Håvard Stavnås Markhus

Reinforced InterFuser for end-to-end autonomous driving in simulated environments

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

Master's thesis

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

NTNU
Norwegian University of
Science and Technology

Håvard Stavnås Markhus

Reinforced InterFuser for end-to-end autonomous driving in simulated 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

NTNU
Norwegian University of
Science and Technology

Abstract

Reinforced InterFuser is presented as an approach for utilizing recent sensor fusion approaches in a reinforcement learning context for end-to-end autonomous driving in simulated environments. The architecture uses both a pre-trained and a custom-trained InterFuser model as a visual encoder to a standard reinforcement learning agent and is compared to a baseline RL agent's training performance. Safety mechanisms deduced from explicit predictions from the InterFuser model is also explored to gauge the benefits of additional safety mechanisms applied to an RL agent.

The approach shows promising results for specific configurations of utilizing the InterFuser architecture as a visual encoder, outperforming the baseline agent on unseen evaluation routes after a limited amount of training. Applying safety mechanisms to Reinforced InterFuser vastly improves the ability to stop at red lights and avoid collisions, improving the agent's ability to navigate through unseen challenging scenarios.

Sammendrag

Reinforced InterFuser presenteres som en metode for å utnytte nyere sensorfusjonsmetoder i en "reinforcement learning" kontektst for ende-til-ende autonom kjøring i simulerte miljøer. Arkitekturen bruker en forhåndstrent InterFuser modell, samt en spesialtilpasset InterFuser modell som en visuell enkoder til en standard "reinforcement learning" algoritme og sammenlignes med en grunnleggende RL agent's treningsprosess og kjøreatferd. Sikkerhetsmekanismer dedusert fra eksplisitte prediksjoner fra InterFuser modellen utforskes også for å finne fordelene med ekstra sikkerhetsmekanismer på en RL agent.

Metoden viser lovende resultater for spesifikke konfigurasjoner av å bruke InterFuser arkitekturen som en visuell enkoder, og overgår den grunnleggende agenten på ukjente evalueringsruter etter en begrenset mengde trening. Å bruke sikkerhetsmekanismer til Reinforced InterFuser forbedrer evnen til å stoppe ved røde lys og unngå kollisjoner, og forbedrer agentens evne til å navigere gjennom ukjente utfordrende scenarioer.

Contents

Figures

Chapter 1

Introduction

1.1 Motivation

Research and deployment of autonomous vehicles are nearing reality. Such technology is already operational in several cities in the US, providing taxi services in specifically dedicated areas. The primary aim of this technology is not to supplant human drivers but to enhance safety and comfort during commutes. Given that most traffic accidents result from human error, full automation and the subsequent surpassing of human driving capabilities could significantly mitigate the frequency of these incidents. Presently, the technology is advancing towards exceeding human driving capabilities. However, the most successful current approaches depend on a combination of sensors and highly detailed pre-existing maps of the environment. This approach does not scale well to new environments, considering the infeasibility of maintaining live 3D maps of the entire world.

Therefore, research into enabling autonomous vehicles to surpass human driving capabilities using live sensor data is vital for making this technology scalable to new environments. Humans can drive sufficiently using vision supported by mirrors and potential signals from the vehicle's interface. However, our driving capabilities are restricted by our field of view and attention span, which limit us to focusing on one thing at a time. In contrast, autonomous vehicles can utilize a constant 360-degree vision and additional information from Lidar sensors, thereby enabling potential for better decision-making. Recent high-performing approaches for autonomous vehicles in simulated environments rely on imitation learning, which, however, is constrained by the performance of an expert demonstrator.

Although deep reinforcement learning has shown promising results for autonomous driving, it is currently outperformed by imitation learning in standardized benchmarks. The latest advancements have utilized transformer architectures for efficient sensor data fusion and have yielded encouraging results.

This sensor fusion approach holds promise and has not yet been extensively explored within the context of reinforcement learning agents. Therefore, this study seeks to fill this gap in the literature.

1.2 Goals and research questions

The primary objective of this thesis is to delve into the advancements of sensor fusion in imitation learning for autonomous driving and assess their applicability in the reinforcement learning space. Although the fusion of sensor data from images and LiDAR has been successfully employed in imitation learning, its exploration within reinforcement learning approaches remains limited. The goal is to ascertain if there are methods where sensor fusion could enhance or improve existing reinforcement learning approaches for autonomous driving within simulated environments. This leads to the first research question:

RQ1: Can reinforcement learning approaches in autonomous driving benefit from novel sensor fusion model outputs as visual state representations?

Predicting interpretable aspects of the environment holds several advantages. For instance, these predictions can be used to improve the interpretability of decisions made by an agent. Furthermore, explicit calculations derived from these predictions can guide when an agent should brake, effectively instituting rule-based safety mechanisms. However, such predictions have not been extensively explored in the reinforcement learning context, which brings us to our second research question:

RQ2: Can reinforcement learning approaches in autonomous driving benefit from safety mechanisms deduced from explicit predictions of the environment?

1.3 Methods

The methodology of the thesis involves setting up a baseline reinforcement learning agent along with a proposed method for utilizing novel sensor fusion methods as part of a reinforcement learning agent. Different approaches of combining the sensor fusion methods are trained and evaluated within a common process that is configured with stability in mind. Part of this process is a reward function with the purpose of providing a simple and stable driving behavior goal for each agent.

1.4 Contributions

The following are the contributions of this thesis to the reasearch field of deep reinforcement learning for autonomous driving in simulated environments:

- A summary and analysis of the state-of-the-art in deep reinforcement learning for autonomous driving and sensor fusion for autonomous driving.
- A novel combination of sensor fusion architecture and reinforcement learning for autonomous driving.
- A study of the usage of various output features from the InterFuser architecture as input to a reinforcement learning agent.
- A simplified framework for configuring agents and running route sequences for reinforcement learning approaches using the CARLA simulator.

1.5 Thesis Outline

Chapter 1: Introduction This chapter introduces the thesis by providing a background and motivation for the thesis. It also provides some context on the goals and questions to explore in the thesis.

Chapter 2: Background and related works This chapter provides the needed theoretical background for the thesis and introduces many concepts used and explored in the thesis. It also presents works heavily related to the methodology of this thesis.

Chapter 3: Methodology This chapter explains the method employed and explored in the practical part of the thesis. It also presents the methods used for conducting the experiments in this thesis.

Chapter 4: Experiments and results This chapter presents the results collected during the experiments.

Chapter 5: Discussion This chapter discusses and analyzes the results presented in the previous chapter and tries to learn from and identify potential shortcomings and strengths of the approach.

Chapter 6: Conclusion and future work This chapter draws conclusions based on the analysis and discussions from the previous chapter, as well as presenting strategies for future work and what can be done to potentially further investigate the research question in mind.

Appendix The appendix contains additional information that is not necessary to understand the thesis, but can be useful for further investigation.

Chapter 2

Background and related work

2.1 Autonomous driving

Autonomous driving can be defined as "The capability of a vehicle to drive partly or fully by itself, with limited or no human intervention." This encapsulates many different levels of autonomy. The Society of Automotive Engineers (SAE) International [1] categorizes self-driving cars into six levels:

- Level 0 - No automation: The driver performs all the tasks.
- Level 1 - Hands-on/shared control: The vehicle can automatically conduct some parts of driving, like steering or acceleration, but not both simultaneously.
- Level 2 - Hands-off: The vehicle can control both steering and acceleration/deceleration.
- Level 3 - Eyes off: The vehicle becomes a fully autonomous system but operates under certain traffic or environmental conditions.
- Level 4 - Mind off: The vehicle is fully autonomous in select conditions defined by factors such as road type or geographic area.
- Level 5 - Optional steering wheel: Full-time performance by an automated driving system under all roadway and environmental conditions.

Currently, the self-driving car market primarily operates at level 3 and above. This means autonomous vehicles can already navigate certain areas without requiring the driver to keep their eyes on the road. An example of such vehicles in operation includes Waymo vehicles [2], which provide a driverless taxi service in specific parts of the US.

The autonomy of these vehicles stems from reading sensor inputs and converting these into actions such as steering adjustments or brake and speed controls. Autonomous driving technology employs a variety of sensors to interpret the environment, which include:

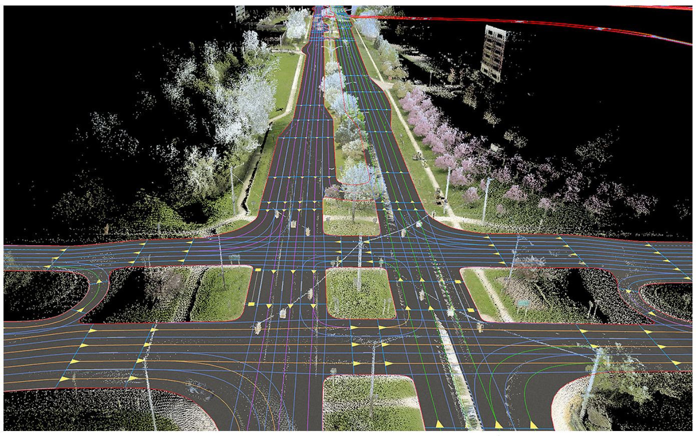


Figure 2.1: HD map example

- **RGB Camera:** A traditional camera affixed to the vehicle captures images at a predetermined rate, acting as the vehicle's eyes. For instance, Tesla's Full Self-Driving feature uses eight cameras installed at different locations on the vehicle.
- **LiDAR Sensor:** This sensor gathers points from the immediate surroundings, which can be used to determine the distance between any physical surface and the vehicle.
- **Inertial Measurement Unit (IMU):** This sensor tracks the force, velocity, orientation, and other data about the vehicle's current state.
- **Global Navigation Satellite System (GNSS):** This technology provides a rough estimate of the vehicle's current location.

These types of sensors create the foundation of how the vehicle understands the world.

Additionally, some autonomous vehicles utilize High Definition (HD) maps, particularly for urban driving. As shown in Figure 2.1, HD maps provide detailed, 3D representations of the environment, offering valuable information for driving decisions. These maps contribute significantly to the performance of some autonomous driving systems, particularly in areas with varying weather conditions such as Nordic winters. During these conditions, vehicle sensors can be obstructed, and roads may be covered in snow, making lane detection challenging. HD maps can supply essential information about the structure and placement of obscured roads. However, these maps are geographically limited and may not be available everywhere. Relying solely on such highly-detailed mapping information is not a practical solution for achieving full autonomy in driving. It is infeasible to maintain detailed data for all possible roads. Thus, fully autonomous systems must be designed to function based on the data gathered by the vehicle's own sensors to ensure universal operation.

Software capable of interpreting this sensor data and converting it into reasonable driving behavior is essential. The development of such software involves intensive machine learning training and frequent performance evaluations. Given the potential for catastrophic results in real-world testing, and considering the inefficiencies of this approach, simulations become crucial. They offer a safer and more efficient environment for training and evaluating autonomous driving software before deployment on a physical vehicle.

2.1.1 Simulated environments

Driving simulators are invaluable tools for training and evaluating self-driving software. They offer the ability to simulate dangerous situations that would be unethical or impractical to generate in the real world. Furthermore, simulators allow for more frequent training and evaluation sessions and can run driving scenarios at an accelerated pace compared to real-world conditions.

However, these advantages come with some caveats. Simulations do not perfectly represent the real world, which could lead to challenges when deploying driving software trained and tested exclusively in simulated environments. This discrepancy, known as "distribution mismatch," can be substantial and poses a challenge to the effective use of simulations. Additionally, while simulations strive to mimic real-world conditions, they inevitably fall short in terms of photorealism, leading to further mismatches between simulated and real-world scenarios.

High-quality simulators are often proprietary, making them inaccessible to many researchers and developers. Nonetheless, there are open-source options specifically designed for self-driving agents, such as the Carla simulator [3]. On the other hand, proprietary alternatives like NVIDIA Drive Sim [4] offer highly photorealistic graphics and physics, but as of June 2023, they remain within an Early Access Program and are not publicly available.

2.1.2 Carla driving simulator

The CARLA simulator is a comprehensive platform designed for the critical tasks of developing, training, and validating autonomous driving systems. This platform is built upon open-source code and protocols, allowing for transparency and adaptability in various research contexts. One of the unique characteristics of CARLA is the provision of open digital assets. These assets, encompassing urban layouts, buildings, and vehicles, have been specifically designed for simulations in the context of autonomous vehicles and are freely accessible. The simulator is characterized by its flexibility, providing the ability to configure a multitude of sensor suites and environmental conditions. This allows for a broad spectrum of testing scenarios, which are essential in the autonomous vehicle research landscape. CARLA also enables users to

Figure 2.2: Carla driving simulator

exercise complete control over all static and dynamic actors within the simulation. This feature is crucial for creating complex and detailed scenarios, enhancing the accuracy and relevance of the simulation results. Moreover, the platform supports map generation, a feature that enriches its capabilities and allows developers to create realistic and varied testing environments. This, in combination with the other aforementioned features, positions CARLA as a crucial tool in the field of autonomous vehicle research and development.

Sensors

To facilitate the development of autonomous driving systems, CARLA provides a wide range of sensors that can be attached to a vehicle. These sensors are designed to replicate the functionality of real-world sensors, providing a realistic simulation environment. The sensors can be attached to any vehicle in the simulation, and the data they capture can be used to train an autonomous driving agent. Some of the important sensors in the suite are:

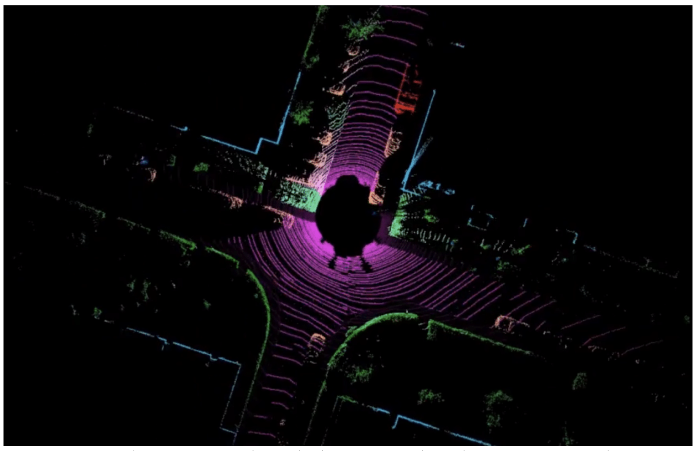


Figure 2.3: Lidar point cloud data as a bird's eye view histogram

- **RGB camera:** Captures footage from the vehicle's point of view.

- **LiDAR sensor:** Captures point cloud data from the agent's surrounding environment. This sensor is a realistic replication of real-world Lidar sensors. The data can be restructured to create a 2D bird's eye view representation as shown in figure 2.3
- **GNSS sensor:** Tracks the vehicle's approximate location, functioning as a GPS for the car.
- **IMU sensor:** Tracks the vehicle's acceleration and orientation.

