. This highlights the need for more standardised or containerised development platforms for reproducibility and scalability.

Pipeline design implications: The current architecture has a staged design: object detection⟶tracking⟶trajectory prediction⟶collision detection. While such a modular design enables interpretability and debugging, each intermediate step results in error propagation. Another way would be to learn an end-to-end model from raw LiDAR data to predict potential collisions without intermediate bounding boxes or tracks. Such a design might achieve higher performance and efficiency, though at the cost of reduced transparency regarding failure points.

Future Extensions: There are several ways in which the pipeline could be extended further:

- Improving the accuracy of trajectory prediction via parameter tuning, alternative architectures.
- Including richer output from collision detection than merely a warning (e.g., graded warnings such as automotive parking sensors, or active safety interventions such as automatic braking or evasive steering).
- Exploring vehicle-to-bicycle (V2X/V2B) communication protocols for cooperative intent sharing and collision avoidance.

By addressing these challenges, this research takes a small step toward safer and more sustainable bicycle mobility.

Overview of Generative AI Tools Used

GPT-3.5 and GPT-4o and the integrated version of Overleaf TextGPT were used to improve writing style and vocabulary. Further, GPT was used for providing LaTeX snippets for integrating figures, tables and circuittikz. TikzMaker was used to generate the pipeline flowchart Graphic 4.1

TextGPT suggestions were applied from the integrated suggestion toolbox within Overleaf.

DeepL was used to generate the basis of the German translations for the abstract and acknowledgements sections.

List of Figures

List of Tables

List of Algorithms

Glossary

BEV Bird's Eye View. A top-down 2D representation of the environment (e.g. LIDAR or camera input).. 8, 9, 14, 23, 26, 29, 38, 39, 43, 72

CARLA An open-source simulator for testing perception and safety pipelines.. 4, 18, 26, 27, 35, 41, 42, 46–49, 52, 53, 56, 57, 67, 68, 72

CNN Convolutional Neural Network. A deep learning model that extracts features from input data using convolution operations.. 8, 13, 14, 44, 71

ego vehicle The main vehicle equipped with sensors in a collision detection system. It is the point of reference for determining whether surrounding objects pose a potential collision risk.. 3, 8, 30, 32, 33, 38, 39, 41, 42, 44, 46, 47, 58, 71

Forward KL Forward Kullback–Leibler Divergence. A statistical measure of how one probability distribution diverges from another.. 43, 44

GRU Gated recurrent unit. A type of recurrent neural network designed for sequential data.. 44

KITTI A benchmark dataset containing real-world sensor recordings, including images, LIDAR scans, and GPS/IMU data.. 4, 7, 9, 10, 14, 18, 19, 27, 29, 33, 41, 42, 45–47, 51–56, 61, 71, 72

LIDAR Light Detection and Ranging. A sensing method that uses laser beams to measure distances to surrounding objects.. 1, 2, 4, 7–10, 14, 16–18, 20, 23–29, 35–37, 39, 64, 71, 72

residual connection A "skip connection" used in deep neural networks, where the input is added directly to the output of a layer.. 12

RNN Recurrent neural network. A model for sequential data processing using residual connections to memorise previous input.. 9, 16, 17, 44

SGD Stochastic gradient descent. An optimisation algorithm for training neural networks.. 43, 44

Acronyms

DOF Degrees of freedom. 18, 20

GPS Global positioning system. xi, 18, 19, 71

IMU Inertial measurement unit. xi, 18

ML Machine learning. 62

RGB Red Green Blue. 2, 3, 9, 18, 24, 28, 71

Bibliography

[1] S. Austria, "Weniger verkehrstote 2021, aber höchstwerte bei verletzten radfahrerinnen und radfahrern." https://www.statistik.at/fileadmin/announcement/2022/05/20220428Unfaelle2021.pdf, 2022. (Accessed on 06/12/2024).

[2] "OpenBikeSensor." https://www.openbikesensor.org, May 2025. [Online; accessed 21. Sep. 2025].

[3] "Separated Bike Lanes—Making Roads Safer for Bicyclists | Innovator | 2024 | March / April." https://www.fhwa.dot.gov/innovation/innovator/issue101/page_02.html, Mar. 2024. [Online; accessed 16. Oct. 2025].

