Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Aksel Lunde Aase
Mathias Wold

Exploring Domain Shift with Autonomous Driving Agents in Simulated and Real Environments

Master's thesis in Computer Science
Supervisor: Frank Lindseth
Co-supervisor: Gabriel Kiss

June 2023

NTNU
Norwegian University of
Science and Technology

Aksel Lunde Aase
Mathias Wold

Exploring Domain Shift with Autonomous Driving Agents in Simulated and Real Environments

Master's thesis in Computer Science
Supervisor: Frank Lindseth
Co-supervisor: Gabriel Kiss
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

The field of autonomous driving has seen great progress in recent years, but large-scale deployment of fully self-driving vehicles is not currently present. Studies show that road traffic accidents are the leading cause of death for people aged 5 to 29 years worldwide, with the number of road traffic deaths in 2016 reaching 1.35 million. Human driving errors are also the cause of 92% of road accidents [1, 2]. Self-driving vehicles could help reduce these accidents. In addition, they can significantly reduce transportation costs for both humans and goods. Other possible benefits are increased productivity during travel, improved mobility options for non-drivers, and lower energy consumption and pollution [3].

The deployment of autonomous vehicles requires extensive testing to ensure the safety of both passengers and road users. While real-world testing most accurately reflects the autonomous system's capabilities, simulators are often utilized during development because of their reduced hardware costs, faster prototyping, and safer testing environments. However, this can introduce a domain shift when systems are exposed to new environments, possibly leading to flawed or unsafe behaviors when used in the real world.

This thesis investigates the impact of domain shift on two state-of-the-art models trained in the CARLA autonomous driving simulator [4]. TransFuser [5] and InterFuser [6], two of the top performers on the CARLA Leaderboard [7], are end-to-end models for autonomous driving that use Imitation Learning to achieve safe point-to-point navigation in urban settings. We first train and evaluate their performance in included CARLA benchmarks, then evaluate them on real-world driving data provided by NAPLab.

Our results show success in reproducing the performance of TransFuser and InterFuser in the simulator benchmarks. We train new models with a sensor configuration that matches that of the NAPLab vehicle, and show that this increases the driving performance on the real-world driving data, due to the reduced domain shift. The experiments are implemented in a reproducible manner to enable further work on our research. Finally, we include multiple suggestions for future work based on the knowledge gained in this thesis.

Sammendrag

Fagfeltet rundt autonome kjøretøy har sett stor fremgang de siste årene, men storskala utrulling av selvkjørende biler finnes fortsatt ikke. Studier viser at trafikkulykker er den ledende dødsårsaken for personer i alderen 5 til 29 år over hele verden, og i 2016 nådde antall omkomne i trafikken 1,35 millioner. Menneskelige feil er årsaken til 92% av trafikkulykker [1, 2]. Selvkjørende biler kan bidra til å redusere disse ulykkene. De kan i tillegg redusere kostnadene betydelig for transport av mennesker og varer. Andre fordeler de kan gi er økt produktivitet under reise, bedre mobilitet for mennesker uten førerkort, lavere energiforbruk og mindre forurensing [3].

Utrullingen av autonome kjøretøy krever omfattende testing for å ivareta sikkerheten til både passasjerer og trafikanter. Mens testing i den virkelige verden gjenspeiler det autonome systemets evner mest nøyaktig, er simulatorer ofte benyttet under utvikling på grunn av lavere maskinvarekostnader, raskere prototyping og tryggere testomgivelser. Samtidig kan dette introdusere en domeneforskyvning når systemene utsettes for nye omgivelser, noe som kan føre til feilaktige eller farlige oppførsler når de brukes i den virkelige verden.

Denne oppgaven undersøker innvirkningen av domeneforskyvning på to "state-of-the-art"-modeller trent i en simulator for autonome kjøretøy med åpen kildekode kalt CARLA [4]. TransFuser [5] og InterFuser [6] er to av modellene med best resultater på CARLA Leaderboard [7]. De er ende-til-ende modeller for autonom kjøring basert på imitasjonslæring, som har som mål å lære sikker punkt-til-punkt-navigasjon i urbane omgivelser. Vi starter med å trene og evaluere ytelsen deres i inkluderte CARLA-benchmarks, før vi deretter evaluerer dem på data innhentet av NAPLab fra den virkelige verden.

Resultatene våre viser suksess i å reprodusere ytelsen til TransFuser og InterFuser i simulatorevalueringene. Vi trener nye modeller med en sensorkonfigurasjon tilsvarende NAPLabs bil, og viser at dette øker kjøreytelsen på den virkelige kjøredataen, på grunn av den reduserte domeneforskyvningen. Eksperimentene er utført på en reproduserbar måte for å muliggjøre videre arbeid med forskningen vår. Til slutt foreslår vi flere forbedringspunkter til fremtidig arbeid basert på kunnskapen som er oppnådd i denne oppgaven.

Preface

This thesis is done in collaboration with NTNU Autonomous Perception Laboratory (NAPLab), a research group at NTNU with the objective of continuously developing and researching state-of-the-art models for autonomous vehicles that are robust to a Nordic environment [8]. We thank our supervisor Frank Lindseth and co-supervisor Gabriel Kiss for their guidance with the project and for providing us with necessary compute resources. We also thank Jan Christian Meyer for granting us a larger Idun storage quota needed for this work, the Idun team for answering questions regarding Idun usage, and finally Florian Wintel for lending us his computer for our final simulator evaluations as well as fruitful discussions.

Note that this thesis is a continuation of our specialization project delivered in autumn 2022: "Training and evaluating a state-of-the-art model for autonomous driving in the CARLA simulator". The specialization project, as defined by NTNU, should provide a solid theoretical platform and is often used as the foundation for the master's thesis [9]. Since our specialization project is not published, we will not cite it directly in this thesis. Some of the content in Chapters 1 and 2 will be improved and updated versions of the text found there, giving the reader the necessary background knowledge from this paper alone.

Contents

Figures

Tables

Acronyms

AD Autonomous Driving. 24, 30, 134

ANN Artificial Neural Network. 21, 24, 25, 134

BC Behavioural Cloning. 22, 23, 32–35, 37, 51

BEV bird's-eye view. 35, 37, 38, 54, 56, 72, 73, 139

CARLA Car Learning to Act. iii, v, 2, 13

CNN Convolutional Neural Network. 25, 26, 28, 29

CV Computer Vision. 29, 30

DPL Direct Policy Learning. 23, 137

DS Driving Score. xiv, xv, 76, 77, 93, 95, 96, 100, 101, 103, 105–107, 123, 124

GNSS Global navigation satellite system. 10, 79

GUI Graphical User Interface. 40

HITL Hardware in the Loop. 12

i.i.d. independent and identically distributed. 23, 134

IL Imitation Learning. iii, 22, 136

IMU Inertial measurement unit. 10

IRL Inverse Reinforcement Learning. 23, 24, 32

IS Infraction Score. 76, 77, 93, 95, 96, 100, 103, 107, 127

JSON JavaScript Object Notation. Textual format for storing data. 73

LIDAR Light Detection and Ranging. xiii, 9, 11, 14, 79

NAPLab NTNU Autonomous Perception Laboratory. iii, v, vii, xiii, 3, 4, 18–20, 39, 40, 43, 45, 49, 65, 77, 78, 91, 92, 106, 109, 127, 128, 130, 133

PCC Pearson correlation coefficient. 88

PID-controller Proportional-Integral-Derivative controller. 12, 56, 88, 139

RC Route Completion. 76, 87, 93, 95, 96, 99–101, 103, 107, 127

RL Reinforcement Learning. 24, 137

RNN Recurrent Neural Network. 27, 134

SITL Software in the Loop. 12

Glossary

activation function A function calculating the output of a neural input given its weighted input. 21, 25

domain shift The change of the distribution of input data when, for instance, exposing an autonomous agent trained in a simulator to the real world. iii, 2–4, 125, 136, 139

loss A mathematical function to evaluate the error or cost in the predictions of an artificial neural network. xiv, 22, 33, 55, 56, 58, 71, 93, 94, 101, 103, 104, 108

positional encoding An encoding of the position of each input token in the Transformer architecture. 27

reinforcement learning Algorithm used to train artificial neural networks using random exploration and rewards. 24

reward function Used in Reinforcement Learning to guide the agent toward positive rewarding actions. 23, 24

supervised learning Algorithm used to train artificial neural networks using labeled data. 22–24

Transformer A neural network architecture utilizing an attention mechanism to focus on important parts of its input. 25, 27, 28, 54

Vision Transformer A neural network architecture for visual data, utilizing an attention mechanism to focus on important parts of its input. 3, 28

Chapter 1

Introduction

This chapter introduces the thesis. Its motivation is discussed in Section 1.1, while Section 1.2 presents its goals and research questions. Section 1.3 describes the applied research method. Contributions made in the thesis are presented in Section 1.4, before the chapter ends with an overview of the thesis outline in Section 1.5.

1.1 Motivation

The field of autonomous driving has seen great progress since the first computer-controlled self-driving car was released in 1986 [10]. With a top speed of 32 km/h, the modified Chevrolet was capable of basic routing and obstacle avoidance. By 1995, prototype cars demonstrated lane keeping, cooperative driving, and better vision of the surroundings. Today, these features are just some of the challenges faced when designing and creating fully autonomous vehicles, and large players in the industry such as Tesla and Google are publicly pushing the technology towards this common goal [11, 12]. Waymo opened its public autonomous vehicle taxi service to the general public in Phoenix in 2020 [13], and Baidu recently got a permit to offer similar driverless services in Beijing [14]. Tesla's Full Self-Driving Beta is soon expanding to Tesla owners outside of North America [15]. A report published last year by the Environmental and Energy Study Institute suggests that half of newly produced vehicles could be fully autonomous by 2050, and half of all vehicles by 2060 [16]. Still, a large-scale deployment of fully self-driving vehicles is not currently present.

The United Nations Economic Commission for Europe (UNECE) publishes statistics every two years on road traffic accidents in Europe and North America [17]. Their latest release shows that almost 100,000 people died as a result of road traffic accidents in 2019, which is equivalent to an average of 270 people per day. Zooming out to a global scale, the numbers are even worse. The World Health Organization (WHO) clearly states that road traffic accidents are the leading cause of death for people aged 5 to 29 worldwide, reporting 1.35 million road traffic deaths in 2016, or 3,700 per day on average [1]. Autonomous driving technology could help reduce or even remove the effect of human driving errors, such as driver inattention, distractions, false assumptions, ignoring speed limits, and driving under the influence. Together, these are the cause of around 92% of road accidents [2]. This is a clear motivation for the development and deployment of autonomous vehicles.

Autonomous vehicles could also help reduce the cost of transport. An analysis on autonomous ships released by the Danish Technical University in 2020 estimated that the gross cost per employee amounts to one million DKK annually, with crew costs typically accounting for 20-30% of the total cost for a cargo ship's journey [18]. We can see the same trends in the context of self-driving cars. Victoria Transport Policy Institute predicts that private autonomous vehicle costs will on average be 55% cheaper than human-operated taxis per mile [3]. If autonomous vehicles are shared instead of private, the costs are predicted to be even lower. Costs of delivered goods and services can also be reduced if autonomous vehicles were to take over for human-driven cars and trucks, which would be beneficial for both the customer and the supplier. Other potential benefits of a deployment of autonomous vehicles include increased productivity while traveling, better mobility options for non-drivers, increased road capacity, reduced parking costs and reduced energy consumption and pollution [3]. The latter will especially be noticeable with a possible full-scale deployment of connected self-driving cars that can intelligently interact with each other to reduce traffic congestion and therefore also reduce inefficient fuel consumption and carbon dioxide emissions. A deployment of autonomous vehicles will also accelerate the transition to electric vehicles, providing further environmental benefits.

When developing autonomous driving technology, there are two choices for evaluating performance: Either test using a real car in the real world, or use a virtual car in a computer-simulated environment. While real-world testing most accurately reflects the agent's capabilities, testing in a simulator is often both necessary and desirable. It is not only safer to use simulated environments, but it is also much cheaper due to the reduced hardware cost. Simulators enable more rapid prototyping and development and are generally more available to the research community. They also make it possible to create and evaluate rare corner cases, something that is difficult to replicate in the real world. However, due to the uncertainties associated, particularly with neural network-based autonomous technology, such technology must always be thoroughly tested in the real world before one can be fully confident in its behavior. While doing so, one might uncover several behaviors and flaws that were not present in the simulated environments, an effect caused by the domain shift occurring when the agent is exposed to new environments. The focus of this thesis is to explore the impact of domain shift on two neural network models that are trained on autonomous driving tasks in the CARLA simulator, but evaluated on real-world data.

1.2 Goals and Research Questions

The main goal of this thesis is to explore and evaluate state-of-the-art end-to-end models for autonomous driving in simulated and real environments. We aim to investigate two such models, TransFuser [5] and InterFuser [6], both utilizing Vision Transformer architectures. This includes upgrading them to work with the latest version of the CARLA simulator, generating data, and setting up a training and evaluation pipeline that is easy to use, both for us and for future researchers. We will also generate data and train the models in CARLA with a sensor configuration that matches NAPLab's Kia e-Niro, which will enable us to investigate the domain shift incurred by the transition from simulation to reality.

To reach this goal, we will answer the following research questions:

RQ1: Are the TransFuser and InterFuser results reproducible in CARLA version 0.9.10?
RQ2: How does the upgrade to version 0.9.14 affect the performance?
RQ3: How do the models perform on real-world data generated with NAPLab's Kia e-Niro?
RQ4: Is it necessary to generate data using a sensor configuration that matches the real car, or would the models perform well in different sensor configurations without modifications?
RQ5: Is TransFuser or InterFuser best suited for real-world data?

An additional goal of this thesis is to enable continued work on the research and results that we present. We therefore strive to include all relevant details of the workflow, such that future researchers may use this document as a guide to rapidly resume from the point where this work ends. Where details are omitted from the thesis, we point to the relevant pieces of implementation, which is also intended to be easily used even without this thesis as a reference.

1.3 Research Method

This thesis is based on an experimental research method. A literature review will be conducted to compare current state-of-the-art methods for autonomous driving. We will then perform experiments using benchmarks in the CARLA simulator and evaluate models using real-world data. The benchmarks consist of a set of routes from different towns, where each route has a unique environmental condition based on a combination of weather and lighting conditions. The routes also contain several adversarial scenarios that test the autonomous agent's driving proficiency in realistic traffic situations.

We will mainly use a quantitative analysis of the results. The benchmarks output performance metrics such as route completion, infraction score, driving score, and infractions per kilometer, all of which can be used to compare different experiments. We will also perform a qualitative analysis of the evaluation runs. This will be done by inspecting the visual outputs of the models while the autonomous vehicle is driving through the evaluation routes. For our real-world experiments, we will create a comparison between the model's driving choices and the ground-truth driving data.

1.4 Contributions

The main contributions of this thesis to the research field of end-to-end autonomous driving are:

- A literature review of the latest state-of-the-art end-to-end models for autonomous driving.
- A performance comparison between two of these models, TransFuser and InterFuser, in both simulated and real environments.
- An investigation into the impact of domain shift on autonomous agents trained in a simulator but evaluated on real-world data.

In addition, the work in this thesis has produced the following technical contributions, most of which can be found in our GitHub organization[1]:

- Updated codebases for TransFuser and InterFuser that are compatible with the newest version of the CARLA simulator (0.9.14).
- Methodology to use the CARLA simulator for data generation on NTNU's HPC cluster Idun.
- Methodology to train and evaluate TransFuser and InterFuser locally in reproducible environments.
- Pipeline to process real-world data through TransFuser and InterFuser.
- A digital version of NAPLab's autonomous vehicle, including its sensor configuration, is added to CARLA.
- Datasets with two different sensor configurations for both TransFuser and InterFuser. Available for future work using NAPLab's compute resources.

1.5 Thesis Outline

This thesis contains six chapters with the following outline:

Chapter 1: Introduction describes the context of the thesis. This includes the motivation and purpose of the thesis, along with research questions and how the thesis contributes.

Chapter 2: Background and Related Work discusses relevant background topics and related work.

[1] `https://github.com/orgs/aasewold`

Chapter 3: Methodology describes in detail the tools and methods we will use to answer our research questions.

Chapter 4: Experiments and Results contains the setups, results, and discussion for each of our experiments.

Chapter 5: Discussion will discuss the overall results of the thesis in light of the research questions.

Chapter 6: Conclusion and Future Work summarizes what was achieved in the thesis and discusses future work that could follow this thesis.

Chapter 2

Background and Related Work

This chapter introduces the concepts and theory relevant to the work presented in this thesis. The chapter begins with a presentation of various approaches to autonomous driving in Section 2.1, before diving into simulated and real environments in Sections 2.2 and 2.3. Sections 2.4 and 2.5 presents necessary theory about deep learning and computer vision, and finally Section 2.6 explores related works.

2.1 Autonomous Driving

Autonomous driving refers to the ability of a vehicle to operate and navigate without direct human input or intervention. Using artificial intelligence, sensors, and on-board computers, the goal is to enable the vehicle to perceive its surroundings, make decisions, and navigate in a safe and efficient manner. This section will explore in detail different levels of driving automation, sensors, and approaches to autonomous driving systems.

2.1.1 Levels of Autonomy

eIt is helpful to have a clear understanding of the capabilities of a given autonomous system. SAE International (formerly known as the Society of Automotive Engineers) has defined a standard for autonomous driving systems that provides a taxonomy with detailed definitions for six levels of driving automation [19]. This standard is often referenced in the literature and is adopted by the United States Department of Transportation [20]. The levels are summarized below, while full details are available in SAE's infographic shown in Figure 2.1. Not only does SAE's standard set clear expectations for drives, but it can also help promote safety by ensuring that drivers understand when they need to be alert and ready to take control of the vehicle.

Level 0: No driving automation The driver is fully in control of the vehicle at all times.

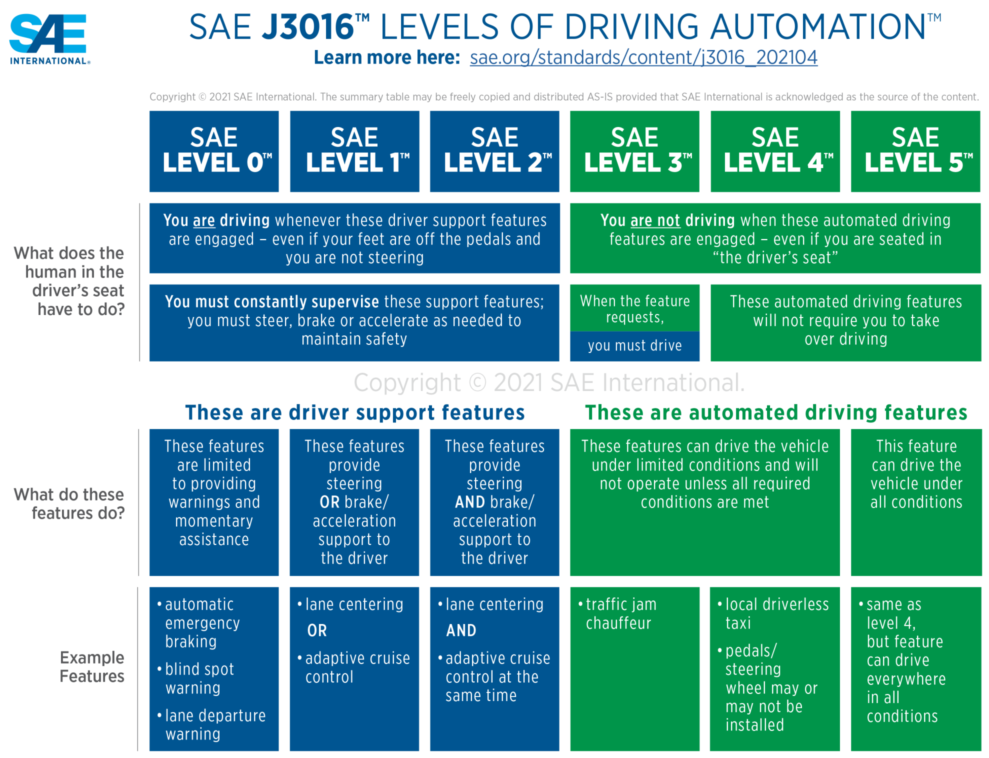
Level 1: Driver assistance The vehicle provides some assistance with driving tasks, but the driver must remain alert and ready to take control at any time.

Level 2: Partial driving automation The vehicle can assist with steering and acceleration/braking, but the driver must remain alert and ready to take control at any moment.

Level 3: Conditional driving automation The vehicle can handle some driving tasks under certain conditions, but the driver must remain available and ready to take control when prompted by the vehicle.

Level 4: High driving automation The vehicle can handle all driving tasks under certain conditions, and the human in the driver's seat is not required to take any active role in driving.

Level 5: Full driving automation The vehicle can handle all driving tasks under all conditions, and the human in the driver's seat is not required to take any active role in driving.



SAE **J3016**™ LEVELS OF DRIVING AUTOMATION™

Learn more here: sae.org/standards/content/j3016_202104

	SAE LEVEL 0™	SAE LEVEL 1™	SAE LEVEL 2™	SAE LEVEL 3™	SAE LEVEL 4™	SAE LEVEL 5™
What does the human in the driver's seat have to do?	**You <u>are</u> driving** whenever these driver support features are engaged – even if your feet are off the pedals and you are not steering			**You <u>are not</u> driving** when these automated driving features are engaged – even if you are seated in "the driver's seat"		
	You must constantly supervise these support features; you must steer, brake or accelerate as needed to maintain safety			When the feature requests, you must drive	These automated driving features will not require you to take over driving	
	These are driver support features			**These are automated driving features**		
What do these features do?	These features are limited to providing warnings and momentary assistance	These features provide steering **OR** brake/ acceleration support to the driver	These features provide steering **AND** brake/ acceleration support to the driver	These features can drive the vehicle under limited conditions and will not operate unless all required conditions are met		This feature can drive the vehicle under all conditions
Example Features	• automatic emergency braking • blind spot warning • lane departure warning	• lane centering **OR** • adaptive cruise control	• lane centering **AND** • adaptive cruise control at the same time	• traffic jam chauffeur	• local driverless taxi • pedals/ steering wheel may or may not be installed	• same as level 4, but feature can drive everywhere in all conditions

Figure 2.1: Levels of autonomy in autonomous driving as defined by SAE International. Source: [21].

We can see that Levels 0 to 2 consider driver support features, while Levels 3 to 5 consider automated driving features. Globally, most vehicles belong to Level 0. Vehicles with cruise control, even adaptive, fall into Level 1. This is the standard in developed countries. Although Tesla advertises both its Autopilot and Full Self-Driving (FSD) Beta as Level 2 systems, legal scholars argue that FSD should be considered Level 4 technology and that Tesla tries to "avoid regulatory oversight and permitting processes required of more highly automated vehicles" with the Level 2 classification [22]. Today, Mercedes is the only car manufacturer with Level 3 certification in the United States [23]. The Waymo Driver, currently used for autonomous taxi services, is restricted to selected areas in the United States and is therefore at Level 4 [24, 25]. Baidu will soon offer similar Level 4 driverless services in Beijing [14]. There are no Level 5 vehicle deployments today [23], although Tesla CEO Elon Musk recently commented at a conference that all of their vehicles since 2016 will be updated to Level 5 when FSD is released [26]. The work in this thesis experiments with situations where a human driver is not needed to operate the vehicle, meaning that we target SAE Levels 4 and 5.

2.1.2 Sensors

Sensors are essential parts of every autonomous system. They enable the system to perceive its surroundings and make informed decisions about how to navigate safely and efficiently. We will in this section look into common types of sensors found in today's autonomous vehicles. As we will see in later chapters, they can complement each other to give a comprehensive view of the vehicle's surroundings through sensor fusion. Figure 2.2 shows some example sensor types.

Cameras

RGB cameras can capture color images and videos of the vehicle's surroundings, and can assist in tasks such as lane detection and traffic light recognition. Choices of resolution, field of view, and relative position to the vehicle will all affect the output of the cameras. Multiple cameras can be combined to create a larger degree view of the environment. Most vendors use cameras as their primary sensor for autonomous driving features, including Tesla [27], Mercedes [28], and Waymo [29].

LIDAR

Light Detection and Ranging (LIDAR) sensors are used to create a 3D map of the vehicle's surroundings. They rotate at high speed while emitting laser beams that reflect off objects in the environment, and by measuring the time of the reflected light to return to the receiver, they can create high-resolution maps showing the distance to and shape of objects. These maps are often referred to as LIDAR point clouds. Although most autonomous cars are equipped with LIDAR, Tesla argues that they are not needed given adequate camera vision processing [27].

Radar

Radars use radio waves to detect objects in front of the sensor. Similar to LIDAR they use reflected waves to calculate distance and speed of objects relative to the vehicle. They provide higher precision than LIDAR, and due to their larger radio waves, they are especially useful in adverse weather conditions where LIDAR and cameras can fail to provide accurate information due to rain, snow, or fog [30]. Waymo [29] and Mercedes [28] are examples of manufacturers that use radars.

Ultrasonic Sensors

Ultrasonic sensors use high frequency sound waves to detect objects in the environment. While radar sensors are useful for long-range detection, these sensors are better used for close-range detection. Therefore, they are often utilized for tasks such as parking assistance and collision avoidance. Like radar, they are also effective in adverse weather conditions [31]. Tesla has controversially removed ultrasonic sensors from its cars in favor of their vision processing system [32], but the sensor is otherwise a standard component for most car makers.

GNSS

Global navigation satellite system (GNSS) receivers are used to determine geolocation of the vehicle based on satellite data. They provide highly accurate information and are therefore well suited for long-term positioning and navigation. However, these sensors can be affected by signal loss or interference, especially in areas with poor GNSS reception, such as tunnels or between large buildings. GNSS is a standard feature in most cars today.

IMU

Inertial measurement unit (IMU) sensors measure vehicle motion and orientation. They contain accelerometers and gyroscopes that detect changes in speed, direction, and orientation. As IMUs will not suffer GNSS signal loss or interference, they are also useful for short-term positioning and navigation in, for example, tunnels. IMUs are used by most autonomous car manufacturers since they do not require a connection or knowledge of the external world.

2.1.3 Modular vs End-to-End Approaches

When developing systems for autonomous driving, we generally differ between two main approaches: modular and end-to-end [33]. The former divides the autonomous system into modules with different areas of responsibility. There can, for example, be different modules responsible for perception, prediction, route planning or vehicle control. These modules vary in complexity. The end-to-end approach, on the other hand, looks at the autonomous system as one complete

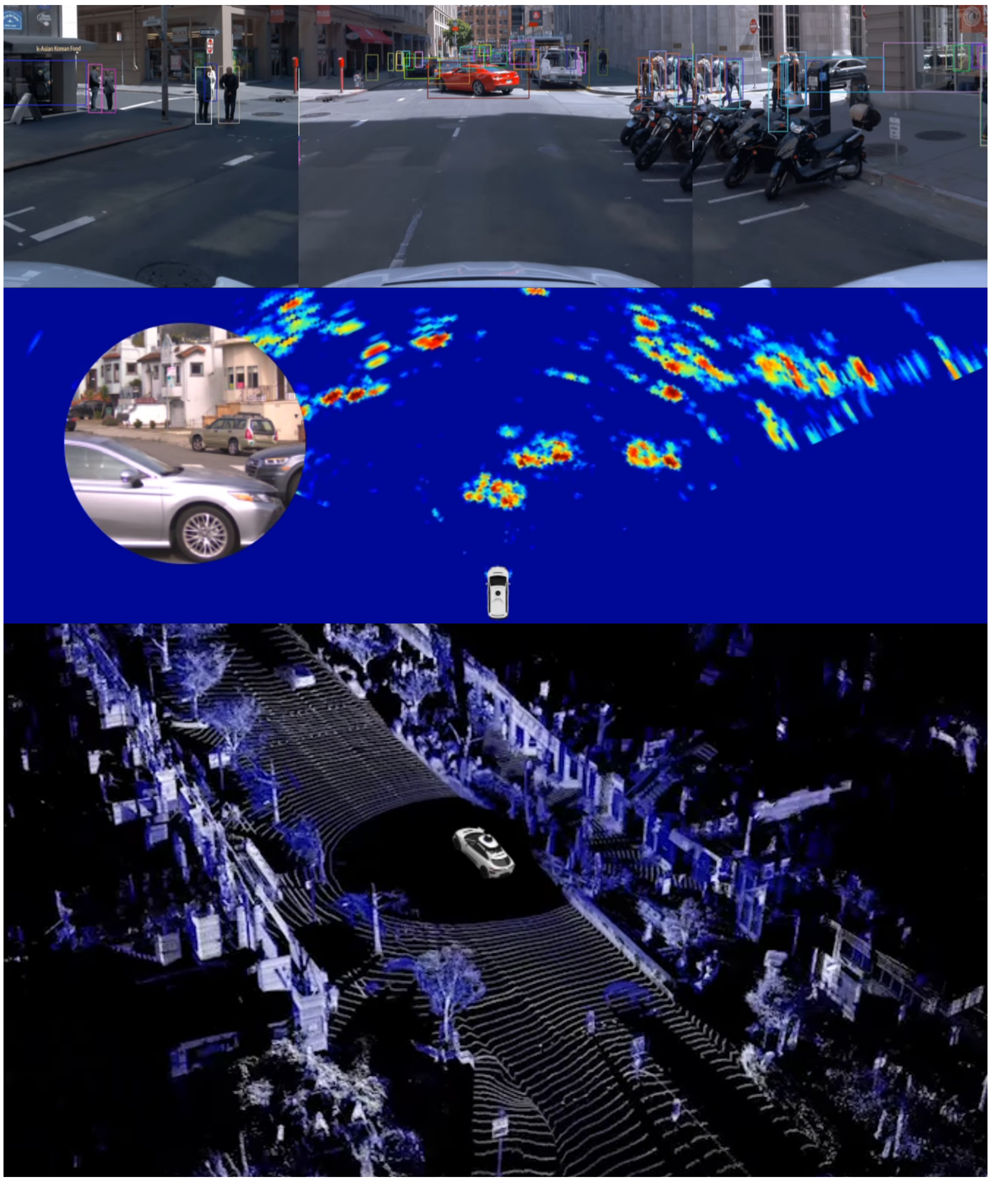


Figure 2.2: Illustrations of camera (top), radar (middle), and LIDAR (bottom) sensor types. The images are part of Waymo's video visualizations [29].

unit. This approach takes raw sensor data (RGB camera, LIDAR, GPS, etc.) as input, and output planning results, often in the form of waypoints. This can then be fed into traditional control algorithms such as PID-controllers to output actions such as acceleration, steering, and braking. Some systems also transform the sensor data into scene representations to provide interpretability and other benefits, something which will be discusses more in detail in Section 2.6.

As discussed in a recent paper comparing end-to-end techniques, and as seen in Section 2.6, there is an ongoing shift toward the latter approach [34]. Although the modular approach offers an explainable and verifiable system in which each module can individually be evaluated, it lacks efficient use of computer resources and awareness of the high-level task that is autonomous driving. In addition, the modular approach requires a huge amount of annotated data for each module. This is in contrast to the end-to-end approach, where the annotations live in the state of the car (steering commands, GPS location, etc.) at a given moment.

2.2 Simulated Environments

The need for safe environments when developing autonomous driving technology cannot be understated, and simulated environments are as safe as you can get. Simulation software can provide a physically and visually realistic environment in which an AI agent can be trained, enabling both safe and greatly simplified development, compared to embedding the AI agent in the real world from its inception. It can also be used to create and test corner case scenarios, something that can be both difficult and unsafe to do in the real world.

Simulator development has much in common with 3D game development, as both disciplines see requirements for physically realistic movement, photorealistic rendering, convincing environments, and artificial intelligence/behavior control for actors situated in the virtual world.

One major difference, however, is how the software is interacted with. For games, the traditional interaction involves a human player providing input in the form of key or mouse presses, for which the game responds by updating the internal state and rendering a new scene, which is then presented to the player. Simulators are similar, but the crucial difference is that the player is another computer program, and that the two programs are usually configured to run in lock-step. That is, the simulator updates its internal state and renders a scene which is passed to the actor. The actor parses the information received and determines an appropriate action, which is sent to the simulator. The simulator, upon receiving the action, updates its internal state again and renders a new scene. This sequence is often called Software in the Loop (SITL), in contrast to Hardware in the Loop (HITL) in which the actor would be embedded in a physical vehicle. The former is naturally both safer and leads to faster progress, but has the downside of not being a true copy of the real world, having both inaccuracies in physics and rendered frames, and often presents less varied environments and challenges than the unpredictable real world. AI agents trained to perfection in simulated environments are therefore not guaranteed to perform well once deployed to the real world, an effect we intend to explore in depth in this thesis. But first, we present our simulator of choice, CARLA.

2.2.1 CARLA

Car Learning to Act (CARLA) is an open-source photo-realistic simulator for research on autonomous driving [4]. It was developed from the ground up to support development, training, and validation of autonomous urban driving systems. It also features an online leaderboard for evaluating autonomous agents' driving performance in realistic traffic scenarios [7]. CARLA provides free and open digital assets such as urban layouts, buildings, and vehicles. Additionally, the simulator includes a suit of sensors, environmental conditions, full control over static and dynamic actors, a traffic manager, and more.

CARLA is built with a scalable client-server architecture in mind. The server uses Unreal Engine [35] as a simulation foundation and is responsible for tasks such as scene and sensor rendering, computation of realistic physics, actor updates, and providing updated world information to the connected clients. Multiple clients can be connected to the CARLA server, and these send commands to the server and receive updated world information in return. Example commands are traffic generation, controlling vehicles, and setting weather conditions.

Python API

To implement the client-server functionality, the CARLA team has developed client APIs in Python [36] and C++ [37] that leverage sockets to establish a connection to the server. The work in this project is based on the latest released version of the Python API, which at the time of writing is version 0.9.14. The Python API can be used to control the core parts of the simulation:

World: The world represents the simulation, and one can change properties such as weather conditions, lighting, and the map to use.

Map: The map represents the actual simulated world, often called the town. The client can manage roads, lanes, and junctions in the map, which are used together with waypoints to provide vehicles with a navigation path. Waypoints are simply points in 3D space oriented in the direction of the lane containing them.

Actors: An actor is anything that plays a role in the simulation. Examples are pedestrians, vehicles, sensors, and traffic lights.

Blueprints: Blueprints are already-made models that are used to spawn new actors into the world. These include properties such as vehicle color, sensor channels, and pedestrian walking speed.

Sensors: Sensors are attached to vehicles. They observe the simulated environment, collect data, and make this available to the client. There are many sensor types in CARLA, including realistic sensors (cameras, LIDAR, GPS, etc.) and ground-truth sensors from the simulator (collision detection, semantic segmentation, etc.). Figure 2.3 illustrates the outputs of different camera sensors and the LIDAR sensor.

Towns and Weather Conditions

CARLA includes several towns (maps) that can be loaded using the client. They are defined using an OpenDRIVE file, which is a standardized annotated road definition format that describes the geometry of roads, lanes, junctions, and environmental objects [39]. This thesis experiments with all included CARLA towns [40], except Town08 and Town09:

Town01: A basic town layout consisting of "T junctions".

Town02: Similar to Town01, but smaller.

Town03: The most complex town, with a 5-lane junction, a roundabout, unevenness, a tunnel, and more.

Town04: An infinite loop with a highway and a small town.

Town05: Squared-grid town with cross junctions and a bridge. It has multiple lanes per direction. Useful to perform lane changes.

Town06: Long highways with many highway entrances and exits. It also has a Michigan left.

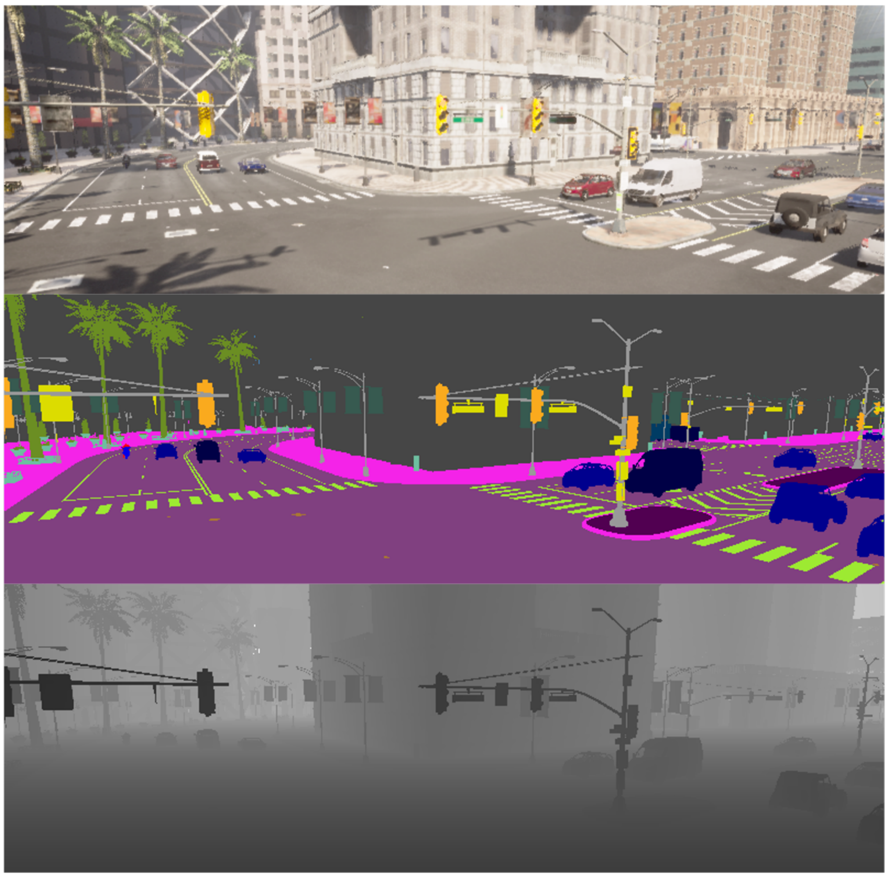
Town07: A rural environment with narrow roads, barns, and hardly any traffic lights.

Town08: A secret "unseen" town used in the CARLA Leaderboard evaluations.

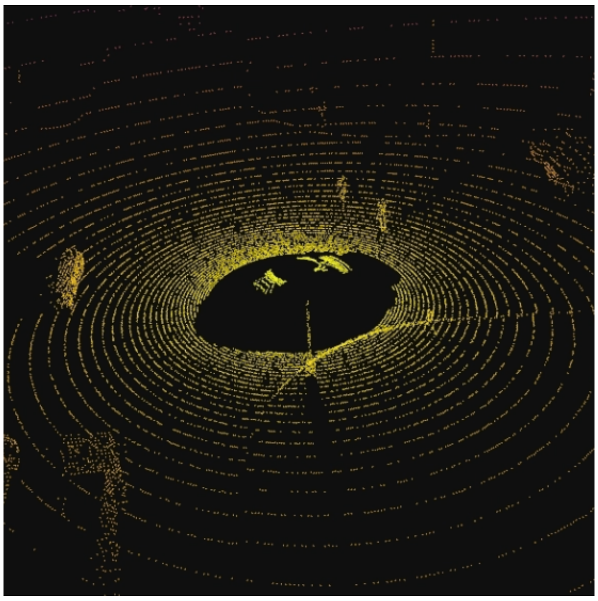
Town09: Same as Town08.

Town10: A city environment with different environments such as an avenue or promenade, and more realistic textures.

Top-down visualizations of each map's road layout can be found in [40], with an example shown in Figure 2.4. Each town is by default loaded with a specific weather configuration, but this can be changed. Using the client, one can change several weather parameters such as sun orientation, fog, cloudiness, and rain intensity. CARLA also provides preset configurations such as clear noon or hard rain sunset.

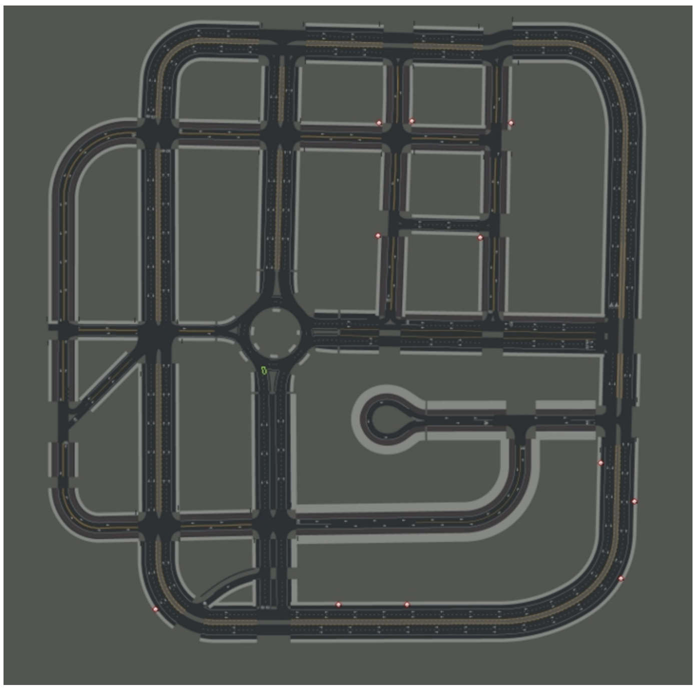


(a) Three different camera types. From top to bottom: RGB image, semantic segmentation image, and depth image.

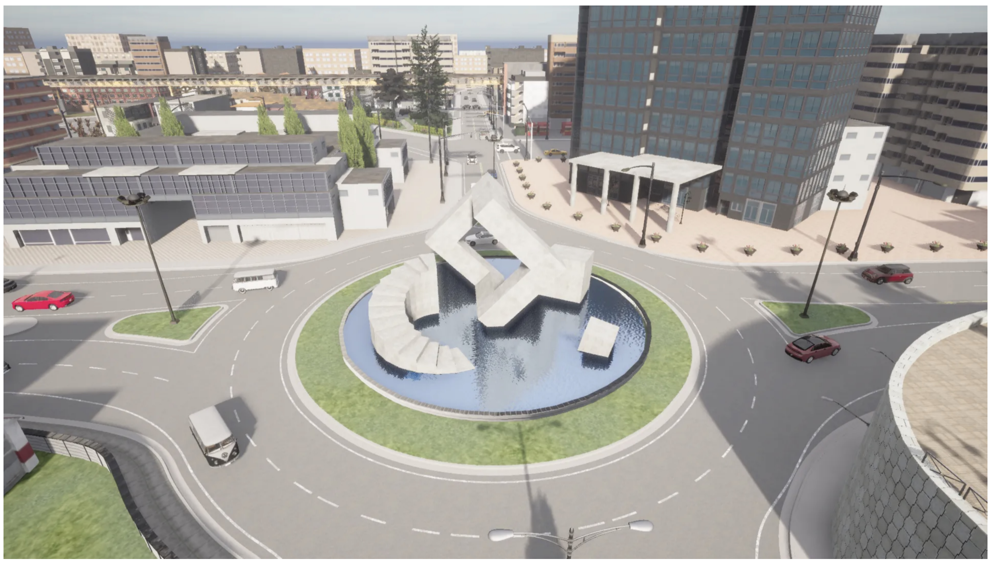


(b) Example LIDAR point cloud. Source: [38].

Figure 2.3: Visualizations of camera types and LIDAR generated in the CARLA simulator.



(a) Top-down view of Town03's road layout. It includes a roundabout, underpasses and overpasses.



(b) Rendered view of Town03, showing the roundabout.

Figure 2.4: Example visualizations of a map in CARLA. Source: [41].

Leaderboard

The CARLA Autonomous Driving Leaderboard [7] is an open platform for evaluation of driving performance by autonomous agents in realistic traffic scenarios. The goal of the platform is to simplify comparisons between different approaches to autonomous driving using a set of predefined routes. The routes vary in weather conditions and world layouts, and contain scenarios such as lane merging, lane changing, intersections, yielding to emergency vehicles, and handling traffic lights. They also have constraints on which type of sensors one can utilize, depending on which leaderboard track they belong to. There are two leaderboard tracks; the "SENSORS track" only allows access to sensors attached to the vehicles, while the "MAP track" allows access to an HD world map in addition to the sensors.

The evaluation is based on two metrics. The first is route completion, which is simply the average percentage of route distance completed. The other metric counts infractions per kilometer and aggregates them into a penalty score. The infraction types counted by the leaderboard are collisions with pedestrians, vehicles and static objects, as well as running red lights and stop signs. The route completion score and the infraction score are combined into a driving score, which is the main metric of the leaderboard.

CARLA Leaderboard uses the CARLA ScenarioRunner [42] during evaluations. ScenarioRunner is used to define and execute the traffic scenarios and routes described above using a Python interface. Example scenarios included are vehicles that run red lights across intersections or cyclists driving into the road in front of the ego vehicle during a turn.

There are two versions of the leaderboard. Leaderboard 1.0 is for CARLA version 0.9.10, while Leaderboard 2.0 is for the latest CARLA version (0.9.14). Although Leaderboard 2.0 is the current version, it contains no entries at the time of writing. Therefore, this thesis will discuss submissions from Leaderboard 1.0.

Ways to run the CARLA server

The primary way to use the CARLA server is by downloading and installing the packaged version of the simulator[1]. The packaged version is not customizable. CARLA can also be built from the source code[2]. This is the recommended approach if one wants to add extra features, assets, or maps to the simulator via Unreal Editor. CARLA can be built from source on Linux and Windows. The last option is to run the simulator in a Docker container[3]. This is useful for running without a display and for running multiple servers with GPU mapping. Different versions of CARLA support different graphics APIs. Versions until 0.9.11 support using both

[1]`https://carla.readthedocs.io/en/0.9.14/start_quickstart/`

[2]`https://carla.readthedocs.io/en/0.9.14/build_carla/`

[3]`https://carla.readthedocs.io/en/0.9.14/build_docker/`

Vulkan and OpenGL, but only OpenGL worked for us when running without a display. Versions equal to and after 0.9.12 only support Vulkan, and worked both with and without a connected display. We used the Docker approach for all our tasks except when adding a custom vehicle, as this required a version of CARLA built from source with access to Unreal Editor. This is explained in more detail in Section 3.3.