The simulator also provides a range of ground truth sensors that can be used for auto-labeled data in training an agent:

- **Semantic segmentation camera:** Gathers 2D images from a set viewpoint where each pixel is semantically labeled based on the object it represents.
- **Instance segmentation camera:** Similar to the semantic segmentation camera, but instead of labeling pixels based on the object they represent, they are labeled based on the instance of the object.
- **Semantic LiDAR sensor:** Adds semantic and instance ground-truth to points in the point clouds. This means the identifier of the object hit (the instance), as well as the semantic tag of the instance, whether it is a vehicle, pedestrian, etc. It also includes the cosine between the angle of incidence and the normal of the surface hit.

These sensors cover most of the needs for running a simulation for testing autonomous driving systems and are the only ones relevant for this thesis. Additional available information that can be used for setting ground truth data includes lane waypoints.

Waypoint API

The Waypoint API is an essential part of the simulator for extracting vital environmental information. It provides access to the position of continuous points from all the lanes in the environment, which includes the implicit direction of the lane at each point. These waypoints can be used as examples to train agents to follow lanes correctly.

Maps

The simulator has a set of default maps that are used for testing and training. These maps are fictional and are designed to simulate real-world environments.

- **Town01**: The original CARLA map featuring a grid layout common to many US cities. It offers a variety of junction types, with a mix of commercial and residential buildings.
- **Town02**: Similar to Town01, but with a unique layout. It provides a comparable urban driving experience with a diverse array of building

types and junctions.

- **Town03**: A suburban map characterized by winding roads and a variety of building types, offering a unique driving challenge.
- **Town04**: This map, primarily a highway layout, is ideal for testing high-speed driving scenarios and overtaking behaviors.
- **Town05**: A rural map with country roads and fewer buildings, offering a distinctly different driving experience compared to other maps.
- **Town06**: A complex urban environment designed to challenge any autonomous vehicle, with a variety of road types, complex junctions, and high-density buildings.

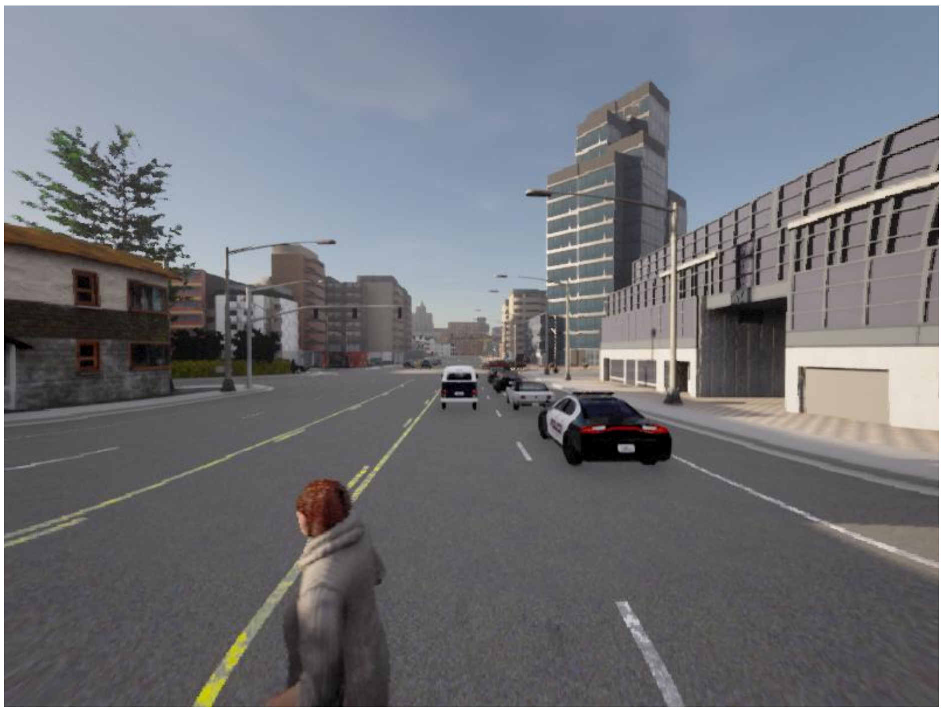


Figure 2.4: Dangerous pedestrian crossing scenario

Scenarios A scenario is an event triggered when a simulated ego vehicle enters a designated area. It comprises a set of actors and conditions that must be fulfilled for the scenario to be considered completed or passed. An example of a scenario is demonstrated in figure 2.4.

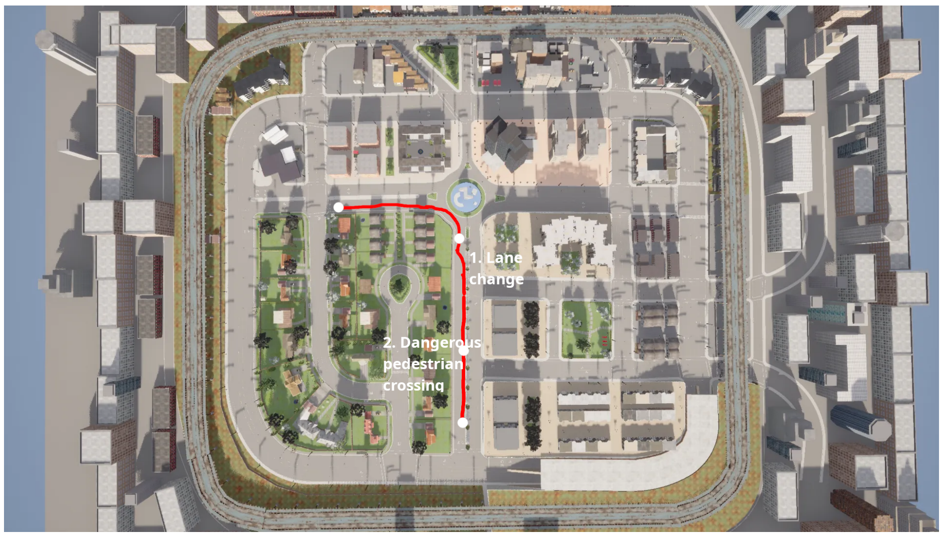


Figure 2.5: Short example of a route for Town03 in Carla

Routes A route, as depicted in figure 2.5, is a sequence of waypoints that the agent must follow. Along with the route, a set of scenarios can be defined to accompany the route. This results in a path through a town where at certain points, scenarios are triggered and must be addressed by the agent. The route consists of a set of high-level waypoints and from these, a set of dense waypoints can be interpolated. Carla provides functionality for interpolating These dense waypoints to create a privileged global plan.

Commands Along with the dense waypoints, there are accompanying high-level commands within a global plan for a route. These high level commands can be one of:

- **Go Straight**: The agent should continue straight ahead.
- **Follow lane**: The agent should follow the lane.
- **Left**: The agent should turn left at the next intersection.
- **Right**: The agent should turn right at the next intersection.
- **Change lane left**: The agent should change to the left lane.
- **Change lane right**: The agent should change to the right lane.

Carla Scenario Runner The Carla Scenario Runner is an open-source tool developed for running driving scenarios. It utilizes the Carla python API to manage traffic and set up routes and scenarios for an agent to react to. To run scenarios or routes with the scenario runner, the user must specify a configured agent, as well as a scenario or list of scenarios with the option to specify a set of routes. The scenario runner then uses the Carla Python API to orchestrate the configured routes and/or scenarios, while injecting the configured agent into the simulation.

Agents For an agent to interact with specified scenarios, a custom implementation overriding specific methods in a simple interface class must be provided to the scenario runner. Sensors and logic based on the data from these sensors have to be specified in the agent class. The sensors specified will be attached to the "hero" vehicle that the scenario runner sets up in the simulation. The logic specification receives data from the sensors at each frame in the simulation, where the logic of the agent can be implemented and based on. This means the logic flow is to receive sensor data, then decide action. The logic specified in the agent class runs every frame of the simulation and is used to control the "hero" vehicle.

The CARLA Leaderboard

To test the proficiency of trained agents, Carla provides an online leaderboard where agents can be benchmarked on their driving proficiency. These benchmarks are a set of held-out routes and scenarios not seen during training of an agent. Agent class specifications with compatible sensor specifications that are within the leaderboard's requirements can be submitted to the leaderboard. Two distinct types of benchmarks are defined: one for agents driving using an HD map and another for agents using sensors only. This thesis will focus on the sensor benchmark.

Offline leaderboard The scenario runner tool provides a set of 76 different routes across the different towns described in 2.1.2. 50 of these are reserved as training routes, and 26 are reserved as test routes. The training set consist of routes for Town01, Town03, Town04, and Town05. The test set consists of routes for Town02, Town04 and Town06. All the routes consist simply of sparse waypoints for the agent to follow within the towns. These routes are however accompanied by a large set of specific scenarios for each town. The routes and scenarios are all specified in files passed to and read by the scenario runner that facilitates these specifications. A link to these files are provided in the Appendix: 6.2.

2.2 Learning for self-driving agents

2.2.1 Modular approach

The methodologies for training self-driving agents primarily fall into two categories: modular approach and end-to-end learning using a neural network. In a modular approach, the problem is often divided into two parts: perception and planning. The perception module is responsible for processing sensor data and creating a comprehensible model of the environment. On the other hand, the planning module interprets this model and decides the most appropriate actions based on the current state of the world. An example of

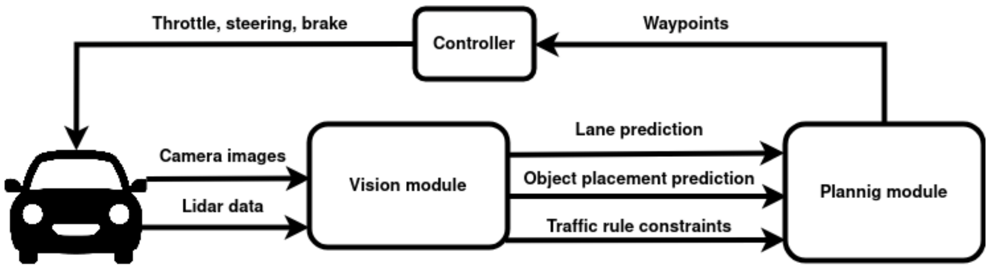


Figure 2.6: Modular system

this approach is Tesla's Full Self Driving Beta, which separates the machine-learning based perception module and rule-based planning module, enabling each module to specialize and excel in its specific domain.

2.2.2 End-to-end approach

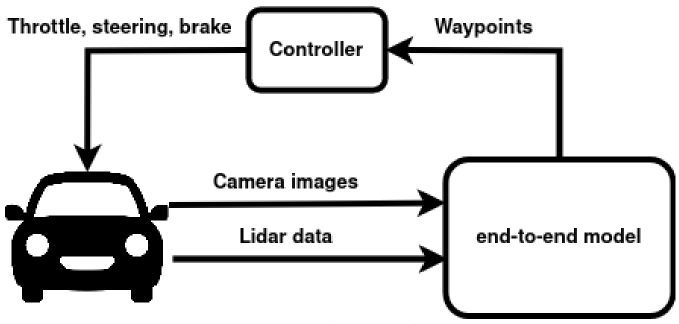


Figure 2.7: End-to-end system

In an end-to-end model, the system processes sensor data and directly generates corresponding actions. An intermediate step in this process might involve the generation of way-points or trajectories, which can be interpreted into specific actions via a controller. The controllers used can range from a simple proportional–integral–derivative controller (PID-controller) to more advanced options such as a model predictive controller (MPC).

A key distinction of end-to-end approaches lies in the decision-making process. Instead of relying on potentially explainable hand-crafted algorithms, these models typically use a neural network or another "black-box" method. While this can introduce ethical challenges due to the lack of transparency in decision-making, it also opens up possibilities for surpassing the capabilities of explicitly defined algorithms.

End-to-end models typically employ two main strategies: reinforcement learning and imitation learning. Some approaches may even combine elements of both. These strategies and their potential advantages and drawbacks will be further discussed in the following sections.

Reinforcement learning

Reinforcement learning is a method for training agents without the need for labeled data. It revolves around the concept of trial and error and is typically formulated within the framework of Markov Decision Processes (MDPs).

Markov Decision Process A Markov Decision Process (MDP) provides a mathematical framework for modeling decision-making scenarios where outcomes are partially random and partially under the control of a decision maker.

An MDP consists of four components: a set of states (S), a set of actions (A), a reward function (R), and a transition probability function (P). At each time step, the system is in a state $s \in S$, and the agent selects an action $a \in A$. This results in the environment transitioning to a new state $s' \in S$ and the agent receiving a reward $R(s, a, s')$. The goal of the agent is to find the optimal policy π that maximizes the expected cumulative reward. The policy π is a mapping from states to actions that dictates the agent's behavior.

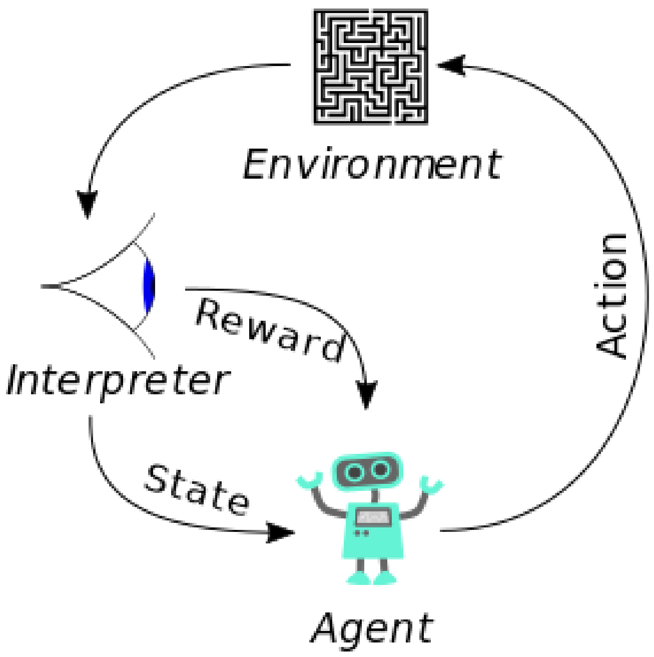


Figure 2.9: Reinforcement learning diagram - By Megajuice - Own work, CC0, https://commons.wikimedia.org/w/index.php?curid=57895741

Environments Reinforcement learning environments are interactive settings where an agent can learn to make decisions. Gymnasium is a common tool for developing and comparing reinforcement learning environments. It provides pre-defined environments for agents to interact with and learn from. For example, env = gym.make('CartPole-v1') creates a CartPole environment. The step function of an environment lets the agent interact with the environment, returning the new state, reward, and a done flag. The reset function is used to initialize the environment back to its starting state. Gymnasium's simplicity makes it ideal for testing and benchmarking reinforcement learning algorithms. These environments are compatible with many reinforcement learning libraries, including Stable Baselines [5] and Ray RLlib [6]. Creating a custom environment in Gym is achieved by defining a class that inherits from gym.Env and implementing the specific methods, "init", "step", "reset" and optionally "render".