[4] B. Nassi, Y. Mirsky, D. Nassi, R. Ben-Netanel, O. Drokin, and Y. Elovici, "Phantom of the adas: Securing advanced driver-assistance systems from split-second phantom attacks," in *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, pp. 293–308, 2020.

[5] C. Mike Horton, "IMU Technology Forms the Brains of the Autonomous Vehicle," *5G Technology World*, Feb. 2019.

[6] "Rerun." https://rerun.io, July 2025. [Online; accessed 8. Jul. 2025].

[7] G. Elghazaly, R. Frank, S. Harvey, and S. Safko, "High-definition maps: Comprehensive survey, challenges, and future perspectives," *IEEE Open Journal of Intelligent Transportation Systems*, vol. 4, pp. 527–550, 2023.

[8] D. A. Pomerleau, "Alvinn: An autonomous land vehicle in a neural network," *Advances in neural information processing systems*, vol. 1, 1988.

[9] W. Luo, B. Yang, and R. Urtasun, "Fast and furious: Real time end-to-end 3d detection, tracking and motion forecasting with a single convolutional net," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pp. 3569–3577, 2018.

[10] R. Chandra, U. Bhattacharya, C. Roncal, A. Bera, and D. Manocha, "Robusttp: End-to-end trajectory prediction for heterogeneous road-agents in dense traffic with

noisy sensor inputs," in *Proceedings of the 3rd ACM Computer Science in Cars Symposium*, pp. 1–9, 2019.

[11] M. Shah, Z. Huang, A. Laddha, M. Langford, B. Barber, S. Zhang, C. Vallespi-Gonzalez, and R. Urtasun, "Liranet: End-to-end trajectory prediction using spatio-temporal radar fusion," *arXiv preprint arXiv:2010.00731*, 2020.

[12] M. Liang, B. Yang, W. Zeng, Y. Chen, R. Hu, S. Casas, and R. Urtasun, "Pnpnet: End-to-end perception and prediction with tracking in the loop," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 11553–11562, 2020.

[13] A. Laddha, S. Gautam, S. Palombo, S. Pandey, and C. Vallespi-Gonzalez, "Mvfusenet: Improving end-to-end object detection and motion forecasting through multi-view fusion of lidar data," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2865–2874, 2021.

[14] H. Zhao, J. Gao, T. Lan, C. Sun, B. Sapp, B. Varadarajan, Y. Shen, Y. Shen, Y. Chai, C. Schmid, *et al.*, "Tnt: Target-driven trajectory prediction," in *Conference on robot learning*, pp. 895–904, PMLR, 2021.

[15] J. Gu, C. Sun, and H. Zhao, "Densetnt: End-to-end trajectory prediction from dense goal sets," in *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 15303–15312, 2021.

[16] J. Gu, C. Hu, T. Zhang, X. Chen, Y. Wang, Y. Wang, and H. Zhao, "Vip3d: End-to-end visual trajectory prediction via 3d agent queries," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 5496–5506, 2023.

[17] Z. Xiong, S. Liu, N. Helgesen, J. Johnander, and P.-E. Forssen, "Catplan: Loss-based collision prediction in end-to-end autonomous driving," *arXiv preprint arXiv:2503.07425*, 2025.

[18] A. Alahi, K. Goel, V. Ramanathan, A. Robicquet, L. Fei-Fei, and S. Savarese, "Social lstm: Human trajectory prediction in crowded spaces," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 961–971, 2016.

[19] A. Gupta, J. Johnson, L. Fei-Fei, S. Savarese, and A. Alahi, "Social gan: Socially acceptable trajectories with generative adversarial networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2255–2264, 2018.

[20] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, "Vision meets robotics: The kitti dataset," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1231–1237, 2013.

[21] P. Sun, H. Kretzschmar, X. Dotiwalla, A. Chouard, V. Patnaik, P. Tsui, J. Guo, Y. Zhou, Y. Chai, B. Caine, *et al.*, "Scalability in perception for autonomous driving: Waymo open dataset," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 2446–2454, 2020.