Alternatives to CARLA

There exist multiple simulator alternatives to CARLA, each with its own advantages and disadvantages. They often have a trade-off between visual fidelity of the simulated 3D environment and the level of realism in the computed vehicle physics. According to Malik *et al.* [43], who have done an extensive comparison between different alternatives, the state-of-the-art simulators that best meet this trade-off are CARLA and LGSVL [44]. Similarly to CARLA, the open source LGSVL simulator provides tools which allow users to customize sensors, controllable objects, and other modules. Additionally, LGSVL provides integration with autonomous driving software, making it a complete end-to-end tool. Note that the simulator stopped receiving updates from the original authors as of January 2022. However, there still exist usable forks of the project which can be used. Other popular open-source alternatives are AirSim [45], DeepDrive [46], and Torcs [47].

An interesting upcoming simulator is NVIDIA Drive Sim [48]. It is an end-to-end simulation platform with focus on being physically accurate, fast, and efficient at scale. While this is a project showing impressive photo-realistic simulation, it is currently in an early access phase, meaning that you need to apply to access it. Another disadvantage is the commercial and closed nature of the product, meaning that one will have less control and configurability over the simulator.

2.3 Real Environments

Although simulated environments are great for training and evaluating autonomous agents in a safe and efficient manner, the goal is always to deploy them in the real world. However, simulations often lack the unpredictable interactions and properties of reality. It is difficult, if not impossible, to model and simulate all possible events that an autonomous agent will have to deal with in the real world, especially in the domain of autonomous driving. The agent may encounter unexpected scenarios, various weather conditions, and different traffic patterns that might not have been seen in the simulator. Simulators also often provide idealized sensor data, whereas the real world exposes the agent to sensor inputs with noise, errors, and inaccuracies.

The resulting performance gap when transferring an agent from simulation to reality must be addressed before it can be deployed. In this thesis we will explore this gap by utilizing NAPLab's modified car for testing our models.

2.3.1 NAPLab Car

NAPLab owns a 2019 Kia e-Niro modified to be used for autonomous driving research. The car is equipped with several sensors, a computer, and other hardware to enable data recording and vehicle control. In 2022 Gusev [49] explored the possibility of remote controlling the car over a 5G network and included in his thesis a detailed description of the car's hardware and software setup. We will in this section summarize the setup, and the reader is encouraged to consult [49] for more details.

A NVIDIA DRIVE AGX Xavier computer [50] acts as the core of the autonomous system and lies in the trunk of the car. It is specifically built for reading and processing data from sensors to perceive the surrounding environment, and is connected to the vehicle's CAN bus through a drive-by-wire kit. It provides access to the car's hardware through NVIDIA DRIVE OS [51]. The operating system is built securely for the development and deployment of autonomous vehicle applications. The computer comes installed with NVIDIA DriveWorks Software Development Kit [52], which enables development through comprehensive modules, tools, and applications. Some features it provides include a unified interface to the vehicle's sensors, camera calibration tools, and a recording tool. This end-to-end NVIDIA stack fully utilizes the throughput limits of the system, enabling real-time self-driving applications.

Connected to the computer is a NETGEAR GS116E gigabit Ethernet switch that interfaces with some of the sensors and an ARCUS 5G router by Celerway. The router is built to support high-capacity connectivity on the move and outdoors, and uses four antennas attached to the front and back windows to provide cellular connectivity. The following sensors are added to the car:

- Eight SEKONIX NVIDIA DRIVE cameras with 60° and 120° FOV (SF3325 and SF3324). All have a resolution of 1928×1208 pixels and 12-bit RGB color depth. They are mounted to cover all sides of the car (three in front, two on each side mirror, and one in the back) and are connected directly to the computer.
- One large Ouster LIDAR sensor with 128 beams (OS2-128). It is mounted on the roof and is directly connected to the computer.
- Two smaller Ouster LIDAR sensors with 16 beams (OS1-16). They are mounted on the front and right rear sides and are connected to the switch.
- Two Continental ARS 408-21 Long Range 77Ghz radar sensors, one in the front and one in the rear. They are directly connected to the computer.
- Two Swift GNSS mini-survey antennas on the roof, connected to the switch.

Figure 2.5 shows the inside mounted equipment, while Figure 2.6 visualizes the positionings of the camera and LIDAR sensors. The relevant sensors for this thesis are the front 120° camera, the front-facing cameras mounted on the side mirrors, and the large rooftop LIDAR.

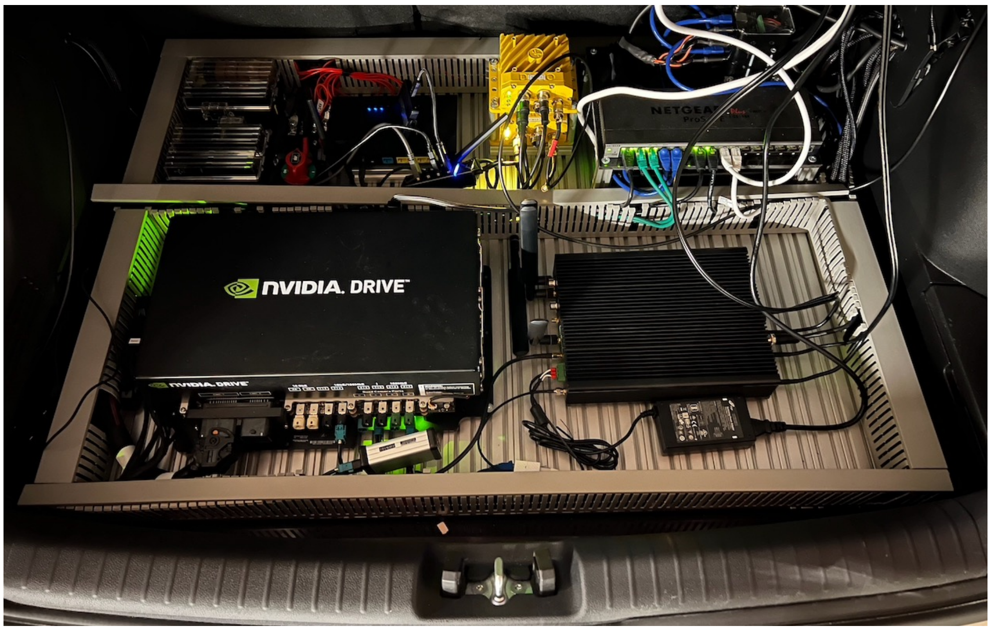


Figure 2.5: Overview of the trunk of NAPLab's car. In the bottom of the image we see the NVIDIA AGX Xavier computer on the left and the ARCUS 5G router on the right. In the top we see the switch, GNSS receivers, and some additional equipment.

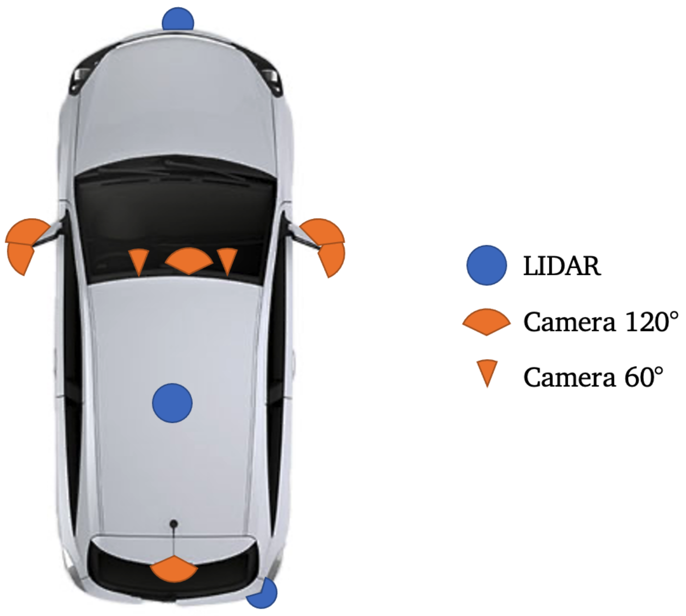


Figure 2.6: Placements of the LIDAR and camera sensors on NAPLab's Kia e-Niro.

2.4 Deep Learning

Deep learning is a subset of machine learning that uses Artificial Neural Networks (ANNs) to model complex patterns in data. ANNs are biologically inspired structures built to approximate arbitrary mathematical functions, composed of simple building blocks like convolutions, matrix multiplications, and various nonlinear functions. With the right combination of operations, neural networks are able to learn complex tasks with very high accuracy, with costs much lower than those of alternative methods.

In the context of self-driving vehicles, deep learning algorithms are heavily utilized to interpret camera data and provide an understanding of the environment around the car, in addition to actually controlling the vehicle. In contrast to previous rule-based AI systems, deep learning agents show much greater potential to successfully navigate a vehicle under varied conditions. ANNs also prove useful in smaller tasks such as automatic detection of speed limits, lane markers, and other roadside information.

The most common representation of an artificial neuron is perhaps the following mathematical expression for calculating its output value:

$$a(\mathbf{x}) = f(\mathbf{w} \cdot \mathbf{x} + \mathbf{b})$$

where the output a is determined by the dot product between the neuron's weights $\mathbf{w}$ and input $\mathbf{x}$, the neurons bias $\mathbf{b}$, and the choice of activation function f. However, in practice, this expression is broken into even smaller building blocks, and is more commonly formulated as a sequence of basic operations:

$$\begin{aligned} \mathbf{a}_1 &= \mathbf{w} \cdot \mathbf{x} \\ \mathbf{a}_2 &= \mathbf{a}_1 + \mathbf{b} \\ \mathbf{a}_3 &= f(\mathbf{a}_2) \end{aligned}$$

This representation enables much greater flexibility, allowing reordering of the computations or building arbitrarily complicated computation graphs. The task of the ANN designer is to create a suitable computation graph (network architecture) for the problem at hand, while the task of the training algorithm is to find optimal values for all trainable parameters $(\mathbf{w}, \mathbf{b}, \dots)$. This section will further explore various network architectures and training algorithms.

2.4.1 Training Methods

Countless algorithms have been created to infer the optimal values for the trainable parameters in an ANN. They can generally be classified into three distinct categories: supervised learning, unsupervised learning, and reinforcement learning. Both supervised learning and reinforcement learning are commonly used in autonomous driving tasks, and supervised learning in particular is relevant for the models we explore later on. Therefore, we take a closer look at these two categories.

Supervised Learning

Supervised learning is a type of machine learning algorithm that uses a labeled dataset. That is, the dataset consists of pairs of input and output data, and the model is trained to predict the output ("label") given the input. The goal of supervised learning is to make a model capable of making predictions on new data based on the patterns it learned from the training data. In the case of self-driving vehicles, the labeled data might consist of dash cam images of traffic and corresponding bounding boxes of each car in the image, with the task of the model being to detect cars in new, unseen images.

A fundamental algorithm used in supervised learning is gradient descent. Gradient descent works by repeatedly presenting an input from the dataset to the model and comparing its prediction with the sample label. The error, also called loss, is used to calculate the partial derivative of the loss with respect to each of the learnable parameters. These parameters are then gently adjusted in the opposite direction of the calculated gradient to reduce the loss. This process is repeated until a satisfactorily low error is achieved or until the training stagnates.

Formally, consider a dataset $\mathcal{D}$ consisting of input-label pairs $\mathcal{D}_i = (\mathbf{x}_i, \mathbf{y}_i)$, and a loss function $\mathcal{L}$ quantifying the error between $\mathbf{y}_i$ and the model prediction $\hat{\mathbf{y}}_i$. Iterating through the dataset, all learnable weights θ are updated according to the following rule:

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}(\theta \mid \mathbf{y}_i, \hat{\mathbf{y}}_i) \tag{2.1}$$

where η is the learning rate and ∇_θ is the gradient operator with respect to θ.

Imitation Learning

Imitation Learning (IL) is an approach to training autonomous agents in which agents are trained to mimic the behavior of an expert agent. The expert agent typically has access to privileged data from a simulator, such as ground truth positions of other cars, or traffic light states. Thus, it is well suited to determine the best course of action a when exposed to various states s, which enables the generation of a dataset consisting of state-action pairs (s, a). Within the IL regime, there are three main algorithms for teaching an agent the expert dataset, which we further explore in detail.

Behavioural Cloning (BC) is one such approach in which the state-action pairs are treated as inputs to a supervised learning algorithm. Given an expert policy π^* that one wishes to mimic, first generate a dataset $\mathcal{D}$ with N samples $\mathcal{D}_i = (s_i^*, a_i^*)$ from the expert policy distribution d^{π^*}. Then, using supervised learning, recover an approximate policy π_θ that minimizes the training loss $\mathcal{L}$:

$$\pi_\theta = \underset{\pi_{\hat{\theta}}}{\operatorname{argmin}} \sum_{i=1}^{N} \mathcal{L}\left(a_i^*, \pi_{\hat{\theta}}\left(s_i^*\right)\right)$$

BC greatly benefits from a large amount of research on supervised learning. However, there is a significant drawback to the method, the Distribution Shift problem [53]. In supervised learning, the pairs of input and output data are assumed to be independent and identically distributed (i.i.d.) according to the dataset distribution d^{π^*}. When the agent is exposed to another distribution and must predict actions for states not seen during training, it generally performs worse than on states from the dataset $\mathcal{D}$. This can cause the agent to choose actions that further deviate it from its known domain, which can result in cascading and catastrophic failure. Using a self-driving vehicle as an example, consider an expert agent who always manages to drive in the center of its lane. A BC agent trained to mimic this expert may, by chance or error, end up slightly away from the center, causing it to experience states never before seen during training. The BC agent is not trained to handle this state, which may cause it to drift further and further away from the center of the lane.

This can happen not only when the distribution is changed, but also when the i.i.d. assumption is broken. For the task of driving a vehicle, the current state is without doubt dependent on the previous states the agent has visited; thus, this assumption is broken when training self-driving agents, and this may cause problems.

To mitigate this, one method is to increase the variance of the training dataset. For instance, during data generation, instead of always letting the expert agent decide how the vehicle moves, random actions could be chosen with probability ϵ, creating new states. Another approach is to deliberately place the expert agent in states it would not have encountered on its own, such as perturbing its position or speed. These methods are useful, but limited in scope to the creativity of the engineer implementing the methods.

As an alternative to BC, Direct Policy Learning (DPL) includes the expert not only during dataset generation, but also at training time. The agent policy π_θ is trained as in BC using supervised learning; however, the policy is rolled out during training to generate additional states from the learning agent's distribution d^{π_θ}. The expert agent is queried at these new states to generate additional state-action pairs from this new distribution, thus increasing the variance of the dataset.

The requirement of having an interactive expert is often costly and not always feasible. For example, the cost of driving in the real world using a human expert is prohibitive on a large scale. However, in simulated environments, it is often possible due to the reduced costs and safety risks. An example of DPL is the DAgger algorithm [54], which, as described, continuously expands the training dataset by querying the expert agent for corrections to the choices of the imitating agent.

The last regime is the Inverse Reinforcement Learning (IRL) method. Instead of using supervised learning to teach the imitating agent, Inverse Reinforcement Learning (IRL) deduces a reward function from the dataset, and then trains the imitating agent using this reward function and Reinforcement Learning. We do not go into further detail on IRL in this thesis.

Reinforcement Learning

Reinforcement learning (RL) is an alternative method to supervised learning in which the agent is not directly presented with state-action pairs, but instead explores the state-action space to be rewarded for desirable choices. What is desirable is encoded in the reward function $R(s, s')$, which maps state transitions from s to s' to a numeric reward. Positive rewards reinforce actions that led to the state s', while negative rewards reduce the probability of choosing this action later. The reward function is commonly specified manually by the developer, but, as seen in IRL, can also itself be learnable. RL is widely used when developing autonomous agents,but is not further relevant to the TransFuser and InterFuser agents as used in this thesis.

2.4.2 Network Architectures

Nevertheless, a training algorithm is only as good as the model it trains. Model design is an essential part of achieving high performance ANNs, and to do so requires an understanding of both the various network architectures and the target domain for which one designs. ANNs targeting natural language processing might have other suitable architectures than those targeting financial models. These, again, might be different from architectures designed to process visual information.

Although the former focuses on text, speech, and time-series data, computer vision is the field of artificial intelligence that focuses on interpreting visual data from the physical world. It involves the development of systems that can automatically extract and analyze information from visual sources, such as images and videos. This allows computers to perform tasks such as recognizing objects, detecting faces, and understanding scenes in real time. In the context of Autonomous Driving (AD), computer vision plays a crucial role in enabling self-driving cars to perceive and understand their surroundings in order to navigate safely and efficiently.

Computer vision systems are typically trained using supervised learning on large datasets of labeled images and videos, enabling them to learn to identify and classify objects, scenes, and actions. In the case of AD, objects such as other vehicles, pedestrians, road signs, and traffic lights are particularly important and crucial to identify correctly to drive safely.

However, approaches differ according to the overall framework. While modular approaches to AD include modules to explicitly detect and track such objects, end-to-end approaches may only barely acknowledge that the objects exist at all. The former approach is therefore easier to inspect and understand from a human perspective, while the latter might have the benefit of improved performance due to being free from human-imposed restrictions, and is thus able to learn alternate, perhaps more efficient, representations of the environment. In any case, both approaches lean heavily on the Convolutional Neural Network (CNN) structure, which we will explore next. In addition, we take a look at the Transformer architecture, which has made great strides in image processing in recent years, being used in both TransFuser and InterFuser, and in a variety of other models.

Convolutional Neural Networks

CNNs are a specialized kind of ANN for processing data that have a known grid-like topology, which makes them suitable for image processing. The main difference from ANNs is that they replace matrix multiplication with the convolution operation in at least one of their layers [55]. In machine learning applications, we often work with the cross-correlation function in place of the normal convolution operation. Given a two-dimensional input image I and a two-dimensional kernel K, the output feature map S is given by:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n), \tag{2.2}$$

where the learnable parameters are located in K. In a two-dimensional context, we can say that the kernel slides over the input image to create the feature map.

The convolutional layers in CNNs are based on three fundamental ideas [55]:

Sparse interactions means that each neuron in a layer is only connected to a subset of the neurons in the previous layer, instead of being fully connected like in a traditional ANNs. The use of kernels allows CNNs to detect small and meaningful features, such as edges, when processing large images. It also reduces the number of parameters in the network, giving smaller memory requirements and improved efficiency.

Parameter sharing refers to using the same parameters (weights) across different spatial locations of the input, which in the case of CNNs refers to the kernel parameters. Using the same kernel at every position of the input greatly reduces the memory requirements compared to dense matrix multiplication.

Equivariant representations means that if the input changes, the output changes in the same way. Parameter sharing causes CNNs to be equivariant to translation. Therefore, the network can efficiently learn and recognize patterns and objects regardless of their location in the input image.

Note that CNNs can trivially be generalized to higher-dimensional inputs and kernels, such as RGB images with three channels, by extending the summation operation in Equation (2.2) to additional indexes. Convolutional layers are usually followed by a nonlinear activation function before the results are fed into a pooling layer. A pooling function replaces a value of the feature map with a summary statistic of nearby values. Two common pooling functions are max pooling, where the maximum value of the nearby outputs is kept, and average pooling, where the average value of the nearby outputs is calculated and kept. Not only do pooling layers reduce the size of the feature maps and therefore also the memory requirements, but they also help CNNs become approximately invariant to small translations of the input [55].

As a consequence of Equation (2.2), the number of pixels in the input feature map I affecting the value of a single output pixel $S(i, j)$ is given by the size of the convolutional kernel K. For larger ranges of m and n, more pixels from I affect the value $S(i, j)$. This area of influential input pixels is called the receptive field of $S(i, j)$, and is a limit to the range of positional relationships that can be processed by the convolutional layer. Patterns greater than the receptive field cannot be perceived. To solve this problem, the size of K can be increased. However, as the numbers of multiplications required increase quadratically with the width of the kernel, this is prohibitive for large kernel sizes. The more common approach is to consecutively stack multiple convolutional and pooling layers, which greatly increases the receptive fields of neurons in the deeper layers with respect to the network input, although the receptive field at each layer only increases a little.

Thus, a typical CNN consists of several convolutional and pooling layers in series that learn to extract complex and abstract features from the input data. The features are finally flattened and fed through one or more fully connected layers, which perform the final classification or regression task. Figure 2.7 shows the architecture of VGG [56] as an example.

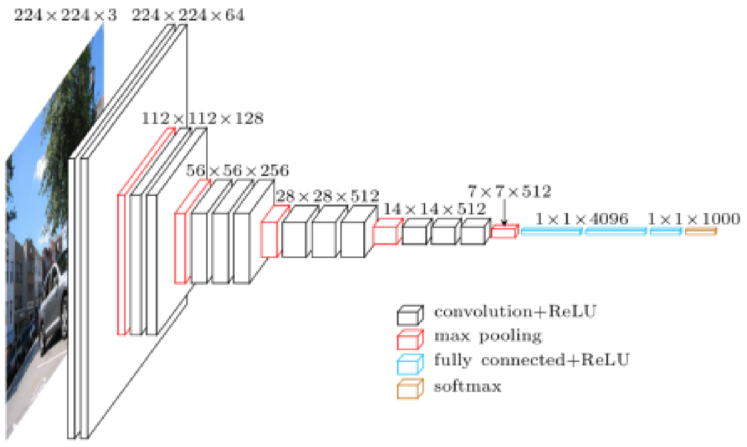


Figure 2.7: VGG [56] is built using a traditional CNN architecture and includes a combination of convolutional, pooling and fully connected layers. This architecture showed great results in both image localization and classification tasks at the time of release. Source: [57]

Transformers

The Transformer architecture, originally proposed by Vaswani *et al.* [58] in 2017, is mainly used for sequence-to-sequence natural language processing tasks such as machine translation, text generation, and document summarization. Previous approaches to these tasks involve Recurrent Neural Network (RNN) that sequentially process one word at a time, thus gradually building an internal state that represents the semantic content of the text. In stark contrast to this, the Transformer architecture simultaneously looks at all words in the text, enabling long-term relationships between words that are otherwise lost in RNNs due to the long temporal distances.

At the core of Transformer models, we find the self-attention mechanism. The idea of this mechanism is to calculate the relevance of one output element to all input elements. This allows the model to capture relationships between different positions in a sequence of data, for example, between words in a sentence.

Each word in the sentence can be turned into a fixed-size input vector using an embedding algorithm. In self-attention, each input vector is represented in three ways: as a query, a key, and a value. Every input vector is transformed through linear projections to create these representations: one transformation for queries, one for keys, and one for values. Each transformation contains learnable parameters. Once we have the query, key, and value representations of each input vector, we can pack them together into the matrices Q, K, and V.

To determine how much each input should be attended to by other inputs, we take the dot product between Q and K. A higher dot product indicates stronger relevance between the keys and the queries. For example, consider the sentence "The pizza was not eaten because it was too cold". When the model processes the word "it", the dot product would be larger between "it" and "pizza" than between "it" and "because", since "it" refers to the pizza. We then scale the dot product by $\frac{1}{\sqrt{d_k}}$, where d_k is the dimension of the input queries and keys, before applying the softmax function to the result. Scaling is done to avoid vanishing gradients after softmax when the dot product is large. Finally, we use the resulting attention scores to weight the value vectors. We have the following resulting formula:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V.$$

For RNNs, the ordering of the input was determined by the temporal order of processing. In Transformers, this implicit ordering is lost; thus, to maintain an ordering of the input, a positional encoding is used. A positional encoding is simply a set of optionally learnable parameters that are combined with the input tokens to indicate their position in the text. The Transformer learns to interpret this encoding to know the order of words in a sentence, and, through the attention mechanism, is able to attend to tokens close to each other more. At the same time, the model is also free to attend to tokens far away if the content of those tokens is relevant.

Transformer models are often built using multi-head attention. This means that the input vectors are projected into multiple sets of queries, keys, and values, where each set has separate learnable parameters. Each attention head can therefore attend to different parts in the input vector, making it possible to model, for example, how different words in a sentence relate to each other in different ways. Attention is calculated in all heads in parallel before the results are concatenated and linear transformed once more to create the final result. The original Transformer architecture is shown in Figure 2.8.

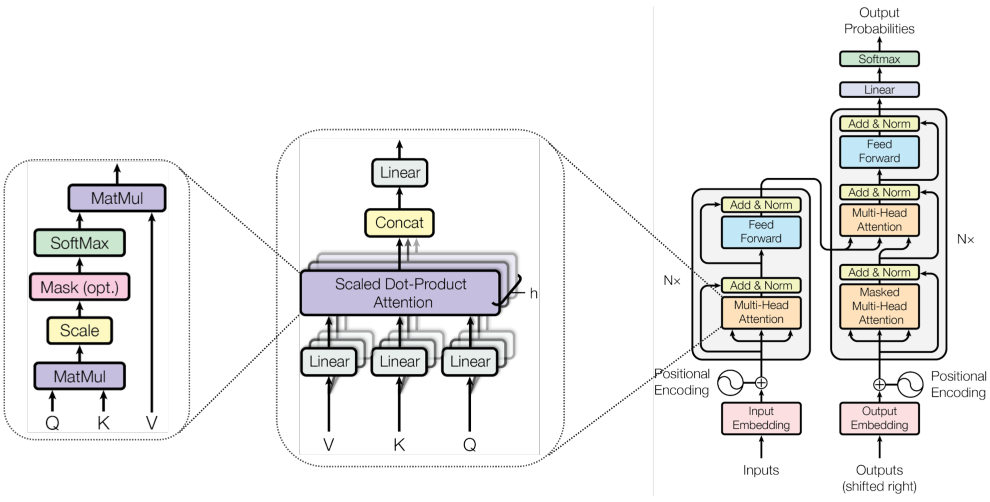


Figure 2.8: The original Transformer is built using an encoder-decoder architecture, where both parts include input embeddings, positional encoding and multi-head attention. In the figure we see how the multi-head attention block is built using multiple attention heads. This figure is a modified version of the figures found in [58]. This figures also shows use of masked attention, which is a variant of self-attention where certain positions of the input vector are ignored. This is often done to preserve causality in certain prediction tasks.

Vision Transformers

Although Transformers were initially designed with natural language processing in mind, their exemplary performance in models such as BERT [59] and GPT-3 [60] has also sparked interest in the field of computer vision, resulting in the development of Vision Transformers.

Similarly to how Transformers simultaneously processes a 1D sequence of words, Vision Transformers is capable of simultaneously processing a 2D grid of tokens, where each token could, for example, be a small part of a larger image. This allows the model to learn relationships over arbitrarily long distances in the input image, which was previously restricted by the receptive field of view in CNNs.

The self-attention mechanism also enables the processing of multiple sensor modalities, such as RGB images and LiDAR using similar processing blocks, and shows great scalability for large networks and datasets [61]. In the next section we will see multiple examples of models that utilize vision transformers to fuse multi-modal input.

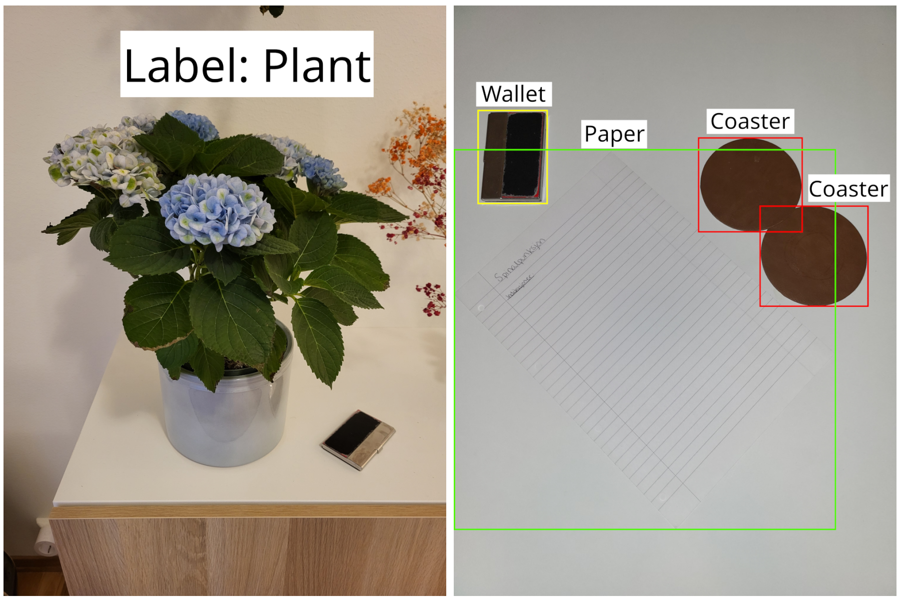
2.5 Computer Vision

Now, let us discuss Computer Vision (CV), the task of interpreting the content of a visual input. Due to the very high dimensionality of digital images, with millions of pixel values in a single image, this task has traditionally been very difficult and thus prevented AI agents from successfully interpreting their surroundings. However, the advent of CNNs and improvements to deep learning in general have revolutionized this field, enabling autonomous agents that even rely solely on cameras alone, for instance Tesla's Full Self-Driving system [27]. These advancements have also unlocked new tasks that were previously infeasible, and this section will briefly explain the differences in CV tasks.

Image classification is one of the most basic computer vision tasks. Given an image containing an object, the task is to correctly label the object with one of the pre-determined classes. The number of classes can vary greatly. The simple task of recognizing handwritten digits requires 10 classes, one for each digit. In contrast, thousands of classes may be required to distinguish common household items. An example of image classification can be shown in Figure 2.9a. As you can see in the figure, there are actually multiple objects present, and this might cause ambiguities in what the correct label should be. Generally, the most prominent object is chosen, but there are sometimes several of those as well. This brings us to the next task.

Object detection is the task of finding and classifying zero or more objects in an image. Instead of predicting one class for the whole image, multiple objects in the image may be labeled separately. It is common that the model also predicts the bounding boxes containing the objects, resulting in a joint optimization problem of both accurate and tight bounding boxes, accurate labels, and high sensitivity. The object detection task is illustrated in Figure 2.9b. Note how some bounding boxes overlap in the figure and that the rectangular shape does not fit all objects equally well.

Segmentation tasks are usually divided into three different techniques. **Semantic segmentation** further increases the fidelity of image classification and object detection tasks by predicting an individual class label per pixel in the input image. This solves the issue of overlapping bounding boxes and the mismatch between the bounding-box rectangles and the shapes of some objects. However, a new ambiguity arises, which is that multiple distinct objects appear as a single continuous object in the segmented image. The task



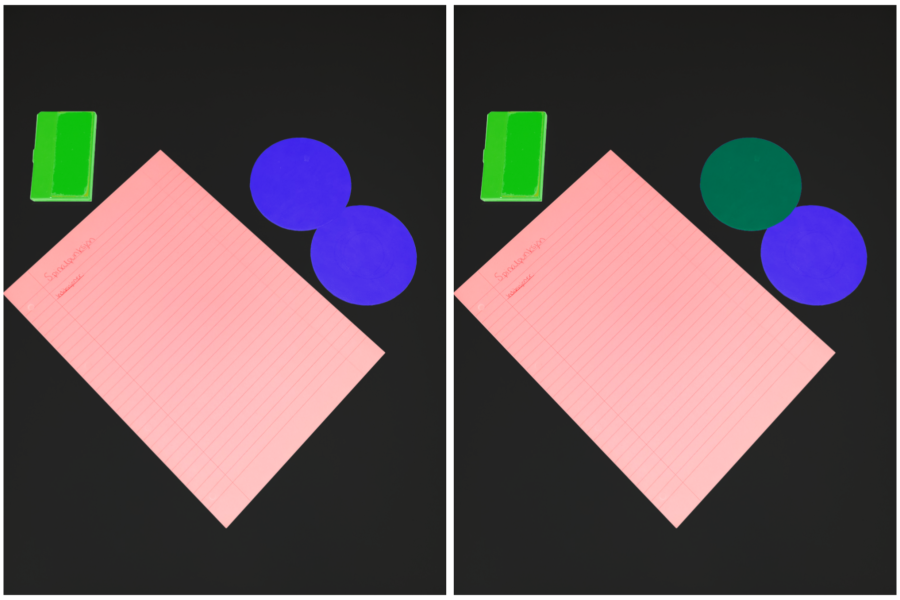
(a) Image classification. **(b)** Object detection.

Figure 2.9: Image classification and object detection. Given an image, the model should determine the correct label(s) potentially object positions.

of distinguishing between distinct objects is known as **Instance segmentation**. This technique assigns unique labels to each instance of an object. Finally, **Panoptic segmentation** is a combination of previous techniques and provides both class and instance labels. The difference between semantic and panoptic segmentation is illustrated in Figure 2.10.

Depth estimation is something humans are very good at, but we carry an unfair advantage called stereoscopic vision, which is not always available to virtual agents. However, depth estimation is still possible by looking at a single image, and an example of this is shown in Figure 2.11.

For autonomous driving, these tasks are critical to safely navigate through traffic and around obstacles. Object detection plays an essential part in modular AD pipelines, where the detected objects such as cars, pedestrians, and traffic signs are explicitly processed and used in planning algorithms. In end-to-end pipelines however, the CV tasks are not always explicitly required, and may sometimes be omitted entirely. Instead, these models usually have a perception backbone that uses input images to create a compact representation of the environment, where the end-to-end learning process ensures that relevant patterns are detected from the images. Still, we will see that their inclusion as auxiliary optimization targets can still improve agent performance. For this purpose, semantic segmentation is perhaps more commonly used, for instance in the TransFuser and InterFuser models, which we will have a closer look at later.



(a) Semantic segmentation. **(b)** Panoptic segmentation.

Figure 2.10: Semantic and panoptic segmentation. Label each pixel individually, and also mark distinct object instances.

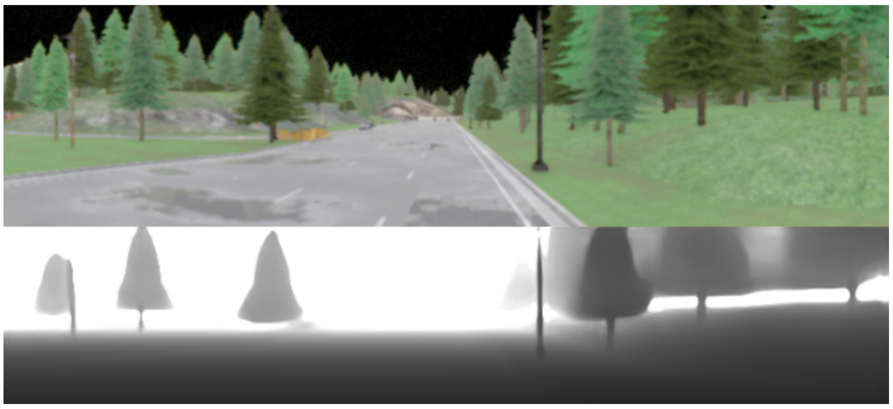


Figure 2.11: Depth estimation. Given an image, predict the distance to the object seen in each pixel.

2.6 Related Work

This section will explore recent work related to the content of this thesis. They are presented chronologically with respect to the publication date.

2.6.1 LAV (March 2022)

Chen and Krähenbühl [62] presents in their paper a system to train models for autonomous driving on data not only from the ego vehicle itself, but also other vehicles it observes, giving the name Learning from All Vehicles (LAV). They argue that training on other vehicle's trajectories will help with sample efficiency, greatly increase the chance that the model sees interesting scenarios, and help the ego-vehicle avoid collisions. Finding these trajectories are however challenging due to partial observability and the lack of sensors in other vehicles.

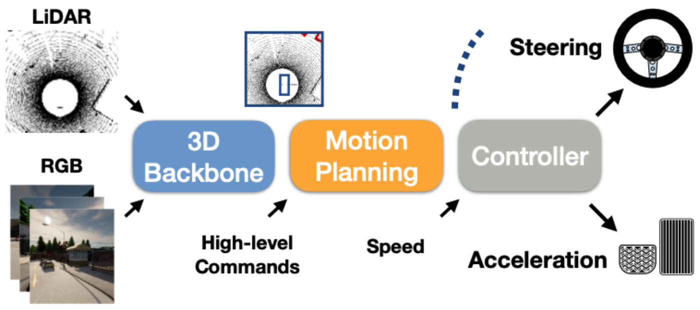


Figure 2.12: An overview of the Learning from All Vehicles inference architecture. It consists of three modules: A perception module, a motion planner and a low-level controller. Source: [62].

Their end-to-end model is based on an IRL approach and consists of a three-stage modular pipeline, as seen in Figure 2.12. The first stage is a perception module that maps raw sensor readings from three front-facing RGB cameras and LIDAR to map-view feature representations. Its goal is to create a robust and generalizable representation of the surrounding world and to build vehicle-invariant features that help supervise the motion planner. The motion planner is the second stage, and it uses these features to produce a series of waypoints describing the future trajectory of all vehicles that surround the ego-vehicle. The last stage is a low-level controller that converts motion plans into actions to be executed on the vehicle. Chen and Krähenbühl [62] reached the top of the CARLA Leaderboard with their submitted paper, and is currently ranked 3rd [62, 63].

2.6.2 TransFuser (May 2022)

Imitation with Transformer-Based Sensor Fusion for Autonomous Driving (TransFuser), ranked 2nd when published and currently ranked 4th on the CARLA Leaderboard, is an end-to-end BC approach to autonomous driving in the CARLA simulator [5, 63, 64]. In their paper, Chitta *et al.* [5] explores how complementary sensors should be integrated for autonomous driving. Their model uses both RGB images and LIDAR data and outputs position waypoints that are passed to a traditional PID-controller for steering. By interleaving Transformer-blocks within both the RGB and LIDAR feature extraction branches, features are fused between the two sensor modalities leading to exchange of information and supposed improved understanding of the environment. An overview of the architecture is shown in Figure 2.13.

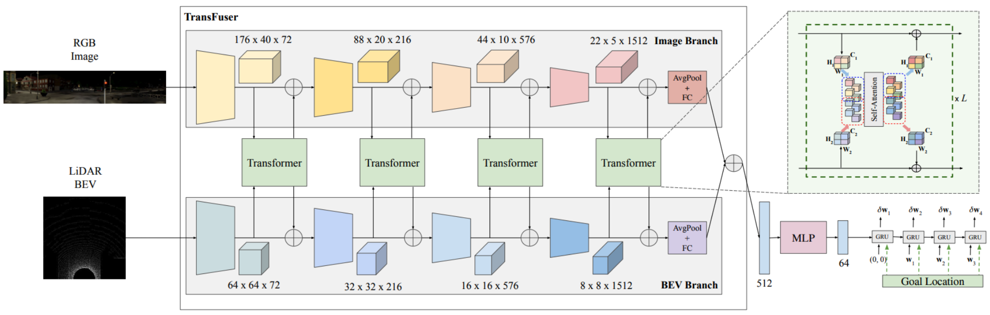


Figure 2.13: The TransFuser architecture uses several transformer modules for the fusion of intermediate feature maps between RGB and LIDAR data. The two branches are combined and fed into an MLP before passing it to an auto-regressive waypoint prediction network. Source: [5].

During training, TransFuser optimizes a multi-task loss consisting of not only predicting waypoints, but also predicting depth and semantic segmentation images, as well as a top-down HD-map and bounding boxes vehicle detection.

Chitta *et al.* [5] also propose a new benchmark for autonomous driving called Longest6. This involves longer routes, increased traffic density, and more challenging pre-crash scenarios than existing closed-loop driving benchmarks. Another advantage of this benchmark over the official CARLA Leaderboard is that it can be used for evaluation on local resources without the online platform's computation budget restrictions.

As this thesis is based on TransFuser, a more detailed description of the model and benchmark can be found in Section 3.5 and Section 3.10 respectively.

2.6.3 TCP (June 2022)

Wu *et al.* [65] introduces the idea of generating control signals by two simultaneous branches; one control branch that predicts future control signals directly, and one trajectory branch that predicts future waypoints, which are then converted to control signals using a PID controller. The architecture follows a BC approach, and is shown in Figure 2.14. They name their model Trajectory-guided Control Prediction (TCP), and argue that the two regimes of trajectory planning or direct actuation have different performance characteristics and are best suited for different situations. The predicted control signals from the two branches are mixed using a dynamic mixing factor α, which is chosen based on the current situation the vehicle is in. If the vehicle is detected to be turning, the mixing factor is set to 0 to solely use the directly predicted control signals. Otherwise, $\alpha = 0.5$ to evenly mix the control signals.

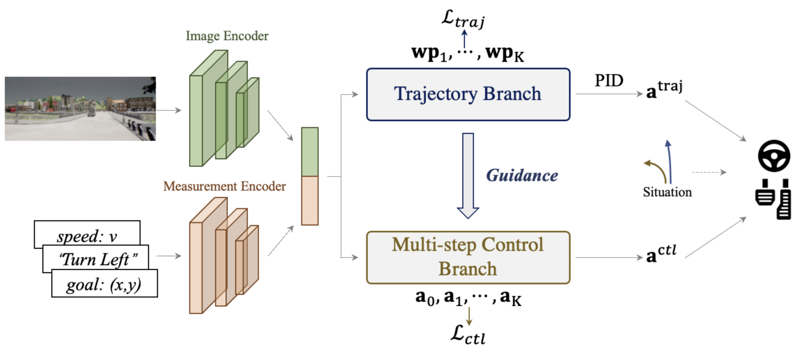


Figure 2.14: The TCP architecture is based on two distinct branches that share encoded image and navigation features. The trajectory branch provides per-step guidance for multi-step control prediction. A situation based fusion scheme generates vehicle actions based on the branch outputs. Source: [65].

Using only a single monocular camera as input and a ResNet34 backbone, TCP achieved first place on the CARLA Leaderboard with a Driving Score of 75% at the time of release.

2.6.4 InterFuser (July 2022)

Shao *et al.* [6] propose Safety-Enhanced Autonomous Driving Using Interpretable Sensor Fusion Transformer (InterFuser), a safety-enhanced autonomous driving framework based on BC. They investigate methods for multi-modal sensor fusion in a more scalable and effective way than TransFuser's multi-stage approach. They also work towards interpretability of their end-to-end model by creating a safety mind map based on immediate features, which they argue can help unveil the model's decision basis and failure causes.

The model follows an encoder-decoder architecture as seen in Figure 2.15. The input consists of three RGB cameras (left, front, and right) and LIDAR. From the RGB cameras, they create four images, one for each direction and one focusing-view image to capture distant traffic lights. The inputs are then fused in a multi-view multi-modal transformer before being fed into the decoder. The decoder takes queries in the form of waypoints, density maps, and a traffic rule to the attention mechanism and outputs predictions of the same attributes. Finally, a safety controller uses the predicted waypoints, object density map, and traffic rule to create a safety mind map, which is utilized to constrain the actions into a safe set. InterFuser is currently the second best performer on the CARLA Leaderboard [6, 63].

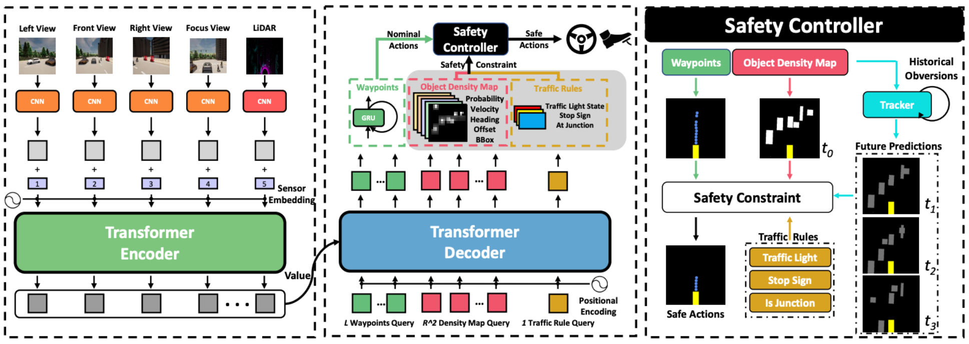


Figure 2.15: The InterFuser architecture. It consists of an encoder-decoder network built on transformers, in addition to a safety controller that constraints the output actions to enhance driving safety. Source: [6].

2.6.5 MILE (October 2022)

Hu *et al.* [66] present Model-Based Imitation Learning for Urban Driving (MILE), a BC approach for autonomous driving that jointly learns a model of the world and a driving policy. MILE operates on high-resolution images and leverages 3D geometry for better understanding of the environment. The 3D features are converted to a bird's-eye view (BEV). It is trained using only expert driving data and does not require interaction with an online environment or access to a reward. The authors claim that the model sets a new state-of-the-art in the CARLA simulator, even surpassing approaches requiring LiDAR input. MILE can predict diverse and plausible future states and actions, and can execute complex driving maneuvers based on plans "predicted from imagination". The model architecture is shown in Figure 2.16.

2.6.6 UniAD (December 2022)

Hu *et al.* [67] propose UniAD, a Unified Autonomous Driving algorithm that incorporates five essential tasks for a safe and robust autonomous driving system. The tasks are defined as tracking, mapping, motion, occupancy, and planning. It follows a planning-oriented philosophy and uses a query-based design to connect all nodes, allowing flexible intermediate representations and knowledge exchange between multiple tasks. The authors claim that UniAD outperforms previous state-of-the-art models in all aspects and hope that it can serve as a starting point for coordinating various driving tasks in autonomous systems.

UniAD consists of four transformer decoder-based perception and prediction modules and one planner in the end (see Figure 2.17). A sequence of images is fed into a feature extractor, and the resulting features are transformed into a unified BEV feature. The *TrackFormer* module learns from the BEV feature to detect and track agents. The *MapFormer* performs panoptic segmentation of the map. Interactions among agents and the map are captured and used by the *MotionFormer* to

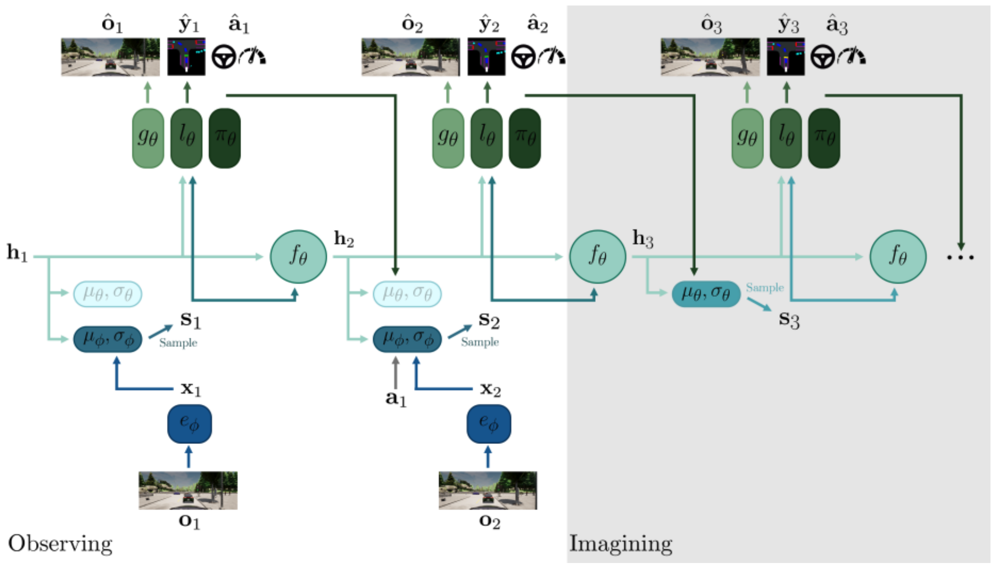


Figure 2.16: A diagram showcasing the MILE architecture observing inputs for $T = 2$ time steps, and then imagining future latent states and actions for one step. It consists of an inference model and a probabilistic generative model which together have the goal of infering the latent dynamics that generated observations, expert actions and BEV labels. The architecture is described in detail in [66].