An example of a simple custom environment:

```python
import gymnasium as gym
from gym import spaces

class CustomEnv(gym.Env):
    def __init__(self):
        super(CustomEnv, self).__init__()
        # example for a binary action space
        self.action_space = spaces.Discrete(2)
        # example for a 1D observation space bounded between 0 and 100
        self.observation_space = spaces.Box(low=0, high=100, shape=(1,))

    def step(self, action):
        # Update environment state
        # Calculate reward
        # Check if episode is done
        return observation, reward, done, info

    def reset(self):
        # Reset state
        return observation

    def render(self, mode='human'):
        # Visualize environment state
```

Learning The learning strategy of reinforcement learning is about learning by interacting with the environment. The agent tries to optimize its expected reward by learning a policy based on rewards gotten from interacting with the environment. An important aspect of reinforcement learning is the "reward signal" coming from the environment, usually implemented as a reward function calculated from the state (s'). If the environment is a scenario in the CARLA simulator, the reward could be calculated based on the agent's placement in a lane, whether it's following traffic rules and if the current speed is desired based on the condition of the world. This way reward signals can be given to a driving agent, in which reinforcement learning can be used to train a driving Policy.

Classical reinforcement learning includes learning a direct mapping from state to optimal action from given state based on experiences gathered during training. This approach is however not sufficient in cases where the state space is massive, like in the self-driving problem. Luckily, reinforcement learning can utilize function approximators, typically neural networks to account for the massive state space. This way, information like sensor data can be used as part of the state for determining actions in reinforcement learning.

Off-policy, on-policy, exploration and exploitation We can split policies into a behavior policy π_b, and an update policy π_p. The behavior policy is used during interaction with the environment and describes what actions the agent should take given a current state s. The update policy is the policy that is trained based on the decisions and reward signal gathered from the behavior policy. In on-policy reinforcement learning, the behavior policy and the update policy is the same. For off-policy reinforcement learning, the behavior policy and update policy can be different. The update policy that is learned could be different from the behavior policy used during exploration of the environment. In CARLA, an expert agent with "cheat sensors" and an optimal policy can be used as a behavior policy, in which the reinforcement learning algorithm can use the experiences gathered from the behavior policy to learn optimal behavior. This can be seen as a type of imitation learning, or a special case of imitation learning where reinforcement learning is used to achieve it.

Imitation learning

Imitation learning is a way of learning that focuses on imitating and generalizing the behavior of an expert demonstrator. When it comes to imitation learning, no reward function is needed for learning a driving policy. The learned policy is simply trying to mimic the behavior showcased in expert demonstrations and should generalize to novel situations and still perform well. The drawback of this approach is that technically, the learner cannot be better than the expert demonstrations in any way, however learning is more efficient than with reinforcement learning, as the process does not have to go through a rigorous phase of trial and error to coincidentally find beneficial actions. Any action the demonstrator makes is implicitly the optimal action to take. Pure behavior cloning does however suffer from large error propagation when differing from the distribution found during training. If an expert agent is optimal, it won't make any mistakes. Expert data used as training data for a learning agent will then not include experiences where a small mistake has been made and how to correct this small mistake. This means a small mistake might push the agent outside of experiences found during training, making it increasingly difficult to remedy the mistake. Appropriate noise in the training data is needed for avoiding this problem, but has the drawback of introducing non-optimal decisions in the training data. Another approach to handle this is to use Generative Adversarial Imitation Learning (GAIL) [7], which treats the imitation learning as an adversarial game.

For autonomous driving, expert demonstration data can include sensor information as input and a trajectory as well as semantically labeled data about the environment as target data. In CARLA there are autopilot agents that can generate expert data including waypoints at any time step, as well as information from privileged sensors like semantic maps or other labeled data.

For any type of self-driving agent, it is important for it to be able to decode the world around it. The information gathered from sensors can be very exhaustive and has to be interpreted efficiently in order for the system to properly learn how to drive. In other words, any driving agent must either way learn to see and understand its environment in a meaningful way.

Combined approaches

Imitation learning and reinforcement learning can be combined to speed up the learning process, like in AlphaGO [8] and GRI [9]. These combine imitation learning and reinforcement learning to handle issues of a slow initial learning phase to trial and error in classical reinforcement learning by providing expert demonstrations with a constant reward as part of the training loop. These expert demonstrations can practically speed up the initial tedious training phase of reinforcement learning. For complex learning tasks like self-driving, where a lot of trial and error has to be done for even learning the simplest tasks of following the road, expert demonstrations can be very helpful for guiding the learning process initially.

2.3 Vision

Human drivers rely on just two eyes to understand the state of their driving environment. Despite this, humans can effectively navigate the complex dynamics of driving. However, a self-driving agent potentially has access to significantly more information, enabling superior decision-making capabilities and faster reaction times. This is made possible by equipping vehicles with a multitude of sensors, far surpassing the capabilities of the human eye alone. To decode the vast amount of information that originates from cameras, LiDAR, and other sensors integral to autonomous vehicles, visual intelligence is required. Developing manual algorithms for processing this data is challenging, thus necessitating the use of neural networks for decoding information from complex datasets such as images. Convolutional neural networks are typically employed due to their proven efficacy in image processing.

2.3.1 Convolutional Neural networks

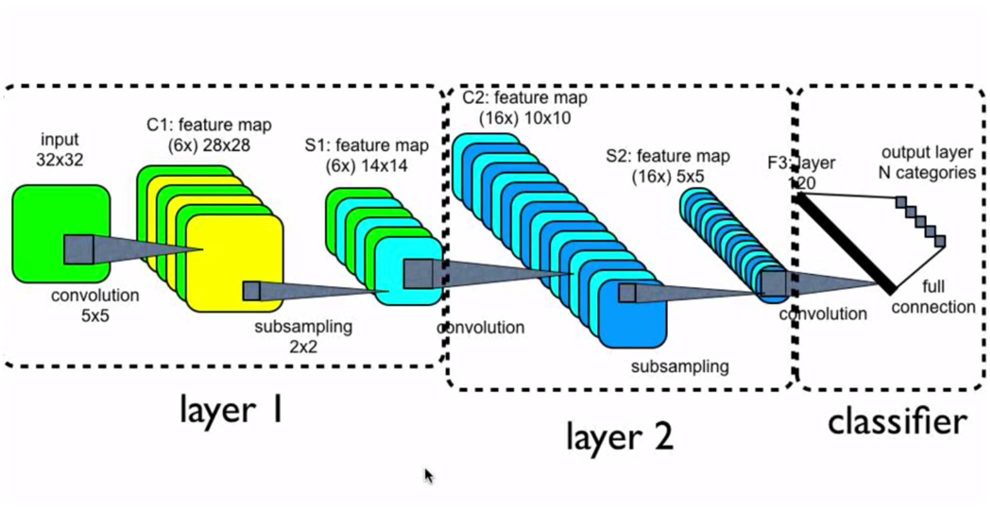


Figure 2.10: Convolutional Neural network for image classification

Convolutional Neural Networks have been the state-of-the-art for different image classification and visual intelligence tasks since it started gaining traction in the 2000s when the GPU implementations started popping up. A Convolutional layer in neural networks works as feature extractors as a way of learning important features about images needed for tasks in visual intelligence. CNNs have for a long time been the most important factor in the success of visual intelligence, and is still widely used in the field. In self-driving they are mainly used for semantic segmentation and object detection tasks.

2.3.2 Transformers

In recent years, the field of neural network architecture has witnessed the rise of transformers, originally used for Natural Language Processing (NLP) tasks. Introduced in 2017 by Vaswani et al. [10], transformers have effectively superseded RNNs [11] and LSTMs [12] within NLP. The primary reason for this shift is the efficiency of transformers in capturing context through their self-attention mechanism. They are also parallelizable, enabling effective training on modern GPUs. Unlike transformers, RNNs and LSTMs must process tokens sequentially to predict the next token in a sentence, which is required to construct the final output in tasks such as next sentence prediction. On the other hand, transformers can process each token in a sentence in parallel, resulting in significant speed improvements over traditional NLP neural network architectures. Furthermore, RNNs suffer from the vanishing gradient problem, meaning that the context of a token is determined by the outcome of the previous token, and hence, each token prediction lacks direct, precise information about preceding tokens. In tasks such as language translation, understanding the context of a word is crucial. Therefore, attention mechanisms were integrated into RNNs to account for this context. However, during the development of the transformer architecture, these attention mechanisms were found to be sufficiently powerful in their own right.

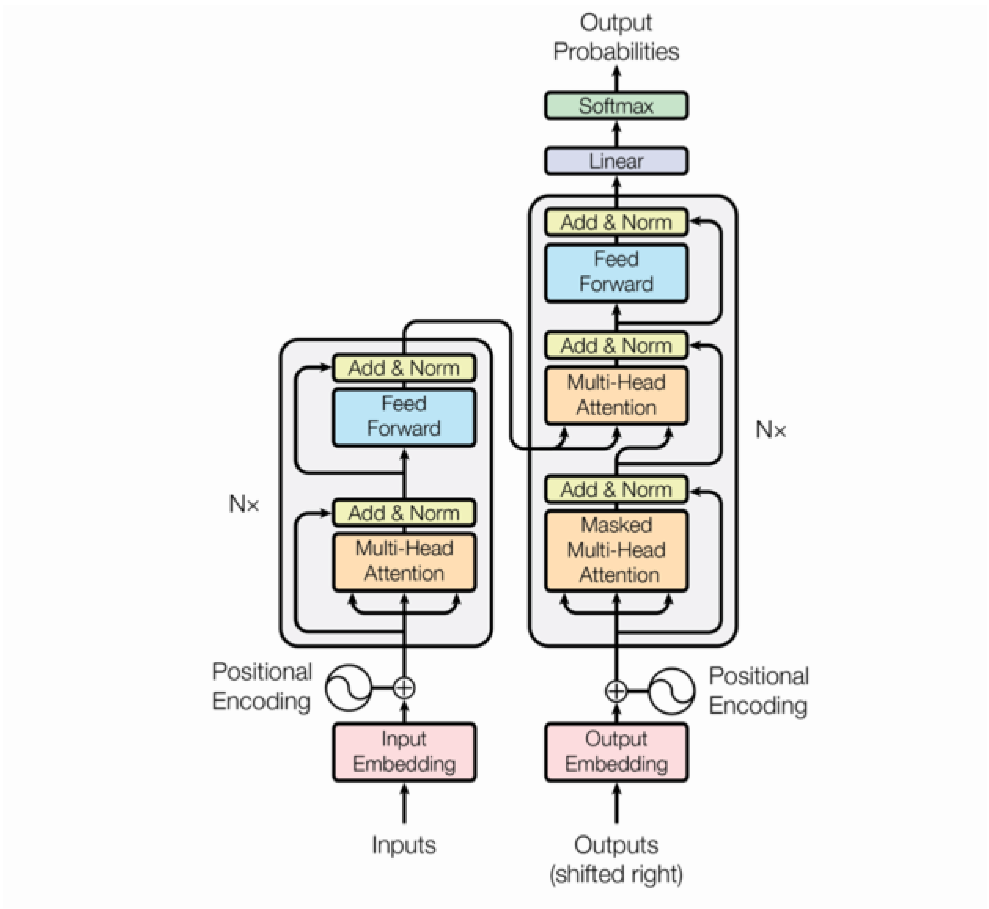


Figure 2.11: Transformer architecture. Illustration from Vaswani et al. [10]

As illustrated in Figure 2.11, the transformer architecture consists of two main components: the encoder (left) and the decoder (right). In a sequence-to-sequence language task, the encoder receives a set of tokens to be translated, while the decoder processes the output tokens to which the input should be translated. Observing Figure 2.11, it's evident that the output serves as an input in the decoder. During the training process, this output forms part of the input, and the ground truth predictions are simply the output tokens shifted one position forward. This design allows each word to train the network in parallel, with each word's output prediction being the next token in the sentence.

Encoder

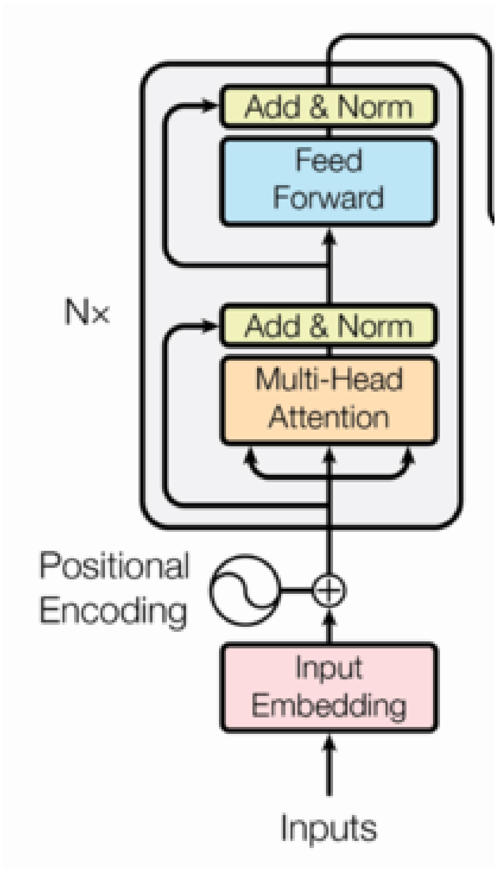


Figure 2.12: Transformer encoder. Illustration from Vaswani et al. [10]

Initially, words are converted into embedded vector representations using a pre-trained vectorization model. This word embedding encapsulates the "meaning" of a word. It is then summed with a positional encoding to represent the "context" of the word. Each resulting vector undergoes processing in a global self-attention block, which includes a Multi-head attention layer followed by an Add & Normalization layer. The attention is computed using vectors of queries (Q), keys (K), and values (V), as illustrated in equation 2.1:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.1}$$

During the first block in the encoder, the global self-attention block, Q, K, and V all derive from the input vector. Each attention vector calculated reflects the relevance of a word in relation to others within the sentence. During the multi-head attention block, the vectors are projected into multiple learned versions of Q, K, and V. These are processed in parallel through the block, then weighted-summed using other learned values. This approach allows the representation of information from different subspaces, which according to Vaswani et al. [10], enhances performance. The outcome of the entire block is an attention vector for each word. These vectors are simultaneously processed in the next block, a feed-forward neural network. A combination of these two blocks constitutes one layer in the encoder, which can be stacked N times

Decoder

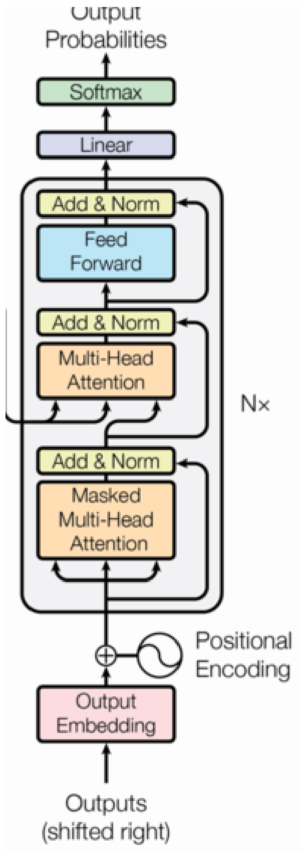


Figure 2.13: Transformer decoder. Illustration from Vaswani et al. [10]

The output from the encoder is passed to the decoder, serving as keys (K) and values (V) in the decoder's second attention layer. The decoder's input operates similarly to the encoder's: each sentence is converted into a set of word embeddings, which are summed with a positional encoding for each word. During this block, the attention vectors are masked so that the i^{th} word lacks information about subsequent words ($i + n$). This process is essential for facilitating learning. The encoder's output is then used as K and V values, while the result from the first block of the decoder serves as the query (Q), and is passed to another Multi-head attention layer. Subsequently, the resulting vector undergoes processing in a feed-forward network and an Add & Normalization step, akin to the encoder. This architecture can also be stacked N times. The transformer's final output emerges from a linear projection and a softmax operation, effectively predicting the next word. Since the i^{th} word is not dependent on the result from the $i - 1^{th}$ word, the entire process can occur in parallel.