[22] H. Caesar, V. Bankiti, A. H. Lang, S. Vora, V. E. Liong, Q. Xu, A. Krishnan, Y. Pan, G. Baldan, and O. Beijbom, "nuscenes: A multimodal dataset for autonomous driving," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 11621–11631, 2020.

[23] M.-F. Chang, J. Lambert, P. Sangkloy, J. Singh, S. Bak, A. Hartnett, D. Wang, P. Carr, S. Lucey, D. Ramanan, *et al.*, "Argoverse: 3d tracking and forecasting with rich maps," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 8748–8757, 2019.

[24] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788, 2016.

[25] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *European conference on computer vision*, pp. 21–37, Springer, 2016.

[26] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," in *Proceedings of the IEEE international conference on computer vision*, pp. 2980–2988, 2017.

[27] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *European conference on computer vision*, pp. 740–755, Springer, 2014.

[28] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 580–587, 2014.

[29] Y. Zhou and O. Tuzel, "Voxelnet: End-to-end learning for point cloud based 3d object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4490–4499, 2018.

[30] A. H. Lang, S. Vora, H. Caesar, L. Zhou, J. Yang, and O. Beijbom, "Pointpillars: Fast encoders for object detection from point clouds," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 12697–12705, 2019.

[31] X. Chen, H. Ma, J. Wan, B. Li, and T. Xia, "Multi-view 3d object detection network for autonomous driving," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pp. 1907–1915, 2017.

[32] J. Ku, M. Mozifian, J. Lee, A. Harakeh, and S. L. Waslander, "Joint 3d proposal generation and object detection from view aggregation," in *2018 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pp. 1–8, IEEE, 2018.

[33] A. Bewley, Z. Ge, L. Ott, F. Ramos, and B. Upcroft, "Simple online and realtime tracking," in *2016 IEEE international conference on image processing (ICIP)*, pp. 3464–3468, Ieee, 2016.

[34] X. Weng, J. Wang, D. Held, and K. Kitani, "3d multi-object tracking: A baseline and new evaluation metrics," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 10359–10366, IEEE, 2020.

[35] N. Wojke, A. Bewley, and D. Paulus, "Simple online and realtime tracking with a deep association metric," in *2017 IEEE international conference on image processing (ICIP)*, pp. 3645–3649, IEEE, 2017.

[36] Y. Zhang, C. Wang, X. Wang, W. Zeng, and W. Liu, "Fairmot: On the fairness of detection and re-identification in multiple object tracking," *International journal of computer vision*, vol. 129, no. 11, pp. 3069–3087, 2021.

[37] Z. Lu, V. Rathod, R. Votel, and J. Huang, "Retinatrack: Online single stage joint detection and tracking," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 14668–14678, 2020.

[38] T. Meinhardt, A. Kirillov, L. Leal-Taixe, and C. Feichtenhofer, "Trackformer: Multi-object tracking with transformers," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 8844–8854, 2022.

[39] T. Yin, X. Zhou, and P. Krahenbuhl, "Center-based 3d object detection and tracking," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 11784–11793, 2021.

[40] L. Hui, L. Wang, M. Cheng, J. Xie, and J. Yang, "3d siamese voxel-to-bev tracker for sparse point clouds," *Advances in Neural Information Processing Systems*, vol. 34, pp. 28714–28727, 2021.

[41] F. Zeng, B. Dong, Y. Zhang, T. Wang, X. Zhang, and Y. Wei, "Motr: End-to-end multiple-object tracking with transformer," in *European conference on computer vision*, pp. 659–675, Springer, 2022.

[42] K. Pauwels and D. Kragic, "Simtrack: A simulation-based framework for scalable real-time object pose detection and tracking," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 1300–1307, IEEE, 2015.

[43] T. Tang, L. Zhou, P. Hao, Z. He, K. Ho, S. Gu, Z. Hao, H. Sun, K. Zhan, P. Jia, *et al.*, "S2-track: A simple yet strong approach for end-to-end 3d multi-object tracking," *arXiv preprint arXiv:2406.02147*, 2024.

[44] A. Singh, "Trajectory-prediction with vision: A survey," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 3318–3323, 2023.