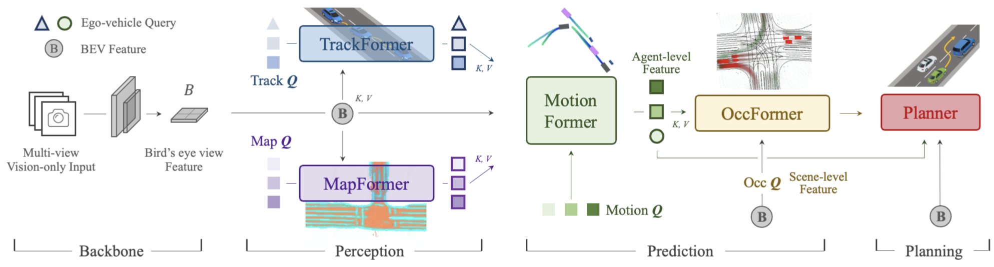


Figure 2.17: The UniAD pipeline. It consists of four modules: the feature extracting backbone, the perception module, the prediction module and the planning module. Source: [67].

forecast future trajectories. The *OccFormer* predicts future occupancy for agents. Finally, the *Planner* predicts a plan based on results from the two prior modules.

2.6.7 VAD (March 2023)

Jiang *et al.* [68] present the end-to-end driving framework Vectorized Autonomous Driving (VAD). In contrast to the rasterized scene representations previous works use for planning, such as occupancy and semantic maps, VAD models the scene's agents and map with vectors. They argue that this representation is more efficient in terms of computation and that it provides more accurate information on road structure, traffic movement, and the future agent trajectory. Vectorized information is used by the autonomous agent for both implicit and explicit planning. VAD adopts map queries and agent queries to implicitly learn map features and agent motion features from sensor data, giving planning information via query interaction. The model also uses three planning constraints based on the explicit vectorized scene representation: an agent collision constraint, a boundary overstepping constraint, and a lane direction constraint. Figure 2.18 shows the complete model architecture. The authors claim that the vectorized representation and constraints together effectively improve planning performance without being computationally expensive, and the model outperforms UniAD by reducing average planning displacements and collision rates while running 2.5 times faster.

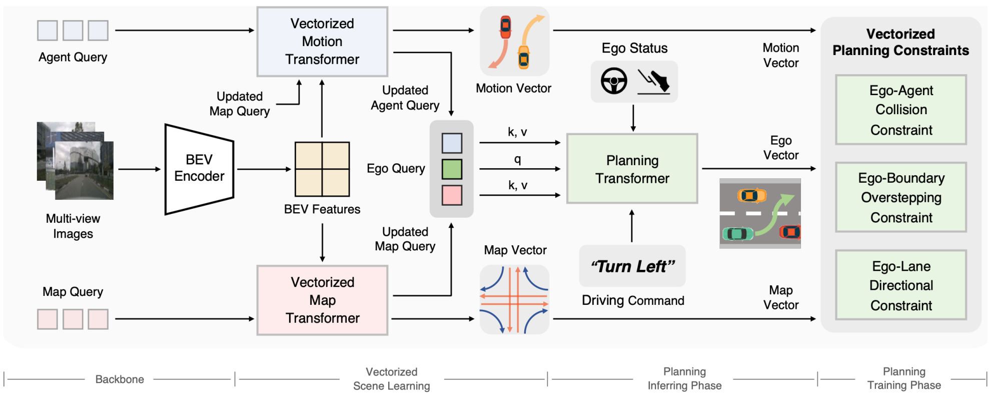


Figure 2.18: The VAD architecture is divided into four parts. The backbone projects image features to BEV features. During Vectorized Scene Learning the model encodes scene information into agent and map queries, and outputs the scene with motion and map vectors. During the Planning Inferring Phase the model extracts map and agent information via query interactions and outputs the planned trajectory. Finally, the Planning Training Phase regularizes the planned trajectory using three different constraints. Source: [68].

2.6.8 ReasonNet (May 2023)

Shao *et al.* [69], the authors of InterFuser, present ReasonNet, a BC end-to-end framework that takes advantage of both temporal and global information about the driving scene. They focus on improving driving performance in dense urban traffic scenarios, where many dynamic objects, road structures, and road user interactions are involved, and argue that high-fidelity future predictions require temporal reasoning over the historical information of the scene. They therefore propose a temporal reasoning module which fuses information from different frames for better driving performance. In addition, they claim that the autonomous vehicle must make use of global reasoning to have a good perception of the scene, especially when dealing with occluded objects. By understanding global information about the scene, such as road geometry and driving interaction patterns, the agent can anticipate potential dangerous events. The authors propose a transformer-based global reasoning module for fusing information from the environment and its objects, and analyze their interactions for better scene understanding. This also provides benefits such as better identification of traffic light status and more accurate future trajectory forecasting based on interactions between other vehicles. The ReasonNet architecture is shown in Figure 2.19. The model is currently the top performer on the CARLA Leaderboard, beating Interfuser with an almost 4% better driving score. The authors also provide a new benchmark called Driving in Occlusion Simulation (DOS). It consists of various occlusion scenarios in urban scenes and is designed to serve as a comprehensive occlusion event evaluation benchmark.

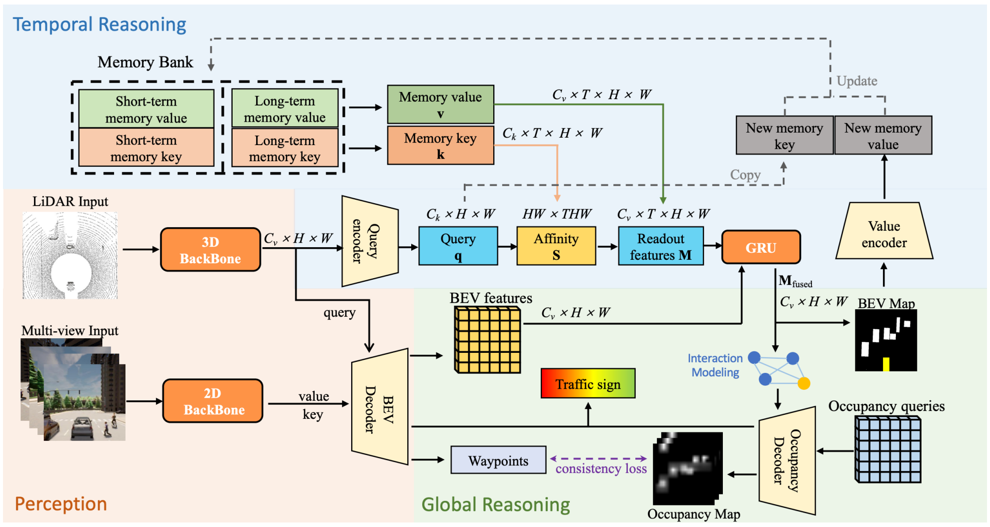


Figure 2.19: ReasonNet consists of three modules. The first is a perception module that fuses multi-view RGB images and LIDAR data to generate BEV, traffic sign, and waypoint features. Next is a temporal reasoning module which processes current and past features while also storing past features in a memory bank. The last module employs global reasoning by modelling the interaction between objects and the environment to detect adverse events and improve perception performance. Source: [69].

Chapter 3

Methodology

This chapter describes in detail the methods used to answer the research questions of the thesis. It begins with an overview of our tools and resources in Section 3.1, while Section 3.2 describes our development environment. Section 3.3 shows how a digital model of NAPLab's Kia e-Niro was created and imported to CARLA, and Section 3.4 shows the camera calibration process for the real-world car. Sections 3.5 and 3.6 give a deeper understanding of the autonomous driving models used in this thesis, and Section 3.7 explains how they were upgraded to work with the latest CARLA version. Finally, the data generation, training, and evaluation pipelines are presented in Sections 3.8 to 3.11.

3.1 Tools and Resources

Training vision transformers is a computationally heavy task. To fulfill the computational demands during our research, we have utilized three main computational resources, which will be described in the following sections. To develop on these resources, we mainly used the free and open-source code editor Visual Studio Code (VS Code) [70]. VS Code has a large ecosystem of extensions enabling support for debugging, smart completions, and more in most programming languages. This includes an extension for connecting to remote machines in the editor using secure shell (SSH) [71], a tool that we use extensively when working with the project. Note that all of the resources below require a connection through NTNU's campus network or the NTNU VPN. Finally, to synchronize work among the researchers and to manage code versioning, we utilize a GitHub organization that is intended to store all relevant project code.[1] This also made it easy to synchronize code changes between the different resources we used when developing. Our development environment is further described in Section 3.2.

[1]https://github.com/orgs/aasewold

3.1.1 IDI Horizon

For initial development and small-scale testing, we used virtual machines from IDI Horizon - Virtual Desktop Infrastructure for Visual Computing. These are equipped with NVIDIA A10-24Q GPUs with CUDA version 11.4. Access to a desktop graphical interface was available through VMWare Horizon [72], however, the latency through this solution was often quite high, thus our primary workflow consisted of SSH based access both through a terminal emulator and with VS Code. We also experienced lower compute and GPU performance when running workflows on these computers than other options, which has led to these resources mainly being left idle.

3.1.2 `nap02`

Another computing resource we got access to early in the project is the `nap02`-server, a bare-bones computer equipped with 2x Intel Xeon Gold 6342 CPUs, 2x NVIDIA A100 80G GPUs with CUDA version 11.6, 512 GiB RAM, and 14 TiB SSD-backed storage. This server is shared between members of NAPLab, and used for various AI research in addition to autonomous driving. The impressive hardware specifications have proven to be very useful in our research, making `nap02` the main development platform.

3.1.3 Idun

Some workloads however have proven too great even for `nap02`, in which case the Idun High Performance Computing cluster has been used. Idun provides 864 CPU cores and 80 NVIDIA GPUs to the Department of Computer Science [73], and thus enables a much higher degree of parallelization than what is possible on `nap02`. The cluster is managed using the Slurm Workload Manager [74], which coordinates and schedules jobs that are queued by all users of Idun.

3.1.4 `vcxr12`

The final computing resource that we have utilized is a brand new desktop computer called `vcxr12`. This computer is equipped with a 24 core Intel i9-13900KF CPU, a NVIDIA GeForce RTX 4090 24G GPU and 32 GiB RAM. It runs Ubuntu 20.04 with CUDA version 12. The main advantage of this machine was to have a physically available desktop not shared with others, such that we were free to manage software on the computer and have access to a low-latency graphical interface. This was necessary to be able to build CARLA from source, model a new vehicle in 3D in Blender, and work with the Unreal Editor (see Section 3.3). These tasks were not possible on `nap02` or Idun due to those environments being headless (offering no graphical environment), and difficult to perform on the IDI Horizon virtual machine due to the high GUI latency. The increased rasterization performance of the RTX 4090 compared to the A100 GPUs also significantly increase

CARLA performance, and thus improve all workflows involving CARLA.

3.2 Development Environment

With multiple computational resources available, care must be taken during development to be able to fully utilize the resources. This involves ensuring that the code is compatible with the different environments, and requires proper workflows to prevent accidental data loss or confusion due to code being scattered across platforms. To combat these problems, we synchronize all code through the GitHub code-sharing platform, using a layout that separates infrastructure from implementation. We use both Docker and Singularity containers depending on availability to ensure a consistent runtime environment, reducing the need for complex dependency setup steps. This section describes these approaches in more detail.

3.2.1 Code Organization

All code produced in this work is available in the following six repositories on GitHub:[2]

experiments contains infrastructure and tooling to easily run experiments independent of host environment. To manage this, a set of Dockerfiles and necessary scripts manage the runtime environment of the various experiments performed.

transfuser contains the TransFuser agent with CARLA 0.9.14 compatibility fixes and support for evaluation using Kia e-Niro data.

interfuser contains the InterFuser agent with CARLA 0.9.14 compatibility fixes and support for evaluation using Kia e-Niro data.

carla contains a fork of the CARLA simulator with some small patches that have been submitted for inclusion upstream.

carlo contains the Carlo client used for simple experiments in the CARLA simulator.

carla-python-stubs contains Python type stub files for type annotations of the CARLA client-side Python API.

Particularly important are the first three repositories, which contain the major developments in this thesis. While the `transfuser` and `interfuser` repositories are mostly similar to the original agents, the creation and usage of the `experiments` repository brings several benefits, such as:

[2] `https://github.com/orgs/aasewold`

- Simple usage accelerates development. By writing a *run.sh* script once, we do not need to remember complex order of commands to run experiments.
- Works on multiple machines. Reuse the run-script on other machines without difficulties.
- Reproducibility. Ability to ensure running the same environment as last time. Detailed logs are also stored.
- Organization. Folder structure and code which are simple to reason about.
- Decoupling from the model. The code is mostly reused between different models and CARLA versions, which makes it easy to evaluate different models. Compare this to having the evaluation in-line in the model repositories, which would have made the evaluation scripts scattered between TransFuser and InterFuser and made it hard to keep track of bug fixes across models and CARLA versions.
- Everything you need in one place. No need to setup different folders for different experiments. One can just clone the experiment repository and run the scripts to reproduce the results in the thesis.

The `experiments`-repository contains several folders, each dedicated to its own purpose. Folders that produce evaluation results are prefixed with `ex-` to indicate their status as an experiment. Note, however, that the experiments are further parameterized by the choice of model weights and the selected benchmark; thus, there is no one-to-one correspondence between experiments as presented in Chapter 4 and these folders. The following is a description of each folder's purpose and workings.

`common/` Dockerfiles for running CARLA and TransFuser/InterFuser, shell utilities and scripts to download pre-trained weights, all of which are utilized in our experiments. Not intended to be used directly.

`datagen-transfuser/` Generate training data using TransFuser's expert agent. Parameterized by which CARLA and TransFuser version to use. Routes must first be prepared locally using the scripts in `routegen/`, after which data generation jobs can be submitted to Idun using the tooling in `idun/`. Data generation is currently not implemented for local hosts. See further details in Section 3.8.2.

`datagen-interfuser/` Generate training data using InterFuser's expert agent. Similarly to the above, however, data generation is also possible using local hosts. See additional details in Section 3.8.3.

`train-transfuser/` Train a TransFuser model on a given dataset. Run using the script `run.sh`, which is parameterized by which TransFuser version to use and the dataset path. Additional details are described in Section 3.9.1.

`train-interfuser/` Train a InterFuser model on a given dataset. Similar to the above, with additional details in Section 3.9.2.

`ex-transfuser-0.9.10/` Evaluate a TransFuser model on the Longest6, 42routes, or Town05 benchmark with CARLA version 0.9.10. Run using `run.sh`, which is parameterized by which benchmark to perform and the model weights to use. Additional details are found in Section 3.10.

`ex-transfuser-0.9.14/` Identical to the above, except using CARLA version 0.9.14. Also parameterized by which sensor configuration to use (original or Kia-like).

`ex-interfuser-0.9.10/` Evaluate an InterFuser model on the Longest6, 42routes, or Town05 benchmark with CARLA version 0.9.10. Run using `run.sh`, which is parameterized by which benchmark to perform and the model weights to use. Additional details are found in Section 3.10.

`ex-interfuser-0.9.14/` Identical to the above, except using CARLA version 0.9.14. Also parameterized by which sensor configuration to use (original or Kia-like).

`ex-kia/` Evaluate TransFuser and InterFuser on real-world data collected using NAPLab's Kia e-Niro. To do so, first create a plan using the tooling in `make-plan/`, and then perform inference using the tooling in `eval/`. Additional details can be found in Section 3.11.

`ex-transfuser-leaderboard/` Submit and evaluate a TransFuser model on the official CARLA Leaderboard.

Later in this work we will refer to specific files and folders in the `experiments` repository using the notation $^{\text{ex}}$`<path>`, for instance $^{\text{ex}}$`ex-kia/eval`. The notation also functions as a clickable hyperlink to the corresponding path on GitHub.

3.2.2 Containerization and Idun

To run CARLA, TransFuser, and InterFuser in a reproducible and stable environment, we defined Docker containers [75] that contain all required dependencies. The container dependencies can then be run isolated from the host OS. We also utilize Docker Compose [76] to run multiple containers at once from a single configuration file. To simplify the management of multiple training and evaluation runs, we use GNU Screen [77], which allows creating multiple virtual terminals independent from each other. Once detached from a new `screen`-session the process will continue in the background, even when the SSH session is disconnected. The `screen`-session can then be resumed when reconnecting through SSH.

The containerization approach enables reliable reproducibility as well as robust deployment regardless of the target environment. Our main motivation for this approach is the simplified deployment required to generate data, train, and evaluate using TransFuser and InterFuser. All configuration files used with Docker and Docker Compose are stored in the experiments repository, for the most part in $^{\text{ex}}$`common`.

However, the Docker container runtime is not well-suited for all environments. Specifically, for HPC clusters such as Idun, the Singularity container runtime [78] is often preferred [79]. Singularity provides several benefits for HPC administrators, such as preventing users from requiring root privileges, and its immutable container images that simplify concurrent access from multiple compute nodes.

Therefore, we have developed methods to run the CARLA simulator and clients on Idun using Singularity. This is done by converting our existing Docker images to Singularity images using the `singularity build` tool. These Singularity images can be run on Idun both manually and in batch jobs using the Slurm workload manager [74]. The executions of these can be configured using Slurm job scripts, a variant of normal shell scripts that contain Slurm options at the start of the file. Some options that can be defined are job name, time allocated to the job, and, perhaps most importantly, the required compute resources. This includes the number of CPU cores, amount of memory, number of GPUs, and constraints on which GPU models to use. Finally, multiple jobs can be run concurrently from the same Slurm job script using Slurm job arrays. See Idun's documentation for a more in-depth overview of the available options.[3]

When the job is submitted, it is queued in the cluster and executed once the requested compute resources are available. One can easily check the status of the submitted jobs using the `squeue -u <username>` command. Note that Idun has set per-user limits on both size of usable disk space and number of stored files. The current usage of the quota can be checked with the command `lfs quota -uh <username> /cluster`. The default limits per user are at the time of writing 1 TiB and 2 million files. At our request, our disk usage limit was raised to 10 TiB for both authors for the duration of this thesis, due to the large datasets involved in this work.

3.2.3 Creating Python Stubs for CARLA

Although CARLA provides an extensive Python interface for its client-server functionality, it lacks proper typing of classes and methods. This means that we lose the possibility of auto-completion, documentation, and type checking in the code editor while developing, features that normally speed up the process and help prevent unwanted mistakes. An issue requesting this was raised in 2019 in the CARLA repository, but has not been resolved to date [80]. Therefore, we created a new tool for automatically generating Python stubs files based on the CARLA Python API docs [36]. The stub files contain type hints to be used by the type checker during development, and are not used during runtime. Currently we have only generated stub files for CARLA version 0.9.14, but they can also be generated for other versions. The generator source code, stub files, and installation instructions are available at our GitHub repository.[4]

[3] `https://www.hpc.ntnu.no/idun/getting-started-on-idun/running-jobs/`
[4] `https://github.com/aasewold/carla-python-stubs`

3.3 Creating a Digital Version of NAPLab's Car

To help reduce the gap between simulation and reality, we created and imported a digital model of NAPLab's Kia e-Niro to the CARLA simulator. In order to do this we first had to build CARLA from source to get access to the Unreal Editor, since this is not included in the packaged versions of the simulator. We follow CARLA's documentation to build CARLA from source on Ubuntu 20.04.[5] Since this requires a computer with a display and an adequate GPU, we used the new `vcxr12` computer equipped with a NVIDIA RTX 4090 GPU.

Once the build process was completed and we had access to the Unreal Editor, we could follow the guide on content authoring to start creating our custom vehicle. Note that there exist two guides describing this process that differ slightly in their instructions ([6] and [7]). However, before doing any work in Unreal Editor, we had to model our vehicle in Blender [81].

3.3.1 Modelling in Blender

We were provided a 3D model (SDL file) of the Kia e-Niro that NAPLab uses from our supervisors. We imported this into Blender as seen in Figure 3.1.

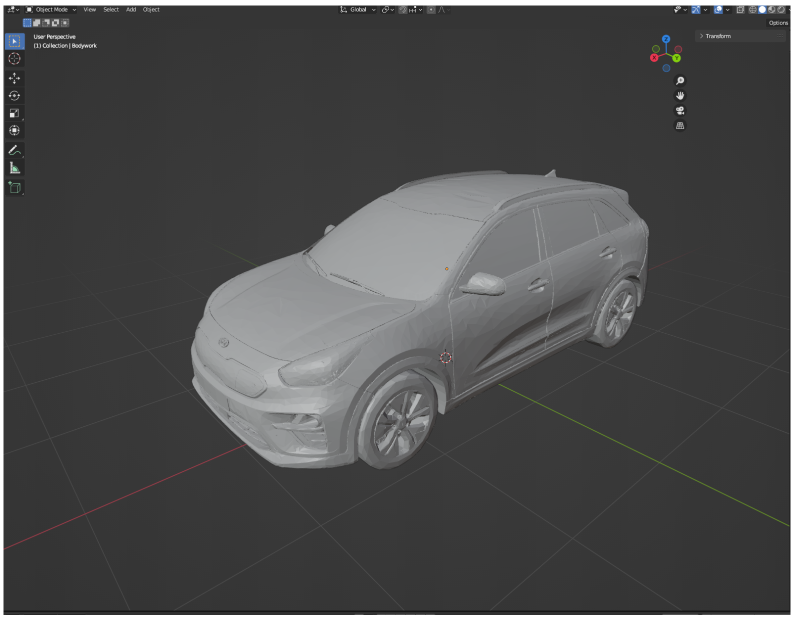
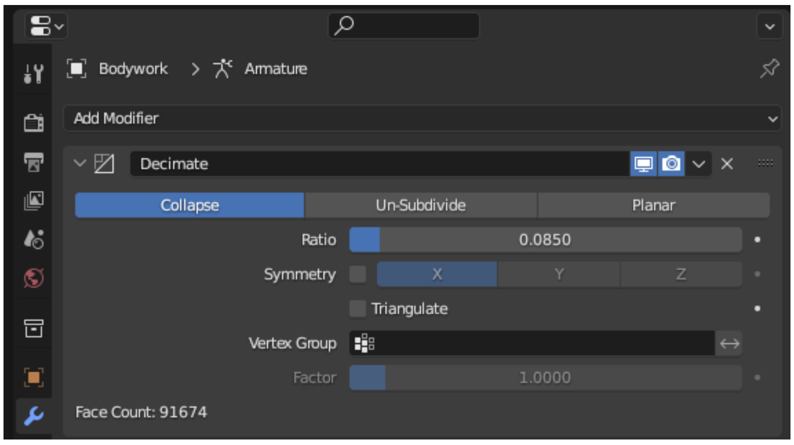


Figure 3.1: Kia SDL file imported into Blender.

As recommended by CARLA we reduced the face count of the vehicle from more than 1,000,000 down to around 100,000. This was done using Blender's decimate

[5]`https://carla.readthedocs.io/en/0.9.14/build_linux/`
[6]`https://carla.readthedocs.io/en/0.9.14/tuto_content_authoring_vehicles/`
[7]`https://carla.readthedocs.io/en/0.9.14/tuto_A_add_vehicle/`

tool, as seen in Figure 3.2.


Bodywork > Armature

Add Modifier

Decimate

Collapse | Un-Subdivide | Planar

Ratio 0.0850

Symmetry X Y Z

Triangulate

Vertex Group

Factor 1.0000

Face Count: 91674

Figure 3.2: The decimate settings used to reduce the face count of the vehicle.

CARLA requires that the wheels are separated from the main vehicle body. Our model consisted of one mesh, so we used Blender's tool for mesh selection and separation to create a standalone mesh for each wheel. When this was done we could move the wheels freely as seen in Figure 3.3

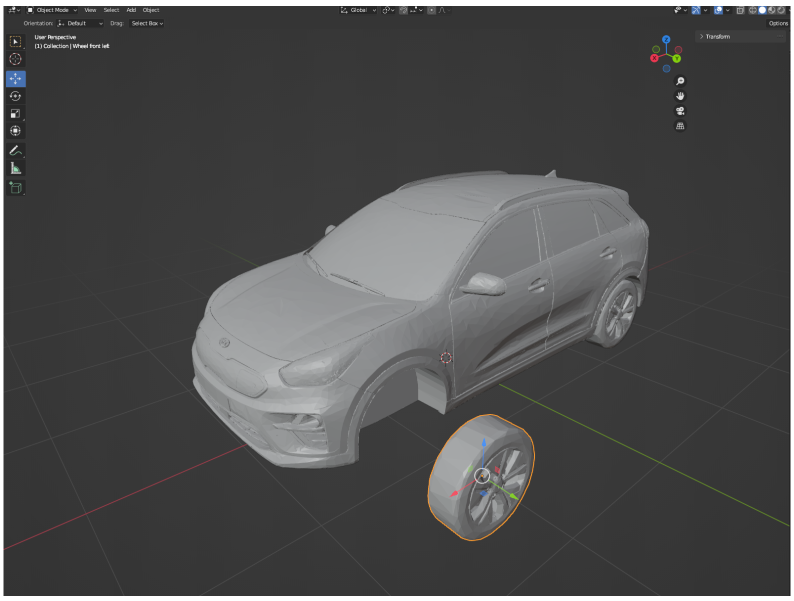


Figure 3.3: The wheels could be moved independently after being separated from the main body.

The last step in Blender was to rig an armature to our vehicle to identify the wheels and allow their movement. The armature consists of one main bone at the center of the vehicle and one connected bone per wheel, shown in Figure 3.4. The four wheel bones were placed in the center of each wheel and allowed each wheel to rotate freely, while the main bone moves the entire vehicle.

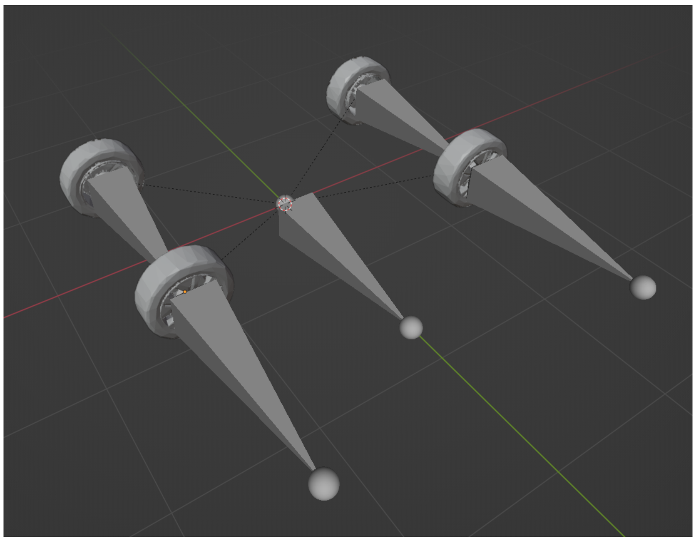


Figure 3.4: The armature layout of the vehicle. There is one for each wheel and one for the vehicle as a whole.

The finished model was exported from Blender to FBX format for import into Unreal Editor.

3.3.2 Importing into Unreal Editor

We could now open the Unreal Editor. In Figure 3.5 we see a preview of the simulated world and the content browser at the bottom.

After importing the Kia FBX into Unreal Editor, we adjusted the physics asset to fit the car model as seen in Figure 3.6.

We then created an animation and blueprints for both the wheels and the car as a whole as specified in the documentation. To add lights to our vehicle, we copied the front headlight objects from the Tesla Model 3 blueprint to our blueprint as seen in Figure 3.7. Figure 3.8 shows the effect of adding headlights.

The final list of assets is shown in Figure 3.9. We can preview the vehicle in the simulator by dragging the car blueprint into the simulator window and pressing play, as shown in Figure 3.10. Note that we also added a white car texture available in the simulator to the vehicle. This closely matches the material of the real world car, and were good enough for our needs since we only see the vehicle from the perspective of the mounted cameras.

This concludes the process of adding our custom vehicle to the CARLA simulator. We have uploaded a custom CARLA docker image that contains the new car for

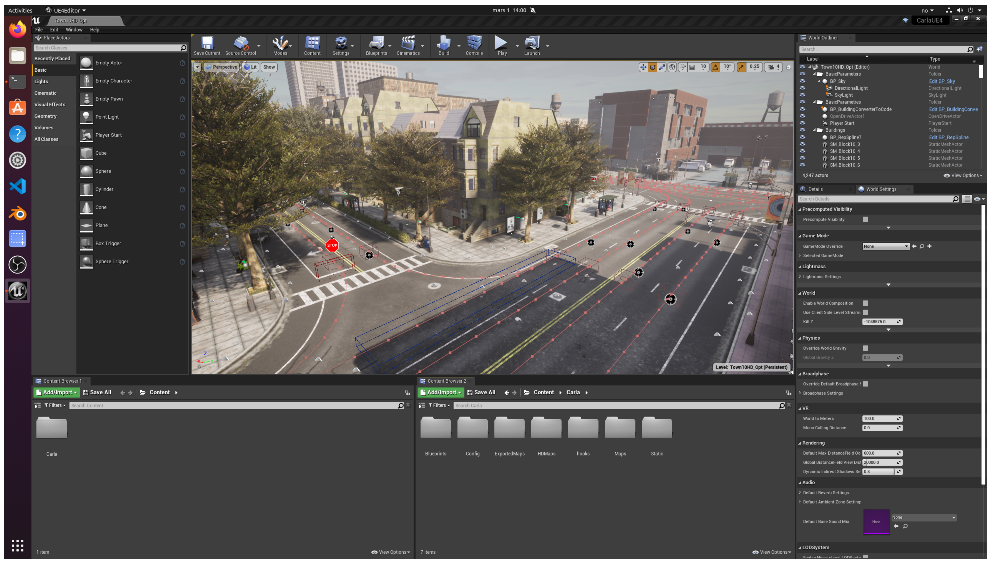


Figure 3.5: Custom version of Unreal Editor used for CARLA. The image showcases the simulator preview, content browser and properties panel.

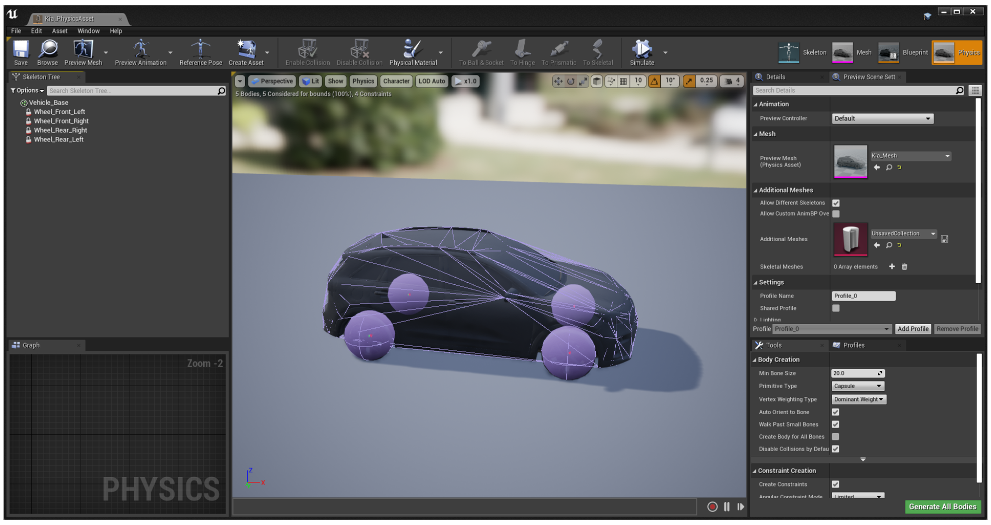


Figure 3.6: The Kia physics asset. It showcases the collision meshes for the wheels and main body.

others to use. It was packaged using the commands found in the CARLA documentation[8]. The docker image can be downloaded by running `docker pull mathiaswold/carla:0.9.14`, and the blueprint name is '`vehicle.kia.e-niro`'.

[8]`https://carla.readthedocs.io/en/0.9.14/tuto_A_create_standalone/`

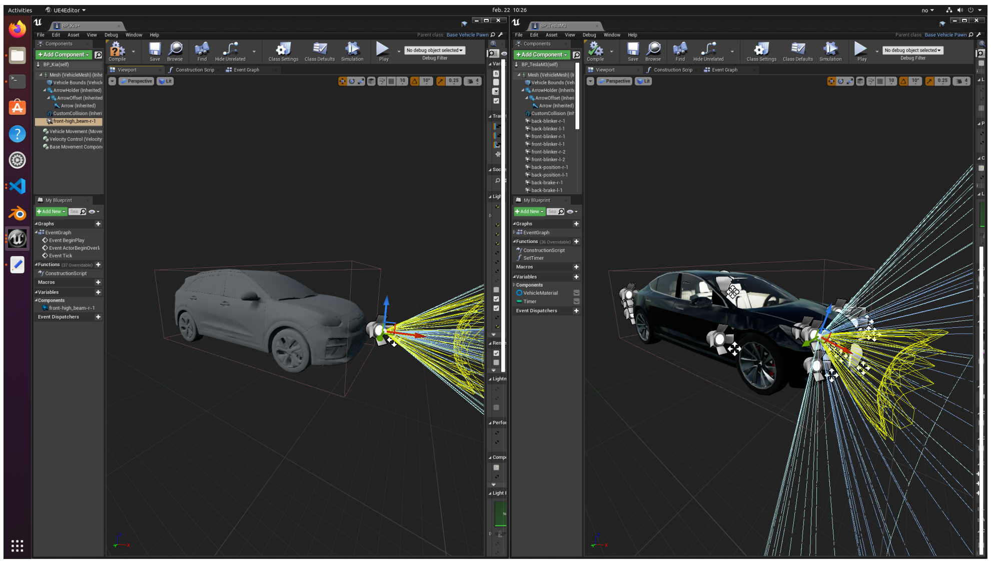
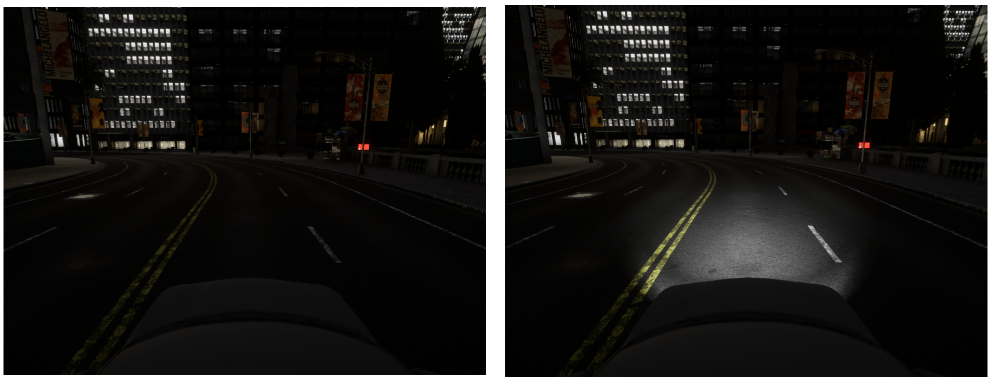


Figure 3.7: To add lights to the new vehicle, we copied existing light objects from the Tesla Model 3 blueprint to the Kia blueprint.

3.4 Calibrating and Using the NAPLab Car's Sensors

As mentioned in Section 2.3.1, NAPLab's Kia e-Niro is equipped with a computer running the NVIDIA DriveWorks SDK, which is responsible for interfacing with the vehicle's sensors. DriveWorks considers the car as a rig with several rigidly attached sensors [82]. Therefore, it is important to know the exact positions and orientations of the sensors on the rig in order to receive accurate data. These properties can be estimated through a calibration process.

As we already had the LIDAR sensor positions, we only needed to calibrate the RGB camera sensors. Doing this would ensure accurate mapping of objects per-



(a) Lights off. **(b)** Lights on.

Figure 3.8: Comparing before and after turning the vehicle lights on.

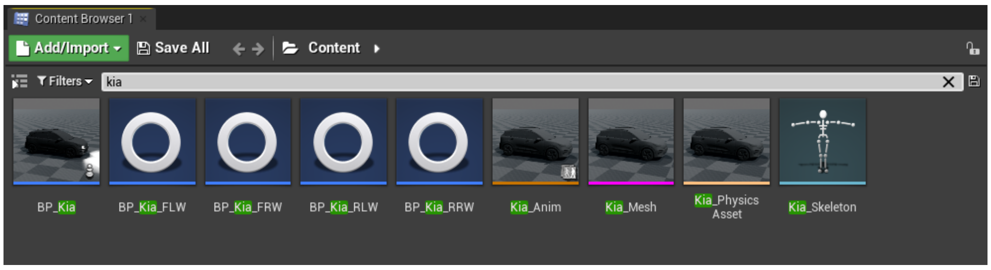


Figure 3.9: Final list of assets added for the new vehicle.

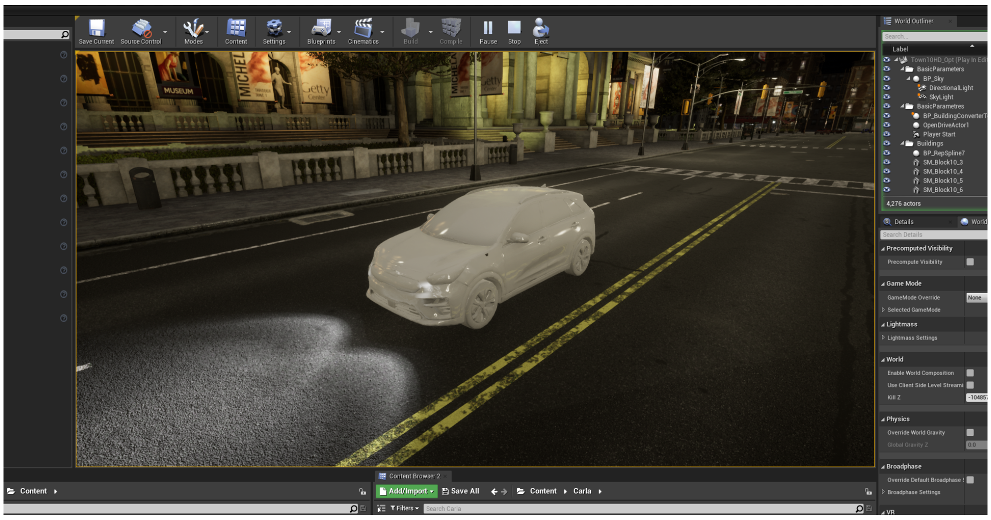


Figure 3.10: Preview of the new vehicle in the simulator.

ceived with an individual camera to other camera frames or the real world. Using NVIDIA DriveWorks' calibration tools [83], we performed calibration of the cameras with respect to the car while stationary. This is known as static calibration, and the calibration parameters include the intrinsic model and the extrinsic orientation and position. The intrinsic model calculates the geometrical relationship between pixels and optical rays for a camera, whereas the extrinsic model calculates the orientation and position in relation to the car. The two models are shown in Figure 3.11.

Figures 3.12 and 3.13 show and describe the calibration process in more detail. While intrinsic data was captured by the mounted cameras only, extrinsic data had to be captured both from the mounted cameras and an external camera.

3.4.1 Using the Sensors in CARLA

The calibration tool generates a rig configuration file when finished. This is a JSON file containing two main parts: one that describes the properties of all the sensors on the vehicle and one that describes properties of the vehicle itself. For

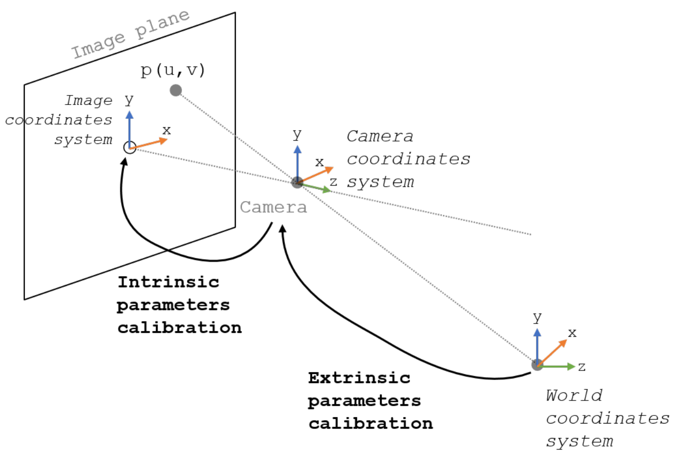


Image plane
p(u,v)
Image coordinates system
y
x
y
x
z
Camera coordinates system
Camera
Intrinsic parameters calibration
Extrinsic parameters calibration
y
x
z
World coordinates system

Figure 3.11: Intrinsic vs. extrinsic parameters calibration. Source: [84].

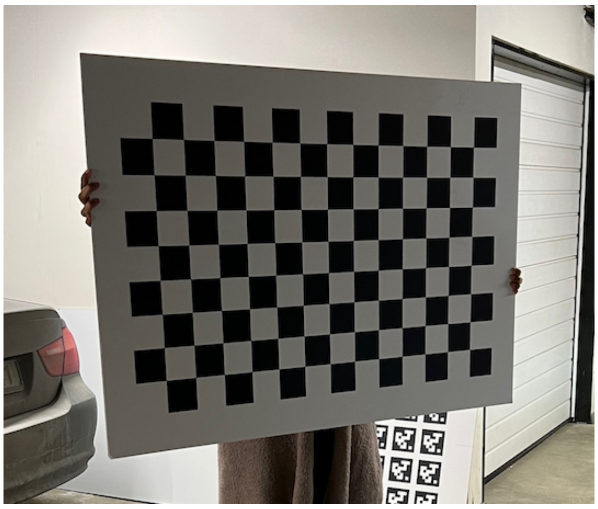
our use case, the most important sensor property is the rig transformation. This relates the camera and the rig coordinate system to each other, and contains the x, y, and z translations in meters and roll, pitch, and yaw in degrees. The origin in camera coordinates is therefore the position of the camera in rig coordinates. A full description of the rig file format can be found in NVIDIA's documentation [85].

The origin of the rig coordinate system is the middle of the back axle. The CARLA simulator, on the other hand, uses the center of the vehicle as coordinate system origin when attaching sensors. This means that a conversion is needed in order to replicate the real-world sensor positions in CARLA. In short, we do this by finding the back axle coordinates of the vehicle in CARLA, and calculating its offset from the center of the vehicle. This offset can then be used to get the desired sensor locations relative to the CARLA vehicle. Note, however, that DriveWorks uses a right-handed coordinate system with the y-axis pointed to the left, while CARLA uses a left-handed system with the y-axis pointed to the right. To get the correct transformation from rig coordinates to CARLA coordinates we therefore have to invert the rig y-axis and yaw. The full implementation is available on GitHub.[9] Once we had the correct sensor positions and rotations, we could attach them to the digital version of the Kia e-Niro created in Section 3.3.

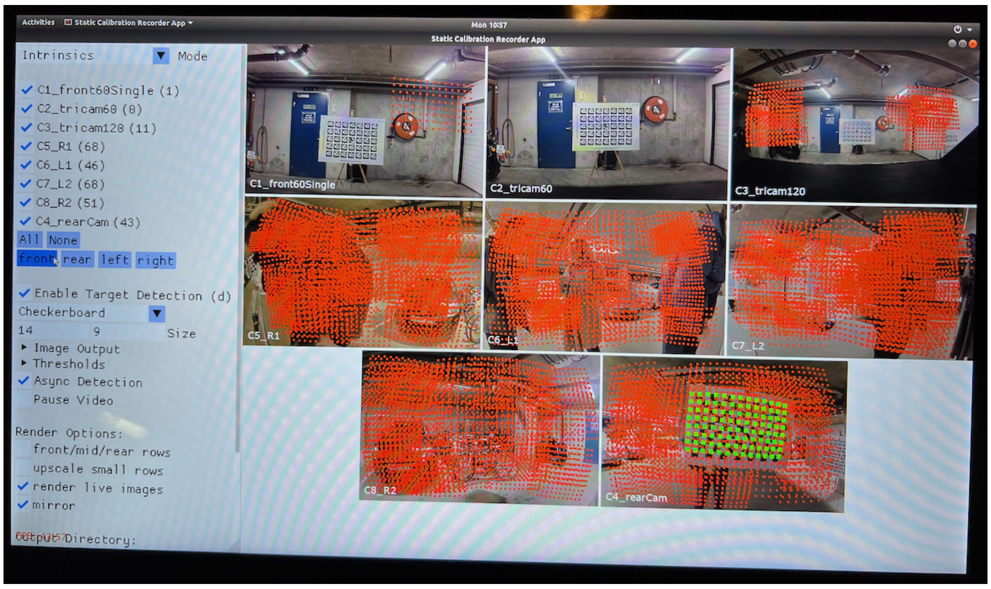
3.5 TransFuser

This thesis experiments with the work proposed by Chitta *et al.* [5]. This section will therefore describe the TransFuser model in more detail and is heavily based on their paper. They consider a Behavioural Cloning approach where the goal is safe driving directions in an urban setting.

[9]`https://github.com/aasewold/carlo/blob/main/src/scripts/camera_from_rig.py`



(a) Checkerboard used for intrinsic camera calibration.