In recent years, transformers have propelled the success of Large Language Models (LLMs). Notable examples include ChatGPT with GPT-4 [13] — which is constructed with extensive stacks of transformer decoders and fine-tuned using reinforcement learning with human feedback — and BERT [14], characterized by its use of multiple transformer encoders. Despite these achievements in language processing, this thesis shifts the focus to images and sensor data. The subsequent section will explore the application of this architectural framework in the computer vision tasks necessary for autonomous driving.

2.3.3 Vision Transformers

Vision Transformers (ViTs) apply transformer encoder architecture for tasks such as image classification or other vision-related assignments. The concept was introduced by Sharir et. al. [15] and achieved state-of-the-art results on object detection datasets like COCO [16].

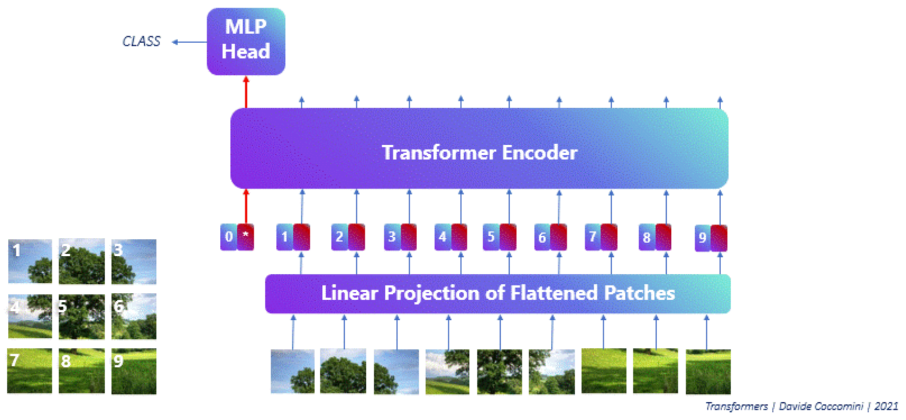

CLASS
MLP Head
Transformer Encoder
0
*
1
2
3
4
5
6
7
8
9
Linear Projection of Flattened Patches
Transformers | Davide Coccomini | 2021

Figure 2.14: Vision transformer, By Davide Coccomini - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=110678226

The transformer scales quadratically with the input size, as the multiattention block performs calculations based on every other input value for each token. This process generates an attention vector signifying the token's relevance to the remainder of the input. When dealing with images, each pixel must possess an attention value for each other pixel, making it computationally infeasible due to the rapidly expanding size of attention vectors. With e.g an image of size $224x224$, the amount of patches would add up to $50,176$ To facilitate the transformer's work with images, each image is segmented into 16x16 pixel patches. These patches are subsequently flattened into a vector using a linear projection and affixed with a positional encoding. Alongside the patches, an extra learned parameter is incorporated into the input as a predictive value. For image classification tasks, the output for this additional learned parameter is transferred to a Multilayer Perceptron (MLP) head which predicts the class, utilizing conventional loss functions for model learning. ViTs are highly data-dependent, necessitating vast quantities of training data to perform effectively on vision tasks. The authors of "attention is all you need" [15] recommend pretraining ViTs on large datasets, despite the high computational expense, and then fine-tuning them on task-specific, smaller datasets. When fine-tuning, the required training parameters are significantly reduced, as they only include parameters of a prediction head outside the transformer encoder.

An alternative to the Vision Transformer is the Shifted Windows Transformer (Swin Transformer), introduced by Liu et al. [17]. Swin Transformers are conceptually similar to ViTs but address the challenge of scaling to high-resolution images. Instead of applying self-attention to all patches globally at each layer, Swin Transformers limit the self-attention mechanism to a subset of vectors for each layer. To maintain a global context, the self-attention mechanism shifts at every layer, effectively acting as a "sliding window" across the patches iteratively along the layers. This method allows the complexity of the encoder-blocks to remain manageable, even when the patch quantity is increased to handle higher-resolution images.

2.4 Related works

2.4.1 Implicit Affordances - 2019

In this work, Toromanoff et al. [18] introduced a technique coined *Implicit Affordances* to address the challenges associated with applying reinforcement learning (RL) to complex environments such as autonomous driving. Implicit Affordances refer to intermediate encoded vectors used to predict driving-relevant state information from the environment.

Before the RL stage of learning how to drive is initiated, a visual encoder and decoder are trained to predict crucial information about the state of the environment. During the RL phase, the output from the visual encoder is utilized as part of the environment's state during training. The weights of the visual encoder are kept constant during this RL training. In essence, the agent first learns to "see" before it learns to drive.

Architecture

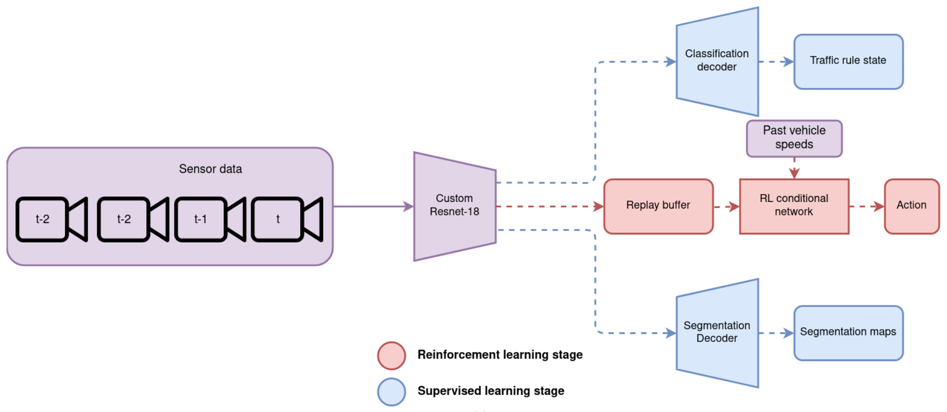


Figure 2.15: Implicit affordances architecture

As depicted in Figure 2.15, the system uses the previous four frames from a center camera as its input. During the supervised phase, the blue and purple components of the system are trained. The weights from these components are frozen during the subsequent RL phase, and training is only performed on the Conditional Reinforcement Learning network. This conditional network is a combination of 6 equivalent fully connected neural networks, each individually chosen and trained for a specific higher-order input command from the simulation, such as "turn left," "turn right," or "go straight." The RL head uses the output from the visual encoder as the world state, in addition to the four previous vehicle speeds.

Visual Encoder and Decoder

Training the visual encoder involves the use of a custom ResNet-18 architecture. The encoder processes inputs of size $(4 \cdot 288 \cdot 288 \cdot 3)$, which represent sequential sets of 4 RGB images, each with a resolution of 288x288 pixels. The output is a $522 \cdot 4 \cdot 4$ vector, serving as an embedded vector used as implicit features. These features are then passed onto a Semantic Segmentation Decoder, which predicts the semantic segmentation of the images. Additionally, a High-Level Traffic Information Decoder uses these vectors to predict the traffic light state, the presence of an intersection, and the lane position. The training data for this process is derived from the CARLA autopilot, which provides examples where the agent is driving in the center of the lane. This presents a challenge when training the RL agent, as it often veers off-road early in the training process - a condition not accounted for in the training data. The authors address this problem by augmenting the viewport of the autopilot during data collection. This involves randomly shifting the camera positions offset to the vehicle, generating various perspectives of the environment from outside the center of the lane. This process introduces sufficient noise in the training data to effectively train the RL agent.

Reinforcement Learning Head

The RL head builds upon the authors' previous work, Rainbow-IQN APE-X [19]. This represents a distributed value-based reinforcement learning approach, similar in nature to DQN. Utilizing a distributed approach enables multiple CARLA simulations to run simultaneously during the RL loop, expediting the training process. The use of a visual encoder reduces the state of the environment significantly compared to using raw sensor data. Considering that RL demands substantial data, it is essential for the model to be compact enough for efficient inference. Another critical point is that most RL approaches leverage a replay buffer for training, which, when populated with numerous full RGB images for each saved transition, can lead to memory issues. The visual encoder mitigates this problem by significantly reducing the size of the replay buffer per transition.

Reward function

The reward function used during training mostly depends on information extracted from the waypoint API described in 2.1.2 from CARLA. The reward function is split into three main components:

- Desired speed
  The reward is given if the agent is driving at the desired speed. The desired speed scales linearly based on traffic light state and the distance to obstacles in its direction. Otherwise, the desired speed is set to the speed limit.
- Desired position
  This depends heavily on the waypoint API. The closer the agent is to the middle of the lane, the higher the reward.
- Desired rotation
  Using the waypoint API, the further the agent's heading differs from the heading of the closest waypoint, the lower the reward.

As well as giving rewards for these three main components, the agent is also punished for colliding into anything, driving too far from the lane and stopping for no reason.

The approach solves many of the problems present if trying to apply reinforcement learning on the driving problem. The approach won the camera

Only track of the CARLA challenge at the time, showing that pure reinforcement learning approaches can be effective in the self-driving task, at least in a simulated environment.

2.4.2 GRIAD - 2021

This work, *General Reinforced Imitation for Autonomous Driving* (GRIAD), from Chekroun et al. [9] is in many ways similar to the approach in section 2.4.1, and is placed 7th on the Carla SENSOR track leaderboard. The main difference in this work is that it uses the author's proposed General Reinforced Imitation (GRI) reinforcement learning strategy for teaching the agent how to drive (AD). This means that instead of only learning from experiences during simulation, examples from an expert demonstration agent are also incorporated into the training of the autonomous driving agent. This is done by setting a constant reward r_{demo} for all actions taken by the expert agent and adding episodes with these to a common replay buffer together with classic exploration state transitions.

GRI algorithm

Algorithm 1: GRI algorithm

```
Input: $r_{demo}$ demonstration reward value, $p_{demo}$ probability of using demonstration agent

Initialize empty buffer $\mathcal{B}$
while not converged do
    if $len(\mathcal{B}) \geq minbuffer$ then
        DRL network update;
    end
    if $random.random() \geq p_{demo}$ then
        $e \leftarrow$ collect episode from exploration agent;
        $\mathcal{B} \leftarrow \mathcal{B} \cup e$
    else
        $e \leftarrow$ collect episode from expert agent ;
        $\mathcal{B} \leftarrow \mathcal{B} \cup e$
    end
end
```

Algorithm 1 shows the GRI algorithm. At each step, the algorithm collects an episode comprising a sequence of state transitions, each of which includes the state, the action taken, the resulting reward, and the new state. For a probability p_{demo} this episode is drawn from an expert demonstration dataset, otherwise the episode is gathered by an online exploration agent, in essence normal reinforcement learning.

For the autonomous driving task, collecting an episode as the exploration agent means running the agent through a CARLA simulated scenario and saving the state transitions until the episode ends. An episode concludes when either the agent crashes or the maximum episode length is reached. The demonstration agent episode data, which consists of 200k state transitions, is gathered using a CARLA Autopilot. The reward for each state transition sampled from the demonstration agent is set to a constant r_{demo}.

GRI architecture

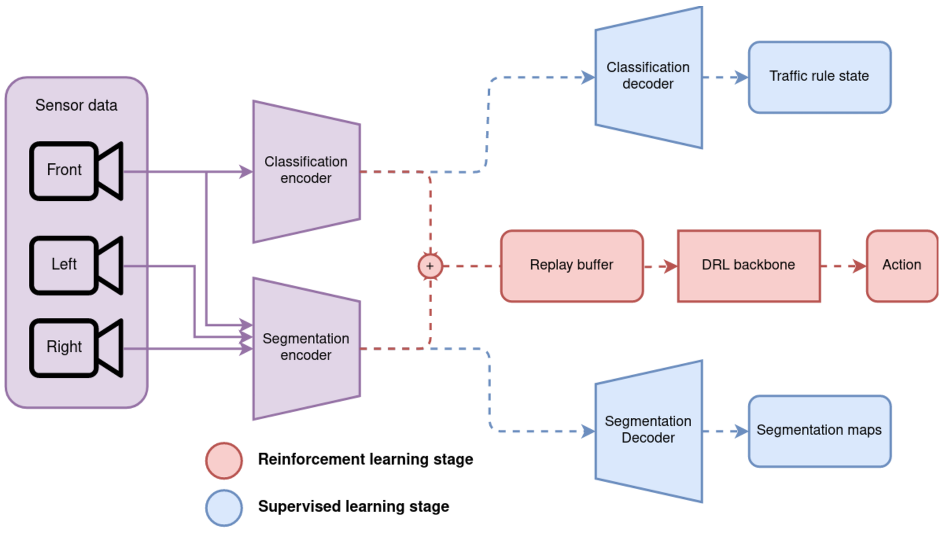


Figure 2.16: System architecture of GRI for autonomous driving

As seen in figure 2.16, the system uses two distinct modules for classification and segmentation encoding. The system also employs three cameras, one front, left and right RGB camera as the sensor input. For the classification encoder, only the input from the center camera is utilized. During the supervised learning stage, the encoders and decoders are optimized and a perception dataset is used for training the system. During the reinforcement learning stage, the visual subsystem, the weights for the blue and purple components are frozen, while the DRL weights are optimized instead. For distributed training of the DRL network, the system also utilizes Rainbow-IQN Ape-X [19].

2.4.3 TransFuser - 2022

This work is an approach by Chitta et al. [20] that, at the time of its publication, surpassed all other submissions on the CARLA sensor track leaderboard and achieved state-of-the-art performance on this specific self-driving benchmark. As of this writing, the submission holds 5th place. The work is an imitation learning approach to self-driving and focuses on sensor fusion using the transformer architecture.