[45] R. S. Tomar, S. Verma, and G. S. Tomar, "Svm based trajectory predictions of lane changing vehicles," in *2011 International Conference on Computational Intelligence and Communication Networks*, pp. 716–721, IEEE, 2011.

[46] N. Lee, W. Choi, P. Vernaza, C. B. Choy, P. H. Torr, and M. Chandraker, "Desire: Distant future prediction in dynamic scenes with interacting agents," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 336–345, 2017.

[47] B. Ivanovic and M. Pavone, "The trajectron: Probabilistic multi-agent trajectory modeling with dynamic spatiotemporal graphs," in *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 2375–2384, 2019.

[48] T. Salzmann, B. Ivanovic, P. Chakravarty, and M. Pavone, "Trajectron++: Dynamically-feasible trajectory forecasting with heterogeneous data," in *European Conference on Computer Vision*, pp. 683–700, Springer, 2020.

[49] N. Rhinehart, R. McAllister, K. Kitani, and S. Levine, "Precog: Prediction conditioned on goals in visual multi-agent settings," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 2821–2830, 2019.

[50] "The KITTI Vision Benchmark Suite." https://www.cvlibs.net/datasets/kitti, Oct. 2025. [Online; accessed 11. Oct. 2025].

[51] "nuScenes." https://www.nuscenes.org, Dec. 2024. [Online; accessed 11. Oct. 2025].

[52] "Home." https://www.argoverse.org, May 2025. [Online; accessed 11. Oct. 2025].

[53] "About – Waymo Open Dataset." https://waymo.com/open, Oct. 2025. [Online; accessed 11. Oct. 2025].

[54] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "Carla: An open urban driving simulator," in *Conference on robot learning*, pp. 1–16, PMLR, 2017.

[55] "Traffic Manager - CARLA Simulator." `https://carla.readthedocs.io/en/latest/adv_traffic_manager`, Aug. 2025. [Online; accessed 27. Aug. 2025].

[56] maudzung, "Complex-YOLOv4-Pytorch." https://github.com/maudzung/Complex-YOLOv4-Pytorch, Nov. 2025. [Online; accessed 26. Nov. 2025].

[57] N. M. Dung, "Super-Fast-Accurate-3D-Object-Detection-PyTorch." `https://github.com/maudzung/Super-Fast-Accurate-3D-Object-Detection`, 2020.

[58] P. Li, H. Zhao, P. Liu, and F. Cao, "Rtm3d: Real-time monocular 3d detection from object keypoints for autonomous driving," in *European Conference on Computer Vision*, pp. 644–660, Springer, 2020.

[59] abewley, "SORT." https://github.com/abewley/sort, Nov. 2025. [Online; accessed 26. Nov. 2025].

[60] nwojke, "DeepSORT." `https://github.com/nwojke/deep_sort`, Nov. 2025. [Online; accessed 26. Nov. 2025].

[61] H. Wu, W. Han, C. Wen, X. Li, and C. Wang, "3d multi-object tracking in point clouds based on prediction confidence-guided data association," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 6, pp. 5668–5677, 2021.

[62] F. Marchetti, F. Becattini, L. Seidenari, and A. D. Bimbo, "Mantra: Memory augmented networks for multiple trajectory prediction," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 7143–7152, 2020.

[63] nrhinehart, "precog." https://github.com/nrhinehart/precog, Oct. 2025. [Online; accessed 26. Oct. 2025].

[64] nrhinehart, "deep_imitative_models." `https://github.com/nrhinehart/deep_imitative_models`, Oct. 2025. [Online; accessed 26. Oct. 2025].

[65] J. He, C. Fu, and X. Wang, "3d multi-object tracking based on uncertainty-guided data association," *arXiv preprint arXiv:2303.01786*, 2023.

[66] hailanyi, "3D-Multi-Object-Tracker." https://github.com/hailanyi/3D-Multi-Object-Tracker?tab=readme-ov-file, May 2025. [Online; accessed 30. May 2025].

[67] J.-E. Deschaud, "KITTI-CARLA: a KITTI-like dataset generated by CARLA Simulator," *arXiv e-prints*, 2021.

[68] "pyglm." https://pypi.org/project/pyglm, Oct. 2025. [Online; accessed 3. Oct. 2025].