Intrinsics ▼ Mode

✓ C1_front60Single (1)
✓ C2_tricam60 (0)
✓ C3_tricam120 (11)
✓ C5_R1 (68)
✓ C6_L1 (46)
✓ C7_L2 (68)
✓ C8_R2 (51)
✓ C4_rearCam (43)

All None
front rear left right

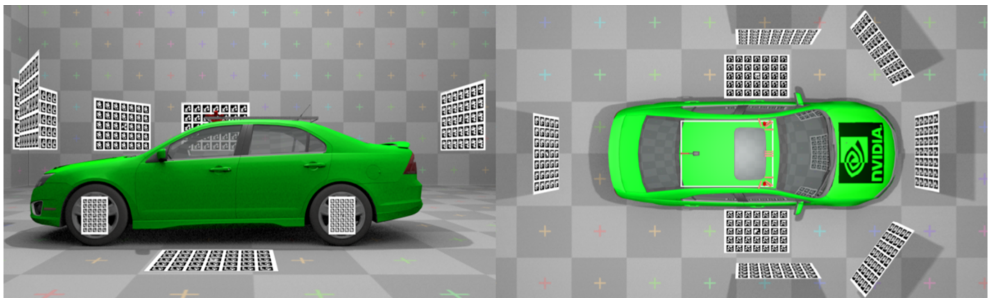
✓ Enable Target Detection (d)
Checkerboard ▼
14 9 Size
▸ Image Output
▸ Thresholds
✓ Async Detection
Pause Video

Render Options:
front/mid/rear rows
upscale small rows
✓ render live images
✓ mirror

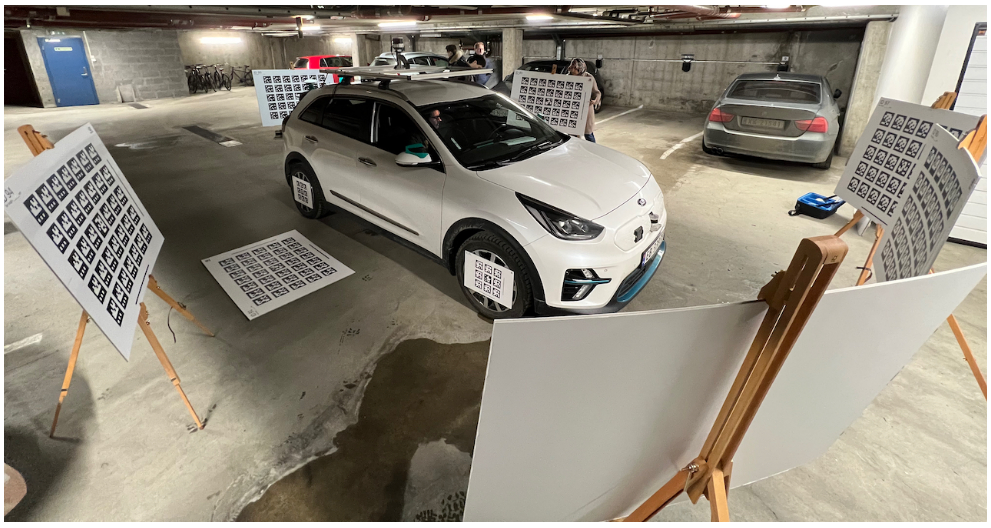
Output Directory:

(b) Software used for intrinsic camera calibration.

Figure 3.12: To perform intrinsic camera calibration, we moved a checkerboard across the camera view both vertically and horizontally until it had covered the whole view. We additionally rotated the checkerboard and moved it closer and further away from the camera to calibrate its rotation and scale. This was repeated for all cameras. The DriveWorks software [83] helped us to ensure that we covered the entire area. The red dots are snapshots of the checkerboard, while the green dots show the current location of the checkerboard.



(a) Ideal target placement according to NVIDIA. Source: [48].

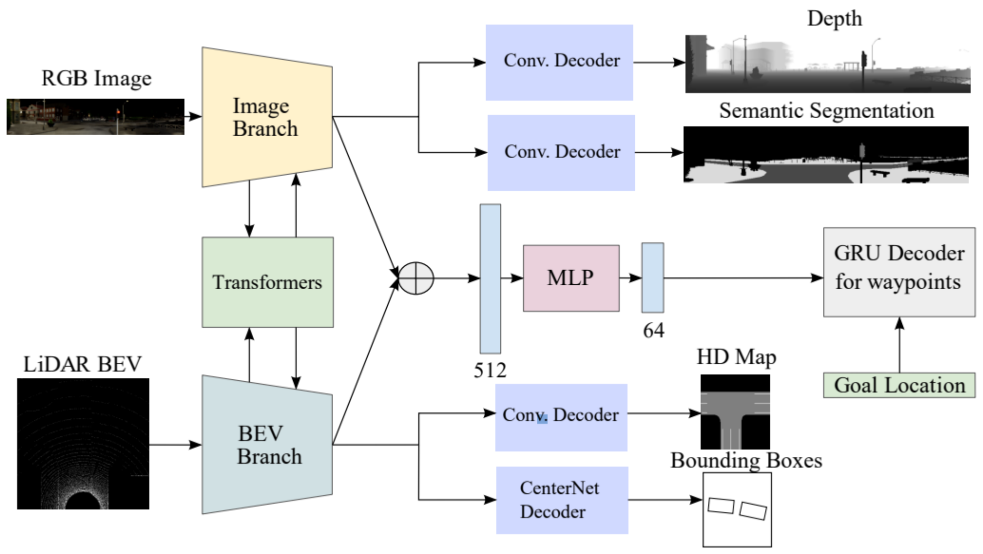


(b) Our target placement around the NAPLab car.

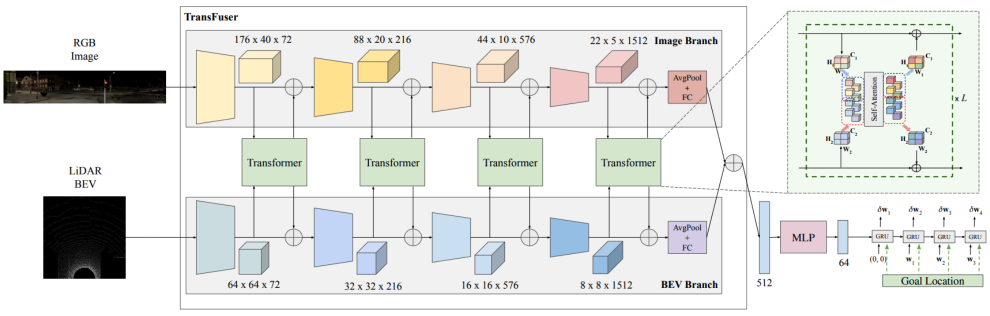
Figure 3.13: To calibrate extrinsics, we placed eight large targets around the car and four small targets on the wheels. Each camera had to observe at least one large target. In addition to capturing images from the mounter cameras, a set of images also had to be captured by an external camera. These were used to create a link between targets that the cameras on the car itself could not observe.

3.5.1 Model Architecture

The TransFuser model has two main components. A multi-modal fusion Transformer integrates information from RGB images and LIDAR data. This is then fed into an auto-regressive network for predicting waypoints that are used to control the autonomous vehicle. The following subsections will describe the components seen in Figure 3.14.



(a) TransFuser uses Transformers for the fusion of intermediate feature maps between RGB and LIDAR data. They are combined and fed into an MLP before passing it to an auto-regressive waypoint prediction network. The two branches are also used for auxiliary tasks.



(b) A more detailed view of the image and BEV branches, consisting of several Transformer modules, as well as the GRU decoder.

Figure 3.14: An overview of the TransFuser architecture. Source: [5].

Input

The model receives data from two modalities as input. The first is a three-channel image of size 256×256 pixels representing a 2D LIDAR BEV around the vehicle. The first two channels consist of a 2-bin histogram created from the raw LIDAR point cloud, where each bin represents the point on/below and above the ground plane. The last channel includes the 2D goal location converted into the same BEV space. The other modality comes from three front-facing RGB cameras with 60° spacing. The three images are combined into a single image with resolution 704×160 pixels and 132° field of view.

Multi-Modal Fusion Transformer

Chitta *et al.* [5] use the self-attention mechanism of Transformers to incorporate global context from complementary RGB images and LIDAR data. Using self-attention in natural language processing tasks normally means working with token input and output structures, but when working with image data, they instead work on grid structured feature maps. TransFuser performs dense feature fusion at multiple resolutions to obtain a feature map from both the RGB images and the BEV LIDAR data. They are then reduced to a dimension of 512 by average pooling and followed by a fully connected layer at each branch. The output feature vectors from each modality is then combined via summation, which results in a compact representation of the environment that encodes the global context of the 3D scene. See Section 2.6.2 for a detailed visualization of this component.

Waypoint Prediction Network

The 512-dimensional feature vector from the Multi-Modal Fusion Transformer is passed through a multilayer perceptron to reduce its dimensionality to 64 to increase computational efficiency. This output, in addition to the current position and goal location, is then fed into a waypoint prediction network implemented using gated recurrent units (GRUs). This results in waypoint predictions for four future time steps in the ego-vehicle coordinate frame.

Loss Functions and Auxiliary Tasks

The primary loss function is based on the L_1 distance between the above predicted waypoints and the expert trajectory. Let $\mathbf{w}_t^{gt}$ be the ground truth waypoint for timestep t. The loss function is given by

$$\mathcal{L} = \sum_{t=1}^{T} ||\mathbf{w}_t - \mathbf{w}_t^{gt}||_1$$

Additionally, several auxiliary losses are used to increase the interpretability and robustness of the model. 2D depth and semantic segmentation are decoded from the image branch features and are supervised with L_1 loss and cross-entropy loss, respectively. A high definition BEV map classifying roads, lane markings, and more is used to encourage the model to encode information regarding drivable and non-drivable areas. This is created from the LIDAR branch using a convolutional decoder and uses cross-entropy loss. Finally, a position map containing bounding boxes for other vehicles is predicted from the LIDAR branch using a convolutional decoder.

Output

The final output of the model is the predicted future trajectory of the ego-vehicle in BEV space. This is represented by a sequence of four 2D waypoints.

3.5.2 Controlling the Vehicle

PID-controllers use the predicted waypoints to perform low-level vehicle control, such as acceleration, steering, and braking. TransFuser implements one controller each for the lateral and longitudinal direction. The model also implements two functions for more refined control. Creeping is used to wake up the vehicle if it has been stationary for a long period of time, which is useful to counteract how vehicles tend to sit still in much of the training data due to dense traffic. Still, creeping would not be safe in a situation where the vehicle is actually stuck in traffic with another vehicle in front of it. Therefore, a safety heuristic is implemented that disables the creeping behavior based on LIDAR data in a small area in front of the car.

3.5.3 Training Data

An expert policy is first rolled out in the environment to collect a high-dimensional dataset that consists of observations of the environment and corresponding expert trajectory. Each time step in the dataset includes a front camera image input and LIDAR point cloud data, while the expert trajectory is defined by a set of 2D waypoints in bird's-eye view (BEV) space which uses the coordinate frame of the ego-vehicle. Chitta *et al.* [5] use all of the 8 included towns in CARLA version 0.9.10 to generate data. The dataset is based on around 2500 routes through junctions with an average length of 100 meters, and 1000 routes along curved highways with an average length of 400 meters. This results in a dataset containing 228,000 frames in total.

The expert policy uses information made available by the simulator to generate data at 2 FPS. The policy is constructed using an A* planner followed by two separate PID-controllers for lateral and longitudinal control. Lateral control is done by minimizing the angle of the car towards the next waypoint in the route. Longitudinal control is done using a version of model predictive control. The standard target speed is 4.0 m/s. This is reduced to 3.0 m/s while the expert is in an intersection. Finally, a predicted infraction sets the target speed to 0.0 m/s. Collision infractions are predicted by forecasting the position of all traffic participants based on a pre-trained kinematic bicycle model, while red light infractions are predicted by performing intersection tests with trigger boxes provided by CARLA.

The expert trajectory is determined by peeking at the expert agent's position four time steps ahead, which at 2Hz corresponds to 0.5, 1, 1.5, and 2 seconds ahead.

3.6 InterFuser

This thesis is also based on the work of Shao *et al.* [6], and therefore this section will describe InterFuser in detail based on their paper. We will see that their model has many commonalities with TransFuser, with additional focus on enhancing the safety and interpretability of end-to-end autonomous driving.

3.6.1 Model Architecture

The following subsections will explain the architecture seen in Figure 3.15. The authors describe three main components: a multi-view multi-modal fusion transformer encoder, a transformer decoder, and a safety controller.

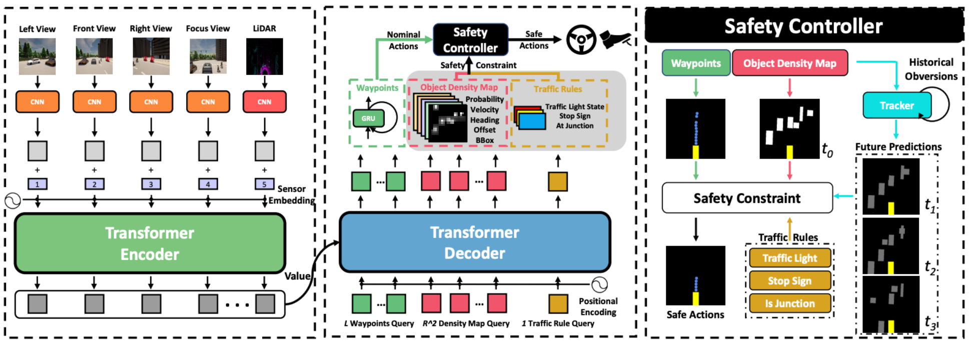


Figure 3.15: The InterFuser architecture. It consists of an encoder-decoder network built on transformers, in addition to a safety controller that constraints the output actions to enhance driving safety. Source: [6].

Input

The model receives data from two modalities as input, similar to TransFuser. Three cameras with a raw resolution of 800×600 pixels and a 100° field of view produce images which make up the first modality. The center camera is scaled and cropped to 224×224 pixels to create the front input image. It is also used to create a focusing view image to capture distant traffic lights in detail. This is created by cropping the center camera to 128×128 pixels without scaling. Two side cameras are angled ±60° away from the center and scaled and cropped to 128×128 pixels to produce the left and right input images. For the other modality, the raw LIDAR point cloud is converted into a two-channel LIDAR BEV projection image.

Backbone

The input images and LIDAR data are fed through a ResNet backbone to generate a lower resolution feature map. The features are then forwarded to the transformer encoder.

Transformer Encoder and Decoder

The feature maps go through a convolution to reduce their dimensions, before the spatial dimension of each feature map is collapsed into one-dimensional tokens. Positional encoding is added to each token, and learnable sensor embeddings are added to distinguish tokens from the different sensors. The tokens from all sensors are then concatenated and passed through a transformer with multiple standard transformer layers.

The transformer decoder follows, where three types of queries are created by leveraging the attention mechanism: waypoint queries, density map queries, and a traffic rule query. Learnable positional embeddings are added to the query embeddings, and they are then fed to the prediction heads.

Prediction Heads

Three parallel predictions modules predict the waypoints, the object density map, and the traffic rule. A single GRU is utilized for auto-regressive prediction of future waypoints. The GRU's initial hidden state is given a vector embedding of the GPS coordinates of the goal location. The density map is predicted using a 3-layer MLP. The traffic rule prediction is based on a single linear layer that predicts the state of traffic lights ahead, whether there is a stop sign ahead, and whether the ego vehicle is at an intersection.

Loss Function

InterFuser's loss function is based on three predictions above; it encourages predicting the desired waypoints ($\mathcal{L}_{pt}$), object density map ($\mathcal{L}_{map}$) and traffic information ($\mathcal{L}_{tf}$):

$$\mathcal{L} = \lambda_{pt}\mathcal{L}_{pt} + \lambda_{map}\mathcal{L}_{map} + \lambda_{tf}\mathcal{L}_{tf},$$

where the λ constants balance the three loss terms. Similar to TransFuser, the waypoint loss is based on the L_1 distance between the generated waypoints and the expert waypoint. The object density map loss is based on a combination of predicting whether an object exists in the map and what its features are (offset, heading, velocity and bounding box). Lastly, the traffic information loss is calculated using binary cross-entropy loss for traffic light status, stop signs, and whether the vehicle is at an intersection.

Output

The output of the model is separated based on whether the output is safety sensitive. Safety-insensitive output consists of predicted waypoints that guide the future driving route of the ego vehicle. To complement these waypoints, the model also generates safety-sensitive outputs in the form of object density map and traffic rule information. These help the vehicle maintain a safe speed, avoid objects around it, and avoid violations of traffic rules.

3.6.2 Using the Safety Controller to Control the Vehicle

The output waypoints, object density map, and traffic rule are used to constrain the vehicle actions into a safe set. InterFuser uses PID controllers to obtain lateral steering action and longitudinal acceleration action. The vehicle steering is based on the average heading of the first two waypoints ahead of it. The safety controller sets a limit on the target velocity based on the maximum safe distance the vehicle can drive at a given time step, and tracks the current state of surrounding objects in addition to considering their future trajectory. The predicted traffic rule is used to make an emergency stop if the traffic light is not green or if there is a stop sign ahead.

3.6.3 Training Data

Similar to Chitta *et al.* [5], Shao *et al.* [6] run a rule-based expert on all 8 included towns in CARLA version 0.9.10.1 to generate data at 2 FPS for their model. They use 21 different weather setups, and the expert agent has access to privileged information in the simulator. The expert agent's target speed is 6.5 m/s, which is reduced to 4 m/s if the expert agent needs to take a sharp turn. If the agent predicts a possible collision, a red light, or a stop sign ahead, the target speed is reduced to 0.0 m/s.

The resulting dataset is generated from randomly chosen routes, dynamically spawned objects, and adversarial scenarios. Each time step includes images from camera sensors, LIDAR point cloud data and information about the world and surrounding objects. It has in total 3 million frames, which is much larger than TransFuser's dataset.

The expert trajectory generated by the rule-based agent consists of 10 waypoints forward in space along the route planned by A*, with approximately 1 meter distance between each. This is in contrast to TransFuser's expert trajectory, which predicts forward in time, and therefore encodes velocity information in the waypoints.

3.7 Upgrading to CARLA Version 0.9.14

At the time of writing, the top submissions on the CARLA Leaderboard are based on CARLA version 0.9.10. There is however a significant graphical difference between version 0.9.10 and the latest versions. Figure 3.16 showcases this. The primary reason for the differences is the upgrade to Unreal Engine 4.26 in CARLA version 0.9.12. The upgrade gave a much improved lighting engine, which is especially noticeable in darker scenarios. As our thesis works towards closing the gap between simulation and reality, we therefore wanted to utilize the more realistic environment found in version 0.9.14. We also noticed some graphical glitches when running CARLA version 0.9.10 without a display using OpenGL in Docker (Figure 3.17). This was a known issue when running version 0.9.10 using OpenGL and Linux [86], and in order to use Vulkan with Docker without a display, we had to upgrade to the newer CARLA version. This improved support for headless mode was also a requirement for running the simulator on the Idun HPC cluster.

3.7.1 TransFuser and InterFuser

Both TransFuser and InterFuser were originally designed for CARLA version 0.9.10. Motivated by the simulator improvements described above, we therefore wanted to upgrade the models to work with version 0.9.14.

To do this, we forked the projects and made the necessary adjustments to be compatible with the new CARLA version's API. Our forked versions are publicly available at our GitHub organization under `aasewold/transfuser`[10] and `aasewold/InterFuser`[11].

No major structural changes were needed; the API changes consisted mainly of renaming classes and functions. Specific details can be found in the git log on the branch named `experiments/0.9.14` in both repositories.

One change impacting TransFuser in particular was the reorganization of the class IDs for CARLA's semantic segmentation camera. TransFuser uses an index-based mapping to combine some of CARLA's many semantic classes into a lesser amount used in TransFuser. When CARLA's semantic class indices changed, this mapping also had to be updated. Unfortunately, the CARLA upgrade combines the previously separate classes for red, yellow, and green traffic lights into one, which makes it no longer possible for TransFuser to learn to classify them separately. Thus, the model now learns to detect traffic lights in general, instead of just the red and yellow states as before. InterFuser does not use semantic segmentation during training, and thus was not impacted by this change.

An additional change we made to both repositories was to enable headlights for all vehicles in the simulation, including the expert agent vehicle. Not only was

[10]`https://github.com/aasewold/transfuser`
[11]`https://github.com/aasewold/InterFuser`

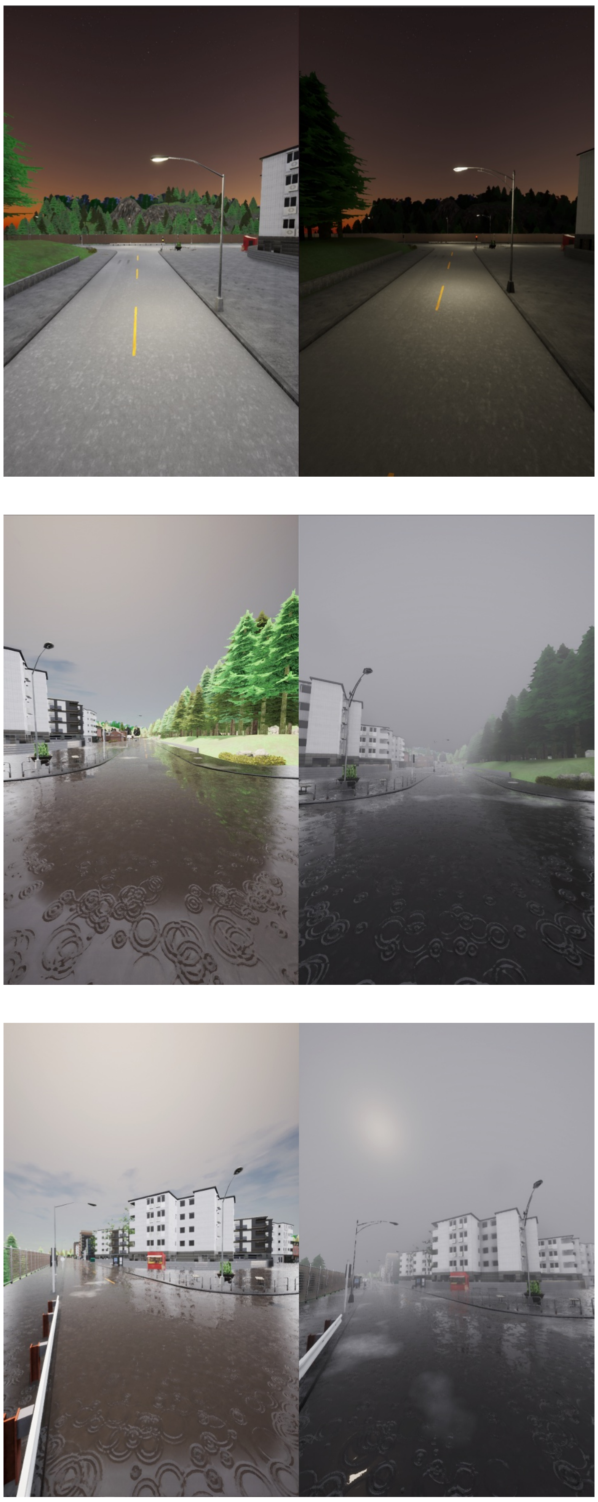
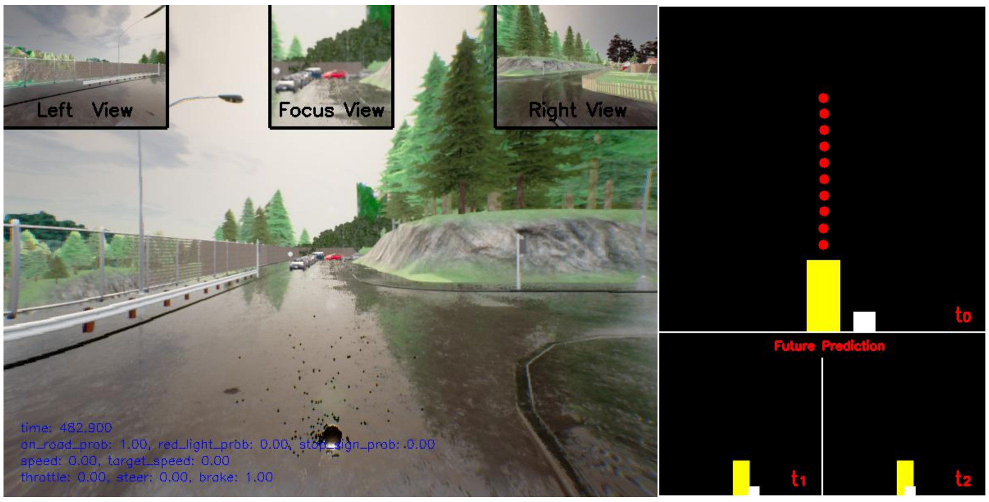


Figure 3.16: Three examples comparing the graphical differences between CARLA versions 0.9.10 (left) and 0.9.14 (right). In general, there is less ambient light and the fog is more dense. Additionally, CARLA 0.9.14 features a new effect where rain drops on the camera lens can distort parts of the image (lower right).



```
time: 482.900
on_road_prob: 1.00, red_light_prob: 0.00, stop_sign_prob: .0.00
speed: 0.00, target_speed: 0.00
throttle: 0.00, steer: 0.00, brake: 1.00
```

Figure 3.17: Running CARLA version 0.9.10 with OpenGL in Docker resulted in graphical glitches in the form of black dots on the cameras. The black dots can be seen in the bottom center of the left image. In this example one can see that the model thinks the road in front is blocked (the red circles in the top right image).

this essential for seeing the road and environment in the darker scenes found in CARLA version 0.9.14, but this also follows the Norwegian law stating that it is mandatory to always have driving lights on while driving.

3.7.2 Graphical Glitches in CARLA Version 0.9.14

Both `nap02` and Idun are equipped with NVIDIA A100 graphics cards as described in Section 3.1. Unfortunately, we encountered graphical glitches when running the latest CARLA version on these cards. During both data generation and evaluation, we sometimes saw black pixelated artifacts rendered on the RGB camera sensor output, as shown in Figure 3.18. Note that these artifacts were not present on all rendered images, but occurred randomly. This is another known issue with the simulator, but no solutions were described at the time of writing [87]. The A100 cards were heavily utilized for our methods and experiments, and therefore this issue could affect our results.

However, it is not certain that the effect is negative, since the glitches bear some similarity to the commonly used dropout technique used to improve robustness and generalization when training neural networks. The technique involves zeroing a random subset of weights in the network during forward inference to stimulate the learning of multiple redundant paths for information flow. In our case, random subsets of the input image is zeroed, in which case the agent is stimulated to learn correct behavior from looking at the rest of the image, or alternatively the LiDAR input. Therefore, we do not make any further effort to eliminate these glitched images from our training data.

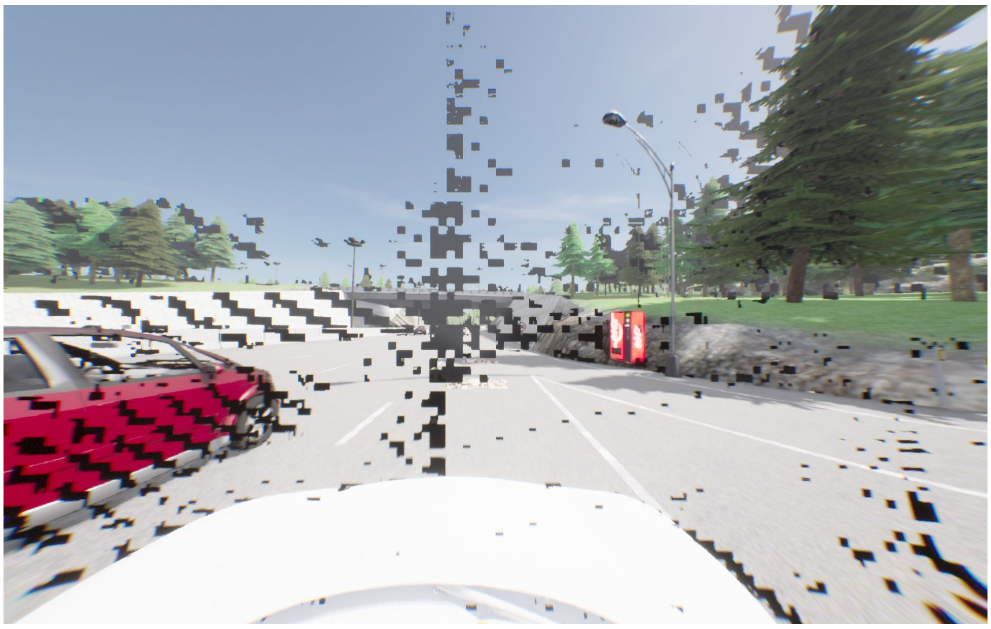


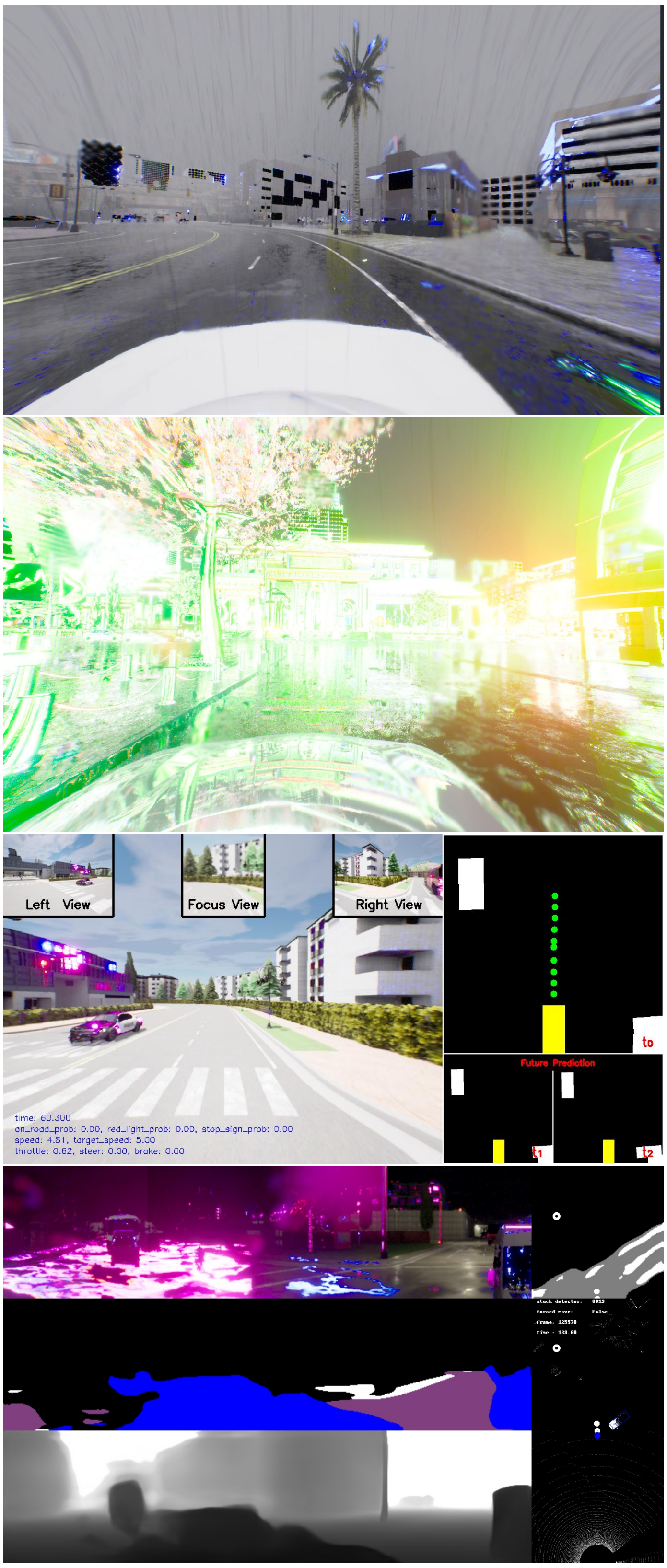
Figure 3.18: Running CARLA version 0.9.14 on NVIDIA A100 graphics cards sometimes produced images with black pixelated artifacts. This image is an example from our data generation process.

We also encountered graphical glitches when running CARLA version 0.9.14 on NVIDIA 4090 graphics cards. Examples of these are given in Figure 3.19, and they indicate that there are problems with the rendering of reflections and specific lighting conditions. This happened during both data generation and evaluation and, although they were not very common, they could still affect our results.

3.8 Dataset Generation

To train TransFuser and InterFuser, we need appropriate training data as described in Sections 3.5.3 and 3.6.3. While TransFuser has a publicly available dataset [88], it is based on CARLA version 0.9.10 and contains a different sensor configuration than the real-world Kia e-Niro we will use in our experiments. InterFuser does not offer a publicly available dataset.

To generate training data, we follow and adapt the method used by TransFuser and InterFuser, which is to first generate a set of routes for an expert agent to follow, and then launch an expert agent to follow these planned routes in the CARLA simulator. While the expert is running, it's sensory inputs and driving decisions are recorded at regular intervals. Each recorded time step forms a labeled data sample for use in the supervised training process. Note that the expert agent has access to privileged data, such as semantic camera images and ground truth positions of surrounding vehicles, which enables it to drive safely without utiliz-



```
time: 60.300
on_road_prob: 0.00, red_light_prob: 0.00, stop_sign_prob: 0.00
speed: 4.81, target_speed: 5.00
throttle: 0.62, steer: 0.00, brake: 0.00
```

```
stuck detector:   0019
forced move:      False
Frame: 125578
Time : 189.60
```

Figure 3.19: Examples of graphical glitches encountered during data generation and evaluation in CARLA version 0.9.14 using NVIDIA RTX 4090 GPUs.

ing complex image processing. These privileged data are also recorded and later used during training. In addition to recreating the original datasets, we create new datasets with modified sensor configurations to match NAPLab's Kia e-Niro using the method described in Section 3.4.1. The resulting camera setup is visualized in Figure 3.20.

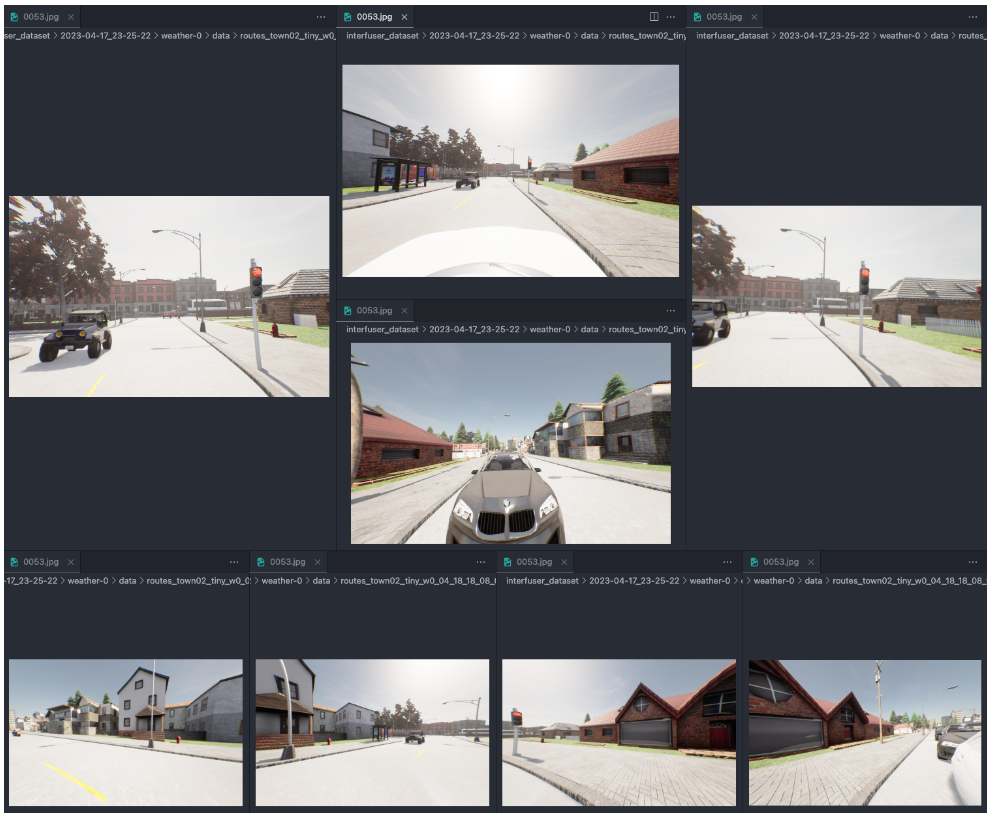


Figure 3.20: Screenshot of the Kia e-Niro camera setup in the simulator used to generate training data. The setup matches the physical cameras shown in Figure 2.6. The top middle, left, and right images show the three front cameras (60°, 120° and 60° FOV). The middle center image shows the rear camera (120° FOV). The bottom row contains the side mirror cameras (all 120° FOV).

During simulation, various events known as *scenarios* are executed to diversify the dataset and challenge the agent. For instance, pedestrians and bikers may suddenly appear and cross the road in front of the agent, requiring corrective maneuvers to avoid collision. One example of this is the red vending machine seen in Figure 3.18, which may appear suddenly with a pedestrian running onto the road from behind the machine.

3.8.1 Using Idun

Due to the large amount of data to be generated, ranging from about 100 GiB to over 1 TiB, we utilized the Idun cluster and its ability to parallelize data generation jobs. Since each route is independent of all other routes, they can be run concurrently, greatly speeding up the process. Each job runs a CARLA server instance and a Python client instance. The client is responsible for collecting data with the expert agent. Since the CARLA simulator tends to crash every so often, the Slurm job scripts include logic to restart and resume the data generation scripts in case of a crash. When all job routes and scenarios are finished, the job creates an archive of the data to help avoid reaching Idun's per-user storage and file limit. This was essential as the number of files generated per frame from all sensors and other metadata quickly filled up the 2 million file quota. By archiving the generated data, we reduced several million files into one. We then transfer the archived data to `nap02` for later use during training. The implementation of the data generation scripts and Idun Slurm jobs can be found in ^{ex}`datagen-transfuser/idun` and ^{ex}`datagen-interfuser/idun`. The following sections describe the implementation details for both models.

3.8.2 TransFuser

TransFuser originally follows a three-step data generation procedure:

1. Generate scenarios by sampling allowed event locations from all available CARLA maps.
2. Generate routes that pass through the scenario locations.
3. Use an expert agent to successfully traverse the route and record the data required for training.

In addition to the complete dataset, Chitta *et al.* [5] has also published the scenarios and routes from steps 1 and 2. When generating custom datasets, it is therefore possible to skip the first two steps and utilize the existing routes. These routes were however generated targeting CARLA version 0.9.10, and are therefore susceptible to incompatibilities when used together with CARLA 0.9.14. This, together with our goal of recreating the TransFuser training method from scratch, motivated the choice to re-generate scenario and route definitions. The original scenario and route generation procedure consists of launching a CARLA server and then running the provided scripts `gen_scenarios.sh` and `gen_routes.sh`. Our approach to route generation is to re-create the expected runtime environment in Docker containers and launch the appropriate scripts using a Makefile. The implementation can be found in ^{ex}`datagen-transfuser/routegen`. From this folder, the command `make scenarios` will launch a CARLA server and run TransFuser's scenario generation scripts until completion, after which `make routes` will execute the next step of the procedure. As part of our adaptations for using the Idun cluster, we also generate a CSV file containing pairs of filesystem paths to connect routes with their correct scenarios using the command `make csv`. An excerpt of the generated CSV file is shown in Code listing 3.1. Each line of this CSV file specifies configuration for a data generation job to be run on Idun, and which can be executed concurrently with jobs specified by the other lines. The CSV file for TransFuser's route configuration specifies 64 route-scenario pairs, which enables up to 64 concurrently executing jobs.

To help submit data generation jobs to Idun, we built specific tooling in ^{ex}`datagen-transfuser/idun`. From this folder, one can run the commands:

```
export CARLA_IMAGE=carlasim/carla:0.9.14
export TRANSFUSER_COMMIT=experiments/0.9.14
make job
```

to submit a data generation job to Idun. This will execute the following steps:

1. Build a Docker image for running CARLA 0.9.14
2. Convert the CARLA docker image to the Singularity `.sif` format
3. Build a Docker image containing the specified TransFuser version
4. Convert the TransFuser docker image to the Singularity `.sif` format
5. Transfer both `.sif` files to Idun
6. Transfer any additional files from the `files/` directory to Idun
7. Execute the script `files/submit.sh` on Idun using SSH, which then:
    a. Prepares a working directory containing the scenario and route definitions, as well as the CSV file specifying the route-scenario pairs.
    b. Submits a Slurm job for each of the pairs in the CSV file.

Each Slurm job then executes step 3 of the data generation procedure by launching a CARLA server and the expert agent to traverse the routes it is assigned. Due to frequent unhandled program errors in both CARLA and the TransFuser expert agent, a loop is set up so that data collection is restarted continuously until the job is fully completed. The outline of this retry loop can be seen in Algorithm 1.

Algorithm 1 Retry loop for data-generation.

```
while $completed\ routes < assigned\ routes$ do
    $p \leftarrow$ find available TCP port
    launch CARLA in Singularity on port $p$
    launch TransFuser client in Singularity connecting to port $p$
    wait until client exits
    kill CARLA server
    $completed\ routes \leftarrow$ parse progress from $checkpoint.json$
end while
```

In addition to the original TransFuser dataset **ds-orig**, we generate three datasets used for training TransFuser:

```
/routes/routes/Scenario7/Town02_Scenario7.xml,/routes/scenarios/Scenario7/
    Town02_Scenario7.json
/routes/routes/Scenario7/Town03_Scenario7.xml,/routes/scenarios/Scenario7/
    Town03_Scenario7.json
/routes/routes/Scenario9/Town10HD_Scenario9.xml,/routes/scenarios/Scenario9/
    Town10HD_Scenario9.json
/routes/routes/Scenario9/Town01_Scenario9.xml,/routes/scenarios/Scenario9/
    Town01_Scenario9.json
```

Code listing 3.1: Excerpt of TransFuser data generation route configuration CSV file.

ds-feb consists of 154 026 frames with TransFuser's original sensor configuration, with routes generated for CARLA 0.9.14 using TransFuser's route generation scripts.

ds-mar consists of 381 867 frames with the same sensor configuration, using routes generated using InterFuser's route generation scripts.

ds-apr consists of 215 913 frames using the Kia's sensor configuration, using the same routes as ds-feb. The increased number of frames is due to an improved restart policy developed to resume data generation from a checkpoint in the case of unhandled program errors, instead of simply aborting early.

Due to the increase in both number of cameras and camera resolution, the data generation process for ds-apr exhibited much larger requirements for disk space. To fit within the available space on `nap02`, the generated RGB images were converted to the JPEG image format, whereas they before were stored as PNGs. Due to JPEG being a lossy compression algorithm, enabling greater compression ratios at the expense of some amount of data loss, this conversion reduced the size of each RGB image by approximately 90%. This was deemed acceptable after visually inspecting a test sample of images before and after conversion and finding no visible compression artifacts. However, we opted against applying the same compression to the equally large depth images, due to the possibility of losing the high-frequency content contained in those images. The semantic segmentation images required no such conversion, as the PNG format already adequately compressed their content. We applied the same conversion to the previously generated ds-feb and ds-mar, as well as the original TransFuser dataset. Table 3.1 lists the space savings achieved for each dataset. The increased camera resolution and greater number of cameras also impacts the data generation time, causing a 3x increase in time-per-frame. In total, ds-apr required 68 days of GPU compute time, while ds-mar used 35.5 days, as shown in Table 3.2.

Dataset	Disk usage (PNG)	Disk usage (JPEG)	Reduction
ds-orig	234 GiB	195 GiB	16.7%
ds-feb	141 GiB	117 GiB	17%
ds-mar	345 GiB	286 GiB	17.1%
ds-apr (partial)	6.2 TiB	3.6 TiB	42%
ds-apr (final)	-	4.2 TiB	-

Table 3.1: Summary of dataset disk space requirements, before and after conversion to JPEG. ds-apr was converted to JPEG partially during generation, which is why its final disk usage in PNG format is not available.

Dataset	Time to generate	Time per frame
ds-feb	-	-
ds-mar	35.5 days	8 seconds
ds-apr	68 days	27 seconds

Table 3.2: Summary of data generation time requirements. Elapsed time is not available for ds-feb due to missing log files from that run. Note that the data generation was parallelized, and the time listed is the total compute time. The wall-clock time on the other hand was in the range of one to seven days.

3.8.3 InterFuser

We used the methods defined in InterFuser's repository `README` as a starting point for our data generation scripts [89]. InterFuser's code does not include scripts to generate routes and scenarios like TransFuser, therefore we used the included routes and scenarios which covered all towns in CARLA. For each CARLA town, there are three types of routes included: long, short, and tiny. The shorter routes primarily contain single scenarios, such as a turn or a lane change, while the longer routes also include scenarios such as pedestrians crossing the road. Each route configuration is also paired with a weather type. InterFuser includes 20 different weather types to choose from, ranging from a clear blue day to a hard rain night. Together these options create a balanced dataset that should model a good variety of driving trips in the real world. Code listing 3.2 contains an example list of route configurations.

Once we have the list of route, scenario, and weather configurations, we start the data generation process. This works as described for TransFuser in Section 3.8.3 by building Sigularity `.sif` files, transferring them to Idun, and executing a submit script that creates one Slurm job per route configuration. The InterFuser-specific implementation is available in ^{ex}`datagen-interfuser`. Due to the large amount of data to be generated with InterFuser and a long job queue on Idun, we also implemented a way to generate data on local hosts through Docker Compose. This configuration mimics the approach done on Idun, but replaces the Slurm job

```
route,scenario,weather_id
training_routes/routes_town01_tiny.xml,scenarios/town01_all_scenarios.json,0
additional_routes/routes_town01_long.xml,scenarios/town01_all_scenarios.json,0
training_routes/routes_town01_tiny.xml,scenarios/town01_all_scenarios.json,2
training_routes/routes_town01_short.xml,scenarios/town01_all_scenarios.json,2
additional_routes/routes_town01_long.xml,scenarios/town01_all_scenarios.json,6
training_routes/routes_town02_tiny.xml,scenarios/town02_all_scenarios.json,6
```

Code listing 3.2: Example InterFuser data generation route configuration CSV file.

parallel execution with GNU Parallel [90]. In our case, we used `vcxr12` for local data generation.

We have pushed multiple changes to InterFuser's data generation code. The primary changes concern the use of our custom vehicle and the type of data that is generated. InterFuser's original dataset structure contains the following:

- RGB images (front, left right and rear)
- Corresponding segmentation images
- Corresponding depth images
- Lidar point cloud
- A topdown segmentation image (bird view)
- 2D and 3D bounding boxes for different agents
- Different types of affordances
- Measurements of the ego-vehicle's position, velocity and other metadata
- Positions, velocities and other metadata of surrounding vehicles and traffic lights

Segmentation, depth, bounding boxes, and top-down data were all removed since they were not used in InterFuser's default training configuration. Overall, the reduction of data types resulted in less sensor computation per frame, which sped up the data generation jobs. One additional change we made to InterFuser's data generation code was to create an archive of the generated data after each route. This was done to help reduce the number of files generated, which was needed to avoid reaching Idun's per-user file limit.