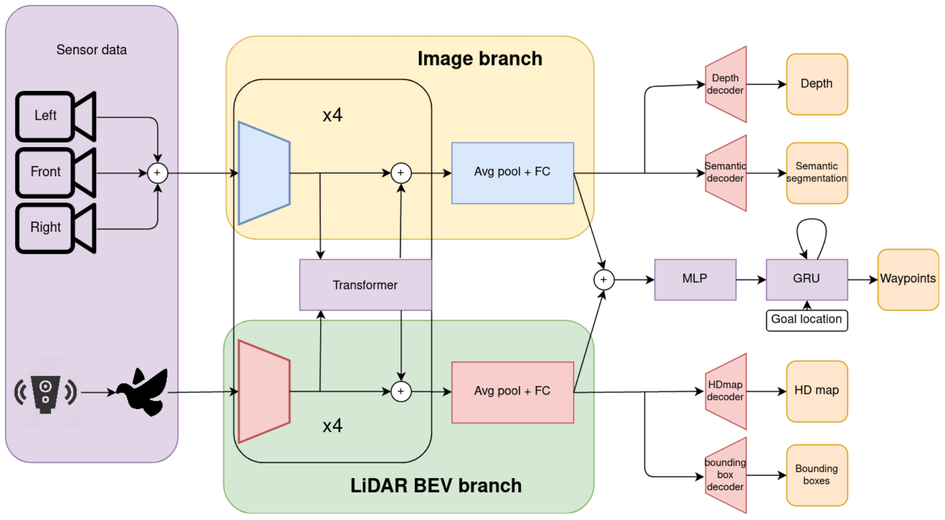


Figure 2.17: Simplified TransFuser architecture

Architecture

The TransFuser model takes three RGB images and a LiDAR bird's eye view as inputs. As depicted in Figure 2.17, RGB images from front-facing, left-facing, and right-facing cameras are concatenated and then input to the model. Raw LiDAR data is converted to a 2 bin histogram to construct a bird's eye view map before being passed to the model. The architecture is split into two branches, one for image data and another for LiDAR bird's eye view data. Both the image branch and Lidar branch have convolutional blocks, followed by both branches passing through a transformer block and element-wise summing the output before moving to the next block. This repeats 4 times at different resolutions. The output of the branches are two flattened vectors which are element-wise summed, then passed to an MLP. Finally, a GRU (Gated recurrent Unit) is used to predict the next n waypoints, while providing an approximate goal location to the GRU at each recurrent step.

Neural attention field decoders [21] are utilized for other prediction heads associated with auxiliary computer vision tasks.

Training

Since the learning task follows an imitation learning approach, the agent undergoes supervised training using a dataset collected by the CARLA autopilot. During training, the system tries to predict several auxiliary tasks; Depth, semantic segmentation, an HD map, bounding boxes of other vehicles in the scene and waypoints for the ego vehicle to follow in the next time-step. All the ground-truth information needed is collected using the autopilot expert demonstrator within the CARLA simulator.

2.4.4 InterFuser - 2022

InterFuser is a self-driving end-to-end imitation learning approach by Shao et al.[22]. This work's submission to the CARLA sensor track leaderboard is, as of writing placed 2nd, only behind "ReasonNet", which is a submission published by the same team. The work focuses on sensor fusion, leveraging both RGB and LiDAR data, while also having interpretable state representation and extra safety features employed. The work is similar to that described in section 2.4.3, but has additional focus on safety and interpretability and leverages the transformer architecture very differently from the TransFuser approach.

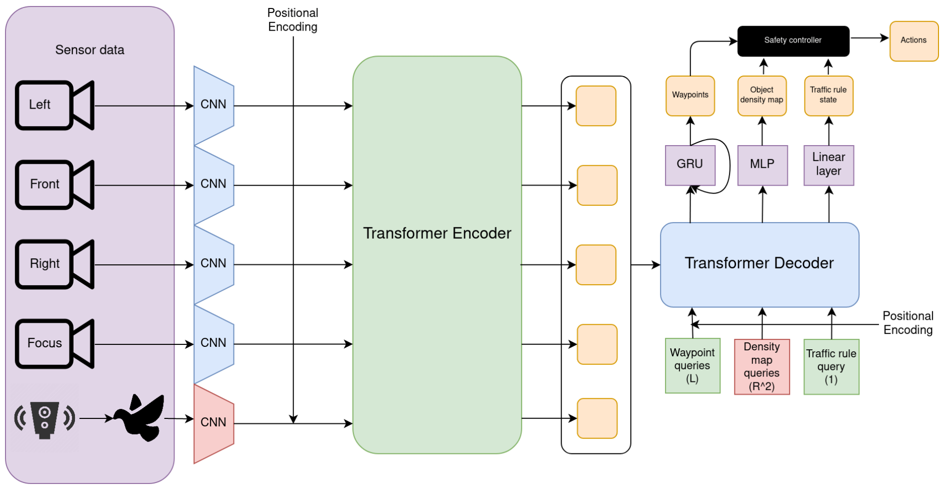


Figure 2.18: System architecture of InterFuser

The system uses five different input channels, which are all passed into the system in parallel. The LiDAR data, in the same way as in section 2.4.3 is converted to a bird's eye view representation. As well as using a front, right and left camera view, a focused view is also used, in an effort to effectively capture information about distant traffic lights. The focused view is a cropped part of the front view, resulting in higher resolution of an important part of the image, i.e. where distant traffic lights are caught.

Sensor Fusion

The sensor fusion differs from the Transfuser approach in that it treats each sensor input as a different modality to fuse, rather than having two explicit modalities for RGB and LiDAR. It also employs transformers similarly to how they are used in language task, rather than in the way vision transformers are used. While Transfuser uses several transformers to fuse information between the two branches at several steps in the pipeline at different resolutions. Interfuser initially converts all of the sensor inputs into one-dimensional vector representations, which is more similar to how transformers are used in classical machine translation tasks, instead of the vision transformer approach used in Transfuser.

As seen in figure 2.18, each sensor input is initially passed through a convolutional neural network. Each RGB camera input uses a pre-trained ResNet-50, while the LiDAR bird's eye view uses a ResNet-50 trained from scratch. The outputs of these are then converted to a one-dimensional vector representation, which then have a positional encoding added to them. Each token is then passed through a Transformer encoder and used in the same way as with standard transformer architectures, where the output of the encoder is used as input to the decoder.

Prediction types

The InterFuser architecture, in addition to predicting waypoints for the vehicle to follow the next time-step, it also predicts an object density map and traffic rule state. Each of these prediction types have a specific set of queries and learnable positional encodings that are passed to the transformer decoder. These queries are learnable parameters designed to prompt the decoder to output the desired information. For the object density map, R^2 queries are used to output an R^2xC traffic feature map, where R^2 is the resolution of a bird's eye view representation of the surrounding objects and C is a constant representing the amount of hidden features for each cell. Each query represents a cell in the predicted traffic features, which in turn is used as input to the prediction head for an object density map. The waypoints use L queries, one for each waypoint to predict. For prompting the decoder predictions for the traffic state, such as whether there is a red traffic light, a stop sign or whether the ego vehicle is in a junction, 1 query is used. Together with the output from the Encoder, all of queries these are passed through the decoder in parallel, which are finally used in three different prediction heads.

Waypoints The waypoint prediction head is a GRU, similar to section 2.4.3. To inject information about the goal location, the GRU's hidden state is initialized with a vector embedded by the GPS coordinates of the goal location.

The loss function used for the waypoint prediction is an L_1 norm loss:

$$\mathcal{L}_{waypoint} = \frac{1}{L}\sum_{i=1}^{L} \|\hat{w}_i - w_i\|_1 \tag{2.2}$$

where L is the number of waypoints to predict, $\hat{w}_i$ is the predicted waypoint and w_i is the ground truth waypoint.

Object density map The 7 channels of the object density map represent the probability of the existence of an object, an offset from the center of the cell, the size of the objects bounding box, the object heading, and the velocity of the object. It is predicted using MLPs to output an R^2x7 feature map, which is then reshaped into a map $M \in R^{RxRx7}$. The loss function used for the object density map is a custom L_1 loss where the object probabilities that are ≥ 0.01 and ≤ 0.01 is split and calculated separately, then averaged together:

$$\mathcal{L}_{object-probabilities} = 0.5 \cdot \mathcal{L}_{\text{prob\_0}} + 0.5 \cdot \mathcal{L}_{\text{prob\_1}}$$

The next channels are calculated for only the values where the object probabilities are ≥ 0.01. The combined loss for traffic is then:

$$\mathcal{L}_{traffic} = 0.5 \cdot \mathcal{L}_{object-probabilities} \cdot \lambda_{lambda} + 0.5 \cdot \mathcal{L}_{traffic}$$

The losses for the velocity $\mathcal{L}_{velocity}$ are returned separately, and weighted individually along with the rest of the losses from the waypoints and traffic rule states.

Traffic rule state The traffic rule state is predicted using a single linear layer for each traffic category. The traffic state predictions consist of three prediction heads for:

- **Stop sign**: Whether a stop sign is present.
- **Traffic light**: Whether a traffic light showing red is present.
- **At junction**: Whether the agent is at a junction.

The loss function used for all traffic predictions is cross-entropy loss with label smoothing:

$$\mathcal{L}_{traffic-state} = (1-\epsilon)ce(i) + \epsilon \sum \frac{ce(j)}{N} \tag{2.3}$$

In the loss function 2.3 $ce(x)$ is standard cross-entropy loss, like $-log(p(x))$, ϵ is the label smoothing parameter, i is the correct class, j is the predicted class and N is the number of classes.

The total loss adds up to:

$$\begin{aligned}\mathcal{L} = {} & \lambda_{traffic} \cdot \mathcal{L}_{traffic} \\ & + \lambda_{velocity} \cdot \mathcal{L}_{velocity} \\ & + \lambda_{junction} \cdot \mathcal{L}_{junction} \\ & + \lambda_{traffic-light} \cdot \mathcal{L}_{traffic-light} \\ & + \lambda_{stop-sign} \cdot \mathcal{L}_{stop-sign} \\ & + \lambda_{waypoints} \cdot \mathcal{L}_{waypoints}\end{aligned}$$

Safety controller

The enhanced safety features of the InterFuser system are driven by the safety controller, as illustrated from figure 2.18. This controller utilizes predictions from each head to constrain control actions extracted from the predicted waypoints. From the waypoints, a desired heading α_d is determined by averaging the heading of the first two waypoints. From the desired heading, a lateral steering action is found using a PID controller. The longitudinal acceleration action is also found from the waypoints and used to attain the desired speed v_d, but is also constrained by factoring in the surrounding objects.

This is achieved by using explicit predictions from the object density maps to recreate the presence of objects. In this context, 'objects' refer to vehicles, pedestrians, or any other actors excluding the ego vehicle. The object density map is a bird's eye view representation with 1x1 meter cells, and an object is recognized as present in a cell if the existence probability in the cell is higher than $threshold_1$, or if the existence probability in the cell is the local maximum in surrounding cells and greater than $threshold_2$, where $threshold_1 > threshold_2$.

Combined with predictions for the heading position and object velocity, the safety controller can monitor the state of surrounding objects. To predict the future trajectory of each object, a tracker records the historical dynamics of each object. The future trajectory is then determined by propagating the historical dynamics forward in time using the moving average. With the future trajectories of all objects present in the nearest environment, the safety controller calculates the maximum safe distance s_t. It does this by checking if any predicted waypoints for the ego vehicle collide with any of the predicted trajectories. The controller then records the shortest distance to any identified collision. From the maximum safe distance s_t, a desired speed v_d is found by solving a linear programming problem to maximize v_d with the following constraints:

$$\begin{aligned}
&(v_0 + v_d^1)T \leq s_1 \\
&(v_0 + v_d^1)T + (v_d^1 + v_d^2)T \leq s_2 \\
&|v_d^1 - v_0|T \leq a_{max} \\
&|v_d^2 - v_0|T \leq a_{max} \\
&0 \leq v_d^1 \leq v_{max} \\
&0 \leq v_d^2 \leq v_{max}
\end{aligned} \tag{2.4}$$

In this context, v_d^1 represents the desired speed that should be maximized, while v_d^2 is a variable representing the desired speed in the second time step. T is a constant for the time step duration, v_0 is the current speed of the ego vehicle, s_1 and s_2 are the maximum safe distances at the first and second future time steps. v_{max} and a_{max} denote the maximum allowed speed and acceleration for the vehicle. The found speed v_d^1 is then finally sent to a PID controller to decide the control actions for throttle and brake.

2.4.5 Imitation is not enough - 2022

This research from Waymo by Lu et al. [23] centers on the enhancement of imitation learning methodologies via reinforcement learning. The authors address a notable restriction of imitation learning - the inability of the trained model to recover from situations that diverge from those encountered during expert agent training. For example, if the expert agent consistently navigates the center of the road, no training instances exist where the agent must return to the lane after necessary deviation. While the expert agent faces no issue, the imitation learning model lacks any knowledge on how to realign itself within the lane. This unpredictability persists unless handled separately or with emergent behavior developed during training. This scenario illustrates the critical issue of deviation from the expert data distribution; the moment an agent must diverge from familiar examples, it loses its ability to 'course-correct'. It is not feasible to assume that expert data can cover all unforeseen circumstances that may arise during actual deployment. The authors' experiments present a promising example of a solution to this problem, "BC-SAC," integrating behavioral cloning with the Soft Actor Critic (SAC) reinforcement learning algorithm. This method exhibits substantial improvements over the standard imitation learning approach, especially in challenging scenarios. The addition of reinforcement learning significantly enhances performance on challenging datasets compared to the baseline imitation learning model.

The authors propose a method to solve this problem by using reinforcement learning in the cases where the agent deviates from the training distribution. The authors hypothesize that reward signals are more effective for scenarios of lower frequency, while demonstrations are more effective for scenarios of higher frequency.

Implementation

GRIAD, outlined in 2.4.2, blends reinforcement and imitation by incorporating demonstration and exploration agents into the RL loop. In contrast, this work formulates a policy using a weighted blend of action selection, based on maximum expected reward, and the log-probability of action selection by the expert demonstrator.

$$\underset{\pi}{max}\ \mathbb{E}\mathscr{T},\pi,\rho 0\left[\sum_{t=0}^{\infty}\gamma^{t}R(s_t,a_t)\right]+\lambda\mathbb{E}_{s,a\sim\mathcal{D}}\left[\log\pi(a|s)\right] \tag{2.5}$$

Equation 2.5 integrates the objective function that maximizes the expected discounted sum of rewards with the objective function that maximizes the log-likelihood of the expert demonstrations. The parameter λ is employed to balance these two objectives. The authors opted to use the Soft-actor-critic (SAC) as the RL algorithm, given its simplicity in adding the imitation learning objective to the expected value of the critic function (Q-function). The resulting comprehensive actor objective is:

$$\mathbb{E}s,a\sim\pi\left[Q(s,a)+\mathcal{H}(\pi(\cdot|s))\right]+\lambda\mathbb{E}s,a\sim\mathcal{D}\left[\log\pi(a|s)\right] \tag{2.6}$$

By selecting an appropriate value for λ, this enhanced SAC objective function promotes policy imitation of expert demonstrations whenever the current state is within the data distribution ($\mathcal{D}$). When regions of the state space are visited that lie outside the data distribution, the policy primarily relies on the reward signal to guide its actions.