The LIDAR sensor originally had its rotation frequency equal to 10. The simulation runs at 20 FPS, meaning we only stored one half of the LIDAR point cloud. This was fine since InterFuser only uses the front half of the point cloud for training, but an unknown change between CARLA version 0.9.10.1 and 0.9.14 made the sensor swap halves when generating data. This resulted in the model getting no LIDAR data. To fix this, we simply changed the LIDAR rotation frequency to 20 so that it stored the whole point cloud. The model then cuts off the data it needs during training from the LIDAR point cloud.

In contrast to TransFuser, InterFuser generated JPEG images by default. We generate the following datasets for training InterFuser:

ds-custom consists of 556 192 frames using InterFuser's original sensor configuration, generated for CARLA 0.9.14. It includes 10 weather types in towns 1 to 7. The total disk usage is about 300 GiB. The goal of this dataset was to verify that our data generation and training scripts worked as expected. The size is therefore smaller than InterFuser's original dataset and our Kia-like dataset.

ds-kia consists of 2 261 519 frames using the Kia's sensor configuration, generated for CARLA 0.9.14. It includes data from 10 weather types in all eight CARLA towns. The total disk usage is 3.4 TiB.

3.9 Training

Both TransFuser and InterFuser offer pre-trained weights in their corresponding GitHub repositories [88, 91]. This was helpful for testing our evaluation pipeline and as a baseline for our experiments. However, as mentioned in Section 3.8, their weights are trained using data from CARLA version 0.9.10, and their data use a different sensor setup than we have on the real world Kia e-Niro. InterFuser additionally only shares a pre-trained model based on part of the full dataset.

Therefore, with the goal of evaluating simulator data from Section 3.8 and real-world data from the Kia e-Niro in mind, we created a pipeline for training our own weights for both TransFuser and InterFuser using CARLA version 0.9.14. The code is publicly available in the experiments repository under ^{ex}`train-transfuser` and ^{ex}`train-interfuser`. The implementation of both is described in the sections below.

Both models are trained on `nap02`, utilizing its two NVIDIA A100 80G GPUs, 96 CPU cores, 512 GiB RAM, and docker setup. We use PyTorch for distributed training [92] and monitor progress using TensorBoard [93]. TensorBoard visualizes the loss metric graphs and, in the case of InterFuser, also previews the images and LIDAR data during training.

3.9.1 TransFuser

TransFuser is originally trained using the AdamW optimizer with learning rate 1×10^{-4} and batch size 12 for 41 epochs. After 30 epochs, the learning rate is reduced to 1×10^{-5}, and after 40 epochs, it is reduced further to 1×10^{-6}. In this work, we additionally perform experiments where the model is trained for 26 epochs, with learning rate reduction instead occuring at epoch 15 and 25.

The training algorithm expects a dataset with the structure shown in Figure 3.21, where the routes follow the highlighted original dataset structure with a single folder for each camera type (RGB, depth, and semantics). In this case, the camera images are a pre-stitched combination of the three cameras used during data generation.

The training algorithm is also modified to support the loading of named camera images following the Kia dataset structure. In this case, the images of different names are stitched together when loaded to form continuous camera images, subject to configuration when training is initialized. When training on the Kia dataset, we have configured the data loader to stitch together images from cameras named `C3_tricam120`, `C7_L2`, and `C8_R2`, as these closely resemble the stitches in the original dataset. More details on this decision are described in Section 3.11.2.

During training, TransFuser employs data augmentation to artificially increase the variance of the dataset. Specifically, each frame is transformed according to a random rotation of the vehicle about the up axis at the time of data generation. When a frame is requested during training, augmentation occurs with a 90% probability. When this happens, a random angle α is chosen from the interval $(-20°, 20°)$, which is used to rotate the LiDAR data, bird's-eye view (BEV) image, and target waypoints around the vertical axis. The camera images are horizontally shifted, corresponding to a rotation of the camera by α, concluding the augmentation. This serves to improve the agent's invariance and robustness to perturbations in the input data, particularly with respect to turns and rotations.

When training on the Kia dataset, the augmentation itself is augmented. Since the stitching of images from different cameras occurs during data loading, it is possible to apply augmentation at this step. We do this by applying a random scale and shift to each of the three images that are stitched together, with the intention to lessen the impact of the imperfect seams. The depth and segmentation images are augmented exactly in the same way to ensure consistency between the three camera types.

All necessary tooling to train a new TransFuser model is gathered in ^{ex}`train-transfuser`. The `run.sh` script requires a name for the model to be trained, and a path to the dataset, after which it will start a containerized training session.

3.9.2 InterFuser

We perform several steps to prepare InterFuser's data for training using the script located at ^{ex}`train-interfuser/scripts/dataset.py`. The first is to copy the archived data generated at Idun to `nap02` and extract it to the format InterFuser expects, see Figure 3.22.

Each of the `routes`-folders contains the sensor data described in Section 3.8.3 (RGB images, LIDAR, etc.), and the routes are grouped by weather configuration. Once the data is extracted, we go through all the routes and look for frames where the ego vehicle is blocked for many consecutive frames. This can happen for multiple minutes due to traffic jams in the traffic simulation. Since it is not optimal to train on lots of almost equal frames, we remove this blocked data to increase the

```
<dataset-root>
├── <collection>
│   ├── <subcollection>
│   │   ├── <route>.................................... Original dataset structure
│   │   │   ├── rgb ..................................................... RGB images
│   │   │   ├── depth ................................................. Depth images
│   │   │   ├── semantics ......................... Semantic segmentation images
│   │   │   ├── lidar .................................................. LiDAR scans
│   │   │   ├── topdown ......................................... BEV image of scene
│   │   │   ├── measurements .......... JSON data with ground-truth waypoints
│   │   │   └── label_raw ........................... JSON data about nearby cars
│   │   ├── <route>.......................................... Kia dataset structure
│   │   │   ├── rgb_<name>
│   │   │   ├── depth_<name>
│   │   │   ├── semantics_<name>
│   │   │   ├── lidar
│   │   │   └── ...
│   │   └── ...
│   ├── <subcollection>
│   │   ├── <route>
│   │   └── ...
│   └── ...
├── <collection>
│   ├── <subcollection>
│   └── ...
└── ...
```

Figure 3.21: TransFuser's dataset directory structure required for training. The original dataset structure contains a single set of camera images per route. The Kia dataset structure contains multiple named cameras per route, each with their own RGB, depth, and semantic images.

```
<dataset-root>
├── weather-0
│   └── data
│       ├── routes_town01_long_w0_04_18_04_04_15
│       │   ├── lidar ............................................... LiDAR scans
│       │   ├── C3_tricam120..................Front facing 120° RBG camera
│       │   ├── C7_L2..........................Left mirror front 60° RGB camera
│       │   ├── C8_R2........................Right mirror front 60° RGB camera
│       │   ├── measurements ............... Position, velocity, etc. of ego-vehicle
│       │   ├── actors_data......Positions, velocities, etc. of surrounding actors
│       │   ├── affordances ........................ Different types of affordances
│       │   └── ...
│       ├── routes_town02_short_w0_04_19_05_42_10
│       └── ...
├── weather-1
│   └── data
│       ├── routes_town01_tiny_w2_04_19_14_20_40
│       ├── routes_town03_short_w2_04_19_14_22_23
│       └── ...
├── weather-2
└── ...
```

Figure 3.22: InterFuser's dataset directory structure required for training. This figure shows the Kia dataset structure. The original dataset is identical, but replaces `C3_tricam120`, `C7_L2`, and `C8_R2` with `rgb_front`, `rgb_left`, and `rgb_right`, respectively.

```
weather-7/data/routes_town06_long_w7_11_28_18_28_35/ 1062
weather-2/data/routes_town01_short_w2_11_16_08_27_10/ 1785
weather-2/data/routes_town01_short_w2_11_16_09_55_05/ 918
```

Code listing 3.3: Example InterFuser dataset index file.

variety of the remaining data. Finally, the model expects a `dataset_index.txt` file describing where to find the route data and how many frames each route contains. See Code listing 3.3 for an example.

We can now move on to training. Running the `run.sh` script will start distributed training in a docker container using PyTorch. InterFuser requires you to define which towns and weather configurations to use for training and optional validation. As an example, you can decide to use data from towns 1 and 2 with weather ids 1, 3 and 4 for training, and town 5 with weather id 2 for evaluation. This is set up in the `train.sh` script. The AdamW optimizer is used for training. The initial learning rate is set to 5×10^{-4} for the encoder and decoder and to 2×10^{-4} for the CNN backbones. The weight decay is set to 0.05. We train the model for a total of 35 epochs, where the first five epochs are used for warm-up. To increase the variance in the dataset, the input RBG images are first augmented by randomly scaling their sizes with a factor of 0.9 to 1.1. The images are then augmented with color jitter, a process that randomly changes color, brightness, saturation, and hue. We follow the authors by leaving out data from Town05 for validation during the training process.

3.10 Evaluation in CARLA

The CARLA Leaderboard [7] presented in Section 2.2.1 is useful to publicly compare individual approaches to autonomous driving by different authors. It also has a simple submission process which only requires uploading a Docker image of the autonomous agent code using their CLI [94]. However, the online leaderboard evaluation has a long execution time (around 225 hours for the TransFuser submission and around 80 hours for the InterFuser submission [95]), and participants are limited to using the platform for only 200 hours each month. We therefore need other ways to evaluate our models during development.

Both TransFuser and InterFuser include several benchmarks that can be run locally. They are based on the same principles as the CARLA Leaderboard evaluation, meaning they consist of several predefined routes with adversarial scenarios that test the autonomous agent's driving proficiency in realistic traffic situations. Some also claim to be more challenging than the CARLA Leaderboard. We run benchmarks for CARLA versions 0.9.10 and 0.9.14.

3.10.1 Overview of Benchmarks

We chose three benchmarks for our experiments. All share several similarities with the official CARLA Leaderboard, but can be used for evaluation on local resources without the online platform's computation budget restrictions. They also report a set of metrics during the evaluation. The first is Route Completion (RC), which is the percentage of route distance completed averaged across all routes. The second metric is Infraction Score (IS). By tracking several types of infractions, the IS is reduced from an ideal score of 1.0 each time an infraction is committed. Examples of infractions tracked are collisions with pedestrians, vehicles and static elements, running a red light, off-road driving, route deviations, timeouts, and the vehicle getting stuck. Finally, using IS and RC, they also report Driving Score (DS), which is the main metric of the CARLA Leaderboard. It is calculated as the product of the route completion percentage and the infraction penalty, and thus gives an overall indication of the autonomous agent's driving performance. Since these metrics are computed as averages over all the routes in the benchmark, they might not always satisfy $DS = RC \cdot IS$ exactly.

Town05

Prakash *et al.* [64] introduced the Town05 Long benchmark in their first paper on TransFuser. In this benchmark, Town05 is selected for evaluation due to the large diversity in road layouts compared to other towns in the CARLA simulator, such as multi-lane roads, single-lane roads, bridges, highways, underpasses, and exits. The Town05 benchmark consists of 10 long routes with length 1000-2000 meters, each with 10 intersections.

Longest6

In their second paper on TransFuser Chitta *et al.* [5] introduced the Longest6 benchmark. To counteract the imbalance of routes per town in the CARLA Leaderboard, Longest6 uses the six longest routes per town chosen from a total of 76 routes across six towns. The result is 36 routes with an average length of 1.5 kilometers, which is similar to the average route length of the CARLA Leaderboard. The benchmark additionally ensures high-density traffic, uses unique combinations of weather and lighting conditions for each route, and includes some of CARLA Leaderboard's adversarial scenarios. Note that the authors chose to omit stop sign infractions for this benchmark because they did not observe any infractions of this kind in their testing.

3.10.2 Running the Benchmarks

The implementations of our CARLA evaluation experiments can be found in ^{ex}`ex-transfuser-0.9.10`, ^{ex}`ex-transfuser-0.9.14`, ^{ex}`ex-interfuser-0.9.10`, and ^{ex}`ex-interfuser-0.9.14`. Upon execution, it asks the user which model to evaluate, which evaluation to run (see Section 3.10.1) and, for the experiments using CARLA version 0.9.14, which sensor configuration to use (original TransFuser/InterFuser or Kia-like). The script then creates an output directory containing information about the specific evaluation run, and starts two containers using Docker Compose; one for the CARLA server and one for the trained autonomous agent. This is all done in a `screen`-session so that the process is detached from the SSH session. It also enables the easy execution of multiple evaluation runs at once on different `screen`-sessions.

By defining restart policies in the Docker Compose configuration, we ensure that the evaluation restarts in case of either the server or the client crashing, a problem that often occurred during our runs. The evaluations are resumable, which means that they can be resumed at the current route progress when it crashes instead of restarting from the first route. The process continues until all routes are completed or timed out due to the agent being stuck. The final results of the evaluation are stored in the output directory.

3.10.3 Results Analysis

There are two ways to analyze agents' driving proficiency in CARLA. The first is the output JSON file that contains results such as how many routes were completed, specific infractions per route, and the quantitative metrics described in Section 3.10.1 (DS, IS, etc.). These numbers are reported as is.

The other way is to visually inspect the driving during evaluation. Both TransFuser and InterFuser output visualization images at a given frame interval during the evaluation process. For TransFuser these visualizations include the RGB camera views, the predicted segmentation and depth views, and top-down views containing the predicted map, waypoints, and bounding boxes. InterFuser outputs the RGB camera views and a top-down view of the predicted map, waypoints, and bounding boxes. We convert the frame-by-frame visualization images into videos to enable simple qualitative analysis.

3.11 Evaluation on Real-World Data

In addition to simulator-based benchmarks, we perform evaluations using real-world data recorded using NAPLab's modified Kia e-Niro. By combining data from the various sensors on the vehicle, a full frame containing the necessary sensory inputs can be constructed and used to predict steering data using TransFuser and InterFuser. However, multiple pre-processing steps are required to successfully process the inputs, due to significant differences in the domain of the real world sensors as compared to the CARLA sensors. In addition, a new set of metrics that can be computed using the available data are necessary. This section explains the methods used for data collection, processing, and evaluation on real-world data.

3.11.1 Data Collection

To gather the required data, NAPLab's Kia e-Niro was used. As described in Section 2.3.1, the vehicle is equipped with a variety of sensors, including cameras, LiDARs, and GNSS receivers. Using NVIDIA Drive, data is recorded simultaneously from the cameras, GNSS receivers, and the vehicle's CAN bus, and stored to disk along with timing information for each sensor measurement.

Due to technical limitations, the LiDAR data is recorded separately on another computer using Ouster Studio 4.2.1. A consequence of this is that the LiDAR recording is not synchronized to the sensors recorded by NVIDIA Drive, as both the human action of starting two separate records introduces delays, as well as possible offset between the two computers' clocks. Thus, the LiDAR must be manually synchronized with the other sensors after the recording session. This is done by visually inspecting the video stream from the cameras to find the time of first movement t_c, and correspondingly inspecting the visualized LiDAR point cloud to find the time t_l when this movement is first seen. The difference $dt = t_c - t_l$ is the delay with which the LiDAR recording was started, and is used to synchronize the LiDAR during later processing. This method is assisted by the tooling in ^{ex}`ex-kia`.

In practice, lateral motion is easier to detect visually than longitudinal motion, due to the large effect rotation has on points far away. As a note to future experimenters, this makes synchronization easier when the recorded trip starts with a sharp turn instead of forward acceleration.

After a complete drive, the recorded data are transferred to `nap02`, where they are later processed offline.

3.11.2 Data Processing

This section describes the required pre-processing applied to each sensor, how complete sensor-input-frames are built to present a single instant in time to the agents, and details adjustments to the agents themselves which are required to correctly process real-world data.

Due to the AI agents being developed with CARLA as a Simulator-in-the-Loop (SITL) in mind, some of the data pre-processing steps performed in the agents are highly CARLA-specific. To successfully evaluate the agents using real-world recordings, the data therefore has to be presented to the agent in such a way that the resulting data after the agent's internal processing correctly matches the agents assumptions. In some cases, this might result in superfluous processing. For instance, in the case of LiDAR data, the agent internally flips one axis to compensate for CARLA's left-handed coordinate system. Even though the real-world LiDAR data already matches the agents' final representation, the corresponding axis must also be flipped before being presented to the agent, to ensure that the final result after the agents' internal transformation is correct.

	Cameras				Other			
	Left	Front	Right	Focus*	LIDAR	Yaw	GNSS	Waypoint
Cam C1								
Cam C2								
Cam C3		x		x				
Cam C4								
Cam C5								
Cam C6								
Cam C7	x							
Cam C8			x					
Lidar 1					x			
Lidar 2								
Lidar 3								
GNSS 1						x	x	x
GNSS 2						x	x	x
CAN bus								

Table 3.3: Overview of which real-world sensors (rows) are used to form inputs to agent sensors (columns). ***Focus**: Only used by InterFuser.

As described earlier, the TransFuser model requires the following input data during inference:

1. Three RGB images of size 960×480, directed left, right, and forward.
2. A LiDAR point cloud covering the half-sphere in front of the car.
3. A IMU measurement, in which only the yaw is used.
4. A GNSS measurement, specifically position in geodetic coordinates and velocity in the forward direction.
5. A waypoint in CARLA world coordinates.

The InterFuser model is not much different, only using 800×600 as the resolution for the RGB images.

The sensors available on the Kia e-Niro have also been previously described, but in summary they include:

1. Eight RGB cameras with resolution 1920×1208, of which two are facing left, two are facing right, three are facing forward, and one is facing backwards.
2. Three LiDAR sensors, of which one covers the half sphere in front of the car, one covers the right and the back, and one covers the full sphere.
3. Two GNSS receivers, which measures geodetic position, velocity, and heading.
4. CAN bus, the car's internal data bus, which includes data such as speed, steering angle, and throttle and brake pedal actuations.

Table 3.3 summarizes the chosen mapping from real-world sensors to the agent input requirements. Note that some sensors like the GNSS receivers are utilized multiple times, while others like CAN bus are not used as input at all. This specific mapping is chosen to as close as possible resemble the sensor configuration used to train the agents in CARLA. Continuing, we describe the necessary processing for each sensor.

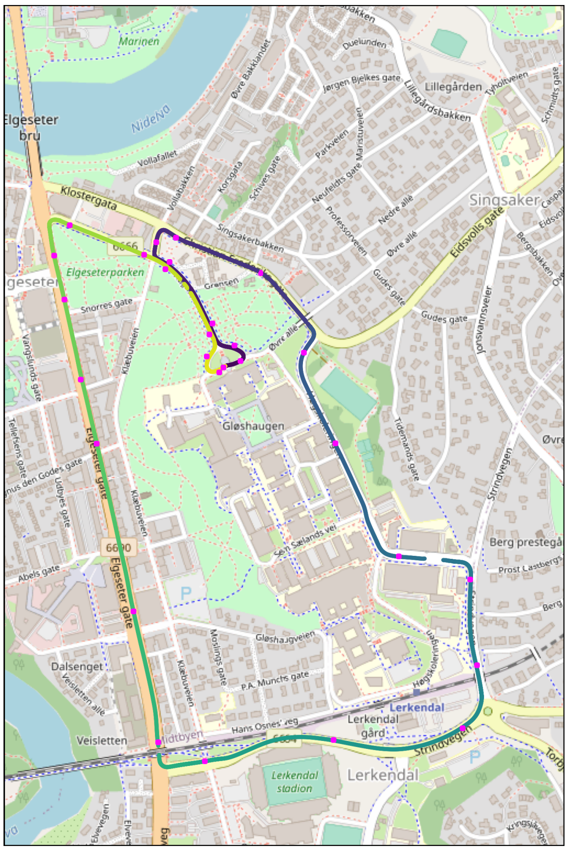
Generating Waypoints

Since both agents need frequent waypoints as input, a simple algorithm has been designed to sample waypoints from the GNSS track. Similar to the waypoints used in the CARLA benchmarks described in Section 3.10, we sample points from the GNSS log at least every 50 meters, in addition to waypoints directly after intersections or lane changes. The latter waypoints are necessary to give the model information about which direction to turn in intersections and when to change lanes. However, due to the lack of labeled data containing information about lane changes and intersections, these waypoints must be created manually, while the regularly spaced waypoints can be generated automatically.

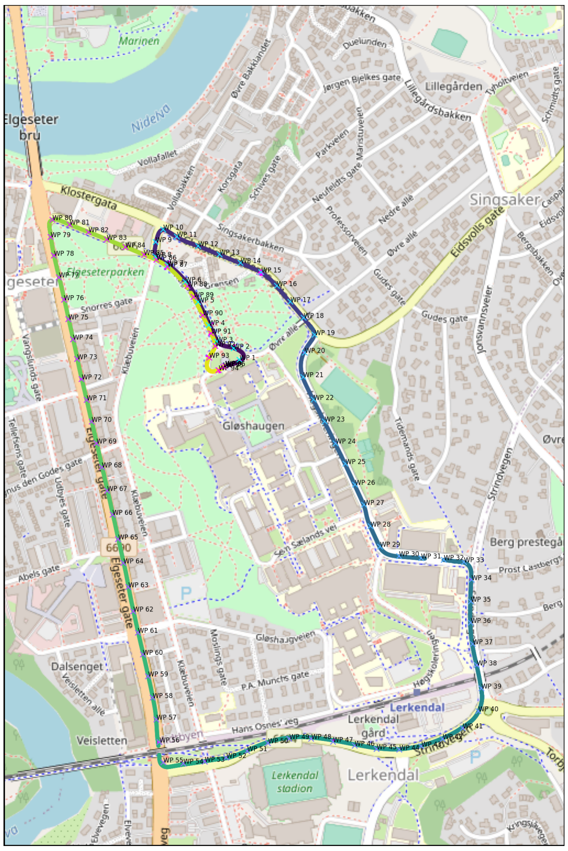
To facilitate this planning, the tooling located in ^{ex}`ex-kia/make-plan` enables loading, plotting, and waypoint generation using recorded GNSS data. A Jupyter notebook provides instructions and code to facilitate the following workflow:

1. Loading a GNSS log and saving a plot of the driven route to the file `fig.png`.
2. Manually, using an image editing program, load `fig.png` and draw pink dots where the waypoints should be placed, saving the file as `plan.png`.
3. Loading `plan.png` into the notebook and detecting the pink dots.
4. Generating waypoints according to the pink dots, snapping to the closest GNSS measurement. Waypoints are also inserted at least once every 50 meters driven.
5. Generating the file `plan_annotated.png` to illustrate the final plan.
6. Shifting all coordinates to be centered at the location (0°N, 0°E) in accordance with CARLA's conventions.
7. Exporting the final plan to `plan.csv` for later use during evaluation.

Figure 3.23a) shows the plot of the GNSS track with pink dots as described in step 2. The final annotated plan produced in step 5, with both manually and automatically generated waypoints, can be seen in Figure 3.23b). Step 6 displaces these waypoints such that they are centered on (0°N, 0°E). This is required by CARLA's choice of coordinate system transformations, which we describe next.



(a) `plan.png`. Pink dots are drawn on top of the route to manually place waypoints at the given locations.



(b) `plan_annotated.png`. The final plan generated with additional waypoints every 50 meters.

Figure 3.23: Products of the waypoint generation procedure.

Coordinate Transformations

CARLA inherits Unreal Engine's choice of a left-handed coordinate system, with axes being defined as x-forward, y-right, and z-up. [12]

The coordinate system is anchored to the real world by defining the origin to correspond to the crossing of the Equator and the Prime Meridian at 0° N, 0° E and 0 meters altitude, with the x-axis pointing east, the y-axis south, and the z-axis up. To transform between Cartesian and geodetic coordinates, CARLA uses the Mercator projection. While other coordinate transformations such as the Earth-centered Earth-fixed frame (ECEF) and local tangent planes such as East-North-Up (ENU) are better suited to handle arbitrary coordinates on Earth, CARLA's choice of a simple Mercator projection is justified by the projection's linearity and small distortion near 0° N 0° E.

The transform from geodetic coordinates to Mercator is given in [97], and CARLA's implementation found at [98]. In summary, CARLA defines the Earth's radius to be $a = 6\,378\,137.0\,\text{m}$, and then computes the East and South coordinates in meters from the geodetic latitude ϕ, longitude λ, and altitude h as follows:

$$\mathbf{p}_{CARLA} = \begin{bmatrix} E \\ S \\ U \end{bmatrix} = \begin{bmatrix} (a\cos\phi)\lambda \\ -(a\cos\phi)\log\left(\tan\left(\frac{2\phi+\pi}{4}\right)\right) \\ h \end{bmatrix} \tag{3.1}$$

As mentioned, the Mercator projection is essentially linear near the origin, due to the following approximations when ϕ is small:

$$\cos\phi \approx 1$$

and

$$\tan\left(\frac{2\phi+\pi}{4}\right) \approx \exp(\phi).$$

Thus

$$\log\left(\tan\left(\frac{2\phi+\pi}{4}\right)\right) \approx \phi,$$

which finally gives

$$\mathbf{p}_{CARLA} = \begin{bmatrix} E \\ S \\ U \end{bmatrix} = \begin{bmatrix} a\lambda \\ -a\phi \\ h \end{bmatrix} \tag{3.2}$$

Since CARLA's GNSS sensor returns geodetic coordinates in degrees, not radians, this must be taken into account. The effective scaling factor is therefore $a' = \pi a/180 = 111319.49$, which is the value found in TransFuser and InterFuser's coordinate conversion routines [99, 100]. However, their conversion routines apply a slightly different scaling factor to latitude than longitude, which

[12]This has made a lot of people very angry and been widely regarded as a bad move, causing Unreal Engine creator Tim Sweeney to publicly apologize for the inconvenience. [96]

indicates that the derivation might be different. The disrepancy is about 0.3%, which causes a difference of only 3 millimeters per meter, thus there is no need to worry about the accuracy of the transformation in practice.

Unfortunately, the assumption that all coordinates are close to 0° N 0° E fails to apply to data recorded in the Trondheim area, which is located at 63.43°N, 10.39°E. Thus, the coordinate transformation built in to TransFuser and InterFuser generates wrong results, resulting in confusion in the agent's current position and direction vector towards its waypoints. As an example, consider an agent driving eastward in Trondheim from coordinates A = 63.4319649°N, 10.3918333°E to coordinates B = 63.4319743°N, 10.3938415°E. In reality, this corresponds to a distance of 100 meters. However, using the simplified transformation above yields different results:

$$
\begin{aligned}
\mathbf{A}_{CARLA} &= \begin{bmatrix} 111319.49 \times 10.3918333 \\ -111319.49 \times 63.4319649 \end{bmatrix} = \begin{bmatrix} 1156813.58 \\ -7061213.98 \end{bmatrix} \\
\mathbf{B}_{CARLA} &= \begin{bmatrix} 111319.49 \times 10.3938415 \\ -111319.49 \times 63.4319743 \end{bmatrix} = \begin{bmatrix} 1157037.13 \\ -7061215.03 \end{bmatrix} \\
||\mathbf{B}_{CARLA} - \mathbf{A}_{CARLA}|| &= \left\| \begin{bmatrix} 223.55 \\ -1.05 \end{bmatrix} \right\| = 223.55\,\text{m}
\end{aligned}
$$

The calculations show that in the east-west direction, distances are scaled by more than a factor of two, leading to incorrect results.

To handle this issue without invasive modifications to the agents themselves (which could render them incompatible with simulator-provided GNSS measurements), the GNSS data from the Kia is instead preprocessed and shifted in space to be centered on the CARLA origin 0°N, 0°E. This is performed using the `geodetic2ned` transformation from the `pymap3d` [101] Python package. Specifically, all points in the GNSS log are transformed into North-East-Down (NED) coordinates relative to the mean GNSS position. This results in the desired CARLA world coordinates in meters, centered on $(0, 0)$. Since the agents require the coordinates in the CARLA geodetic coordinate system, the north and east axes are simply divided by the scaling factor a'.

Estimating Yaw

To fulfill the yaw input requirement, we estimate the vehicle's heading as the tangent vector to the GNSS track. Specifically, we find at least three points behind the car with a distance greater than 0.5 m, and at least three points ahead of the car with the same requirement. Then we calculate the vector between the mean positions of these two sets to find a central-difference approximation to the tangent vector at the current position. This is done in the CARLA world coordinate system so that we directly have the movement components in the east and north directions. The compass heading is then calculated as

$$\theta = \arctan(\Delta E, \Delta N).$$

Due to the yaw being used to transform the current waypoint location into the car and LiDAR's local frame of reference, errors in the yaw estimate directly affect the position of the waypoint as seen from the car, which manifests as errors in the agent's predicted steering angle. Therefore, the requirement of at least three points in the front and back buffers serves to reduce noise in the GNSS measurements.

Other attempts were made using only a single point behind the car and the current position, or a single point behind and in front of the car. These were excessively noisy, causing visible errors in the waypoint location when overlaid on the LiDAR data. The GNSS sensors themselves also provide heading estimates, but these also showed too high variance, in addition to lagging behind temporally. Our method has the advantage of avoiding a phase delay with respect to the car movement, due to the usage of a central difference approach versus a backward difference. However, this does rule out the ability to run the algorithm online while driving the car, but the scheme can be replaced in future work if necessary. Suggestions for future work include the kinematic bicycle model and the Kalman filter, but these were discarded due to the sufficient accuracy of our simple filter.

Reading CAN Bus Data

Among the data collected by NVIDIA Drive is also a log of the vehicle's CAN bus, which contains data such as steering wheel angle, throttle and brake actuations, and vehicle speed. This information is not used as input to the model, but rather as a ground truth comparison for the model's outputs. The recorded CAN bus data is encoded using a proprietary NVIDIA format which is partially documented in [102]. Specifically, the files contain binary data consisting of a 32-byte header, followed by an arbitrary number of messages of equal fixed size N.

The header consists of the 32-bit unsigned magic number `0xc5eeeaf2`, then a signed 32-bit file version number which is either 1 or 2. The magic number can be used to detect and correct for machine endianness mismatch. The message length N is 22 bytes for file version 1, and 78 bytes for file version 2. Each message begins with a 14-byte header, containing first an 8-byte signed UNIX timestamp in microseconds, then a 4-byte unsigned message type ID, followed by a 2-byte unsigned payload length less or equal to $N - 14$. In the case of smaller payloads, the rest of the message is padded with zeros up to length N.

The CAN message payload is also proprietary and vehicle-specific. Fortunately, CommaAI has published information detailing CAN bus messages for Hyundai Kia vehicles, which can be found in [103]. Table 3.4 summarizes relevant CAN

ID	Content
688	Steering wheel angle
881	Throttle and brake pedal position
902	Individual wheel speed
1322	Vehicle speed

Table 3.4: Relevant CAN message IDs and their content.

message IDs. We refer to ex`ex-kia/app/src/nap/kia/canbus.py` for specifics on decoding the message payloads. In this work, the steering wheel angle and vehicle speed is used as ground truth data for comparing model predictions, while the two other messages are currently unused, but may be interesting for future work.

LiDAR Transformation

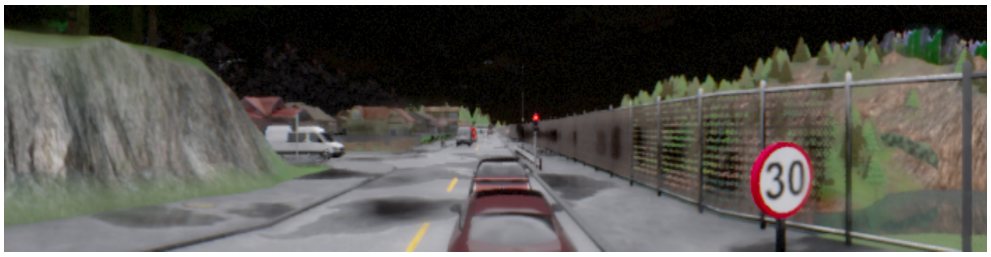
The LiDAR data are loaded using the Ouster SDK [104]. Point clouds are converted to a list of 3D Cartesian points, after which the X- and Y-axes are swapped to match CARLA's conventions.

Camera Transformation

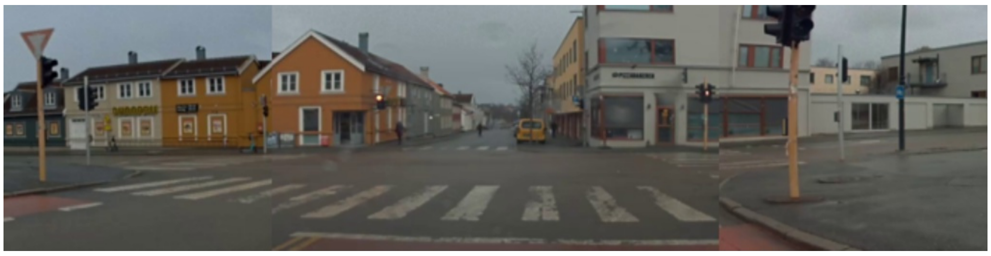
As described in Section 3.5, the TransFuser agent takes three separate camera images as input, which it stitches internally to form a single wide-angle image. Due to the differing mounting positions and properties of the physical cameras mounted on the Kia, the real-world images cannot be directly used as input to the agent without resulting in a warped wide-angle image with obvious discrepancies at the seams. The images must therefore be scaled and cropped so that the internal stitching produces similar results to what the agents experience when driving in CARLA.

Suitable scale and crop parameters were selected manually using the script ex`ex-kia/adjust_crop.py`. The tool facilitates simple visualization and adjustment of the parameters of the final stitched image, resulting in a set of parameters that produce suitable stitches with minimal discrepancies. Figure 3.24 shows a comparison of the final stitched image when running in CARLA versus when evaluating on real-world data. The three cameras `C7_L2`, `C3_tricam120`, and `C8_R2` were chosen due to their similarity to the original sensor configuration in CARLA, with cameras pointing left, forward, and right. The particular crop parameters were chosen to provide a similar field of view to the original TransFuser agent, while minimizing the discontinuities at the seams.

InterFuser, on the other hand, does not perform any stitching, and instead presents the images separately to the model. The only pre-processing applied is to downscale the images to 800 by 600 pixels so that they match the image size the model was trained on.



(a) Stitched RGB input to TransFuser from CARLA cameras.



(b) Stitched RGB input to TransFuser from Kia cameras.

Figure 3.24: Comparison of the stitched RGB input to TransFuser using CARLA vs. real data. Note how the stitches in **(a)** is less noticeable than in **(b)**, due to the cameras in CARLA being positioned exactly in the same position, enabling seamless stitches. The same is not possible for the Kia images, where the cameras are mounted at different positions.

Sensor Synchronization

Although all sensors on the Kia except for the LiDAR are timestamped using the same clock source, they generally do not produce frames at the same rate. For instance, the cameras produce data at 30 Hz, but the GNSS sensors only produce data at about 10 Hz, and only when GNSS coverage is sufficient. The LiDAR also has a frame rate of 10 Hz, however timestamped using a difference clock source; thus, the values are offset from the other sensor timestamps.

To ensure that the agents only look at sensor data from a specific instant in time, there are two different approaches to choose from. Either evaluate the agent at the highest frame rate, that is, on each camera frame. This requires either some sort of nearest-neighbor selection or interpolation of frames from the other sensors. Due to rare, but long, gaps in the GNSS data occurring when the car drives under bridges or otherwise without a clear view of the sky, the nearest-neighbor selection may fail to find measurements close enough in time. Interpolation could in principle work for GNSS data, but this is not possible for LiDAR data. Therefore, the chosen approach is to evaluate the model at the lowest and most irregular frame rate, that is, that of the GNSS. For each GNSS frame, the closest frame in time for each other sensor is found and used as input to the model. The method can be seen in Algorithm 2.

Algorithm 2 Synchronization of sensors when processing real-world data.

```
repeat
    advance $GNSS$
    $t \leftarrow GNSS.timestamp$
    $\Delta t \leftarrow 0$
    for all sensors $s$ do
        advance $s$ to $t_s$ minimizing $\Delta t_s = |t_s - t|$
        $\Delta t \leftarrow \max(\Delta t, \Delta t_s)$
    end for
    if $\Delta t < 100\,\text{ms}$ then
        run inference
    end if
until end-of-stream
```

Agent Adjustments

In addition to data processing, two changes have been applied to TransFuser's configuration. The first is to reduce the agent's `action_repeat` configuration option from 2 to 1, to ensure that the agent actually performs model inference every time it receives input data, instead of skipping every other frame as a performance optimization. The second change is to reduce the agent's `gps_buffer_max_len` configuration option from 100 to 1, to effectively disable the agent's internal GNSS denoising. The denoising algorithm is dependent on an internal model of the car's dynamics, which is only accurate within CARLA and not applicable to the Kia. The algorithm therefore produces incorrect GNSS position estimates when applied to real-world data, so we disable it.

The same `action_repeat` change has been applied to InterFuser, ensuring model inference every time it receives input data. It does not use GNSS denoising or any other functionality that could impact real-world evaluation.

3.11.3 Evaluation Criteria

Due to real-world evaluation being off-policy, that is, it is not the agent that actually controls the car, the metrics used in the simulator benchmarks are not suitable. For instance, Route Completion is meaningless when the route is guaranteed to fully complete, the agent has no influence on whether the route is completed or not. We therefore create new evaluation metrics that can be calculated using the available recorded data, such as GNSS tracks and CAN bus data. In addition to quantitative metrics, we inspect the agents' auxiliary outputs and determine the quality of these in various situations. For instance, we look at TransFuser's segmentation and HD map predictions, as well as both agents' bounding box predictions.

Due to differences in agent behaviour, not all metrics are shared between both agents, and some metrics do not share the same semantics. The most comparable metrics are the ones based on the final outputs from the agents: the steering angle and throttle. However, these final values are not orthogonal to each other: TransFuser, for instance, zeroes the steering output when it decides to brake. This causes the steering value to be adversely affected by the agent's poor speed control. Therefore, we hook into agent's prediction code and collect the pure steering angle before it is modified by other factors, to be able to evaluate these metrics independent from each other. In a similar manner, both agents internally predict a *desired speed*, which is passed through a PID-controller to produce the final throttle output. The throttle output is therefore directly hampered by the mismatch between the real-world and the CARLA speeds for which the PID-controller is tuned for, therefore we instead extract the internal desired speed and evaluate based on this instead.

The predicted steering angles and speed control are both evaluated by comparing with the ground truth steering angle and speed available from the CAN bus. We quantify the similarity using the sample Pearson correlation coefficient (PCC) ρ [105], which is a normalized measure of the covariance between pairs of samples. For two stochastic variables X and Y, PCC is calculated as

$$\rho_{X,Y} = \frac{\mathbb{E}[(X-\mu_X)(Y-\mu_Y)]}{\sigma_X \sigma_Y} \tag{3.3}$$

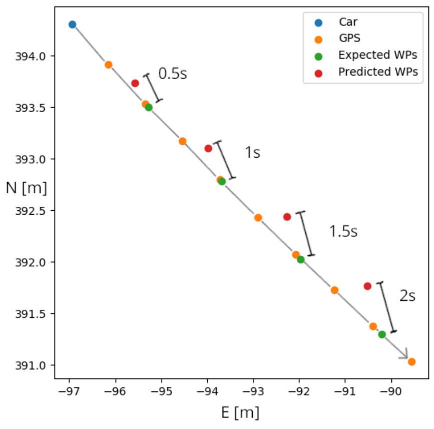
which for the discrete case becomes

$$\rho_{x,y} = \frac{\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i-\bar{x})}\sqrt{\sum_{i=1}^{n}(x_i-\bar{x})}}. \tag{3.4}$$

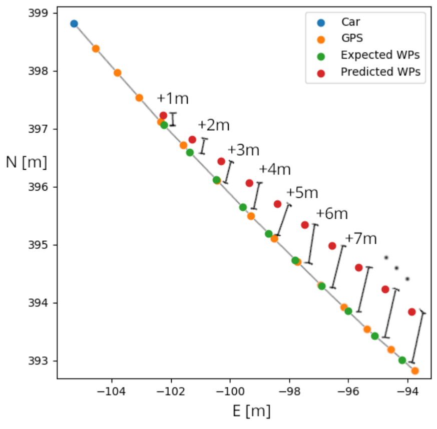
A normalized metric is desirable due to incompatibilities in the data compared. Steering angles from the agent correspond to a desired rotation of the vehicle's orientation in the world, while the CAN bus steering wheel angle corresponds to a rotation *rate*. It is therefore meaningless to measure an absolute difference between the predicted and ground truth values, but it is still expected to see a correlation between large rotation angles and high rotation rates.

The speed predictions also face incompatibilities with real-world data, since the agents were never trained to drive faster than 4 m/sec. Thus, PCC is suitable due to being invariant to the scale of the predicted speeds, while still capturing correlation when speeding up or slowing down. Fully random agents with no correlation with real-world data would show $\rho = 0$, while perfect agents would show a high correlation with real-world data, corresponding to $\rho \approx 1$.

In addition to these two scalar outputs, we also evaluate the average distance of each of the agent's predicted waypoints to some chosen ground truth waypoints. TransFuser predicts $n = 4$ waypoints corresponding to half-second time steps ahead which are compared to ground truth waypoints queried from the



(a) Ground truth waypoints for TransFuser are determined by interpolating the GNSS samples at fixed time steps 0.5 s, 1 s, 1.5 s and 2 s into the future.



(b) Ground truth waypoints for InterFuser are determined by interpolating the GNSS samples at fixed spatial steps with 1 m intervals into the future, with the first point about 3.5 m ahead.

Figure 3.25: Illustrating the quantification of waypoint errors. Each waypoint's distance to the corresponding ground truth waypoint is measured for each frame in the evaluation sequence.

GNSS log at times 0.5, 1, 1.5 and 2 seconds into the future, as illustrated in Figure 3.25a. InterFuser, on the other hand, predicts $n = 10$ waypoints corresponding to 1 meter spatial steps ahead, with the first point about 3.5 m ahead on average. These are compared to ground truth waypoints queried from the GNSS log at the corresponding spatial distances, as illustrated in Figure 3.25b. The error ϵ_i for $i = 1 \ldots n$ is calculated as the Euclidian distance between the predicted and ground truth waypoint, and the distribution of the mean scaled waypoint error $\frac{1}{n}\sum_{i=1}^{n}\frac{\epsilon_i}{i}$ is reported.

Chapter 4

Experiments and Results

This chapter showcases the results of applying the methodology in Chapter 3 in an attempt to answer the research questions of the thesis. Each section consists of setup, results, and discussion for an individual experiment, while a broader discussion is given in Chapter 5. Selected evaluation runs in CARLA are uploaded to YouTube[1], while the real-world evaluation videos are available in the attached document or on request to NAPLab. Table 4.1 shows an overview of the models evaluated in this chapter.

Model	Dataset	Trained by us	Description
TF-paper	Original	-	Results from TransFuser's paper [5]
TF-expert	-	-	TransFuser's expert agent used for data generation
TF-pretrained	Original	-	Pre-trained TransFuser weights [88]
TF-ds-orig	Original	✓	Trained on the provided original dataset
TF-ds-feb	ds-feb	✓	Trained on the generated ds-feb dataset
TF-ds-kia	ds-apr	✓	Trained on the generated Kia-like ds-apr dataset
IF-paper	Original	-	Results from InterFuser's paper [6]
IF-expert	-	-	InterFuser's expert agent used for data generation
IF-pretrained	Original	-	Pre-trained InterFuser weights [91]
IF-ds-custom	ds-custom	✓	Trained on the generated custom-ds dataset
IF-ds-kia	ds-kia	✓	Trained on the generated Kia-like kia-ds dataset

Table 4.1: The TransFuser (TF) and InterFuser (IF) models discussed in this chapter. Details regarding the different TransFuser and InterFuser datasets can be found in Section 3.8.2 and Section 3.8.3, respectively.

Experiment 1 aims to answer **RQ1** and **RQ2** by reproducing the results reported by TransFuser's and InterFuser's authors using the Longest6 and Town05 benchmarks. We evaluate both pre-trained and custom-trained models in CARLA versions 0.9.10 and 0.9.14. Experiments 2 to 4 are smaller and investigate different factors that can affect the evaluation results in simulator benchmarks, and also

[1]`https://www.youtube.com/playlist?list=PLWGDO2Z_62kKDze_p2hgt8DFMOe3VElgA`

the variance between different evaluation runs. These experiments are a consequence of Experiment 1's result and discussion, and aim to help understand our results and how our methods might be improved. Next, to answer **RQ3**, **RQ4**, and **RQ5**, Experiment 5 tests the agents in the simulator with a sensor configuration that matches NAPLab's Kia e-Niro, before Experiments 6 and 7 finally evaluate the agents using real-world data from two different trips in Trondheim. We will then discuss the research questions directly in Chapter 5.

4.1 Experiment 1: Reproducing the Fusers

In this first experiment, we aim to reproduce the results for TransFuser and InterFuser as reported by Chitta *et al.* [5] and Shao *et al.* [6], using both the Longest6 and Town05 benchmarks in CARLA version 0.9.10.

In addition to evaluating the original model weights, we evaluate new weights trained on both original and custom-generated datasets, to examine the success of the data generation procedure and overall reproducibility of the original results. We also run these evaluations in CARLA 0.9.14, to measure the impact of this new environment and to determine whether the version transition was successful.

4.1.1 Setup

This experiment will perform simulator evaluations using the method described in Section 3.10. We evaluate multiple TransFuser and InterFuser models, specifically:

1. The pre-trained models (TF-pretrained and IF-pretrained) available online [88, 91]. These are retrieved by running the `setup.sh` script for the relevant evaluations, for instance the one found in $^{\text{ex}}$`ex-transfuser-0.9.10`.
2. A re-trained TransFuser model (TF-ds-orig) on the original dataset available online [88]. The dataset is downloaded and used to train a model as described in Section 3.9.1, using the tooling in $^{\text{ex}}$`train-transfuser`.
3. New TransFuser and InterFuser models (TF-ds-feb and IF-ds-custom) trained on the re-created datasets. The datasets are created as described in Section 3.8, after which the training method described in Section 3.9 is employed.
4. The expert agents (TF-expert and IF-expert) used to generate training data for TransFuser and InterFuser. These are chosen by selecting the `"autopilot"` option in the dialog when starting the evaluation.