Reward Function Designing a reward function that promotes "good" driving behavior remains an unresolved issue in the field of autonomous driving. However, the authors circumvent this problem by primarily deriving driving behavior from the expert agent's learning. The reward function is explicitly designed to incentivize safety. This function combines reward signals for collision avoidance and lane maintenance. The reward for collision avoidance is given by:

$$R_{collision}=min(d_{collision}-1.0,0), \tag{2.7}$$

where $d_{collision}$ is the Euclidean distance in meters to the closest point on a bounding box of surrounding vehicles. The reward for keeping within the lane is calculated as follows:

$$R_{off-road}=clip(-1.0-d_{to-edge},-2.0,0.0), \tag{2.8}$$

In this case, $d_{to-edge}$ represents the Euclidean distance in meters from the vehicle to the edge of the lane. A negative distance indicates the vehicle is on the road, while a positive distance suggests it is off the road. These rewards are added together to form the final reward function:

$$R = R_{collision} + R_{off-road} \tag{2.9}$$

Architecture The BC-SAC (Behavioral Cloning with Soft-Actor-Critic) approach utilizes dual networks for the actor, critic, and target networks. Each network employs a separate Transformer observation encoder, which encodes all features such as vehicle states, road-graph points, traffic light signals, and route goals. The experimental approach used is modular, wherein visual features are implicitly available for the planning module.

Training

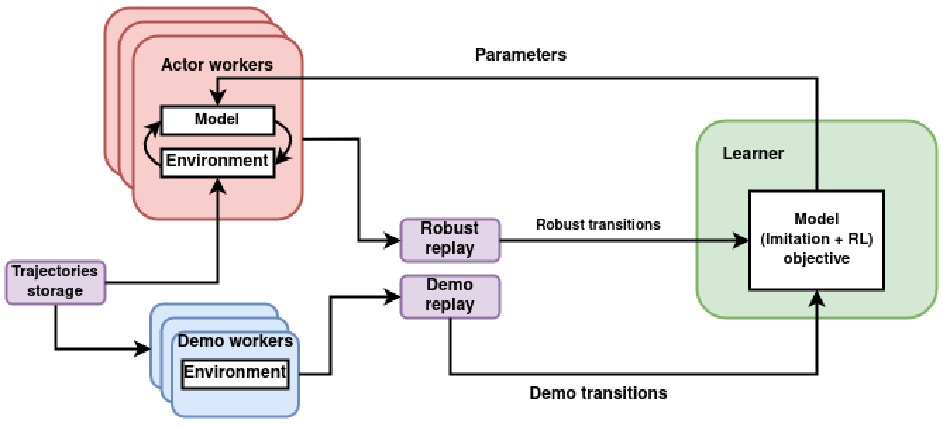


Figure 2.19: Imitation is not enough; architecture

The training process, as depicted in figure 2.19, involves two distinct classes of worker nodes: the demo workers and the actor workers. Replay transitions from each contribute to the optimization of the policy.

Training on Difficult Examples The authors also implement a method allowing the agent to concentrate on learning difficult driving scenarios. This method, as described by Bronstein et al. [24], provides significant advantage in tackling complex conditions.

2.4.6 UniAD - 2023

Many modern autonomous driving systems are built as modular systems, where the system is sequential in that the input data is first parsed through a vision system, then used in a planning module to generate trajectories and actions. This has the inherent problem of errors in the system propagating. That is if the vision system fails, the planning system will be ill-informed and fail consequently. The driving problem is naturally complex and can be categorized through several subtasks, which with sequential pipeline systems can be difficult to coordinate between. In the case of the system described in the previous chapter 2.4.5, a vision system is pre-trained and implied. This means that the vision system cannot directly cooperate with the planning module to optimize on the common goal of planning. It can only produce features that are found by directly optimizing for sets of explicit vision tasks. Planning optimization has not been part of the training loop of the vision module, resulting in errors propagating through the system as a whole. In this work by Hu et al. [25], propose a learning framework that aims to tackle this problem. The authors propose a method "UniAD" based on the idea to orient the entire system towards the ultimate goal of driving, namely planning. To achieve this, the framework focuses on prioritizing perception and prediction tasks such that all sub-optimizations contribute to the ultimate goal of planning.

Architecture

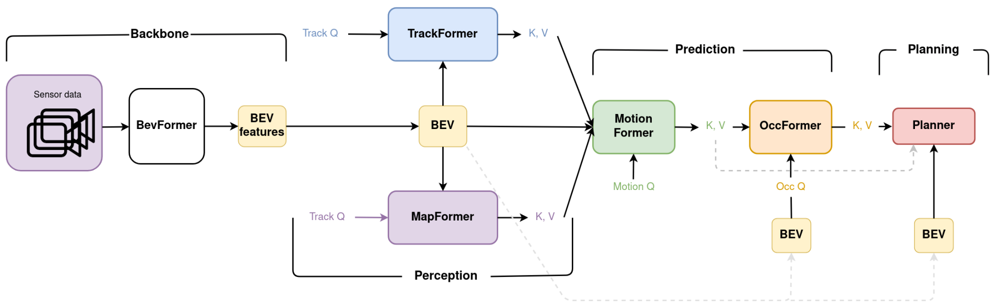


Figure 2.20: System architecture of UniAD

The model is made up of four transformer-decoder based perception and prediction modules, as well as a planner at the end of the pipeline. Different Queries Q, similar to those described in 2.4.4. The difference is that InterFuser uses one common module with different queries for the different prediction heads in the same stack of transformer decoder layers. UniAD also uses learned queries for the different predictions, but have several task-specific modules that have their own sub-optimization problems. The queries and results are also used across modules to connect the pipeline model to different interactions between the different units present in the driving environment. The five different components of the model are the backbone, perception, prediction and planning.

Backbone The backbone is the first step in the pipeline and contains a feature extractor that transforms a multi-view camera only input into a unified

Bird's Eye View (BEV) representation. This is done with an off-the-shelf BEV encoder as specified in BEVFormer [26]. The BEV encoder is built with its own ResNet101 backbone for feature extraction and a custom transformer decoder. This decoder has an initial step of Temporal Self-attention. This initial layer uses The previous BEV representation BEV B_{t-1} together with a set of learned BEV queries Q. The rest of the layers of the decoder are built as a normal transformer decoder, stacked 6 times, and gives the final output of a rich BEV feature representation.

Perception The perception component contains two modules, **TrackFormer** and **MapFormer**, that handles tracking objects and detecting semantic abstractions of road elements, like lanes and dividers. Both use the BEV representation as input and learns embeddings for querying tracking information for TrackFormer prediction heads, and MapFormer prediction heads.

Prediction The **MotionFormer** module is used for predicting the future motions of other agents in the environment. It uses the output from the Track and Map queries instead of the BEV representation and learns Motion prediction embeddings. The **OccFormer** uses the BEV representation as a query while using the agent-wise output of the **BEVFormer** to predict multi-step future occupancy with the identity of the agents preserved.

Planning At the end of the pipeline the Planning component includes only the **Planner** module that uses the ego-vehicle specific agent query from the **MotionFormer** as a query together with the BEV representation as different queries. The planner module takes into context the output from the **OccFormer** to keep away from colliding with obstacles in the scene.

2.4.7 ReasonNet - 2023

This work by Shao et al. [27] shares similarities with InterFuser, as described in section 2.4.4, and is published by the same team. It employs a more expansive architecture than InterFuser, complemented by an additional module for temporal reasoning.

Temporal reasoning The temporal reasoning module serves as a processing step designed to address the challenge of temporally occluded objects, thereby enhancing the agent's capability to predict an object density map. This processing step entails the use of an embedded query based on the current frame's LiDAR input to retrieve the most relevant historic frames from a memory bank. The features of current and historical frames are subsequently fused through an attention mechanism. A feature, which is predicted in parallel using all sensor data (including camera and LiDAR), is similar to the feature for predicting an object density map described in section 2.4.4. This feature, often referred to as the Bird's Eye View (BEV) feature, is combined with the fused historical and current features to predict a BEV map and an occupancy map. This additional processing step equips the agent with the ability to potentially manage situations where other actors are temporarily occluded, facilitating the adjustment needed to avoid collisions.

Chapter 3

Methodology

In this chapter, an approach for utilizing novel sensor fusion architectures to benefit reinforcement learning trained self-driving agents is described. The approach is based on the InterFuser architecture described in section 2.4.4. Along with the proposed method, the tools used and developed along with the processes for training and evaluating are described.

3.1 Reinforced Interfuser

This work proposes a hybrid approach, "Reinforced InterFuser," that utilizes novel sensor fusion approaches within a reinforcement learning agent. The approach is inspired by the work "InterFuser" from Shao et al. [22], as described in section 2.4.4. InterFuser, an end-to-end imitation learning approach, holds the second place on the Carla leaderboard's sensor track. It has only recently been superseded by a submission from the same team called "ReasonNet" [27], which was published on May 17th, 2023.

However, for the scope of this work, InterFuser is considered as the state-of-the-art in the field of sensor fusion for autonomous driving in simulated environments. This is because ReasonNet was published too recently to be included in the scope of this work. Consequently, the aim of this work is to utilize techniques employed in InterFuser as a visual encoder for a reinforcement learning planning agent.

3.1.1 Architecture

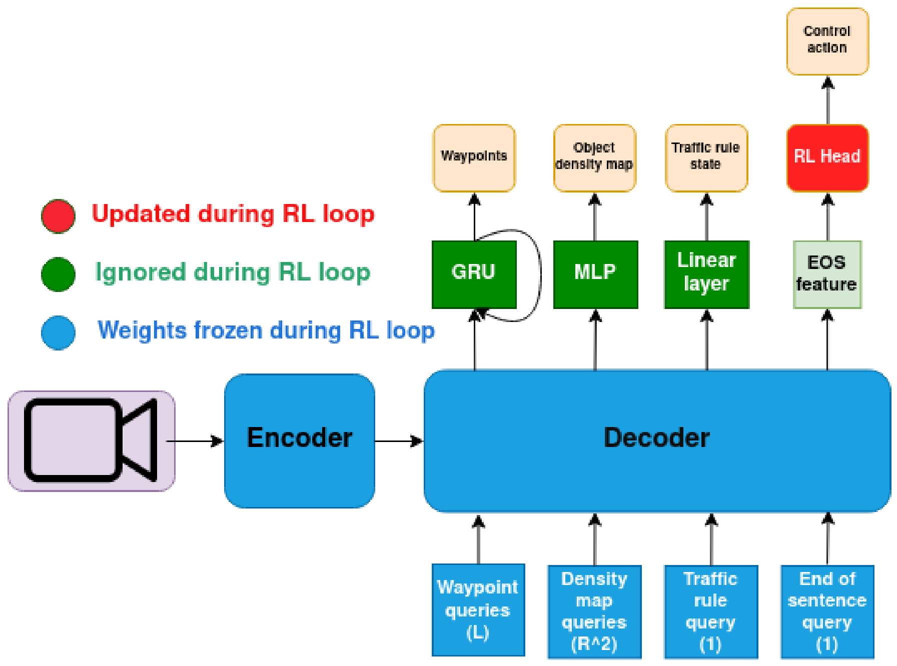


Figure 3.1: Reinforced InterFuser architecture

This work proposes a visual encoder that employs an architecture similar to InterFuser, but with a few modifications. The CNN backbones are downsized to ResNet18 instead of the ResNet50 architecture used in InterFuser. This reduces the number of parameters. Furthermore, the number of stacked transformer encoders and decoders are decreased from six layers to three.

To generate a compact vector for input to the RL agent, an additional query is added to the transformer decoder. This query functions as a special "end of sentence" (EOS) token, similar to those used in language tasks. The output of this query, referred to as the "EOS feature" (as shown in Figure 3.1), is not used during the encoder's training. Instead, it is employed to create a state representation that captures the context of the scene.

This approach draws from strategies used in language tasks, where the output of the EOS token is utilized during fine-tuning for tasks like text classification. In the context of this work, fine-tuning refers to training the RL agent to drive in a simulated environment.

prediction heads The prediction heads for the visual encoder encompass those delineated in 2.4.4, with the exception that the loss function for the waypoint prediction head is replaced with a Gaussian Negative log loss, as described in equation 3.1:

$$GNLL = \frac{1}{2}\left(log(max(\sigma,\epsilon)) + \frac{(x-y)^2}{max(\sigma,\epsilon)}\right) \tag{3.1}$$

In this equation, σ represents the predicted variance, x is the predicted mean, y is the ground truth value, and ϵ is a small constant (defaulting to $1e-6$) introduced to avoid division by zero. Unlike its original form, the output of the waypoint prediction head is modified to predict the mean x and variance σ from equation 3.1 for each coordinate in the waypoint trajectory, rather than predicting the coordinates directly. This alteration allows the output to represent a probability distribution for the predicted plan, instead of a single specific value for each waypoint. Consequently, it can serve as a measure of uncertainty during reinforcement learning and creates an opportunity to explore approaches like BC-SAC, as proposed by Lu et al. [23] (see 2.4.5 for description).

3.1.2 Training strategy

The proposed hybrid approach is split into several stages of training and follows the idea of "learning to see", then "learning to drive" as described in section 2.4.1. The first stage of training employs a training strategy similar to the one described in section 2.4.4. This means pre-training the InterFuser architecture on auxiliary vision tasks, as well as waypoint prediction. The trained InterFuser model is then used with frozen weights during reinforcement learning training to provide visual features in the form of a compact encoded vector to the agent.

Data collection

Data collection follows the same method employed by Shao et al. [22] and is executed for the purpose of training the InterFuser-inspired sensor fusion visual encoder. An expert agent, identical to the one used for InterFuser, is subjected to various driving scenarios delineated by the offline CARLA leaderboard routes. As these routes form part of the default offline leaderboard in the scenario runner, it ensures a consistent exposure of all trained agents to identical scenarios during their training. Moreover, these routes are used during the following reinforcement learning phase, thereby maintaining uniformity in the data distribution across both stages of training.

The agent captures RGB images from five different cameras and approximately 15,000 LIDAR detection points per frame. Additionally, it records the vehicle's state details for every frame, including speed, compass heading, and the vehicle's world coordinates.

Privileged information gathered includes data from Carla's privileged sensors such as depth cameras, segmentation cameras, and data about other environmental actors. This privileged information is utilized to generate the ground truth for the various prediction heads in the InterFuser architecture. These include the object density map, future waypoints, stop sign state, traffic light state, and an indicator determining whether the agent is at a junction.