Note that the original dataset for InterFuser is not available, thus we cannot perform an evaluation similar to (2) for InterFuser. Also note that the new datasets in (3) are created using CARLA version 0.9.14, thus we do not evaluate these models in 0.9.10. Due to the length and time requirements of the Longest6 benchmark, we only select the pre-trained agents for evaluation. The rest of the agents are compared using the shorter Town05 benchmark.

4.1.2 Results

We present the results in three parts, first the output from the training, then quantitative metrics of the evaluation runs, and finally take a look at the visualizations produced during evaluation. The benchmark results are obtained by running each evaluation three times and calculating the mean of each metric.

Training

The training loss for the re-trained TransFuser model on the original dataset, as well as the new models on the new datasets, is shown in Figure 4.1a. We observe a loss minimum at about epoch 7 for the model trained on the original dataset, and at about epoch 14 for the model trained on our new ds-feb dataset. After this minimum, both training runs show increased loss until the reduced learning rate kicks in after epoch 30. Both training runs end with a higher loss than the global minimum.

As seen in Figure 4.1b, the training loss for the re-trained InterFuser model on our new ds-custom dataset shows an even decrease in loss throughout the run, plateauing somewhat after 400k iterations, corresponding to about 23 epochs.

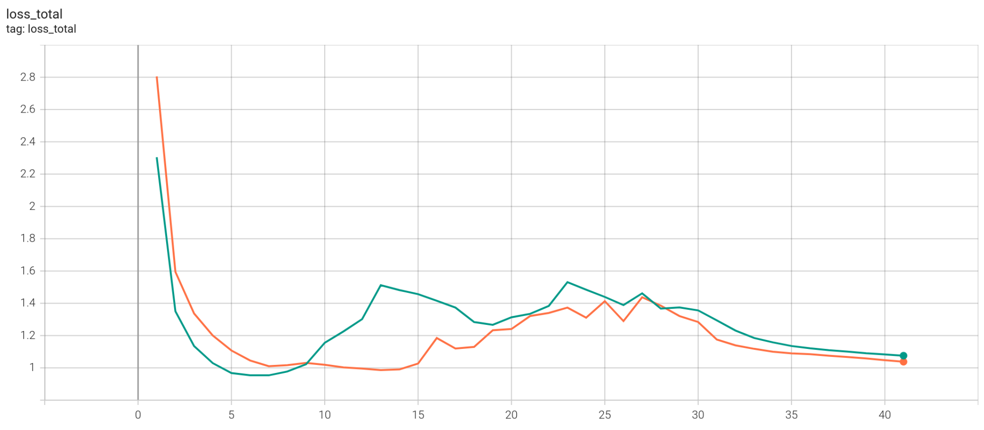
Quantitative Evaluation Results

We start with a comparison of the pre-trained agents using the Longest6 benchmark, with results shown in Table 4.2a. We note that in CARLA 0.9.10, the pre-trained TransFuser agent achieves results close to those reported by Chitta *et al.* [5], with a slightly lower Route Completion (RC) score. The Collision with vehicles (Cv) metric is significantly reduced, while Agent Blocked (AB) is increased. The pre-trained InterFuser agent achieves a slightly lower score than TransFuser, and shows elevated Collision with layout (Cl) and Route Timeout (RT) penalties. On the other hand, it has fewer collisions with vehicles (Cv) than TransFuser.

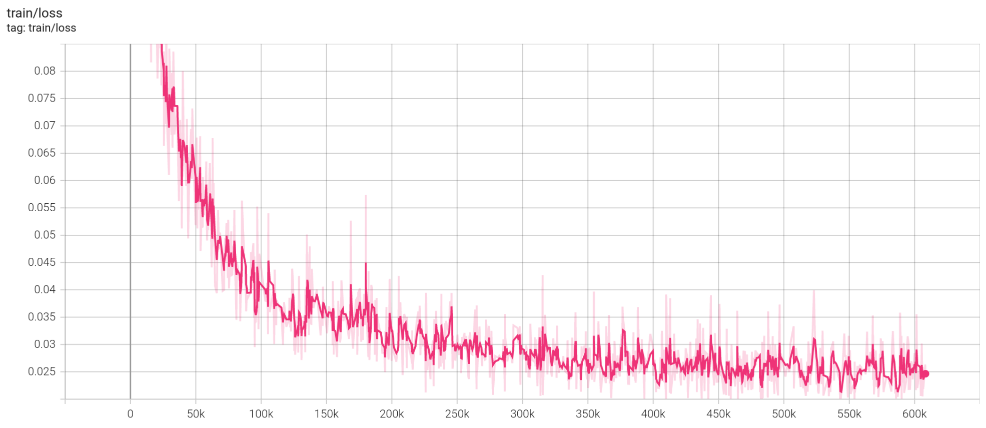
In CARLA 0.9.14, all agents' Driving Score (DS)s are much lower, mostly due to reduced Route Completion. We also see an increased rate of vehicle collisions (Cv) and Agent Blocked (AB) for all three agents. The pre-trained InterFuser agent has the highest DS due to its overall lower infraction penalties.

Continuing with the Town05 benchmark, additional models join the evaluation, and the scores are shown in Table 4.2b. In CARLA 0.9.10, we see that both expert agents perform well, with TF-expert having near-perfect scores. The pre-trained TransFuser model has a significant lead over the pre-trained InterFuser model. Additionally, we see that TF-ds-orig also achieves 100% route completion, although with a lower Infraction Score (IS) than TF-pretrained due to an increased rate of vehicle collisions (Cv). Note that the TransFuser agents also beat the scores reported by Shao *et al.* [6], while IF-pretrained does not.

The scores in 0.9.14 are lower for all agents, including the expert agents. We



(a) Training loss for TF-ds-orig (green) and TF-ds-feb (orange). The x-axis indicates the training epoch.



(b) Training loss for IF-ds-custom. The x-axis indicates the training step. One epoch corresponds to about 17400 steps, with 35 epochs in total.

Figure 4.1: Training loss curves for TransFuser and InterFuser in Experiment 1.

see that the pre-trained TransFuser model still has a lead over the pre-trained InterFuser model. TF-ds-orig has a larger drop in performance than TF-pretrained due to more vehicle collisions. For the agents trained on re-created datasets, we see that TF-ds-feb achieves better scores than TF-ds-orig due to fewer infraction penalties. IF-ds-custom is the least performant model in this benchmark, showing a significantly worse IS due to having a higher vehicle and pedestrian collision rate while also ignoring more red lights (RL).

Qualitative Evaluation Results

A visualization of the output of the pre-trained TransFuser at the start of the Longest6 benchmark in CARLA 0.9.10 is shown in Figure 4.2. The TransFuser model shows very good semantic segmentation and depth perception performance, in addition to correctly interpreting the intersection topology and drawing a crisp map. The predicted waypoints follow a straight line towards the goal location, and the model correctly detects the bounding box of the vehicle in front. A video of this run is available on YouTube[2].

The InterFuser model also demonstrates a correct interpretation of the situation, as shown in Figure 4.3. The two vehicles in front are successfully detected and drawn in the top-down view, although the future prediction indicates that they are in motion and will move ahead. While hard to tell from a still image alone, this prediction may be incorrect due to the vehicles waiting in a queue, and thus are likely to remain still in the future as well. On the other hand, the motor-cycle in the opposite lane is correctly predicted to move past the ego-vehicle in the two future time steps. A video of the run is also available on YouTube[3].

4.1.3 Discussion

Overall, we observe higher scores in the Town05 benchmark than in the Longest6 benchmark. There can be multiple explanations to this. First, Longest6 defaults to requesting 500 background actors during the evaluation, while Town05 only requests 120 actors. The increased amount of traffic gives a higher chance of collisions with vehicles (Cv); a metric we see is much higher in Longest6 than in Town05 across all models. Second, more vehicles also produce more traffic jams, which can cause the agent to get stuck. We see in the visualization videos[4] that most of the Longest6 routes end with a route timeout (RT) because the agent is surrounded by other cars and does not move, which explains the low RC. Finally, the longer routes in Longest6 give more chances of infractions along the way, which can decrease the overall IS.

We see a drop in DS when upgrading from CARLA version 0.9.10 to 0.9.14 in both benchmarks. We assume that the main reason for this is the graphical dif-

[2] `https://youtu.be/PfOsMLSgHsw&list=PLWGDO2Z_62kKDze_p2hgt8DFMOe3VElgA`

[3] `https://youtu.be/TK3uwBmQh54?t=26&list=PLWGDO2Z_62kKDze_p2hgt8DFMOe3VElgA`

[4] `https://youtu.be/vIgbsMDi34k?t=4495&list=PLWGDO2Z_62kKDze_p2hgt8DFMOe3VElgA`

Model	Primary metrics ↑			Secondary metrics ↓							
	DS	RC	IS	Cp	Cv	Cl	RL	SS	OR	RT	AB
CARLA 0.9.10											
TF-paper	*47%*	*93%*	*50%*	*0.03*	*2.45*	*0.07*	*0.16*	-	*0.04*	*0.06*	*0.10*
TF-pretrained†	43%	83%	53%	0.01	0.75	0.06	0.04	0.12	0.03	0.04	0.25
IF-pretrained†	39%	66%	62%	0.01	0.36	0.34	0.13	0.00	0.06	0.47	0.25
CARLA 0.9.14											
TF-pretrained†	23%	52%	48%	0.05	1.51	0.08	0.04	0.08	0.07	0.03	0.71
TF-pretrained-s[1]	21%	53%	48%	0.04	1.55	0.09	0.04	0.13	0.10	0.03	0.81
IF-pretrained†	29%	49%	54%	0.03	1.32	0.19	0.14	0.03	0.33	0.70	0.30

[1] The three models in the TF-pretrained ensamble were evaluated separately and their results averaged.

(a) Longest6 benchmark results for pre-trained TransFuser (TF) and InterFuser (IF) agents.

Model	Primary metrics ↑			Secondary metrics ↓							
	DS	RC	IS	Cp	Cv	Cl	RL	SS	OR	RT	AB
CARLA 0.9.10											
IF-paper	*68%*	*95%*	-	-	-	-	-	-	-	-	-
TF-pretrained†	90%	99%	90%	0.00	0.04	0.00	0.01	0.12	0.00	0.00	0.00
IF-pretrained†	54%	79%	71%	0.01	0.13	0.00	0.01	0.00	0.00	0.13	0.00
TF-ds-orig	79%	100%	79%	0.00	0.05	0.00	0.00	0.15	0.00	0.00	0.00
TF-expert	99%	100%	99%	0.00	0.00	0.00	0.00	0.15	0.00	0.00	0.00
IF-expert	88%	100%	88%	0.01	0.01	0.00	0.01	0.00	0.00	0.00	0.00
CARLA 0.9.14											
TF-pretrained†	84%	100%	84%	0.00	0.03	0.00	0.01	0.16	0.00	0.00	0.00
IF-pretrained†	49%	72%	68%	0.02	0.07	0.00	0.06	0.00	0.00	0.19	0.00
TF-ds-orig	64%	95%	68%	0.00	0.12	0.00	0.00	0.13	0.02	0.00	0.01
TF-ds-feb	71%	88%	77%	0.00	0.07	0.00	0.01	0.09	0.01	0.00	0.04
IF-ds-custom	27%	87%	31%	0.03	0.24	0.00	0.09	0.00	0.03	0.04	0.02
TF-expert	93%	95%	98%	0.00	0.00	0.00	0.00	0.14	0.00	0.00	0.01
IF-expert	78%	100%	78%	0.02	0.02	0.00	0.00	0.00	0.00	0.00	0.00

† Videos of these runs are available at
`https://youtube.com/playlist?list=PLWGDO2Z_62kKDze_p2hgt8DFMOe3VElgA`.

(b) Town05 benchmark results for various TransFuser (TF) and InterFuser (IF) agents.

Table 4.2: Simulator benchmark results. **Primary metrics** (higher is better): Driving Score (DS), Route Completion (RC), and Infraction Score (IS). **Secondary metrics**, in units of infractions per kilometer (lower is better): Collisions with pedestrians (Cp), Collisions with vehicles (Cv), Collisions with static layout (Cl), Red light infractions (RL), Stop sign infractions (SS), Off-road infractions (OR), Route timeouts (RT), and Agent blocked (AB). Route deviations (RD) are not shown, and are zero unless noted explicitly. Dashes indicate unknown values. Agents are grouped by which CARLA version they were evaluated with.

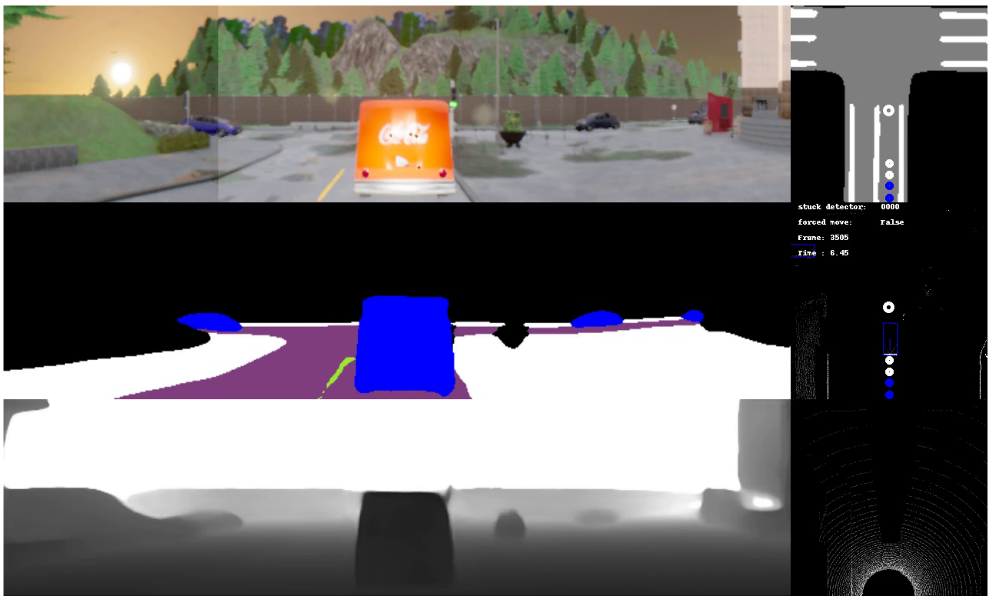
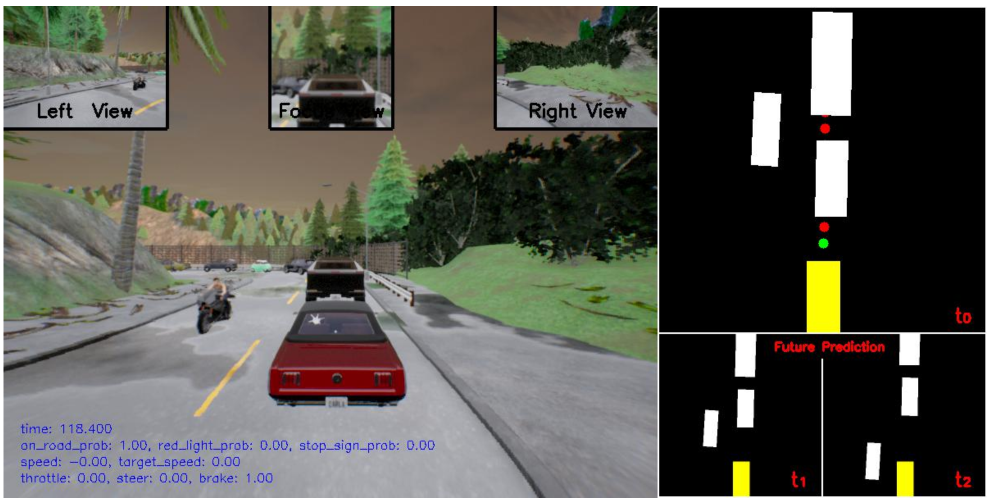


Figure 4.2: Visualization of model inputs and outputs for TF-pretrained evaluated on the first route of Longest6 in CARLA 0.9.10. **Top left:** The camera input, stitched together from the left, center, and right cameras. **Middle left:** The agent's semantic segmentation predictions. Vehicles are blue, roads purple, sidewalks white, road markings a yellow-green, and traffic lights red when they are in the red or yellow state. For CARLA 0.9.14 visualizations, traffic lights are drawn in red for all states. **Bottom left:** The agent's depth predictions. **Top right:** The agent's top-down HD-map prediction. The goal location is indicated by the larger white disk, and the agent's predicted path extends from the bottom. **Middle and bottom right:** The LiDAR data the agent receives as input, split into two images, for points above or below the ground level. The goal location is drawn on the input the agent receives, which is how it knows its target. A video of this evaluation is available at `https://youtu.be/PfOsMLSgHsw&list=PLWGDO2Z_62kKDze_p2hgt8DFMOe3VElgA`.



```
time: 118.400
on_road_prob: 1.00, red_light_prob: 0.00, stop_sign_prob: 0.00
speed: -0.00, target_speed: 0.00
throttle: 0.00, steer: 0.00, brake: 1.00
```

Figure 4.3: Visualization of model inputs and outputs for IF-pretrained evaluated on the first route of Longest6 in CARLA 0.9.10. **Left:** The camera inputs. The largest image shows the center view. In the top left and top right we have the left and right views. The top center image is the focus view, which is a middle crop of the large center view. **Right:** The agent's predicted object density map, consisting of predicted vehicle and pedestrian bounding boxes for the current and two future time steps, as well as predicted waypoints for the current time step. The waypoints are green if the path is predicted to be clear, and red if the model thinks there are objects in the way. **Bottom left text:** The agent's predicted traffic rule information including whether the car is on the road, if it predicts a red light, and if it predicts a stop sign ahead. It also shows the current throttle, steering, and braking commands, as well as the current and target speed. A video of this evaluation is available at `https://youtu.be/TK3uwBmQh54?t=26&list=PLWGDO2Z_62kKDze_p2hgt8DFMOe3VElgA`.

ferences between the two versions, as discussed in Section 3.7. The increased fidelity and improved lighting create images with perhaps less clear distinctions of the scene, especially in darker scenarios. Additionally, as mentioned in Section 3.7.2, we saw several examples of graphical glitches during data generation and evaluation in CARLA version 0.9.14. We can, for example, compare the same route driven through a roundabout by TF-pretrained in the Longest6 benchmark in CARLA 0.9.10[5] and 0.9.14[6]. The agent in 0.9.14 has a harder time predicting the correct segmentation labels, and labels many parts of the road as cars (blue color). It is worth noting that even the expert agents show worse performance in the new version. As these are rule-based agents, graphical differences should not affect them. Therefore, this indicates other potential changes between the simulator versions, such as new traffic behavior, new obstacles, or new scenarios to handle.

Compared to results reported by Chitta *et al.* [5] in Longest6 using 0.9.10, we see that the pre-trained TransFuser model roughly has the same scores, with RC being the main differentiator. However, when comparing the pre-trained Interfuser model with the results reported by Shao *et al.* [6] in Town05, we see an even larger difference in RC. This is likely because the pre-trained InterFuser weights released by Shao *et al.* [6] are not the same as those used on the CARLA Leaderboard and on their Town05 evaluation. Instead, as noted in their GitHub, the pre-trained model is trained only on a part of the full dataset [91]. Perhaps more interesting is that their reported results are worse than all of the TransFuser agents in Town05 using 0.9.10, even though they discussed having better performance than TransFuser on Town05 in their paper. We are not sure of the exact reason for these discrepancies; it might be pure randomness, or our evaluation methods may not be directly comparable to the paper.

The agents trained on our generated datasets show varying results on the Town05 benchmark in CARLA 0.9.14. For TransFuser, the differences in scores between TF-ds-feb and TF-pretrained is not too large, and it even beats TF-ds-orig, which shows that our data generation and training pipeline works, and that the models benefit from training on data generated from the same CARLA version they are evaluated in. In contrast to this, we see much larger differences between the InterFuser models IF-ds-custom and IF-pretrained, mostly due to the first having more infractions during evaluation, for example, three times as many vehicle collisions. As noted in Section 3.8.3, the ds-custom dataset is much smaller than the original InterFuser data set, which shows that more data is beneficial for the models by giving them more examples of different driving scenarios to learn from.

[5]`https://youtu.be/PfOsMLSgHsw?t=1180&list=PLWGDO2Z_62kKDze_p2hgt8DFMOe3VElgA`
[6]`https://youtu.be/BnpHIGfg7g0?t=1004&list=PLWGDO2Z_62kKDze_p2hgt8DFMOe3VElgA`

4.2 Experiment 2: Investigating the Effect of Increased Traffic

Due to the large differences in scores between the scores in the Longest6 and Town05 benchmarks in our first experiment, we investigate whether the reason for this difference is due to the greater amount of background traffic in Longest6 than in Town05. Longest6 by default runs with 500 background actors, while Town05 defaults to 120. We do this by performing evaluations on the Town05 benchmark using both 120 and 500 background actors and comparing the results.

4.2.1 Setup

This experiment requires the pre-trained weights to be downloaded in advance, after which the `run.sh` scripts in ^{ex}`ex-transfuser-0.9.14` and ^{ex}`ex-interfuser-0.9.14` are run. These scripts by default launch evaluations with 120 actors. To instead run the evaluation with 500 actors, we first set the environment variable `ACTOR_AMOUNT=500` before launching the evaluation.

4.2.2 Results

The results of this experiment are shown in Table 4.3. In general, we see a clear reduction in Driving Scores when driving among 500 background actors rather than 120. Particularly TF-pretrained exhibit higher vehicle collision rates (Cv), while other agents see reduced Route Completion (RC) or Infraction Score (IS).

Model	Primary metrics ↑			Secondary metrics ↓							
	DS	RC	IS	Cp	Cv	Cl	RL	SS	OR	RT	AB
120 actors											
TF-expert	94%	96%	98%	0.00	0.00	0.00	0.00	0.15	0.00	0.00	0.01
IF-expert	80%	100%	80%	0.02	0.02	0.00	0.00	0.00	0.00	0.00	0.00
TF-pretrained[1]	84%	100%	84%	0.00	0.03	0.00	0.01	0.16	0.00	0.00	0.00
IF-pretrained[1]	49%	72%	68%	0.02	0.07	0.00	0.06	0.00	0.00	0.19	0.00
500 actors											
TF-expert	70%	75%	93%	0.00	0.01	0.00	0.02	0.02	0.00	0.03	0.06
IF-expert	71%	100%	71%	0.05	0.01	0.00	0.00	0.00	0.00	0.00	0.00
TF-pretrained[1]	46%	70%	61%	0.00	0.24	0.00	0.01	0.01	0.00	0.01	0.11
IF-pretrained[1]	43%	63%	67%	0.01	0.16	0.00	0.03	0.00	0.00	0.16	0.00

[1] Videos of these runs are available at `https://youtube.com/playlist?list=PLWGDO2Z_62kKDze_p2hgt8DFMOe3VElgA`.

Table 4.3: Benchmark results of actors in Town05 with 120 and 500 background actors.

4.2.3 Discussion

This experiment clearly shows that the differences in scores in Experiment 1 between the Longest6 benchmark and the Town05 benchmark are affected by the number of background traffic actors in the simulation. As discussed in Section 4.1.3, it appears that more actors, in fact, give a higher probability of collisions with vehicles and lower RC. This is especially noticable for TF-pretrained, which goes from a DS of 83% to 46 % due to increases in Cv and AB. IF-pretrained is also affected by the increased traffic; however, since its score was already quite low with 120 actors, the change is not as apparent with 500 actors. The expert agents also get reduced scores with the increased actor count, although not by such a large margin.

These differences are also apparent in the video visualizations. By comparing videos of the same route for IF-pretrained, one with 120 actors[7] and one with 500[8], we see that the latter gets stuck in traffic, later colliding with the vehicles in front, while the former gets to continue with the evaluation run. This is also shown in Figure 4.4.

4.3 Experiment 3: Investigating the Impact of Training Time

During training of the TransFuser, training loss was observed to reach its minimum quite early during training. In this experiment we investigate whether the minimum of the training loss also implies a maximum of benchmark performance. We evaluate the TransFuser agent with weights trained on the original dataset for 5, 16, 26, and 41 epochs.

4.3.1 Setup

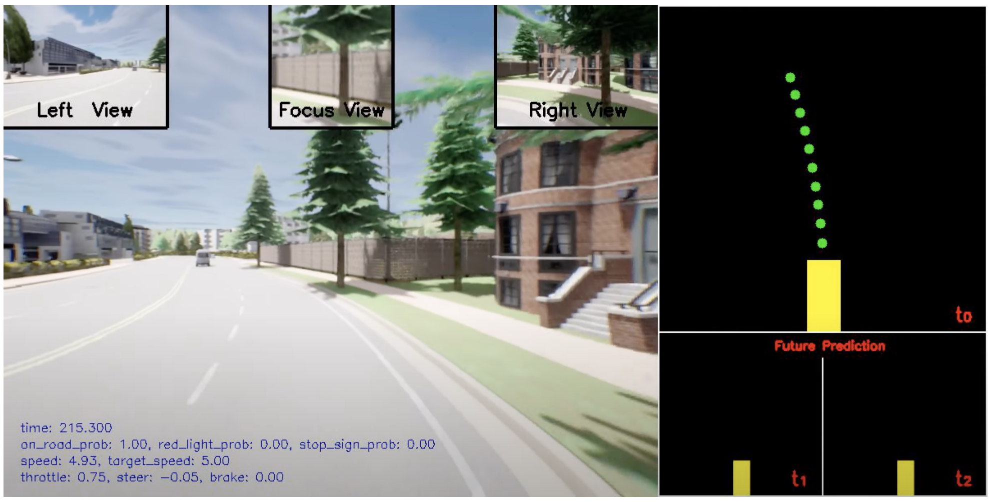
The weights at epochs 5 and 41 are extracted from the training run for TF-ds-orig. Additionally, the weights from epochs 5 and 15 are trained with a reduced learning rate for 11 additional epochs, until they reach 16 and 26 epochs, respectively. All weights are then evaluated in the Town05 benchmark.

4.3.2 Results

The training losses can be seen in Figure 4.5. We see that TF-ds-orig-16 has the lowest training loss, followed by TF-ds-orig-5, TF-ds-orig-26, then finally TF-ds-orig-41.

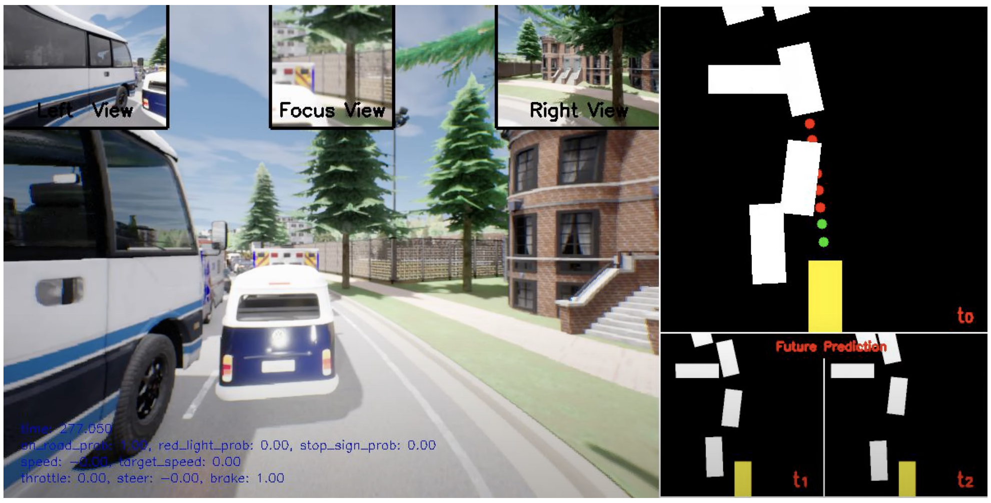
The benchmark results are shown in Table 4.4. Here, we see that the pre-trained TransFuser model performs best, followed by TF-ds-orig-16, TF-ds-orig-5, TF-ds-orig-26, then finally TF-ds-orig-41. We note that this matches the order from lowest to highest training loss.

[7]https://youtu.be/122mBh5KgAQ?t=1040&list=PLWGDO2Z_62kKDze_p2hgt8DFMOe3VElgA
[8]https://youtu.be/IYWmzEHvegg?t=962&list=PLWGDO2Z_62kKDze_p2hgt8DFMOe3VElgA



```
time: 215.300
on_road_prob: 1.00, red_light_prob: 0.00, stop_sign_prob: 0.00
speed: 4.93, target_speed: 5.00
throttle: 0.75, steer: −0.05, brake: 0.00
```

(a) 120 actors. The road is clear and that the agent can continue with the evaluation run. Video is available at `https://youtu.be/122mBh5KgAQ?t=1040&list=PLWGDO2Z_62kKDze_p2hgt8DFMOe3VElgA`.



```
time: 277.050
on_road_prob: 1.00, red_light_prob: 0.00, stop_sign_prob: 0.00
speed: −0.00, target_speed: 0.00
throttle: 0.00, steer: −0.00, brake: 1.00
```

(b) 500 actors. The agent is stuck in traffic, and the evaluation will therefore end in a route timeout. Video is available at `https://youtu.be/IYWmzEHvegg?t=962&list=PLWGDO2Z_62kKDze_p2hgt8DFMOe3VElgA`.

Figure 4.4: Comparing how different actor amounts affects the Town05 evaluation for IF-pretrained.

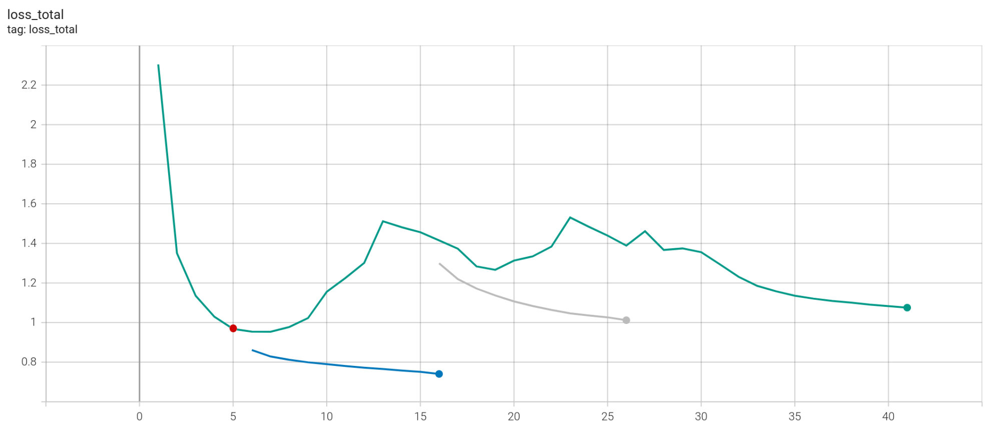


Figure 4.5: Training loss on the ds-orig dataset. TF-ds-orig-41 is shown in green, TF-ds-orig-26 in gray, TF-ds-orig-16 in blue, and TF-ds-orig-5 is marked with a red dot.

Model	Primary metrics ↑			Secondary metrics ↓						
	DS	RC	IS	Cp	Cv	Cl	RL	SS	OR	RT	AB
TF-pretrained	84%	100%	84%	0.00	0.03	0.00	0.01	0.16	0.00	0.00	0.00
TF-ds-orig-5	72%	92%	78%	0.00	0.03	0.00	0.02	0.12	0.02	0.00	0.02
TF-ds-orig-16	79%	90%	84%	0.00	0.07	0.01	0.00	0.12	0.00	0.00	0.03
TF-ds-orig-26	64%	87%	72%	0.00	0.32	0.26	0.01	0.09	0.02	0.01	0.28
TF-ds-orig-41	61%	96%	65%	0.00	0.10	0.00	0.00	0.12	0.02	0.00	0.01

Table 4.4: Town05 benchmark results of TransFuser actors trained for a varying number of epochs.

4.3.3 Discussion

The correlation between low training loss and high evaluation performance indicates that the training loss function functions adequately as a proxy for evaluation performance. It is also clear that continued training with a constant learning rate is not beneficial once the training loss begins to increase. The reduction in learning rate, however, enables further improvements to the model's performance, and may indicate that future experiments exploring other learning rate schedules may be fruitful.

4.4 Experiment 4: Measuring Variance in the Evaluation Results

Due to variance in the results discovered during consecutive evaluations of the same agent in the same benchmark, we decided to investigate the indeterminism of the simulator benchmarks by quantifying the variance of the results. In this experiment, we perform multiple additional runs of various actors in the Town05 evaluation with 120 agents to quantify the variance in results. The Town05 benchmark is chosen due to the shorter run-time, enabling a larger sample size to be drawn. Additionally, since Longest6 consists of 36 routes compared to Town05's 10 routes, Town05 is more likely to exhibit large variance.

4.4.1 Setup

This experiment runs the TransFuser and InterFuser agents using the `run.sh` scripts as in the previous experiments. The agents are evaluated multiple times, with specific focus on the TransFuser ds-feb agent to see the effect of a larger sample size.

4.4.2 Results

The distribution properties for each of the agents are shown in Table 4.5 and visualized in Figure 4.6. We note that all agents have a range of more than 10 percentage points, with some ranges above 20 points and even 30 points. The best performing agent is the TransFuser expert agent, followed by the pre-trained TransFuser before the InterFuser expert agent. Then TF-ds-feb and TF-ds-orig follows, before the pre-trained InterFuser last.

For the variance results, we find that the two InterFuser models have an equal sample standard deviation of 7.8, while TF-expert and TF-pretrained have a std.dev. of about 4.5. The two final TransFuser agents have a standard deviation of about 8.5, with most confidence in the estimate of 8.8 for TF-ds-feb due to the larger sample size. For TF-ds-feb we also note the largest observed range of 37 percentage points.

Agent	Median	Mean	Std. dev.	Range
cIF: Expert agent	82.0	80.4	7.8	23.9
IF: Pre-trained	52.8	51.1	7.8	23.3
TF: Expert agent	92.0	92.6	4.7	14.8
TF: Pre-trained	85.1	83.8	4.5	10.9
TF: ds-orig	65.3	64.4	8.4	18.4
TF: ds-feb	71.4	71.0	8.8	37.0

Table 4.5: Distribution of Driving Score (DS) over multiple evaluation runs for selected agents.

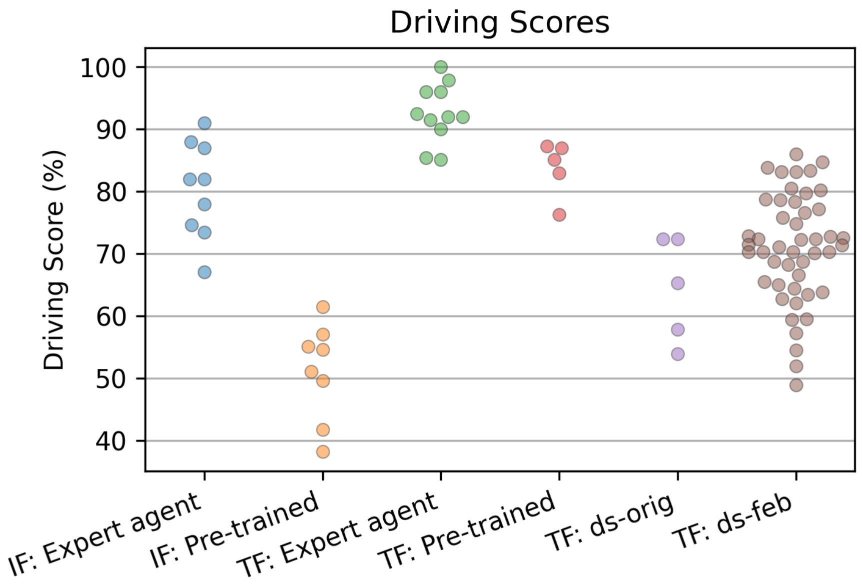


Figure 4.6: Distribution of Driving Score (DS) for selected agents.

4.4.3 Discussion

We see that the variance of the Driving Score metric is quite large, demonstrating that singular measurements using the Town05 benchmark are not sufficient to accurately evaluate agent performance. Variations of over 35 percentage points have been observed, which is greater than many pairwise differences of runs between models. While the results in the previous experiments have been obtained by taking the mean of three runs, we acknowledge that this sample size is likely too small to confidently assess performance differences between models, and we therefore recommend larger sample sizes and more appropriate statistical tests for assessing significance of results in future work.

Although we do not perform a similar experiment for the Longest6 benchmark, the variance of Driving Scores are likely to be lower. This is because the metric is a mean of all n routes in the benchmark, where $n = 10$ for Town05 and $n = 36$ for Longest6. Since the variance of the mean estimator is given as σ^2/n, where σ^2 is the underlying variance of the scores on the individual routes, the larger n for Longest6 will lead to a lower variance in the mean estimate. The online CARLA Leaderboard benchmark is run on $n = 100$ routes, resulting in an even better estimate of mean performance. However, there are other factors that may invalidate this reasoning, for example, that different routes and traffic conditions in the different benchmarks may cause changes to the underlying variance σ^2, and thus affect the variance of the mean estimate in unknown ways.

We therefore recommend that future research pay greater attention to the statistics of these benchmarks. An interesting aspect to investigate is whether the observed variance is due to the agent's behaviour, or inherent to the benchmark. The latter may occur in the case of traffic jams which inadvertently block the agent, and which occur at random intervals depending on the number of background actors and other factors.

4.5 Experiment 5: Testing Agents With New Sensor Configurations

In this experiment, we generate new datasets with a sensor configuration matching NAPLab's Kia e-Niro, train new weights for both TransFuser and InterFuser, and evaluate how the agents perform in the simulator benchmarks using this configuration, compared to the original.

4.5.1 Setup

Two new datasets, tf-ds-kia and if-ds-kia are prepared using the method described in Section 3.8, following the steps to use the Kia sensor configuration. These datasets are then used to train two new models, TF-ds-kia and IF-ds-kia, which are evaluated in the Town05 benchmark in CARLA version 0.9.14 as usual.

4.5.2 Results

The results of training the new models are shown in Figure 4.7. Both training runs show even and gradual descent, and the reduction in learning rate at TF-ds-kia at epoch 16 is also visible.

The evaluation results are shown in Table 4.6. Here we clearly see poor performance for the TransFuser model with the Kia sensor configuration, only achieving a Driving Score (DS) of 11%, compared to the pre-trained model's 84%. On the other hand, the InterFuser model performs comparably to its pre-trained cousin, achieving 48% versus the base-line 49%.

A closer look on the TF-kia-ds results show that infractions are increased across the board, especially collisions with pedestrians (Cp) and vehicles (Cv), off-road infractions (OR), and agent-blocked infractions (AB). This is also the only model that show route deviations, with on average 0.18 deviations per kilometer.

Model	Primary metrics ↑			Secondary metrics ↓							
	DS	RC	IS	Cp	Cv	Cl	RL	SS	OR	RT	AB
TF-pretrained†	84%	100%	84%	0.00	0.03	0.00	0.01	0.16	0.00	0.00	0.00
IF-pretrained†	49%	72%	68%	0.02	0.07	0.00	0.06	0.00	0.00	0.19	0.00
TF-ds-orig-16	79%	90%	84%	0.00	0.07	0.01	0.00	0.12	0.00	0.00	0.03
TF-ds-kia[1,†]	11%	51%	37%	0.00	0.51	0.32	0.01	0.07	0.69	0.03	0.67
IF-ds-kia†	48%	92%	49%	0.01	0.16	0.00	0.04	0.01	0.00	0.02	0.00

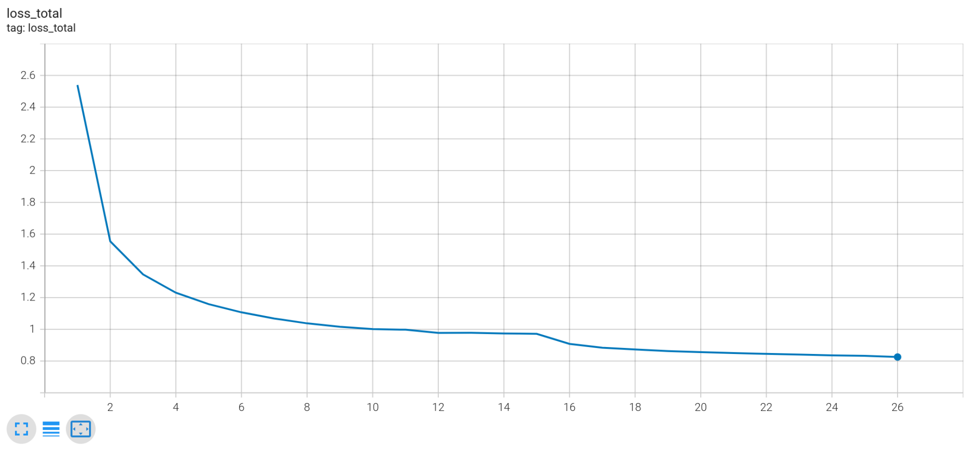
[1] This model also had 0.18 route deviations per kilometer.
† Videos of these runs are available at
`https://youtube.com/playlist?list=PLWGDO2Z_62kKDze_p2hgt8DFMOe3VElgA`.

Table 4.6: Town05 benchmark results of actors trained with the Kia sensor configuration.

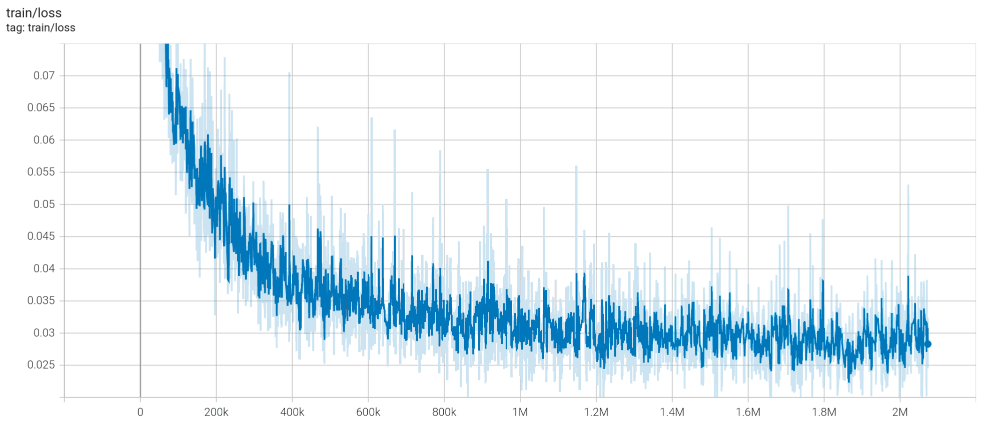
4.5.3 Discussion

The results of the InterFuser models show that IF-ds-kia adapted well to the new sensor configuration, indicating that our pipeline for data generation, training, and evaluation for this model worked well. However, the new TF-ds-kia model unfortunately does not show the same good adaptation. In one of the routes in the model's evaluation run, we can see that the agent turns left at a junction to reach a waypoint[9]. Once the waypoint is reached, the next waypoint is shown on the top-right map to be on in the opposing lane for a split second, before it shifts over to the correct lane (easier to see in lower playback speeds in the video). This makes the agent drive over the road markings to the opposing lane and crash with a vehicle. We also see other bad navigation behaviors, such as the agent driving on the elevated sidewalk [10]. Overall, this can indicate issues with how the TransFuser

[9]`https://youtu.be/Yn_zWBcKmYs?t=113&list=PLWGDO2Z_62kKDze_p2hgt8DFMOe3VElgA`
[10]`https://youtu.be/Yn_zWBcKmYs?t=100&list=PLWGDO2Z_62kKDze_p2hgt8DFMOe3VElgA`



(a) Training loss for TF-ds-kia. The x-axis indicates the training epoch.



(b) Training loss for IF-ds-kia. The x-axis indicates the training step. One epoch corresponds to about 62500 steps, with 35 epochs in total.

Figure 4.7: Training loss curves for TransFuser and InterFuser in Experiment 5.

agent processes target waypoints, and also that the model is not able to properly follow road lane lines.

There could also be an implementation error regarding the cropping and stitching of the input cameras. If the model is trained on one specific camera crop but is evaluated on another camera crop, this could affect its perception performance, since it will fail to understand the correct positions of objects. We note that while its poor actions cause it to gain many infractions during the evaluation, we can see in the visualization video that it is quite good at the auxiliary tasks, including bounding box prediction, mapping, and segmentation. This can indicate that how it processes the input camera images is indeed the main issue.

4.6 Experiment 6: Real-World Evaluation around Gløshaugen Campus

In this experiment, we investigate how the agents perform in an entirely different domain by evaluating them on real-world data collected using NAPLab's Kia e-Niro. Using the method described in Section 3.11, we present the real-world data to the models and query their predictions, resulting in the correlation metrics ρ_{steer} and ρ_{speed}, as well as the waypoint error metrics ϵ_i.

We evaluate the pre-trained TransFuser and InterFuser agents in addition to the agents we train on the Kia datasets, to see whether the specifically adapted dataset improves real-world metrics.

4.6.1 Setup

This experiment requires real-world data collected as described in Section 3.11, and uses the tooling in $^{\text{ex}}$`ex-kia` to perform inference and analyze the results. We prepare **Trip076**, which is a trip starting and stopping at NAPLab's garage in Elgeseter street. It goes around Gløshaugen, passing Studentersamfundet, Berg studentby, and Lerkendal stadium on the way. The weather is sunny.

In summary, the method consists of the following:

1. Generate a series of waypoints from the GNSS log to serve as the agent's targets.
2. Determine the correct LiDAR time offset to synchronize with the other sensors.
3. Run model inference for every frame in the input data.
4. Gather statistics about the results. The correlation coefficients ρ_{steer} and ρ_{speed}, and the waypoint errors ϵ_i are calculated and reported.

The final annotated plan is shown in Figure 4.8.