Waypoint Disturbance A notable challenge when training data for a visual encoder arises from the fact that data collection is facilitated by an expert agent. This agent's driving behavior significantly differs from that of a reinforcement learning agent during training. Consequently, this results in a substantial distribution shift between the data used to train the encoder and the sensor data encountered during reinforcement learning.

To mitigate this issue, the target waypoints of the expert agent undergo significant disturbances during collection, leading to erratic driving behavior. This process infuses the data with additional noise, making it more closely resemble the data encountered during reinforcement learning.

Dataset The resulting dataset consists of 45k frames of information from all scenarios in the offline leaderboard routes. These routes are spread between four different towns in a single weather condition. The dataset used can be downloaded from the link in appendix 6.2. Only data from the training routes are used in the dataset, to ensure that the agent has only been exposed to data from the training routes. No frames from scenarios from the routes in the test set are included. For training of the InterFuser model, The dataset is split randomly (80-20) into a training dataset and an evaluation dataset. Because of this, the validation dataset consists only of samples from the training routes instead. This results in no distribution shift between the training and validation dataset, but this is a trade-off to ensure that the visual encoder has not been in any way biased by the evaluation routes during training.

Training the visual encoder

A custom InterFuser model is trained as a visual encoder for the RL head. This is in place of the stage "learning to see", as with implicit affordances from section 2.4.1. It is trained in a manner similar to the one described in Section 2.4.4. This means the model is trained to predict future waypoints, an object density map and traffic state. The available sensor data for these predictions is the same as the one described in section 2.4.4, which includes a front, left and right -facing camera, a cropped version of the front-facing camera, and LiDAR data in the form of a 2-bin histogram bird's eye view. The objective of the training of the visual encoder is to create a compact vector representing the visual information derived from input sensor data. To assess the viability of this architecture as a visual encoder for an RL agent, the output from the waypoint queries, traffic state query and the target query are made accessible for the RL experiments, as well as the resulting object density map.

Training the reinforcement learning head

The Reinforced InterFuser approach employs a second training step, in place of "learning to drive" after it has learned to "see" in the previous stage of training the visual encoder. Here, the weights of the InterFuser model are frozen, and an added RL head is optimized instead. A vision module that encapsulates the trained InterFuser model is injected to the environment used during RL training, such that the agent can use the output of it as part of the observation space. In the context of figure 3.1, this would mean the "EOS feature". Since other outputs from the InterFuser model are also available, these are also possible to include in the observation space.

3.2 Tools and technologies

To realize the proposed method, a number of tools and technologies are utilized. This section goes over the different tools and technologies used, their purpose, why and how they are used.

3.2.1 Simulator

Among the alternative driving simulators available for researching and developing autonomous driving systems, the CARLA simulator has been employed in this work. Its selection is justified by its open-source nature and a substantial community of dedicated researchers and developers. Furthermore, CARLA offers a highly realistic simulation experience, encompassing a wide variety of environments and weather conditions.

While TORCS [28] could be considered an alternative, it primarily focuses on racing scenarios, making it less relevant to this work. NVIDIA's DriveSim, although promising with its real image-based assets and NeRF techniques for reconstructing assets and environments, is not open-source and can only be accessed through an early access program.

Besides being the most accessible option, CARLA is also the simulator most frequently used in relevant publications for this work. Its use significantly simplifies the task of reproducing the approaches used in this study. Furthermore, CARLA offers robust documentation support and a Python API for managing an active CARLA instance.

3.2.2 Programming language

Python has been selected as the programming language for developing, running, and testing the environment and agents in this work. It is a common choice in autonomous driving research, so adhering to this standard is advantageous. Besides its widespread use and frequent appearance in relevant code examples, Python offers a wealth of readily available tools surpassing other languages.

Python has many available libraries for reinforcement learning, machine learning, deep learning, computer vision, image processing, and support for working with the CARLA simulator. While similar libraries exist for Java, JavaScript, and C++, Python combines the benefits of these libraries with simplicity and efficiency, enabling more output with less effort.

Python's main limitation is its inherent execution speed, but most of the computational heavy lifting is handled by libraries written in C and C++, making this less of an issue for this study.

3.2.3 Carla Episode Manager

In this work, the objective is to train and compare the learning capabilities of reinforcement learning agents with sensor fusion visual encoders. To ensure consistency with benchmark constraints akin to traditional evaluations of imitation learning agents, the scenario runner described in section 2.1.2 is utilized to facilitate scenarios and collect sensor data as part of the reinforcement learning loop.

However, the typical method of exposing agents to scenarios involves injecting an already trained agent, which is primarily used for evaluation or data collection. The scenario runner is not designed for the iterative reinforcement learning loops, as it assumes the injected agent to be fully trained. In reinforcement learning, an environment where the agent can take a step, provide an action, and receive the next state and a reward is needed.

To bridge this gap, a package was developed to control the simulator, simplify configuration, and represent the state of the simulator. The resultant package, named "Carla Episode Manager", requires a sensor configuration and a list of training and evaluation routes. This manager interacts with the existing Scenario Runner code to facilitate a scenario based on a random selection from the provided routes.

Furthermore, the manager oversees the collection of sensor data and the initiation and termination of a Carla simulator instance. Consequently, it provides an interface that controls the Scenario Runner and receives sensor data at each step of a route.

Episode Manager Interface

The manager provides three key methods — "start", "step", and "stop" — to control the episodes collectively. The objective is to efficiently manipulate the scenario runner tool so that it executes one episode at a time, each representing a specific route within the configured set of training or evaluation routes. Essentially, this minimizes the complexity associated with operating scenarios with the scenario runner, transforming it into a streamlined interface that oversees the initiation of a simulator, the facilitation of scenarios, the collection of sensor data, and the provision of statistics for each episode

Start The start method simply selects a random route from the list of routes and initiates the scenario runner with the identified route. If no server is running, a simulator is started and connected to. Once the scenario and all configured sensors are set up, the simulator world is advanced once ('ticked'), and an initial state of the world and sensors is generated, along with the high-level global world plan and a privileged world plan.

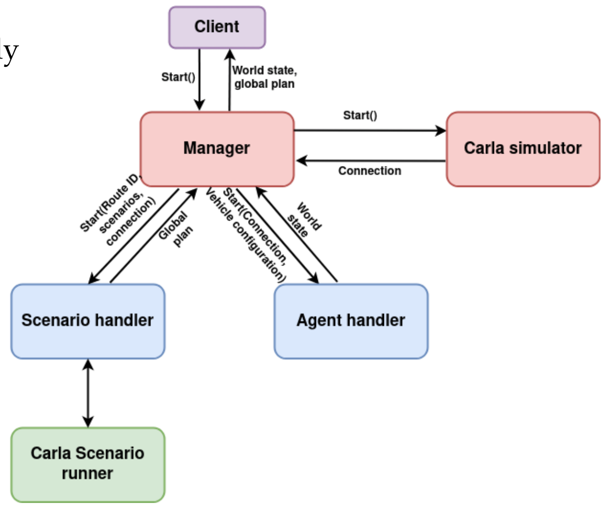


Figure 3.2: Carla episode manager "start" process

Step This method expects a control action, which is applied to the main ("hero") vehicle in the simulator. The simulator is then advanced once again, and the state of the sensors and the world and whether the episode should end after this step is collected and returned.

Stop The stop method halts the currently running scenario, destroys the "hero" vehicle, and returns statistics about the executed route, including route completion, time used, number of collisions, and red light violations.

Managing a simulator A consistent issue when using the Carla simulator are the occasional crashes, as well as the client timing out at random intervals. These crashes usually occur when many scenarios are run in succession. This proved to be a large problem, as it is critically important for the manager to handle running many episodes in sequence. The frequency of crashes would be often enough to make any training process very inefficient. The issue seemed to not happen if the simulator was restarted frequently enough. The episode manager was therefore designed to handle the starting and stopping of the simulator, as well as the restarting of the simulator at a given interval of episodes. This effectively negated the crashes, but did not completely remove the issue of the client timing out from time to time when setting up scenarios. This issue was however ignored, as it did not occur frequently enough to be able to reproduce easily or to argue spending a lot of time handling the issue.

Traffic types

For this research, there is a need to compare approaches in different types of situations to properly gauge the learning capabilities. To achieve this, three different types of difficulty is proposed:

- **Type 1:** No traffic, no pedestrians
- **Type 2:** Normal background traffic
- **Type 3:** Normal background traffic, with challenging scenarios throughout the route.

Realizing the possibility of simpler episodes across the same routes has some complications. This is not an out-of-the box option with the scenario runner, but more reserved to simpler interactions with the simulator. To keep the benefits of the features of the scenario runner while also still being able to simulate simple sequences, some modifications were made to the scenario runner code. One specific default scenario used in every route is the "Background traffic" scenario. To ensure that there is no traffic in the simulation, this default scenario was modified to be configurable, such that if "No traffic" is specified in the episode manager configuration, the scenario is not added to the route configuration. Running a route with no scenarios, and only the default traffic scenario, is already supported by the scenario runner, by simply providing a scenario-list file that is empty whenever specifying a route and coupled scenarios. With these modifications, the episode manager implements and supports these types of traffic, and can be configured when creating an instance of the manager.

The end-goal of the episode manager is intended to be used as a part of a reinforcement learning loop, and can therefore be used as part of a custom Gym environment, as described in 2.2.2. A link to the code for the episode manager can be found in appendix 6.2.

3.2.4 Gym environment

To easily integrate a reinforcement learning loop with the Carla simulator, a Gym environment has been utilized. This simplifies the process of experimenting with different Reinforcement learning solutions, as most off-the-shelf solutions for training are reliant on a Gym environment implementation. There are many pre-existing implementations to Gym environments for Carla, but these are not grounded in the scenario runner and the offline leaderboard. To effectively have an environment that can be used to benchmark the learning capabilities of different approaches, the training experiences should be strictly related to a set of training routes as specified by the offline leaderboard, as well as facilitating evaluation on a different set of held out routes. To achieve this, a custom Gym environment utilizing the Episode Manager described in 4.2.1 has been developed. The implementation simply handles the declaration and facilitation of anything related to "gymnasium", such that the observation space and action space is correctly defined based on the configurations provided.

Action space For the purpose of this work, the action space is confined to a discrete set of actions. Each action comprises a target speed (a_{speed}) and a steering angle (a_{steer}). Moreover, the environment can be configured to utilize continuous intervals for these two parameters, e.g., a_{speed} ranging from 0 to 6, and a_{steer} varying between -1 and 1.

Observation space The observation space is a dictionary of "state", "command" and "images" or "vision_encoding", depending on whether a vision module is used or not.

- **State:** The 'state' is a vector comprising the current speed of the vehicle and the relative position of the next goal waypoint in the global route plan.
- **command:** This part of the observation dictionary is a one-hot encoded vector representing the current high level command in the global route plan
- **images:** This section of the observation dictionary is utilized when a vision module is not configured. It consists of either one entry for each of the images provided by the episode manager at each step, or, if requested by the configuration, a single concatenated image of all the images provided by the episode manager.
- **vision_encoding:** This portion of the observation dictionary is used when a vision module is configured. It represents the encoded visual state of the world and is the output from the vision module.

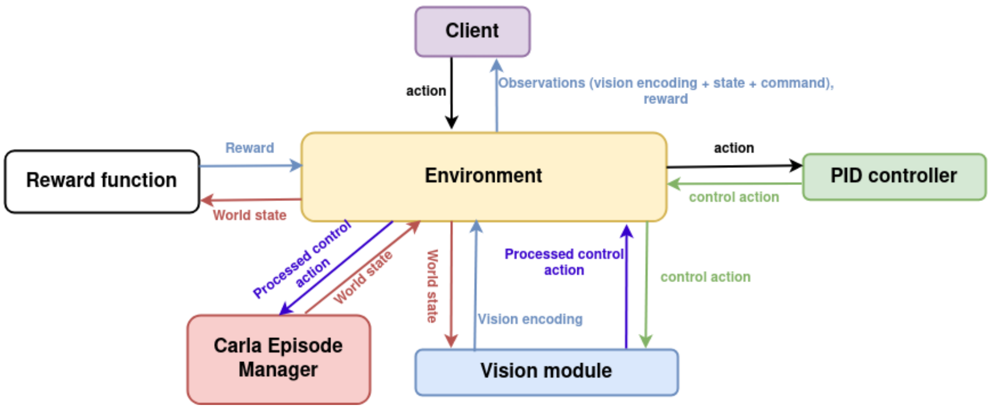


Figure 3.3: Gym environment architecture and communication each step

The environment is dependent on and makes use of several modules that can be provided to the environment:

- Episode Manager (see 4.2.1)
- Vision module
- Speed PID controller
- Reward function

All the components are used within the environment to facilitate the reinforcement learning loop, as illustrated in figure 3.3. This is to ensure that the environment behaves similarly across experiments both with a vision encoder, and to more easily experiment between different reward functions and episode manager configurations.

Vision Module

The vision module is an expected component passed to the environment and is an interface with a set of methods that needs implementation:

- *__call__*(s_t)->$s_{\text{encoded, t}}$: This method should implement the translation of world state s_t to an encoded vector small enough to be used as an observation in a reinforcement learning setting.
- *set_global_plan*$(plan)$: This initializes the global route plan and is called by the environment as soon as an episode is reset.
- $postprocess(a_t)$->$a_{processed}$: This method is called by the environment whenever an action has been selected by the agent and lets the module modify the Carla control action before it is applied to the ego vehicle based on any internal predictions made by the module.

InterFuser vision module implementation In this work, an implementation of the InterFuser vision module was developed to integrate the InterFuser encoder into the environment. This implementation loads a trained InterFuser model and parses the data from the world state into front, center, right, and left images during each call step. Additionally, it includes a Bird's Eye View (BEV) representation of the Lidar point-clouds. This data is then passed to the model. The 'end of sentence' query output from the model, as explained in 3.1.2, is used as the output from the call function of the module. Subsequently, within the environment, this output forms part of the observation for the Reinforcement Learning (RL) agent. An optional post-process step is also implemented. If utilized, it employs the safety controller actions predicted as described in 2.4.4, limiting the throttle and brakes for any emergency braking requirements.

Speed PID controller

To facilitate using target speeds as actions, a PID controller is used to convert these target speeds to throttle and brake actions applied the ego vehicle. The PID controller is used at every step() in the environment and is passed the target speed and the current speed of the ego vehicle, which is obtained from the episode manager.

Reward function

The reward function is passed the state s_t at every step() and is expected to return a reward and whether the episode should terminate.

3.2.5 Reinforcement learning library

To leverage the benefits of a standard interface for reinforcement learning environments, a library for distributed reinforcement learning was selected. After a careful analysis of available options, Ray RL library [6] was selected as the primary technology for this work due to its powerful distributed learning capabilities and a high degree of customizability. It ships with a large set of pre-made reinforcement learning algorithms, which can be used as a starting point for custom implementations. Ray RL library enables parallelization of the training process by utilizing multiple CPU cores and multiple GPUs to run many learning agents in parallel with the training process. This significantly accelerates the training process and scalability of the reinforcement learning process.