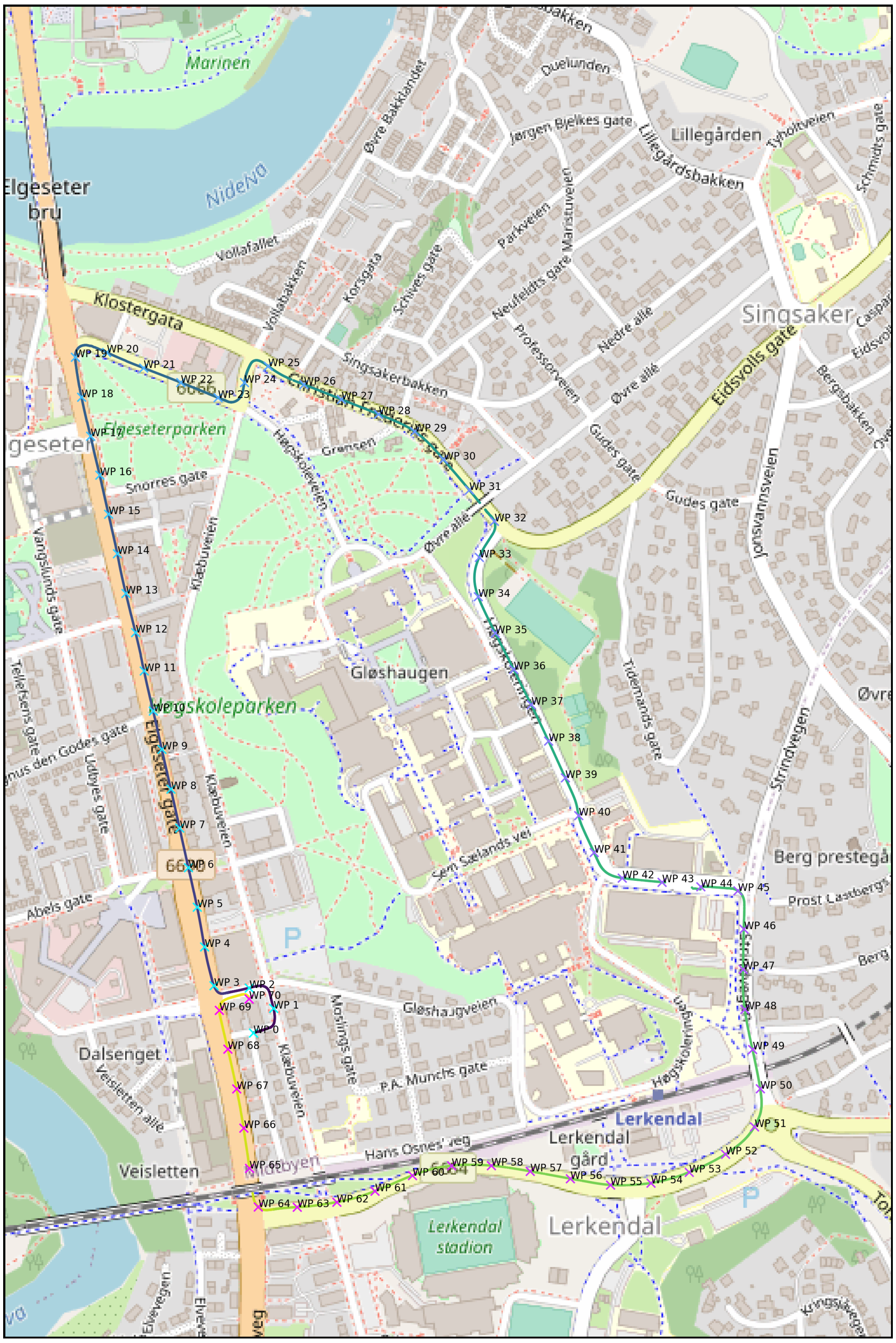


Figure 4.8: The annotated plan for Trip076.

4.6.2 Results

We begin by taking a look at visualization of TF-ds-kia and IF-ds-kia's predictions in Figure 4.9. TransFuser's semantic segmentation and depth predictions look very good, in addition to both models success in detecting bounding boxes for nearby cars. The predicted waypoints also point straight ahead on the correct track towards the goal location shown in the top of the TransFuser HD-map visualization. However, as can be better seen from the videos[11], the predictions can vary significantly from frame-to-frame, causing the noise seen in the following figures. The pre-trained TransFuser agent appears to produce better HD-map predictions, but semantic segmentation quality is about equal. For InterFuser, the pre-trained agent predicts significantly more false-positive bounding boxes than IF-ds-kia.

The errors in the waypoint predictions from TF-pretrained on Trip076 are shown in Figure 4.10. Motivated by the fact that the waypoint errors are approximately proportional to the waypoint index (1 to 4 in TransFuser's case, 1 to 10 for InterFuser), we scale each waypoint down by their index, and take their mean to determine the mean waypoint error shown in the figure. In this case, we see that the agent correctly predicts future waypoints during the red traffic lights at $t = 40\,\text{s}$ to $80\,\text{s}$ and $t = 205\,\text{s}$ to $220\,\text{s}$, but is otherwise often wrong by a meter or more.

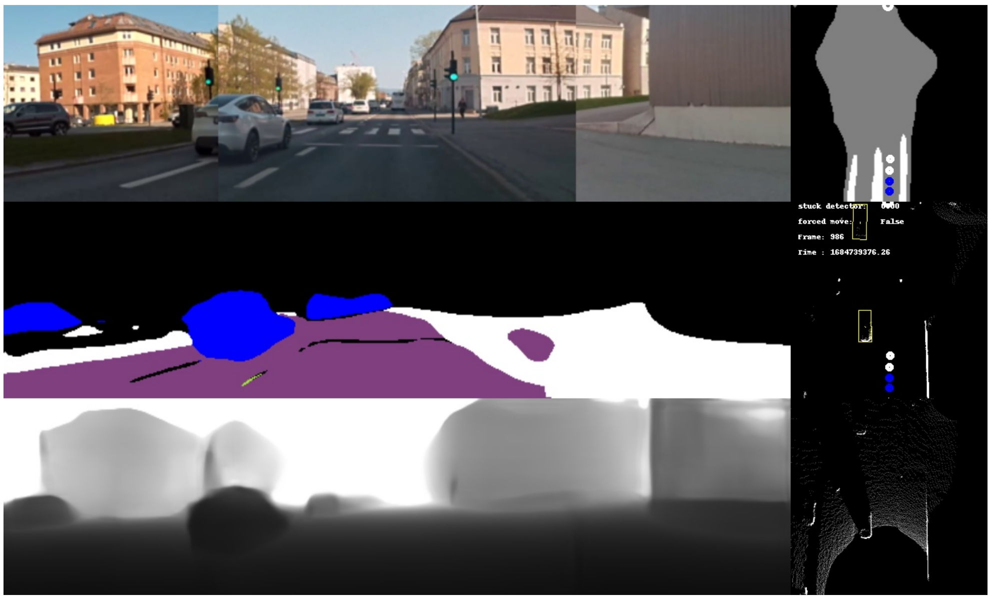
Histograms of the mean waypoint errors for each agent are shown in Figure 4.11. We see that the TransFuser models have a quite uniform distribution from 0.5 m to 1.5 m, with a peak close to 0 m. The maximum error is 2.24 m for TF-pretrained, and 1.95 m for TF-ds-kia, quite close to their 99th percentiles 1.90 m and 1.74 m, respectively.

For the InterFuser models, we see that the shape is more similar to an exponential distribution. Again, there is a peak close to 0 m, with a gradual decline until the 99th percentile at about 0.41 m for both models. The maximum errors are 5.82 m and 2.76 m for IF-pretrained and IF-ds-kia, respectively, significantly greater than the 99th percentiles.

In Figure 4.12, we compare the agents' steering angle predictions to the ground truth wheel angle, and calculate the correlation coefficients ρ_{steer} and ρ_{speed}. We see that while both TransFuser models have noisy predictions, with low correlations of $\rho_{\text{steer}} = 0.35$ for TF-pretrained and $\rho_{\text{steer}} = 0.44$ for TF-ds-kia, the model trained using the Kia dataset achieves a slightly better score. This holds for ρ_{speed}, where TF-ds-kia has a correlation of 0.44, versus 0.34 for TF-pretrained.

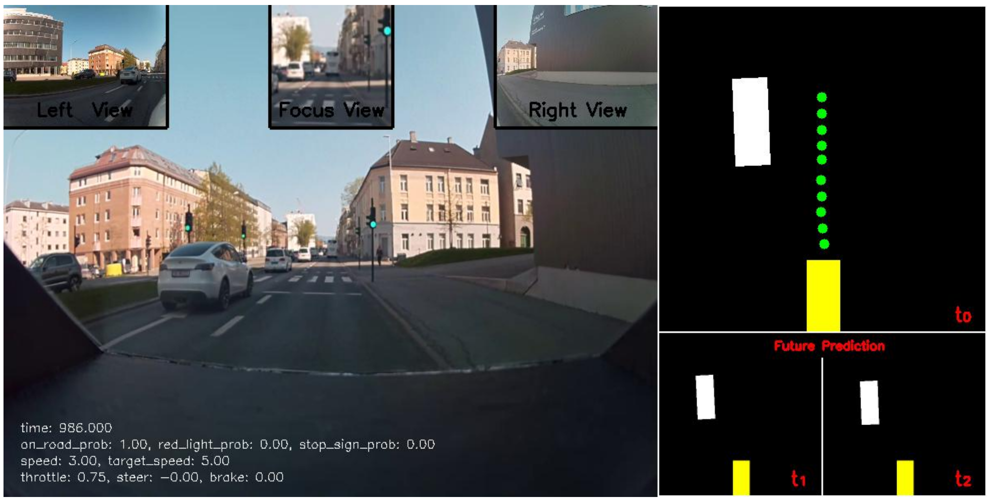
The InterFuser models qualitatively seem much less noisy, and have a high steering correlation of 0.75 for both models. However, the speed correlation is lower, at 0.22 for IF-pretrained and 0.43 for IF-ds-kia.

[11] See links in the attached document.



```
stuck detector:   0.000
forced move:      False
Frame: 986
Time : 1684739376.26
```

(a) TF-ds-kia's predictions. We see generally good semantic segmentation and depth estimation images, despite the seams in the stiched input image. The agent also correctly predicts two bounding boxes. The top-right HD-map is however not very accurate.



```
time: 986.000
on_road_prob: 1.00, red_light_prob: 0.00, stop_sign_prob: 0.00
speed: 3.00, target_speed: 5.00
throttle: 0.75, steer: −0.00, brake: 0.00
```

(b) IF-ds-kia. Here we see the different mechanism of camera input, as different sensors instead of stitched together into one. The model accurately predicts the bounding box for the closest car, as well as correct predictions for 1) being on the road, 2) seeing a red light, and 3) seeing a stop sign.

Figure 4.9: Visualization of model predictions during the real world Trip076.

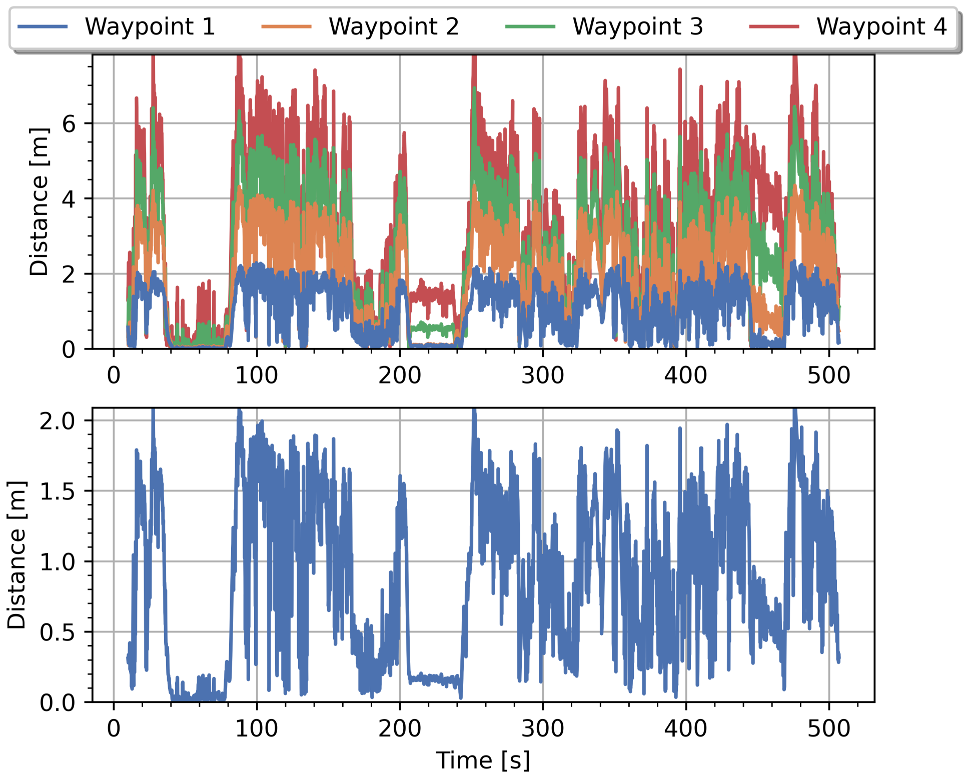
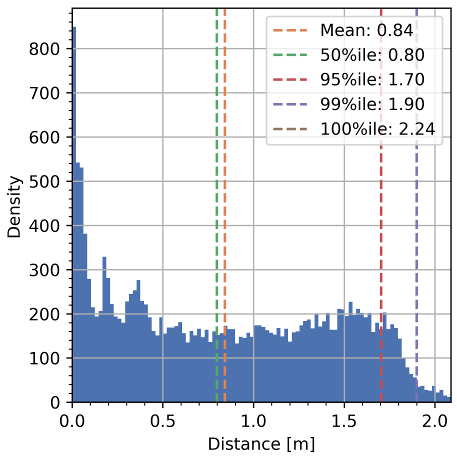
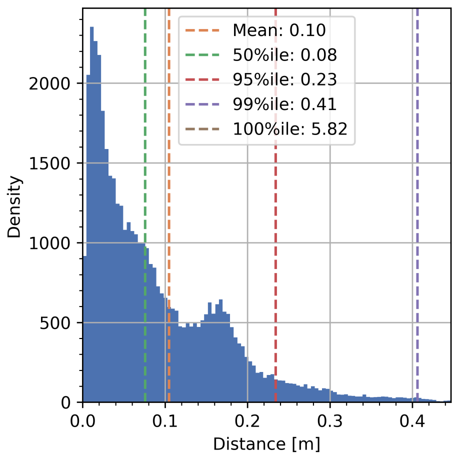


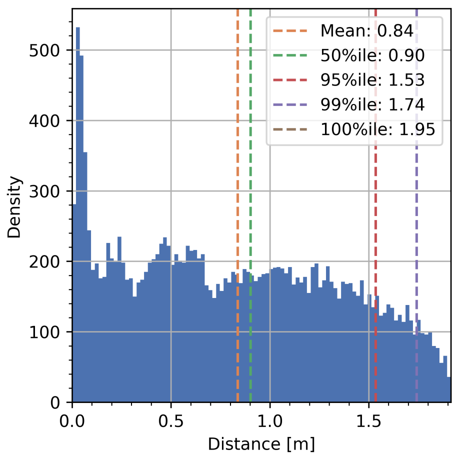
Figure 4.10: Waypoint errors for predictions from TF-pretrained on Trip076. **Upper:** Errors for each individual waypoint. **Lower:** Mean of waypoints scaled down by their waypoint index.



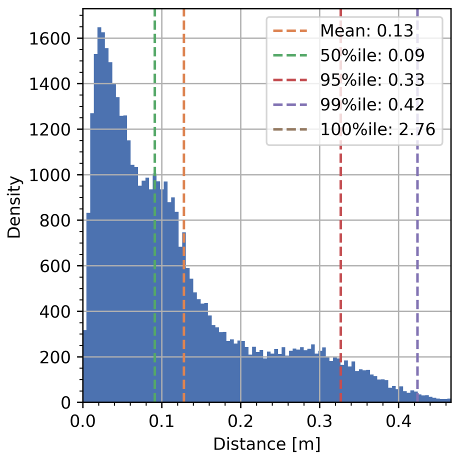
(a) TF-pretrained.



(b) IF-pretrained.

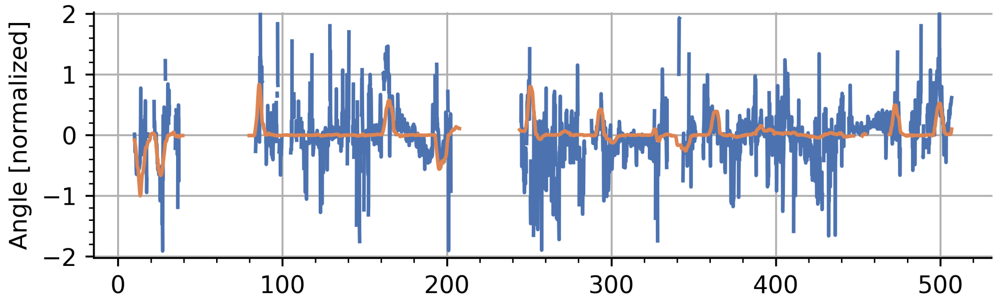


(c) TF-ds-kia.

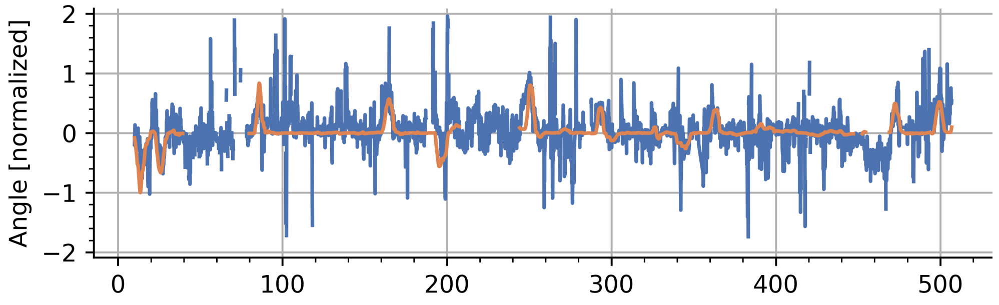


(d) IF-ds-kia.

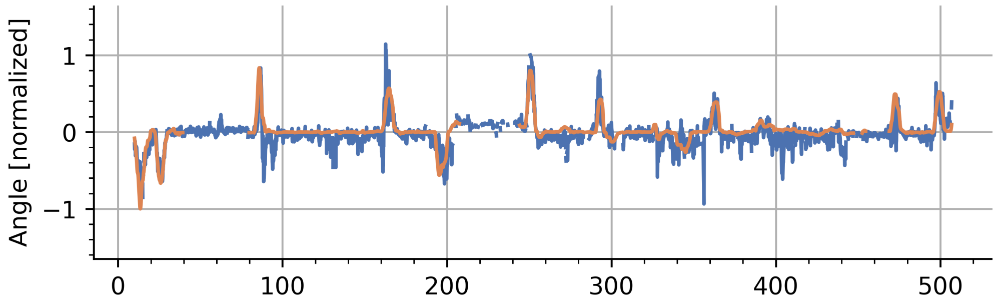
Figure 4.11: Distribution of mean waypoint errors for the four models. Note the differences in the axis scales, which understates how much lower on average InterFuser's errors are than TransFuser's.



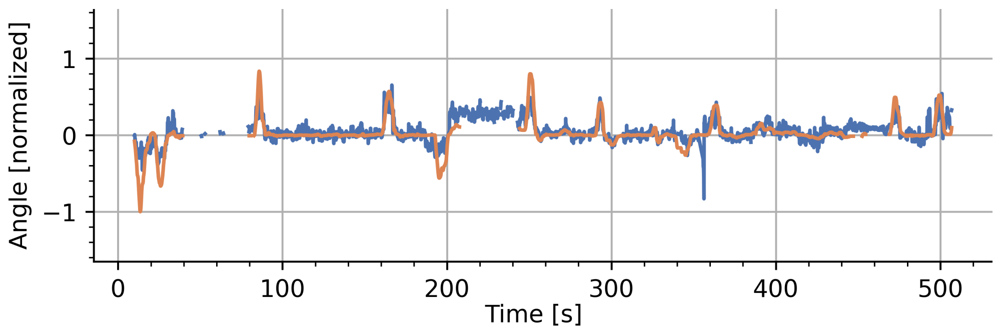
(a) TF-pretrained. $\rho_{\text{steer}} = 0.35,\ \rho_{\text{speed}} = 0.34$



(b) TF-ds-kia. $\rho_{\text{steer}} = 0.44,\ \rho_{\text{speed}} = 0.44$



(c) IF-pretrained. $\rho_{\text{steer}} = 0.75,\ \rho_{\text{speed}} = 0.22$



(d) IF-ds-kia. $\rho_{\text{steer}} = 0.75,\ \rho_{\text{speed}} = 0.43$

Figure 4.12: Correlation between predicted (blue) and ground truth (orange) steering angle for each of the four models on Trip076. Predictions are scaled to visually match the ground truth for easy visual inspection of correlation, this does not affect ρ_{steer}. Points are omitted where the agent's velocity is zero.

4.6.3 Discussion

The waypoint error histograms show significant differences between the TransFuser and InterFuser models. This is likely due to the differences in semantics of a waypoint between the two, with TransFuser combining both steering and speed information into its waypoints by treating it as a target destination for a small time step into the future, while InterFuser only encodes steering information by treating it as a target destination for a small step ahead on its route. This puts TransFuser at a disadvantage in this comparison, due to the fact that errors in its speed predictions also increase the waypoint error, which is not the case for InterFuser. Additionally, the difference in semantics also leads to a different selection of ground-truth waypoints, which by itself may cause incompatibilities in the comparison. Lastly, the different number of waypoints causes TransFuser to predict up to 8 m ahead, while InterFuser may predict waypoints over 12 m ahead. This partly explains why InterFuser's maximum waypoint error is much larger than TransFuser's.

Therefore, we will not further discuss inter-architecture differences in waypoint errors. The comparison within one architecture is more interesting, and here we see that while the median is higher for TF-ds-kia than for TF-pretrained, the 95th to 100th percentiles are lower, indicating better worst-case predictions. The peaks close to 0 m are likely due to the accurate waypoint predictions when waiting for red lights, where the agent correctly brakes. However, both the median and higher percentiles show that the waypoints are commonly wrong by a meter or more, indicating that neither TransFuser model are in agreement with the human driver's actions. For InterFuser, IF-ds-kia also shows better worst-case predictions than IF-pretrained, however both models have generally quite low values and seem to accurately predict future waypoints.

In contrast, the correlation coefficient results are comparable across architectures, and reveal much better steering correlation for InterFuser than TransFuser. Within each architecture, we see that TF-ds-kia improves on both steering and speed correlation compared to TF-pretrained, while IF-ds-kia only improves the speed correlation. The improvement in speed correlation is likely due to the reduced number of false-positive bounding boxes predicted, as false positives in front of the car will cause braking and thus wrong speed control.

This indicates that models trained on the Kia sensor configuration does have an advantage. In addition, the results indicate that InterFuser's waypoint prediction style results in more accurate steering, without apparent downsides in the speed control.

4.7 Experiment 7: Real-World Evaluation on the E6 Highway

This experiment is similar to the previous real-world evaluation, but this time done in alternative traffic conditions found on the highway. We evaluate the same models with the goal of answering the same research questions, namely **RQ3**, **RQ3**, and **RQ4**.

4.7.1 Setup

We follow the same setup procedures as the previous experiment, but prepare **Trip078**, which is a trip on the E6 highway from Nardosenteret, to Angelltrøa and back, in sunny weather. The same agents are evaluated, and the final annotated plan is shown in Figure 4.13.

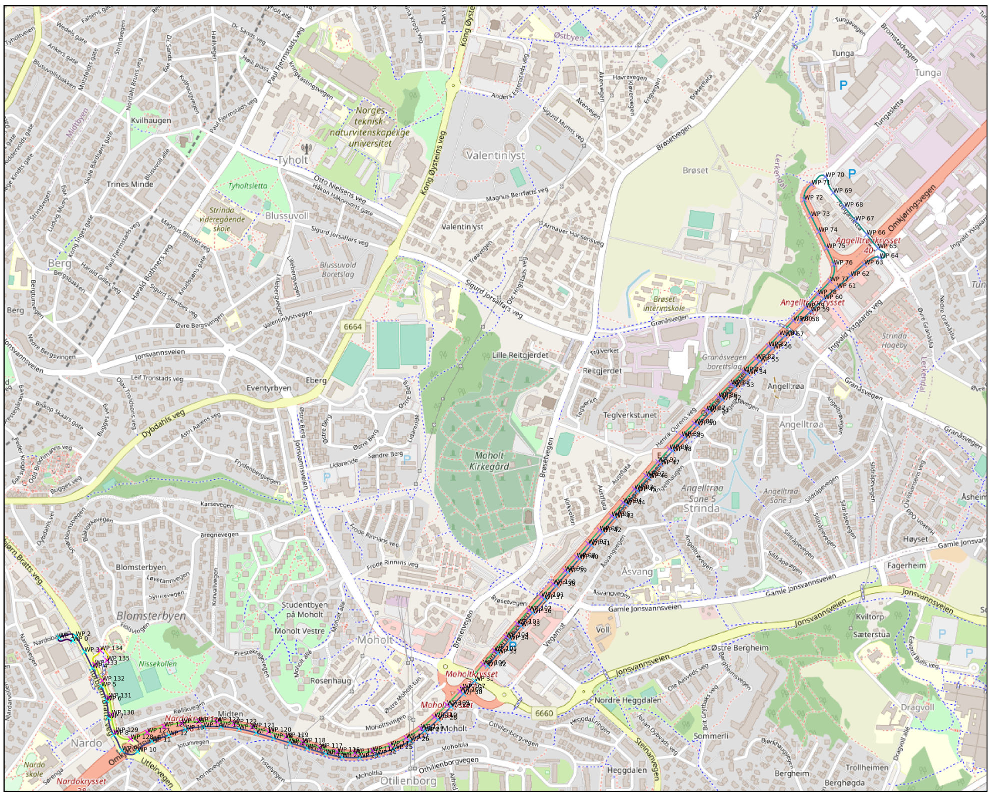


Figure 4.13: The annotated plan for Trip078.

4.7.2 Results

The videos[12] from Trip078 show the agent's predictions for each frame of the evaluation. The TransFuser models both show promising semantic segmentation

[12]See links in the attached document.

and depth results, although the semantic segmentation is often noisy, especially with hallucinations of cars in front of the agent. Both TransFuser models also exhibit waypoint predictions that have collapsed together to a single point in front of the vehicle, indicating a desire to brake. The InterFuser waypoints do not suffer from the same problem, however the agents often predict false positive bounding boxes when nearby other cars.

Figure 4.14 shows the waypoint error distribution for the four agents. In this experiment, the TransFuser predicted waypoints do not exhibit the same peak at 0 m, instead being more symmetric about the mean at 0.97 m for TF-prefuser and 1.15 m for TF-ds-kia. The distributions do not have long and thin tails; as we can see the 99th percentiles of 2.03 m and 1.99 m are quite close to the 100th percentiles of 2.43 m and 2.42 m for TF-prefuser and TF-ds-kia respectively.

For InterFuser, the distributions still resemble exponential distributions in shape, with much lower means of 0.14 m and 0.11 m for IF-pretrained and IF-ds-kia. While the 50th and 95th percentiles are similar for both InterFuser models, IF-ds-kia has much lower 99th and 100th percentiles of 0.58 m and 1.52 m compared to IF-pretrained 1.44 m and 6.40 m.

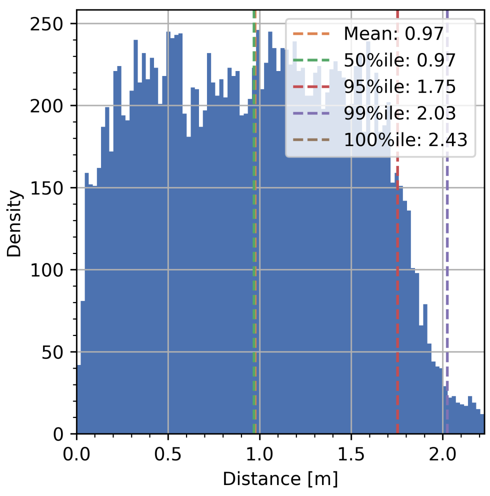
In Figure 4.15, we see the agent's steering angle predictions compared to the ground truth wheel angle. As seen in the previous experiment, TransFuser's predictions vary greatly and are very noisy, although both models seem to correlate well with the ground truth when the car is turning and not driving straight. The InterFuser models are much less noisy, which improves the correlation score.

We note that the TransFuser models have clear difficulties with the increased speed when driving on the highway, showcased by the negative correlation coefficients for the speed predictions. InterFuser on the other hand, still correlate positively for the speed predictions, albeit less so than in the previous lower-speed experiment.

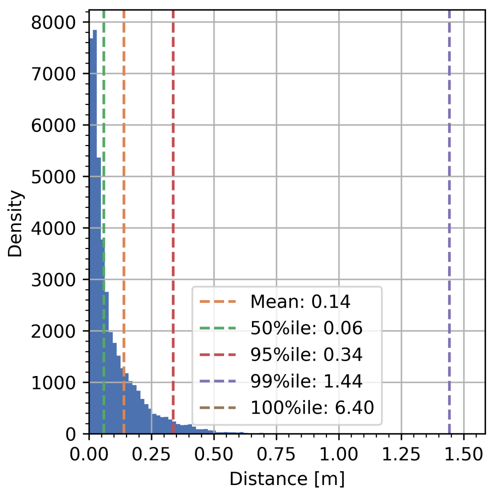
4.7.3 Discussion

The same remarks made in the previous experiment's discussion about TransFuser-InterFuser waypoint error comparisons also apply to this experiment, thus these are not discussed further. Within the TransFuser architecture however, the waypoint error distribution has changed shape significantly from the previous experiment, showing that the trip type greatly impacts the agents' metrics. The loss of the peak at 0 m is likely due to the reduction of situations where the car is standing still due to traffic or in junctions, which allowed the agent to very accurately predict zero movement in the previous experiment. On the highway, the car is in continuous motion, in addition to driving at speeds previously unseen by the agent, which generally causes increased waypoint prediction error.

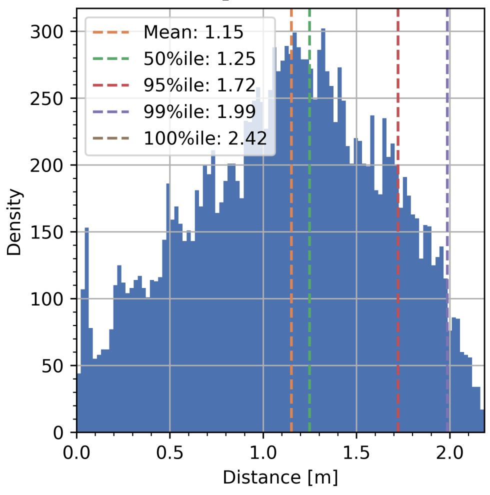
For InterFuser, the significantly shorter tail of IF-ds-kia indicates improved worst-case steering compared to IF-pretrained, however the reduced correlation coef-



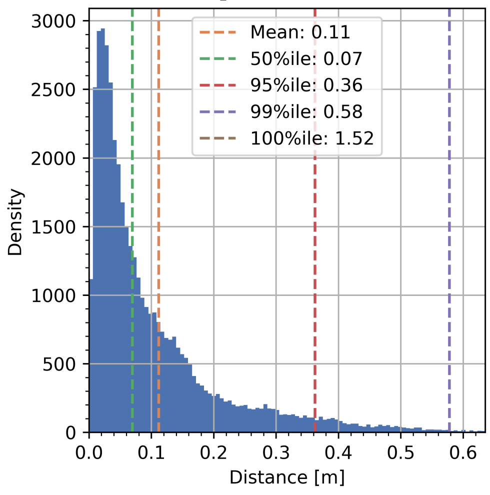
(a) TF-pretrained.



(b) IF-pretrained.

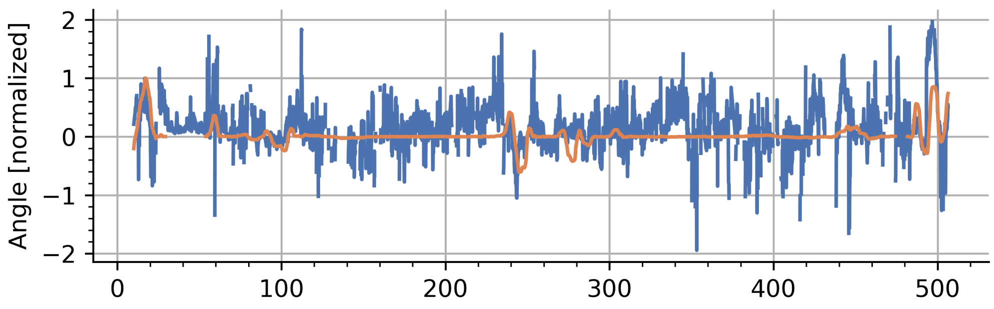


(c) TF-ds-kia.

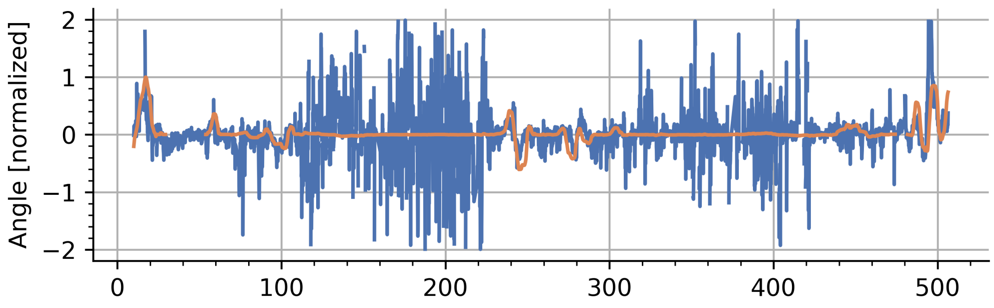


(d) IF-ds-kia.

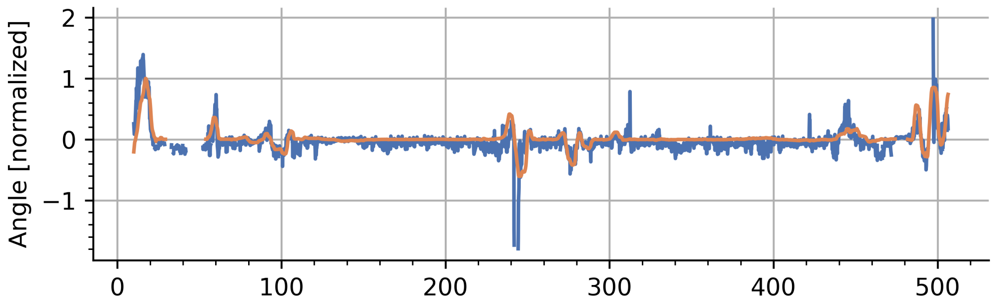
Figure 4.14: Distribution of mean waypoint errors for the four models. Note the differences in the axis scales, which understates how much lower on average InterFuser's errors are than TransFuser's.



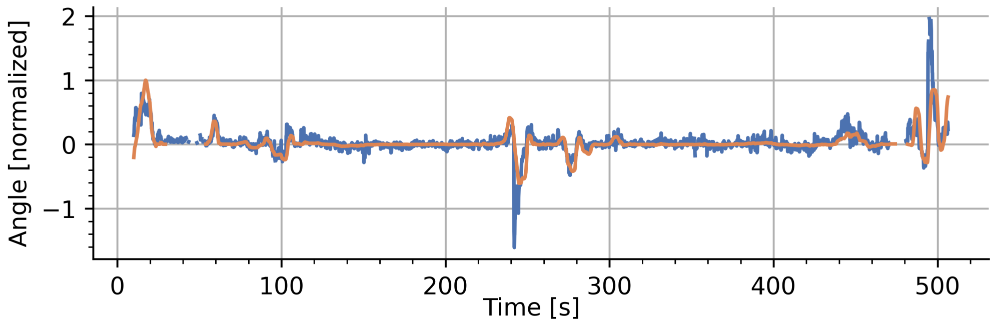
(a) TF-pretrained. $\rho_{\text{steer}} = 0.29,\ \rho_{\text{speed}} = -0.15$



(b) TF-ds-kia. $\rho_{\text{steer}} = 0.25,\ \rho_{\text{speed}} = -0.12$



(c) IF-pretrained. $\rho_{\text{steer}} = 0.73,\ \rho_{\text{speed}} = 0.27$



(d) IF-ds-kia. $\rho_{\text{steer}} = 0.69,\ \rho_{\text{speed}} = 0.35$

Figure 4.15: Correlation between predicted (blue) and ground truth (orange) steering angle for each of the four models on Trip078. Predictions are scaled to visually match the ground truth for easy visual inspection of correlation, this does not affect ρ_{steer}. Points are omitted where the agent's velocity is zero.

ficient ρ_{steer} does not indicate improved steering overall. With regards to speed however, IF-ds-kia appears to improve on the pre-trained model by a small amount. Compared to TransFuser, both InterFuser models show much better both steering and speed correlation, reinforcing the apparent superiority of InterFuser on real-world evaluations.

Chapter 5

Discussion

This chapter discusses the process and results of the thesis. First, Section 5.1 discusses the results of the experiment in relation to the research questions. Next, we evaluate and reflect on our resources and implementations in Section 5.2. Finally, Section 5.3 discusses the shortcomings of the thesis.

5.1 Experiments and Research Questions

This section discusses the results of the experiments in relation to the research questions of the thesis. We split the section in two parts; one for experiments in the CARLA simulator, and one for experiments on real-world data.

5.1.1 Experiments in CARLA

We start by discussing the experiments and research questions related to the CARLA simulator.

RQ1: Are the TransFuser and InterFuser results reproducible in CARLA version 0.9.10?

In Experiment 1, we looked at the scores reported in the TransFuser paper by Chitta *et al.* [5] and the InterFuser paper by Shao *et al.* [6], and attempted to reproduce them in our evaluation setup using their pre-trained weights. Our results show that our evaluation results of the pre-trained TransFuser agent in Longest6 are close to those of the TransFuser paper, with our evaluation having only 4% lower Driving Score (DS). The difference between our evaluation of the pre-trained InterFuser agent and the InterFuser paper in Town05 is a bit larger, with our evaluation having 14% lower DS. However, this was expected since the released InterFuser weights are not the same as those used by Shao *et al.* [6] during their evaluations. Visually inspecting the videos of the 0.9.10 evaluation runs in the experiment[1], we also see that the pre-trained models achieve good performance in their auxiliary tasks, such as waypoint predictions, semantic segmentation, and bounding box prediction. Overall, while the results differ slightly, we therefore find that the original TransFuser and InterFuser evaluation process and their results are reproducible in CARLA version 0.9.10.

We also trained and evaluated a TransFuser agent using the original TransFuser dataset provided by the authors. While this agent only was evaluated in Town05, and therefore is not directly comparable to the results of the TransFuser paper, we saw that its performance was close to that of the pre-trained TransFuser model. This suggests that TransFuser's training pipeline is also reproducible. Unfortunately, InterFuser's authors do not provide such a dataset, so we did not get to test its training pipeline in the same manner.

RQ2: How does the upgrade to version 0.9.14 affect the performance?

Experiment 1 compared results from the Longest6 and Town05 benchmarks in CARLA versions 0.9.10 and 0.9.14. In both benchmarks we saw a decrease in DS for all agents when upgrading to version 0.9.14. More infractions occurred across all runs, especially vehicle collisions. We assume that these performance reductions come primarily from improved graphical fidelity and more varied lighting conditions, as these can make it harder for the agents to perceive the scene correctly. We also experienced some graphical glitches in CARLA 0.9.14 that could have affected the results[2]. Additionally, the rule-based expert agents showed worse scores on the Town05 benchmark in 0.9.14 than in 0.9.10, which indicates that there are other potential changes between the simulator version that affect driving conditions, such as new traffic behavior or new map layouts.

With this in mind, we also saw that the new TransFuser trained on data generated in version 0.9.14 had better scores than the model trained on data from version 0.9.10. This shows that the models benefit from training on a dataset similar to the environment in which they will be evaluated, and also that the TransFuser data generation process is reproducible. The InterFuser model trained on our custom 0.9.14 data did, however, not achieve the same performance. We assume that this is because the dataset is much smaller than what InterFuser normally is trained with, which also tells us that a larger dataset that covers diverese scenarios is vital for driving performance, especially with respect to infractions.

Overall, since we generally see worse driving performance in 0.9.14, even among the expert agents, we conclude that the transition to the new and improved CARLA version negatively affects the performance of the agents.

[1]`https://www.youtube.com/playlist?list=PLWGDO2Z_62kKDze_p2hgt8DFMOe3VElgA`

[2]`https://youtu.be/BnpHIGfg7g0?t=1004&list=PLWGDO2Z_62kKDze_p2hgt8DFMOe3VElgA`

Discussion

In addition to answers to the above research questions, we have found that the choice of benchmark has a large effect on the agent results, with potential pitfalls if results are interpreted without a proper investigation of the score distributions. For instance, the results of Experiment 3 indicate that there is a correlation between training loss and benchmark scores, however, as we determine later in Experiment 4, the sample size of three runs per model is likely not sufficient to assess score differences. In addition, there may be additional variation from training new models on the same dataset, which we do not investigate. This could be an important point in evaluating the suitability of different datasets.

Still, the current results from Experiment 1 and Experiment 5 at least indicate that the datasets do matter, with the IF-ds-custom agent trained on a much smaller dataset scoring significantly lower than IF-pretrained, and TF-ds-kia with the new sensor configuration failing in the Town05 benchmark compared to TF-pretrained. While we show with the TF-ds-feb and IF-ds-kia agents that data generation does indeed produce capable agents, it is clear that the impact of factors such as dataset size, training time, and sensor configuration is not yet fully understood.

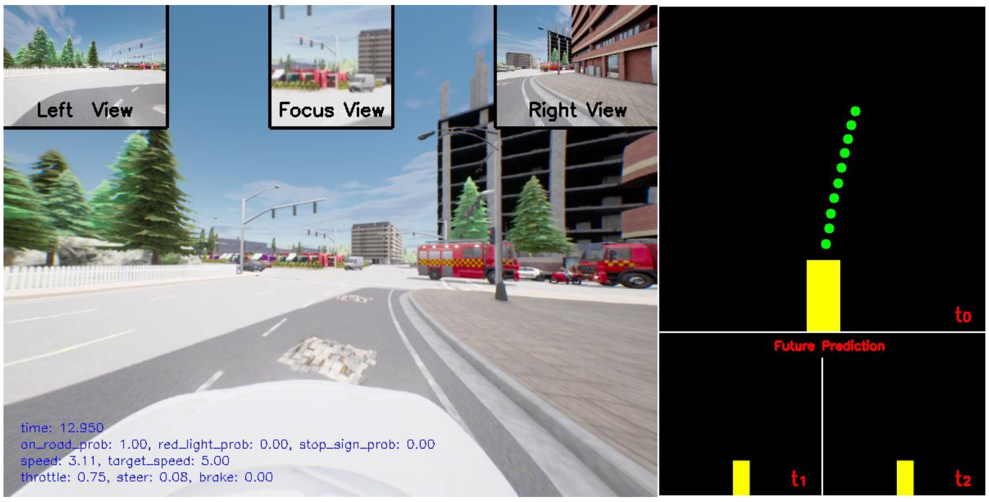
As can be seen in the benchmark videos, the scenario with a vending machine and crossing pedestrian occurs quite frequently, to the point where agents can potentially overtrain on this event, as seen in Figure 5.1. Increasing the variety of unexpected scenarios in the simulator benchmarks could be a useful addition to further close the distribution gap between the simulator and real-world environments.

5.1.2 Experiments on Real-World Data

We continue by discussing the experiments and research questions related to evaluation on real-world data.

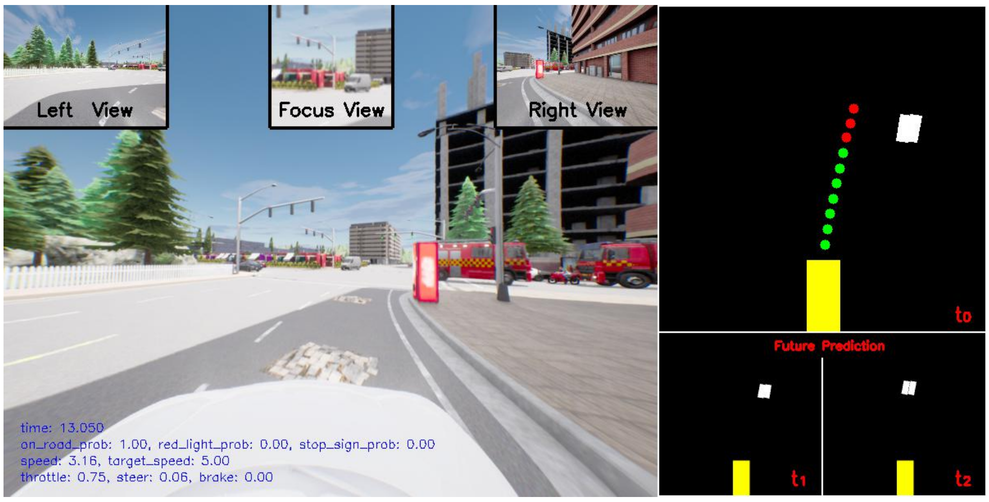
RQ3: How do the models perform on real-world data generated with NAPLab's Kia e-Niro?

As we saw in the real-world Experiments 6 and 7, the agents perform surprisingly well considering the large domain shift from a simulated world to the real world. TransFuser shows promising results in semantic segmentation and depth prediction of real-world camera data, indicating that the variety of weather and lightning conditions in the training dataset is sufficient and has led to generalization instead of overtraining. The agent's bounding box predictions are also good, showing high sensitivity to other cars, although false positives often result from fence posts or other obstacles in the center of the road. InterFuser also shows a high correlation between its steering predictions and the ground-truth steering wheel angle, both in city driving and on the highway. Although speed control is



```
time: 12.950
on_road_prob: 1.00, red_light_prob: 0.00, stop_sign_prob: 0.00
speed: 3.11, target_speed: 5.00
throttle: 0.75, steer: 0.08, brake: 0.00
```

(a) The frame before the vending machine is spawned. The models predicts a clear path ahead (green waypoints).



```
time: 13.050
on_road_prob: 1.00, red_light_prob: 0.00, stop_sign_prob: 0.00
speed: 3.16, target_speed: 5.00
throttle: 0.75, steer: 0.06, brake: 0.00
```

(b) The frame after the vending machine is spawned on the right side of the road. The model instantly predicts the area in front of the vending machine as unsafe (red waypoints), and the future predictions shows that it believes a pedestrian is going to walk left towards the front of the car, even though there are no visible pedestrian yet.

Figure 5.1: Visualizations of two frames from an InterFuser evaluation run showcasing possible overtraining on vending machines. Vending machines are utilized during training and in benchmarks to hide pedestrians when they are spawned, before they walk toward the road.

still flawed, the results generally show great potential for simulator-trained models to interpret and act in the real-world.

RQ4: Is it necessary to generate data using a sensor configuration that matches the real car, or would the models perform well in different sensor configurations without modifications?

In Experiment 5, we saw that the sensor configuration greatly affects the evaluation results on the Town05 benchmark, with TF-ds-kia especially failing to drive properly, and IF-ds-kia seeing increased Route Completion at the cost of decreased Infraction Score. This indicates that optimizing the sensor configuration is a challenge by itself which could be explored further in future work.

The real-world experiments also show differences in results depending on sensor configuration, especially with regards to the correlation coefficients for steering and speed control in Experiment 6. Here we saw improved steering and speed correlation for TF-ds-kia compared to TF-pretrained, and improved speed correlation for IF-ds-kia compared to IF-pretrained. This suggests that the re-training of the agents on new datasets with the Kia sensor configuration matching the NAPLab vehicle is worthwhile and improves performance. The results for the highway experiment in Experiment 7 do not show the same improvement, but neither does it show large degradations, and it appears that the choice of model architecture has a much greater effect on performance.

However, IF-pretrained fares quite well even with its mismatching sensor configuration between training and evaluation time, indicating that the sensor configuration perhaps doesn't need to match exactly, as long as it fulfills the requirements of a left-, center-, and right-facing cameras. This could be due to InterFuser's use of explicit sensory embeddings to distinguish the left, center, and right cameras from each other, while TransFuser is dependent on the implicit positional information in the single stitched camera input, and could therefore be more affected by perturbations in the configuration.

RQ5: Is TransFuser or InterFuser best suited for real-world data?

While the waypoint error metrics show much lower errors on average for InterFuser than TransFuser, the comparison is slightly flawed due to differences in ground-truth selection of waypoints. Therefore, our main metric used to answer this question are the correlation coefficients ρ_{steer} and ρ_{speed}. These also show a clear advantage for InterFuser, with much improved steering correlation on both trips, in addition to much better speed control on the highway trip. The visualization of steering angle over time also shows much less noise in InterFuser's steering predictions, indicating higher-quality predictions. We therefore conclude that InterFuser is better suited than TransFuser for future experiments involving real-world data.

Discussion

These experiments show the great potential of the AD agents in the real world setting, and also illustrate that choices such as sensor configuration and model architecture have a large effect on the results. The traffic environment also plays a significant role, and we see that the performance characteristics clearly differ between the two real-world routes. It is likely possible to quantify the effects from other factors such as weather, road size, pedestrians, and other objects, given enough real world data covering different situations. This could reveal specific agent weaknesses that could be resolved by improving the training data.

As discussed for the CARLA evaluations, there could also be significant variance in real-world evaluations as well. Unfortunately, the data collection process is time-consuming, which has prevented us from collecting additional real-world trips in the same routes to evaluate the agents over a variety of trips recorded at different times. However, the data processing pipeline built in this work enables simple consumption of recorded data if new trips are recorded in future work, allowing evaluation of variance in these evaluations as well.

In conclusion, while the agents exhibit surprisingly good steering performance and interpretation of the traffic situation, there is still much to be done before the agents can safely control a real-world vehicle in proximity of other traffic. However, a combination of this work and the remote control work by Gusev [49] could perhaps enable a simple waypoint-following behavior controlled by the InterFuser agent, in a safe and restricted area. This could serve as an interesting proof-of-concept and would be a milestone for NAPLab.

5.2 Reflections and Evaluation

5.2.1 Compute Resources

Our available resources described in Section 3.1 were vital for the thesis work. `nap02` was heavily used for training and mass storage of the data generated on Idun, as the amount of data exceeded the storage and file quotas there. However, one weakness of `nap02` is that the graphical issues on the A100 GPUs discussed in Section 3.7.2 prevented us from running CARLA benchmarks on the server. We were therefore also dependent on `vcxr12` with its RTX 4090 GPU to run benchmarks. An additional benefit of the RTX 4090 is its superior rasterization performance compared to the A100 GPUs, which significantly reduced the runtime of CARLA evaluations compared to both `nap02` and Idun. For the same reason, we also utilized `vcxr12` for some of InterFuser's data generation, as discussed in Section 3.8.3. However, its VRAM did limit us to running three evaluations simultaneously.

For the rest of the data generation, Idun's parallelization compensated for its slower CARLA performance. The exception was when Idun limited us to five concurrent jobs instead of the standard 20, during a period of high demand in the cluster. A request to Idun's team is therefore to expand available GPU resources, possibly with some GPUs with better render performance such as the RTX series. Finally, although our IDI horizon VMs were mostly unused during the early months of the thesis, they were heavily utilized to complete our final CARLA evaluations during the last weeks. We were also allowed to borrow Florian Wintel's RTX 4090 equipped computer during the last weeks for the same purpose, for which we thank him.

Although we appreciate having access to all these resources, we note that working on up to six different computers at the same time comes with some overhead. With data generation, training, and evaluation happening on different hosts, it becomes difficult to manage where data, models, and results are stored. Keeping up-to-date notes on which tasks are running where proved useful, but we would still recommend a workflow involving fewer hosts, if possible. One possibility would be to implement support for running training and evaluation on Idun too, after which submitting large batches of simultaneous evaluations would be significantly simpler than the current manual approach on the individual hosts.

5.2.2 Technical Implementation

We found that our organization centered on the `experiments` git repository was very helpful, allowing for quick iterations on our implementations and easy code sharing between us. This was especially noticeable when working with the multiple computers discussed above. It enabled us to onboard Florian's computer and start evaluations in under an hour after we got access to the computer, demonstrating the power of containerization. Similarly, when we decided to also utilize the IDI Horizon VMs, the process of deploying the `experiments` repository and starting evaluations took around 10 minutes.

Upgrading TransFuser and InterFuser from CARLA version 0.9.10 to version 0.9.14 provided several benefits. As mentioned in Section 3.7, our main motivation for the upgrade was its much improved graphical quality, making the simulated environment more similar to the real-world data on which we evaluated the models. In addition, the upgrade enabled us to run CARLA in headless environments with Docker easier due to the updated Vulkan graphics API. This was a necessity for data generation on Idun, and made the simulator more robust, resulting in fewer crashes during our evaluations. Note that the latest CARLA version also includes support for multi-GPU rendering, although we were unable to test this during our work. The model upgrades also made them compatible with CARLA Leaderboard 2.0, making them ready for submission once it replaces Leaderboard 1.0.

There were also some disadvantages with the upgrade. We underestimated the amount of work needed to make TransFuser and InterFuser compatible with the latest CARLA client API, and the upgrade introduced many bugs that cost us a considerable amount of time and effort to solve. Examples of this are the changed

LIDAR behavior in the simulator that caused us to train and evaluate models without LIDAR data and the new segmentation IDs that caused TransFuser to label objects incorrectly. These challenges occurred due to a result of poorly documented changes in the simulator release notes, and overly-specialized agent codebases relying on undocumented CARLA behaviors, meaning that there can potentially be other unknown behavior changes that we have not encountered or addressed. The effect of the new graphical glitches that occur on the A100 GPUs (as seen in Section 3.7.2) is also uncertain, potentially affecting training results. Finally, using a new version for our evaluations means that the results are less comparable with the original TransFuser and InterFuser papers.

However, we are satisfied with our own custom implementations for tasks like real-world evaluation. The modules are built using clear Python code with type annotations, and should enable future researchers to inspect and extend our implementations. For example, one can easily load and process data from NAPLab's vehicle, and perform inference using TransFuser and InterFuser in a simple manner.

One task with room for improvement is data generation on Idun, since we used Slurm with a learning-by-doing approach. For example, when resuming data generation after a restart of the Slurm jobs, all jobs would be requeued and wait in line, only for some of them to exit immediately due to their part of the dataset being finished. This wasted long queue durations. It could potentially be more efficient to implement a "process manager" in Python that manages Slurm jobs and distributes work dynamically; however, our approach using CSV files was simple and fast to implement.

5.3 Shortcomings

As discussed in Section 5.2.2, upgrading TransFuser and InterFuser to work with the latest CARLA version took more effort than anticipated, especially when also integrating the models into our reproducible pipelines for dataset generation, training, and evaluation. This, in combination with the finding of many new issues that had to be resolved during our experiments, meant that we did not have time to explore improvements to the models themselves. We instead only trained TransFuser and InterFuser using the same model setups as their authors, missing out on potential improvements from hyperparameter search, architecture changes, or other features demonstrated to perform well by related works. We propose several potential improvements to the models in Section 6.2.1.

The thesis work would benefit from further investigation of possible issues. The results in Experiment 1 show worse results for all models when upgrading from CARLA version 0.9.10 to 0.9.14, even for the expert agent. Although graphical differences between the versions, including improvements and glitches, could be the main culprit, this raises questions about whether the benchmarks work equally after the upgrade or if there are other factors that affect the results, such as new traffic behaviors. Inspections of visualizations from the same route in both versions have shown some map changes, such as new fences at the edges of the road, but time limited us from investigating this further.

We also have low confidence in the correctness of TransFuser's and InterFuser's codebases. They are complex and overwhelming to work with, and their authors have unfortunately not been clear in separating their code implementations from CARLA Leaderboard's code and following best practices for Python code. In retrospective, we probably would have made our own custom model based on their architectures if we had known this from the start. Creating a custom model from scratch would also give greater confidence in our methods and results, and made it easier to iterate on functionality.

As discussed in Section 5.1, another shortcoming of the thesis is the lack of investigation into the dataset methods. We had varying dataset sizes when training our models, and the quality of the data is questionable, especially when considering parts of the dataset included frames with graphical glitches. It is also unclear how much dataset size and variety affect the results. For InterFuser, for example, we had to choose which town and weather configurations to use in the dataset generation, and due to queue limits on Idun, we could not generate data for all combinations like the authors do. On the other hand, the pre-trained TransFuser models achieved good performance even though they are trained on one tenth of the number of frames as InterFuser. There are also potential problems with our training methods such as overtraining, and the training pipeline could benefit from adding a validation step. The lack of large sample sizes and statistical significance is also a shortcoming, as well as the lack of ablation studies that could reveal the importance of camera positions in CARLA or camera cropping and stitching in real-world evaluations.

Chapter 6

Conclusion and Future Work

This chapter summarizes and concludes the thesis work in Section 6.1, and presents suggestions for future work in Section 6.2.

6.1 Conclusion

The research goal of this thesis was to explore and evaluate two state-of-the-art end-to-end models for autonomous driving in simulated and real environments. Seven experiments were carried out to reach this goal and to answer our research questions, beginning with data generation, training, and evaluation in the CARLA simulator, and ending with evaluations on real-world data collected using the NAPLab Kia e-Niro vehicle. The work has resulted in a well-documented method and implementation that simplifies future development within NAPLab.

The results generally show sufficient success in reproducing the performance of the pre-trained TransFuser and InterFuser agents in simulator benchmarks, such that the method can be utilized to also generate data and train models with other sensor configurations. Doing this, we find that the InterFuser model we train with the Kia sensor configuration performs very well on real-world data, achieving high steering correlation to the human driver and low waypoint error. The TransFuser model predicts perhaps surprisingly accurate semantic segmentation and depth images, despite never being trained on anything else than 3D-rendered images, but unfortunately does not score well on the steering metrics, partially due to being affected by the poor speed control.

In addition to both agents being incapable of speed control at real-world velocities, there is significant noise in semantic segmentation, waypoints, steering, and bounding box predictions. These issues currently prevent the agents from successfully navigating a real-world vehicle, and to solve these issues, we present potential improvements in the next section.

6.2 Future Work

Although we have shown that it is feasible to train AD agents in the CARLA simulator and transfer the knowledge to the real world, it is clear that there is significant room for improvement, both in the CARLA benchmarks and in the real world. In this section, we present suggestions for potential future work that may improve performance in both CARLA and real-world evaluations or otherwise expand on issues described in this thesis. We organize the suggestions according to category, that is, whether they relate to 1) model improvements, 2) dataset and training improvements, or 3) evaluation improvements.

6.2.1 Model Improvements

As the models we have evaluated have been designed for CARLA and the Longest6 benchmark, some design choices are not optimal with respect to real-world evaluation. Even within the CARLA domain, there are multiple potential modifications that may improve the agents' performance.

Adding a Temporal Component

As discussed in Section 2.4.1, poor performance may arise when the i.i.d. assumption is broken. For both TransFuser and InterFuser, the dataset contains samples that are not independent, and during evaluation the model does not know its history of previous states for which the current state depends on. This may impact driving performance, for instance, by causing erratic braking and throttling due to being unable to commit to a specific future plan. A potential solution to this problem is to incorporate a temporal component into the model, creating a Recurrent Neural Network. For instance, adding forward layers connecting one or more hidden layers at frame $i-1$ to frame i could enable both better interpretation of the agent surroundings by gathering multiple observations over time, and also better planning and execution by preserving the agents' reasoning across frames. One specific metric that this could improve is the Stop Sign infraction measurement. The current models do not have the ability to wait for a set amount of time at a stop sign, but this could be solved with such a temporal connection.

Predicting Speed

To resolve the speed mismatch, the expert agent must be improved to generate a dataset that contains high-speed driving. In addition to this, we suggest adding a dedicated head to the ANNs to predict speed, for instance, as a one-hot encoded vector with ten different speed levels ranging from 0 to 90 km/h. This could potentially improve speed control by separating it from the difficult waypoint prediction task. TransFuser waypoint prediction would then be modified to predict waypoints at certain spatial distances ahead, similar to InterFuser's approach, instead of predicting waypoints at certain time steps ahead.

Interpretability

In addition to the dedicated speed prediction head, dedicated prediction heads could be added to provide details about the agents' beliefs and reasoning. For instance, similar to InterFuser's red-light- and junction-probability predictions, we propose additional predictions such as "in a left turn", "in a right turn", "changing lanes", and "braking due to crossing pedestrian". This serves to increase the agents' interpretability, and thus assist future researchers in identifying misunderstandings and limitations within the models. The predictions may also improve researchers' confidence in the models' ability to correctly identify its surroundings, which may improve the feeling of safety should one attempt to let the model drive a real car some day.

Exploring Other Models

While the above improvements are mainly related to the models used in this thesis, TransFuser and InterFuser, we also suggest looking at the latest contributions to end-to-end autonomous driving presented in Section 2.6. ReasonNet (Section 2.6.8) improves on InterFuser directly, providing the model with temporal and global reasoning to improve autonomous driving in urban scenarios. Their results show better performance than those of InterFuser, both in the CARLA Leaderboard and in their custom benchmarks, and it would therefore be interesting to evaluate this model on the real-world trips discussed in this thesis. As ReasonNet builds on InterFuser, integrating the model with our real-world evaluation methods should not require much work. Unfortunately, we did not have the opportunity to attempt this, as the authors' paper was not released before late May.

UniAD (Section 2.6.6) uses an alternative approach that optimizes the cooperation between individual perception and prediction tasks so that they all contribute to planning in a single network. Compared to simple multi-task systems, this has the advantage of fully leveraging the potential of each module, while also providing interpretable intermediate representations such as occupancy maps and actor trajectories by extracting knowledge between modules. It would be interesting to explore how this approach performs compared to the end-to-end approach of TransFuser and InterFuser. In addition, since inference speed is critical during real-world autonomous driving, it would also be interesting to explore the vectorized scene representation used by VAD (Section 2.6.7), which they show is more computationally efficient and more accurate than the traditional rasterized representations discussed in this thesis.

6.2.2 Dataset and Training Improvements

There are also multiple potential areas of improvement in the dataset generation and training process, especially with respect to real-world benchmarks.

Incoroprating Real-World Data

We believe that the most important factor is to reduce the distribution shift between training data and evaluation data, which could be done by incorporating real-world data during training. The difficulty lies in the absence of labeled data for the real-world data we have gathered, but there are some options for how to handle this problem.

The most obvious is perhaps to manually label some of the collected real-world data and use this to perform fine-tuning training of the agents. However, this can be both expensive and time-consuming, especially for large amounts of data. Therefore, we suggest another approach we called mixed-data training, in which multiple different datasets are combined, each containing partially labeled data. The training algorithm would then sample from the various datasets, and only calculate and backpropagate the loss for the parts which are labeled. For instance, by including a pre-made dataset with semantic segmentation labels for RGB images, the training algorithm could improve segmentation performance on real-world images by sampling from this dataset and performing backpropagation according to the semantic segmentation loss. The algorithm would still sample from the CARLA-generated dataset to maintain driving performance. By using the GNSS logs from the recorded real-world data, the waypoint loss could be targeted as well, separately from the segmentation loss. Then, if necessary, one could manually label some real-world HD-map and LiDAR bounding box data, and sample from all four datasets during training, potentially improving real-world performance without extensive manual labeling efforts. We direct future researchers to the Waymo Open dataset [106], which includes both semantic segmentation and LiDAR bounding box labels, potentially eliminating the need for manual labeling entirely.

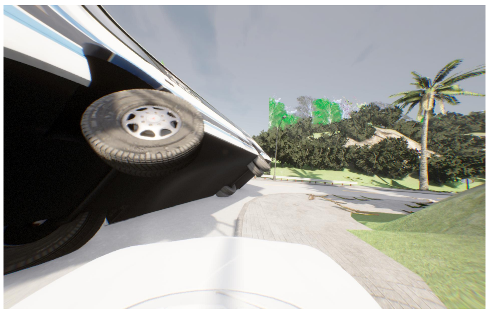
Neural Radiance Fields

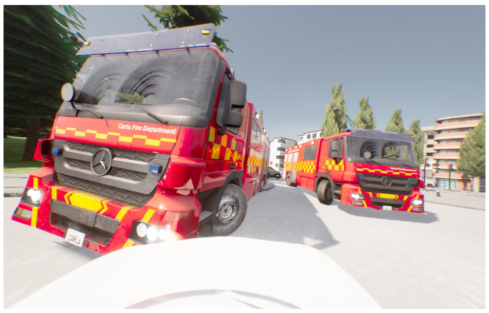
Other approaches to reducing the domain shift may involve simulator-to-real methods, for instance, the combination of digital twins of the target area with Neural Radiance Fields (NeRFs) to generate more realistic looking imagery while still training in a simulator. Wayve recently discussed how advancements in neural rendering have shown great promise in representing 3D scenes more efficiently and robustly [107]. They highlight two main benefits. The first is that NeRFs can be used to generate photorealistic examples to supplement traditional training data. This is especially useful for the IL approaches discussed in this thesis, as they might not generalize well to unseen scenarios, such as the car drifting out of lane. NeRFs makes it possible to augment data to include such examples, for example by slightly moving the vehicle location and position and generating corrective actions that merge back into the original expert driver trajectory. This can result in many extra training examples from one single input frame. The second benefit is that NeRFs can be used to create benchmarks with reliable metrics. In this thesis, we use frames from videos recorded by an expert driver to evaluate model decisions. However, the model decision does not affect the next frame, meaning that we cannot continue evaluating what the model would do if it had actually executed the action on the vehicle. By turning the expert video into a NeRF, one can simulate where the car would end up after its decision, render the new state using NeRF, and repeat until satisfied.

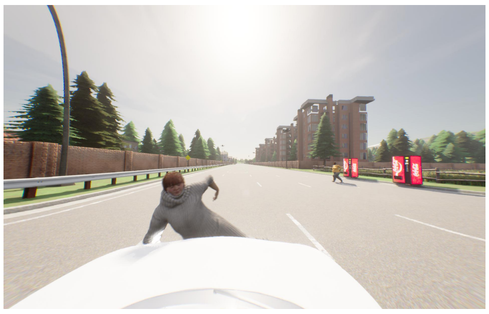
However, these benefits come with limitations [107]. First, creating NeRFs requires pixel-perfect localization of camera locations and orientations to correctly register images. Cameras attached to moving vehicles in the real world may not reach this precision. There exist algorithms to obtain camera properties directly from images, but they are not always accurate. Second, NeRFs scenes are limited in size, often having their quality significantly reduced at distances greater than 150 meters from the camera location. This can, however, be overcome by splitting city-scale NeRFs into smaller segments with some overlap, and then seamlessly swapping NeRFs when reaching their bounderies. Lastly, and perhaps most crucially for autonomous driving, classic NeRFs can only capture static scenes, meaning that we cannot have dynamic objects such as pedestrians and other vehicles that move over time in the scene. Emerging solutions to this problem mix NeRFs and traditional rendering methods by separating dynamic and static elements from the scene. One can then combine the dynamic actors and static scene into a single image, with the freedom of moving the dynamic actors into different locations of the static scene, making it possible to model how actors change over time.

Ground Truth Data

As seen in Figure 6.1, the current expert agents are not perfect drivers themselves. One cannot expect the models to achieve better results than the expert they are trained to imitate, so improvements to the expert agents are likely to improve benchmark performance for the learning agents as well. An alternative to this is to perform fine-tuning training using an algorithm that does not require expert agent data, for example Reinforcement Learning. Experiments using Direct Policy Learning are perhaps simpler and more feasible to implement, in which case we recommend simply running the pre-trained TransFuser or InterFuser model on the routes used in the original dataset, and simultaneously querying the expert agent to collect additional state-action pairs. This new and expanded dataset will then contain a greater variation of states, which serves to reduce the distribution shift occurring from training to evaluation, and thus may improve performance in both simulated and real environments. Though, this assumes that the expert agent is capable of producing sensible actions for these states, and may thus require some improvements to the experts first.


(a) Expert agent taking a shortcut under a bus.


(b) Expert agent preventing firetrucks from saving a house in flames.


(c) Expert agent colliding with a pedestrian in the middle of an open road.

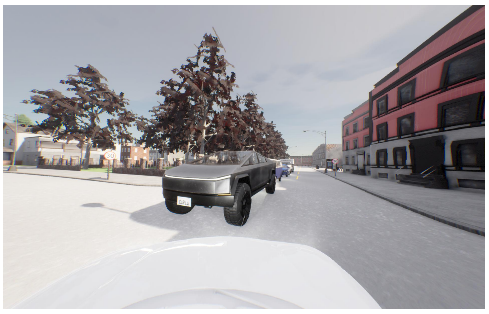

(d) Expert agent ignoring traffic yielding rules when turning, becoming stuck in the middle of the junction.

Figure 6.1: We train our models on data generated by TransFuser's and Inter-Fuser's expert driving agents. Unfortunately, the quality of their driving is not always optimal, as seen in the examples in this figure. Training on this data can therefore lead to unwanted behaviors.

Sensor Configurations

Finally, we recommend additional research on the impact of sensor configuration. As seen in Experiments 5 and 6, the change of sensor configuration had a large impact on the TransFuser model. Adjustments to the camera stitching parameters could potentially improve TransFuser's performance, and perhaps bring it closer to InterFuser's. Additionally, several of the Kia sensors are currently unused, such as the front LiDAR and rear-facing camera. These could provide additional information to the agents, enabling safer and more accurate driving.

On a related note, we recommend applying some form of random perturbations to the simulated sensor rotation and position during data generation, to make the agents less sensitive to errors in the sensor calibration procedure.

6.2.3 Evaluation Improvements

While both CARLA benchmarks and real-world evaluations are useful tools for comparing agent performance, neither is perfect. As noted, the CARLA benchmarks often experience traffic jams outside of the agents' control, and may be lacking in the variety of scenarios encountered. Improvements to the CARLA autopilot and the addition of new scenarios would therefore be valuable contributions to all CARLA-based benchmarks.

One metric the CARLA benchmarks currently do not measure is driving comfort, or smoothness. Agents are not penalized for erratic behavior, such as large and rapid changes to the steering angle or throttle. As agent performance continues to improve on current benchmarks, it is natural to increase the difficulty by adding new penalties for unwanted behavior. The smoothness metric would also be very useful in real-world evaluations, where the erratic behaviors are exaggerated by both the domain shift and the bypassing of the PID-controllers.

For the real-world benchmarks, we recommend two adjustments to the current method. One is to manually label each frame of the evaluation trips according to traffic situation, and investigate driving performance in different scenarios. This could reveal under which conditions the agent struggles and when it succeeds. Second, by manually labeling the bounding boxes in the BEV, one could measure the accuracy and sensitivity of the bounding box predictions. This would be especially useful for measuring differences in detection performance if the mixed-data training method was implemented.

Finally, by using NeRFs to create benchmarks as discussed in Section 6.2.2, evaluations could include additional scenarios not encountered in the real-world recordings. One could, for example, shift the vehicle a meter to the right at a given frame, and see how the model would recover back to the original trajectory.

Bibliography

[1] World Health Organization (WHO). “Global status report on road safety 2018.” (2018), [Online]. Available: `https://www.who.int/publications/i/item/9789241565684` (visited on 2022-12-06).

[2] U. Montanaro, S. Dixit, S. Fallah, M. Dianati, A. Stevens, D. Oxtoby, and A. Mouzakitis, “Towards connected autonomous driving: Review of use-cases,” *Vehicle System Dynamics*, vol. 57, no. 6, pp. 779–814, 2019. DOI: `10.1080/00423114.2018.1492142`. [Online]. Available: `https://doi.org/10.1080/00423114.2018.1492142`.

[3] T. Litman, “Autonomous vehicle implementation predictions: Implications for transport planning,” 2023.

[4] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, “CARLA: An open urban driving simulator,” in *Proceedings of the 1st Annual Conference on Robot Learning*, S. Levine, V. Vanhoucke, and K. Goldberg, Eds., ser. Proceedings of Machine Learning Research, vol. 78, PMLR, 2017-11, pp. 1–16. [Online]. Available: `https://proceedings.mlr.press/v78/dosovitskiy17a.html`.

[5] K. Chitta, A. Prakash, B. Jaeger, Z. Yu, K. Renz, and A. Geiger, “Transfuser: Imitation with transformer-based sensor fusion for autonomous driving,” *Pattern Analysis and Machine Intelligence (PAMI)*, 2022.

[6] H. Shao, L. Wang, R. Chen, H. Li, and Y. Liu, “Safety-enhanced autonomous driving using interpretable sensor fusion transformer,” 2022.

[7] The CARLA team. “CARLA Autonomous Driving Leaderboard,” [Online]. Available: `https://leaderboard.carla.org/` (visited on 2022-11-10).

[8] NTNU. “NTNU Autonomous Perception Laboratory,” [Online]. Available: `https://www.ntnu.edu/idi/naplab/naplab` (visited on 2023-03-14).

[9] NTNU. “Specialization project for Computer Science and preparatory project for Master of Science in Informatics - IDI,” [Online]. Available: `https://i.ntnu.no/wiki/-/wiki/English/Specialization+project+for+Computer+Science+and+preparatory+project+for+Master+of+Science+in+Informatics+-+IDI+` (visited on 2023-05-22).

[10] Á. Takács, I. Rudas, D. Bösl, and T. Haidegger, “Highly automated vehicles and self-driving cars [industry tutorial],” *IEEE Robotics & Automation Magazine*, vol. 25, no. 4, pp. 106–112, 2018. DOI: 10.1109/MRA.2018.2874301.

[11] Tesla. “Tesla AI Day 2022.” (2022), [Online]. Available: https://www.youtube.com/watch?v=ODSJsviD_SU (visited on 2022-12-06).

[12] The Waymo Team. “Waymo at CVPR 2022: A hub for real-world innovation and groundbreaking research.” (2022), [Online]. Available: https://blog.waymo.com/2022/06/%20WaymoAtCVPR2022.html (visited on 2022-11-10).

[13] Waymo. “Waymo is opening its fully driverless service to the general public in Phoenix,” [Online]. Available: https://waymo.com/blog/2020/10/waymo-is-opening-its-fully-driverless.html (visited on 2023-06-07).

[14] Lei Kang. “Baidu’s robotaxi platform Apollo Go gets permit to offer fully driverless rides in Beijing,” [Online]. Available: https://cnevpost.com/2023/03/17/baidus-apollo-go-gets-permit-fully-driverless-rides-beijing/ (visited on 2023-06-07).

[15] F. Lambert. “Tesla FSD Beta takes its first steps outside of North America,” [Online]. Available: https://electrek.co/2023/05/25/tesla-fsd-beta-first-steps-outside-north-america/ (visited on 2023-06-07).

[16] R. Nunno. “Issue brief | autonomous vehicles: State of the technology and potential role as a climate solution.” (2021), [Online]. Available: https://www.eesi.org/papers/view/issue-brief-autonomous-vehicles-state-of-the-technology-and-potential-role-as-a-climate-solution (visited on 2022-11-10).

[17] The United Nations Economic Commission for Europe (UNECE). “2021 Statistics of Road Traffic Accidents in Europe and North America.” (2021), [Online]. Available: https://unece.org/info/publications/pub/364151 (visited on 2022-12-06).

[18] M. Blanke, M. Henriques, and J. Bang, “A pre-analysis on autonomous ships,” 2020.

[19] SAE International. “Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles (J3016_202104).” (2021), [Online]. Available: https://www.sae.org/standards/content/j3016_202104/ (visited on 2023-05-04).

[20] United States Department of Transportation. “Automated Vehicles for Safety,” [Online]. Available: https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety (visited on 2023-05-07).

[21] SAE International. “SAE Levels of Driving Automation™ Refined for Clarity and International Audience.” (2021), [Online]. Available: `https://www.sae.org/blog/sae-j3016-update` (visited on 2023-05-04).

[22] William H. Widen and Philip Koopman. “Do Tesla FSD Beta Releases Violate Public Road Testing Regulations?” [Online]. Available: `https://www.jurist.org/commentary/2021/09/william-widen-philip-koopman-autonomous-vehicles/` (visited on 2023-06-05).

[23] Cat Dow. “What are the six SAE levels of self-driving cars?” (2023), [Online]. Available: `https://www.topgear.com/car%20news/what-are-sae-levels-autonomous-driving-uk` (visited on 2023-05-07).

[24] Waymo. “Waymo One,” [Online]. Available: `https://waymo.com/waymo-one/` (visited on 2023-06-07).

[25] Waymo. “Waymo breaking down what autonomoy means to the Waymo Driver.” (2021), [Online]. Available: `https://twitter.com/Waymo/status/1347286935535017986` (visited on 2023-05-07).

[26] Fred Lambert. “Elon Musk says Tesla’s next vehicle will operate ‘almost entirely in autonomous mode’.” (2023), [Online]. Available: `https://electrek.co/2023/03/07/elon-musk-tesla-next-vehicle-operate-almost-entirely-autonomous-mode/` (visited on 2023-05-07).

[27] Tesla. “Autopilot | Tesla,” [Online]. Available: `https://www.tesla.com/autopilot` (visited on 2023-06-11).

[28] B. Wessling. “Mercedes rolls out Level 3 autonomous driving tech in Germany,” [Online]. Available: `https://www.therobotreport.com/mercedes-rolls-out-level-3-autonomous-driving-tech-in-germany/` (visited on 2023-06-11).

[29] Waymo. “Waymo Driver,” [Online]. Available: `https://waymo.com/waymo-driver/` (visited on 2023-06-11).

[30] Cadence System Analysis. “The Use of RADAR Technology in Autonomous Vehicles.” (2022), [Online]. Available: `https://resources.system-analysis.cadence.com/blog/msa2022-the-use-of-radar-technology-in-autonomous-vehicles` (visited on 2023-05-08).

[31] Babak Shahian Jahromi. “Ultrasonic Sensors in Self-Driving Cars.” (2019), [Online]. Available: `https://medium.com/@BabakShah/ultrasonic-sensors-in-self-driving-cars-d28b63be676f` (visited on 2023-05-08).

[32] B. Templeton. “Tesla Removes Ultrasonic Sensors In Bold Move That Cripples Features But Promises To Restore Them,” [Online]. Available: `https://www.forbes.com/sites/bradtempleton/2022/10/17/tesla-removes-ultrasonic-sensors-in-bold-move-that-cripples-features-but-promises-to-restore-them/` (visited on 2023-06-11).

[33] Y. Xiao, F. Codevilla, A. Gurram, O. Urfalioglu, and A. M. López, "Multimodal end-to-end autonomous driving," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 1, pp. 537–547, 2022. DOI: 10.1109/TITS.2020.3013234.

[34] L. Le Mero, D. Yi, M. Dianati, and A. Mouzakitis, "A survey on imitation learning techniques for end-to-end autonomous vehicles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 9, pp. 14 128–14 147, 2022. DOI: 10.1109/TITS.2022.3144867.

[35] Epic Games, *Unreal engine*, 2019-04-25. [Online]. Available: https://www.unrealengine.com.

[36] The CARLA team, *Python API reference*. [Online]. Available: https://carla.readthedocs.io/en/0.9.13/python_api/ (visited on 2022-11-10).

[37] The CARLA team, *C++ reference*. [Online]. Available: https://carla.readthedocs.io/en/0.9.13/ref_cpp/ (visited on 2022-11-10).

[38] The CARLA Team. "Sensors reference - LIDAR," [Online]. Available: https://carla.readthedocs.io/en/latest/ref_sensors/#lidar-sensor (visited on 2023-05-08).

[39] ASAM. "ASAM OpenDRIVE Standard," [Online]. Available: https://www.asam.net/standards/detail/opendrive/ (visited on 2023-05-09).

[40] The CARLA team. "Maps and navigation," [Online]. Available: https://carla.readthedocs.io/en/0.9.14/core_map/#carla-maps (visited on 2023-05-09).

[41] The CARLA Team. "Town 3," [Online]. Available: https://carla.readthedocs.io/en/latest/map_town03/ (visited on 2023-06-05).

[42] The CARLA team. "ScenarioRunner," [Online]. Available: https://carla-scenariorunner.readthedocs.io/en/latest/ (visited on 2023-05-09).

[43] S. Malik, M. A. Khan, and H. El-Sayed, "Carla: Car learning to act — an inside out," *Procedia Computer Science*, vol. 198, pp. 742–749, 2022, 12th International Conference on Emerging Ubiquitous Systems and Pervasive Networks / 11th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare, ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2021.12.316. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050921025552.

[44] G. Rong, B. H. Shin, H. Tabatabaee, Q. Lu, S. Lemke, M. Možeiko, E. Boise, G. Uhm, M. Gerow, S. Mehta, E. Agafonov, T. H. Kim, E. Sterner, K. Ushiroda, M. Reyes, D. Zelenkovsky, and S. Kim, *Lgsvl simulator: A high fidelity simulator for autonomous driving*, 2020. DOI: 10.48550/ARXIV.2005.03778. [Online]. Available: https://arxiv.org/abs/2005.03778.

[45] S. Shah, D. Dey, C. Lovett, and A. Kapoor, *Airsim: High-fidelity visual and physical simulation for autonomous vehicles*, 2017. DOI: `10.48550/ARXIV.1705.05065`. [Online]. Available: `https://arxiv.org/abs/1705.05065`.

[46] deepdrive, *deepdrive GitHub repository*. [Online]. Available: `https://github.com/deepdrive/deepdrive` (visited on 2022-11-16).

[47] The TORCS team, *Torcs - the open racing car simulator*. [Online]. Available: `https://sourceforge.net/projects/torcs/` (visited on 2022-11-16).

[48] The NVIDIA team, *Nvidia drive sim - powered by omniverse*. [Online]. Available: `https://developer.nvidia.com/drive/simulation` (visited on 2022-11-16).

[49] A. Gusev. "Remote control of autonomous vehicle through 5g network with integrated operator-assistance system." (2022), [Online]. Available: `https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/3019924` (visited on 2023-05-10).

[50] NVIDIA. "Drive agx xavier sensors & accesories," [Online]. Available: `https://developer.nvidia.com/drive/ecoystem-xavier` (visited on 2023-05-10).

[51] NVIDIA. "Nvidia drive os," [Online]. Available: `https://developer.nvidia.com/drive/driveos` (visited on 2023-05-10).

[52] NVIDIA. "Nvidia driveworks sdk," [Online]. Available: `https://developer.nvidia.com/drive/driveworks` (visited on 2023-05-10).

[53] W. Sun, *Introduction to imitation learning & the behavior cloning algorithm*, 2021. [Online]. Available: `https://wensun.github.io/CS4789_data/Imitation_Learning_April_8_annotated.pdf` (visited on 2022-12-13).

[54] S. Ross, G. J. Gordon, and J. A. Bagnell, *A reduction of imitation learning and structured prediction to no-regret online learning*, 2010. DOI: `10.48550/ARXIV.1011.0686`. [Online]. Available: `https://arxiv.org/abs/1011.0686`.

[55] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, `http://www.deeplearningbook.org`. (visited on 2023-05-10).

[56] K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, 2015.

[57] P. with Code. "Vgg," [Online]. Available: `https://paperswithcode.com/method/vgg` (visited on 2023-05-15).

[58] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, *Attention is all you need*, 2017. DOI: `10.48550/ARXIV.1706.03762`. [Online]. Available: `https://arxiv.org/abs/1706.03762`.

[59] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, *Bert: Pre-training of deep bidirectional transformers for language understanding*, 2018. DOI: 10.48550/ARXIV.1810.04805. [Online]. Available: https://arxiv.org/abs/1810.04805.

[60] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, *Language models are few-shot learners*, 2020. DOI: 10.48550/ARXIV.2005.14165. [Online]. Available: https://arxiv.org/abs/2005.14165.

[61] S. Khan, M. Naseer, M. Hayat, S. W. Zamir, F. S. Khan, and M. Shah, "Transformers in vision: A survey," *ACM Computing Surveys*, vol. 54, no. 10s, pp. 1–41, 2022-01. DOI: 10.1145/3505244. [Online]. Available: https://doi.org/10.1145/3505244.

[62] D. Chen and P. Krähenbühl, "Learning from all vehicles," in *CVPR*, 2022.

[63] Papers with Code. "Autonomous Driving on CARLA Leaderboard," [Online]. Available: https://paperswithcode.com/sota/autonomous-driving-on-carla-leaderboard (visited on 2022-12-07).

[64] A. Prakash, K. Chitta, and A. Geiger, "Multi-modal fusion transformer for end-to-end autonomous driving," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021.

[65] P. Wu, X. Jia, L. Chen, J. Yan, H. Li, and Y. Qiao, "Trajectory-guided control prediction for end-to-end autonomous driving: A simple yet strong baseline," *arXiv preprint arXiv:2206.08129*, 2022.

[66] A. Hu, G. Corrado, N. Griffiths, Z. Murez, C. Gurau, H. Yeo, A. Kendall, R. Cipolla, and J. Shotton, *Model-based imitation learning for urban driving*, 2022.

[67] Y. Hu, J. Yang, L. Chen, K. Li, C. Sima, X. Zhu, S. Chai, S. Du, T. Lin, W. Wang, L. Lu, X. Jia, Q. Liu, J. Dai, Y. Qiao, and H. Li, *Planning-oriented autonomous driving*, 2023.

[68] B. Jiang, S. Chen, Q. Xu, B. Liao, J. Chen, H. Zhou, Q. Zhang, W. Liu, C. Huang, and X. Wang, *Vad: Vectorized scene representation for efficient autonomous driving*, 2023.

[69] H. Shao, L. Wang, R. Chen, S. L. Waslander, H. Li, and Y. Liu, *Reasonnet: End-to-end driving with temporal and global reasoning*, 2023.

[70] Microsoft. "Visual Studio Code," [Online]. Available: https://code.visualstudio.com/ (visited on 2022-12-10).

[71] Microsoft. "Visual Studio Code: Remote Development using SSH," [Online]. Available: https://code.visualstudio.com/docs/remote/ssh (visited on 2022-12-10).

[72] VMware. “VMware Horizon,” [Online]. Available: https://www.vmware.com/no/products/horizon.html (visited on 2022-12-06).

[73] NTNU HPC Group. “Idun,” [Online]. Available: https://www.hpc.ntnu.no/idun/ (visited on 2022-12-06).

[74] The Slurm team. “Slurm workload manager,” [Online]. Available: https://slurm.schedmd.com/ (visited on 2022-12-12).

[75] Docker Inc. “Docker: Accelerated, containerized, application development.” (2022), [Online]. Available: https://www.docker.com/ (visited on 2022-12-07).

[76] Docker Inc. “Docker compose overview,” [Online]. Available: https://docs.docker.com/compose/ (visited on 2023-05-11).

[77] GNU Project. “Gnu screen,” [Online]. Available: https://www.gnu.org/software/screen/ (visited on 2023-05-24).

[78] Syslabs. “SingularityCE.” (2022), [Online]. Available: https://sylabs.io/singularity/ (visited on 2022-12-07).

[79] P. R. Computing. “Containers on the hpc clusters,” [Online]. Available: https://researchcomputing.princeton.edu/support/knowledge-base/singularity (visited on 2022-12-07).

[80] analog-cbarber. “Issue 1333: stub files for python api.” (2019), [Online]. Available: https://github.com/carla-simulator/carla/issues/1333 (visited on 2022-12-07).

[81] Blender. “Blender,” [Online]. Available: https://www.blender.org/ (visited on 2023-02-28).

[82] NVIDIA. “Driveworks sdk reference: Rig configuration,” [Online]. Available: https://docs.nvidia.com/drive/driveworks-3.5/rig_mainsection.html (visited on 2023-05-25).

[83] NVIDIA. “DriveWorks SDK Reference - Static Camera Calibration,” [Online]. Available: https://docs.nvidia.com/drive/driveworks-3.5/dwx_camera_calibration.html#dwx_camera_calibration_prereq (visited on 2023-03-14).

[84] University College London. “Intrinsic camera parameters calibration,” [Online]. Available: https://mphy0026.readthedocs.io/en/latest/calibration/camera_calibration.html (visited on 2023-03-14).

[85] NVIDIA. “Driveworks sdk reference: Rig file format,” [Online]. Available: https://docs.nvidia.com/drive/driveworks-3.5/rigconfiguration_usecase0.html (visited on 2023-05-25).

[86] Madecu. “Comment on rendering issues in CARLA 0.9.10,” [Online]. Available: https://github.com/carla-simulator/carla/issues/3377#issuecomment-702610143 (visited on 2023-03-14).

[87] qhaas. “Comment on rendering issues in CARLA 0.9.14 on NVIDIA A100,” [Online]. Available: `https://github.com/carla-simulator/carla/issues/4328#issuecomment-898902233` (visited on 2023-05-01).

[88] Chitta, Kashyap and Prakash, Aditya and Jaeger, Bernhard and Yu, Zehao and Renz, Katrin and Geiger, Andreas. “TransFuser GitHub repository.” (2022), [Online]. Available: `https://github.com/autonomousvision/transfuser` (visited on 2023-05-03).

[89] Hao Shao and Letian Wang and RuoBing Chen and Hongsheng Li and Yu Liu. “InterFuser/README.md.” (2022), [Online]. Available: `https://github.com/opendilab/InterFuser#data-generation` (visited on 2023-04-25).

[90] O. Tange, “Gnu parallel - the command-line power tool,” *;login: The USENIX Magazine*, vol. 36, no. 1, pp. 42–47, 2011. [Online]. Available: `http://www.gnu.org/s/parallel`.

[91] Hao Shao and Letian Wang and RuoBing Chen and Hongsheng Li and Yu Liu. “InterFuser GitHub repository.” (2022), [Online]. Available: `https://github.com/opendilab/InterFuser` (visited on 2023-05-03).

[92] PyTorch. “PyTorch Distributed Overview,” [Online]. Available: `https://pytorch.org/tutorials/beginner/dist_overview.html` (visited on 2023-05-03).

[93] Tensorflow. “TensorBoard: TensorFlow’s visualization toolkit,” [Online]. Available: `https://www.tensorflow.org/tensorboard` (visited on 2023-05-03).

[94] The CARLA team. “Submitting an agent for Leaderboard,” [Online]. Available: `https://leaderboard.carla.org/submit/` (visited on 2023-05-04).

[95] The CARLA team. “Leaderboard - CARLA Leaderboard.” (2022), [Online]. Available: `https://leaderboard.carla.org/leaderboard/` (visited on 2023-05-04).

[96] T. Sweeney. “Tweet.” (2018), [Online]. Available: `https://twitter.com/TimSweeneyEpic/status/952661474501111808?s=20` (visited on 2023-04-23).

[97] Wikipedia. “Mercator projection,” [Online]. Available: `https://en.wikipedia.org/wiki/Mercator_projection#Derivation` (visited on 2023-04-24).

[98] N. Subirón. “Carla/geolocation.cpp,” [Online]. Available: `https://github.com/carla-simulator/carla/blob/f14acb2/LibCarla/source/carla/geom/GeoLocation.cpp` (visited on 2023-04-24).

[99] Chitta, Kashyap and Prakash, Aditya and Jaeger, Bernhard and Yu, Zehao and Renz, Katrin and Geiger, Andreas. “transfuser/submission_agent.py.” (2022), [Online]. Available: `https://github.com/autonomousvision/transfuser/blob/a7d4db684c160095dec03851aff5ce92e36b2387/team_code_transfuser/submission_agent.py#L613` (visited on 2023-04-24).

[100] Hao Shao and Letian Wang and RuoBing Chen and Hongsheng Li and Yu Liu. “InterFuser/planner.py.” (2022), [Online]. Available: `https://github.com/opendilab/InterFuser/blob/e0682c3/leaderboard/team_code/planner.py#L48` (visited on 2023-04-24).

[101] M. Hirsch. “pymap3d API documentation,” [Online]. Available: `https://geospace-code.github.io/pymap3d/` (visited on 2023-05-29).

[102] NVIDIA. “Binary CAN File Format - DRIVE AGX Xavier / DRIVE AGX Xavier General - NVIDIA Developer Forums.” (2022), [Online]. Available: `https://forums.developer.nvidia.com/t/binary-can-file-format/154781/8` (visited on 2023-05-02).

[103] CommaAI. “opendbc/hyundai_kia_generic.dbc at d40e429 · commaai/opendbc · GitHub.” (2022), [Online]. Available: `https://github.com/commaai/opendbc/blob/d40e429914a8bc5c2c630726fe097f85f7108185/hyundai_kia_generic.dbc` (visited on 2023-05-02).

[104] Ouster, Inc. “Ouster SDK - Ouster Sensor SDK 0.8.1 documentation,” [Online]. Available: `https://static.ouster.dev/sdk-docs/` (visited on 2023-05-18).

[105] Wikipedia. “Pearson correlation coefficient - Wikipedia,” [Online]. Available: `https://en.wikipedia.org/wiki/Pearson_correlation_coefficient` (visited on 2023-05-29).

[106] Waymo. “About - Waymo,” [Online]. Available: `https://waymo.com/open/about` (visited on 2023-05-29).

[107] Wayve. “Building City Scale Neural Radiance Fields for Autonomous Driving,” [Online]. Available: `https://wayve.ai/neural-rendering/` (visited on 2023-06-07).