Parallelization

Initially, the choice of reinforcement learning library was set on using Stable-baselines3 [5], as it is a well-known and simple to use library for reinforcement learning. However, issues quickly arose when trying to scale the training process up across several parallel processes. The main issue being that the library does not support asynchronous training of agents. This was an issue because of the way the environment was implemented. Since the environment is dependent on the Carla Scenario runner, this meant that the environment had to wait for the scenario runner to finish the scenario, then use time to load the new map and route and facilitate the scenarios included for the selected town. For each reset() call to the environment, this time-consuming process occurs. The only Parallelization option for Stable-baselines3 is to run the environment in a "SubProcVecEnv", which runs N environments in separate processes synchronously, i.e each step() is called in parallel but awaited the result for. To handle whenever one of the environments needs a reset, the entire training process halts while the single environment is reset, before continuing the collection of transitions with step() calls on each environment. This was a major bottleneck in the training process, and effectively made training the agents not much faster even when running on hardware capable of running many environments in parallel.

Parallelization with the ray RL library is achieved by utilizing rollout workers. These workers are each loaded with a configuration for the environment, and initializes it in a separate process from the main training process. This architecture supports asynchronous collection of experiences, which means there is no need to wait for step() to finish on all environments before continuing. This has tradeoffs in that there is no guarantee that the worker is at any time up-to-date with the latest weights of the policy for on-policy algorithms. With frequent enough weight-syncing between the main learning thread and the rollout workers, this is however not an issue. The environment is very complex and requires a lot of time steps to be able to learn anything meaningful, so having out-of-date policies creating experiences is not a major issue. With an appropriate choice of parameters and algorithm, the policy is not likely to change drastically between each weight-syncing, meaning the out-of-date experiences collected with asynchronous rollouts are still useful for improving the policy.

Evaluation The framework supports several ways to include evaluation of the agent during training. Separate rollout-workers specifically for evaluation can be included, and can be used in parallel with the rollout workers collecting experiences, or at a set interval. Problems with this approach for this work in particular is that the environment is dependent on an external Carla simulator. Since the Episode manager from 4.2.1 is used to facilitate the scenarios, as well as starting, connecting and communicating with the simulator, this means that each environment at any time might have a simulator instance up and running, which takes up a lot of VRAM from the assigned GPU. Since evaluation cannot be accomplished with the same workers as the ones collecting experiences by default in the framework, this means that using additional workers for evaluation would require a lot more VRAM than is being used at any time, unless evaluation is run in parallel. For this work's case, constantly evaluating is not necessary, and only needed between a larger amount of training iterations. Parallelization of evaluation is therefore not really beneficial, but creates the issue of using unnecessary large amounts of extra VRAM while training. To solve this issue, the episode manager and environment have to expose a function for stopping the simulator, and dynamically start a simulator whenever the simulator is off when an episode is started. This way, the simulator can be stopped for all environments set up by the rollout workers whenever evaluation is being performed, and in turn the simulators for the evaluation workers can be stopped when the evaluation is finished.

3.2.6 Logging and Visualization

To log and visualize the training process, an ML-Ops platform has been utilized. For this work, the important metrics are the general results of the routes, and the gathered reward during training rollouts, as well as rewards and route statistics from evaluation rollouts. To visualize the driving progress, videos also have to be recorded while the agents are collecting experiences to be able to make some qualitative assumptions on the learned behavior. There are several alternatives to libraries that cover these requirements. The most popular ones are *TensorBoard* [29], *Weights & Biases* [30], and *Comet ML* [31]. Which are all free to use and provide a similar set of features. However, for this work, the most fitting solution was *Weights & Biases*, as uploading video files is well-supported and there are pre-implemented integrations for this in the chosen RL library. To support creating the needed data during training, several key metrics are logged during training with weights and biases, each which result in timeline graphs of the metrics over time in the training process. The logging strategy includes logging the minimum, mean and max of all collected statistics for each episode rollout, this includes:

- Accumulated reward
- Length of the episode
- Number of collisions
- Route completion percentage
- Number of red light infractions

On top of this, videos are recorded of entire episode rollouts at a set interval. e.g. every 20th episode rollout is recorded and uploaded to the platform. All of these metrics and videos are automatically kept separate between training and evaluation rollouts in the integration. Some modifications had to be made to include the statistics from the routes, including route completion and red light infractions. To include video upload in the integration, a custom callback for each rollout was implemented, which is used

Chapter 4

Experiments and Results

In this chapter the conducted experiments set up for answering the research questions for this thesis, described in section 1.2 and their results are presented and discussed.

4.1 Experiment 1: Reducing InterFuser

The experiment is conducted as part of the investigation of **RQ1**. As part of trying to answer the question of the benefits of sensor fusion in reinforcement learning, the InterFuser model is trained with a more compact architecture and a smaller dataset. Additionally, an "end of sentence" query is added as a learnable parameter to investigate if the token can be used to produce a more compact feature vector that encompasses the global context of the scene in the following experiment 2a in section 4.3

4.1.1 Setup

The number of stacked transformer encoders and decoders are reduced from 6 to 3, and the ResNet-50 backbones are replaced with a ResNet-18 backbone. The used dataset is the expert frames collected and described in 3.1.2, which consists of expert frames from all training routes from the offline Carla leaderboard for one weather configuration. The dataset is split randomly into a training set of 80% and a validation set of 20%. The architecture's prediction heads and loss functions are changed in the way described in the Reinforced InterFuser architecture in section 3.1.1, and the training of the model is done in the way described in section 3.1.2 To evaluate the performance of the model, the final loss values and metrics found on the evaluation split of the dataset are used.

4.1.2 Results

The results for training the reduced model are reported in parts of their different prediction heads.

Waypoints

The used loss function for the waypoints are different from the original InterFuser loss, which is the L_1 norm between the predicted waypoints and the target waypoints from the sequence generated by the global planner. For this work, a Gaussian Negative Log Loss (GNLL) is used, where the mean and variance of the predicted waypoints' distribution is compared to the target waypoints samples. The GNLL is described in section 3.1.1.

The loss is weighted for each of the 10 predicted waypoints $\{w_0 ... w_9\}$ as shown in figure 4.1.

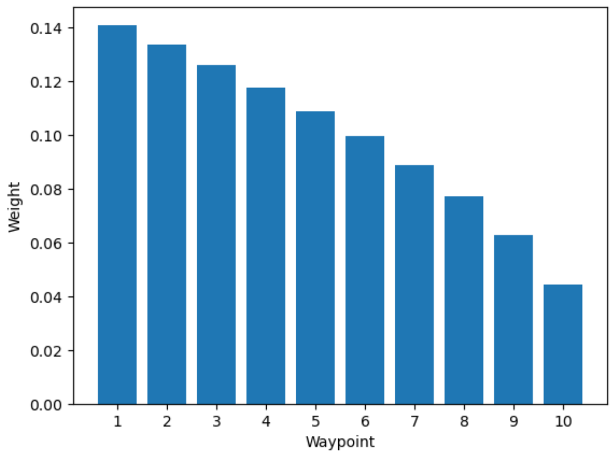


Figure 4.1: Waypoint loss weights

Figure 4.2 shows the loss rapidly decreases over the first few epochs and evens out after around 20 epochs. The final loss value after 35 epochs, or 1153 steps is 0.003651.

To sanity-check whether these predictions are reasonable, The mean for each of the predicted waypoint distributions are compared to the target global plan waypoints for each frame with random samples from each batch in the evaluation dataset after 35 epochs:

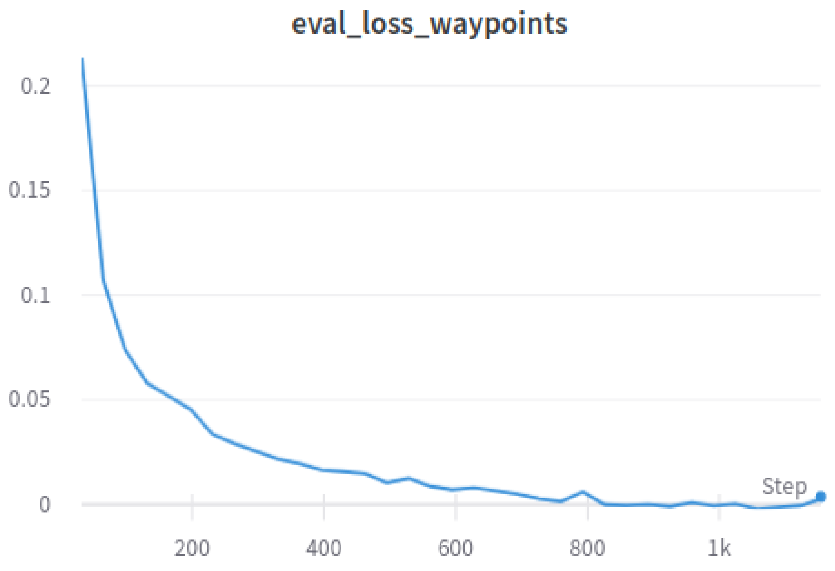


Figure 4.2: Waypoints loss, 1 epoch = 33 steps

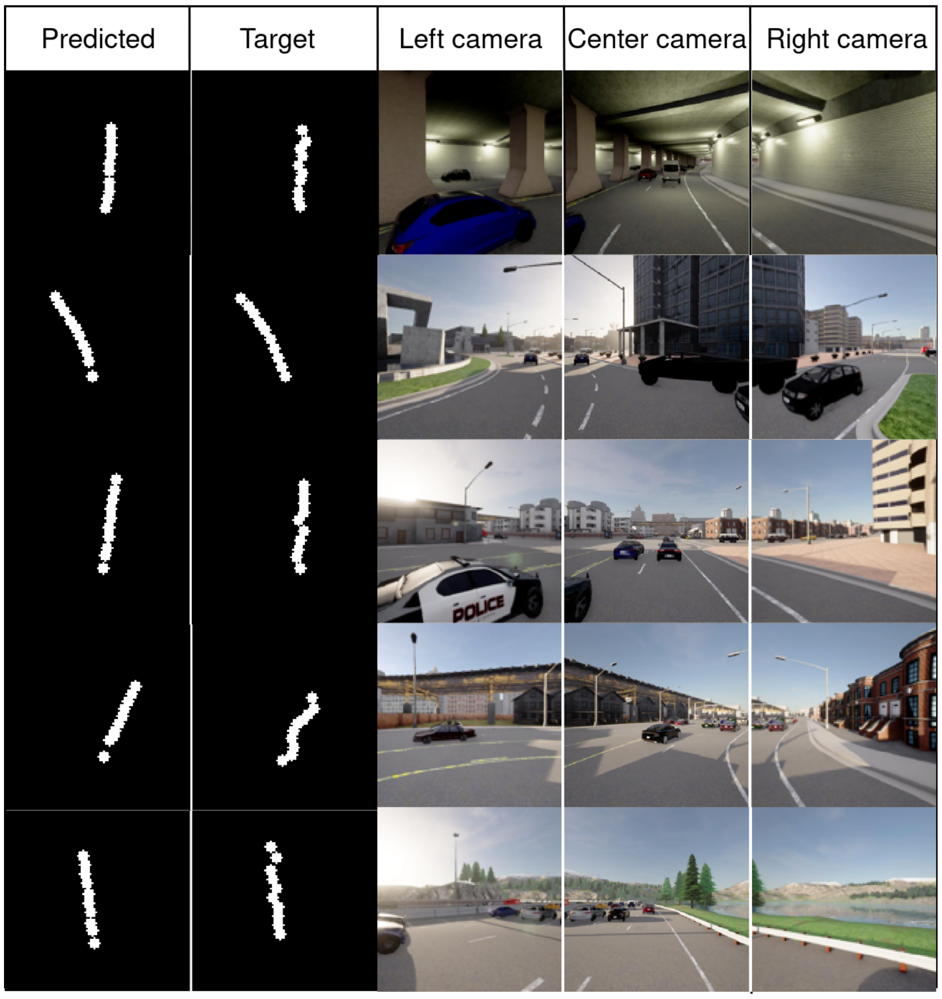


Figure 4.3: Waypoint prediction samples

Figure 4.3 shows the relative coordinates of each predicted mean for the waypoints, along with the camera image of the same frame. Each example show a reasonable trajectory compared to the situation of the vehicle. The target waypoints have some zigzagging, which is intended as the waypoints fed to the global plan in the dataset are disturbed during the collection process. The predicted waypoints however, still end up having smooth transitions between each waypoint. This is a result of the GRU architecture of the waypoint prediction head.

Traffic State

The traffic state predictions consist of 3 classes:

- **Stop sign**: Whether a stop sign is present.
- **Traffic light**: Whether a traffic light showing red is present.
- **At junction**: Whether the agent is at a junction.

The loss functions used during the experiment is equal to the one used in the original InterFuser model described in 2.4.4 To evaluate the performance of the model on the traffic state predictions, the accuracy of the predictions are used. These results are from the same evaluations as the waypoint predictions in the previous section.

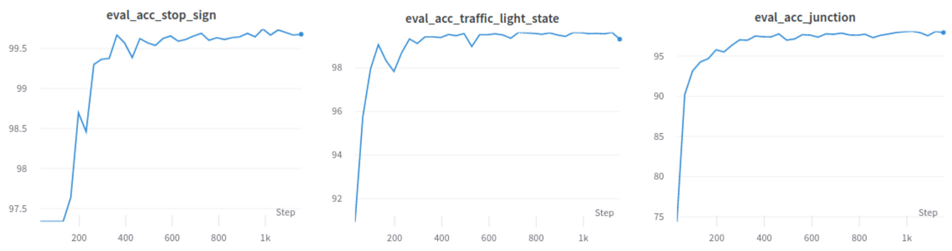


Figure 4.4: Traffic state accuracies, each epoch is 33 steps

The final accuracies for the predictions are:

- **Stop sign**: 99.678%
- **Traffic light**: 99.313%
- **At junction**: 97.939

The accuracies are all very high, likely because of the distribution shift not being very prominent in the evaluation dataset. The tasks are all very simple, and most frames in the dataset are negative examples for all the tasks. Meaning a model that predicts only negative would have an accuracy of around 90% for all the tasks. The worst performing task is the junction task. This might be because of the vague definition of what a junction might look like. Stop signs are simple, as they are always the same shape and color, same with traffic lights. Junctions are however more of a concept, than a physical object with limited variations. It requires a wider understanding of the environment than the other tasks. 97.939% is still a very high accuracy, but this means that even with this minimal distribution shift, there are still a handful of frames where the model predicts incorrectly.

Object density map

The loss for the object density map are the same as the original InterFuser and are described in section 2.4.4. This model output's main purpose is to predict explicit information that can be used to constrain the speed of the vehicle, whenever the planned waypoint trajectory might run into a moving object.

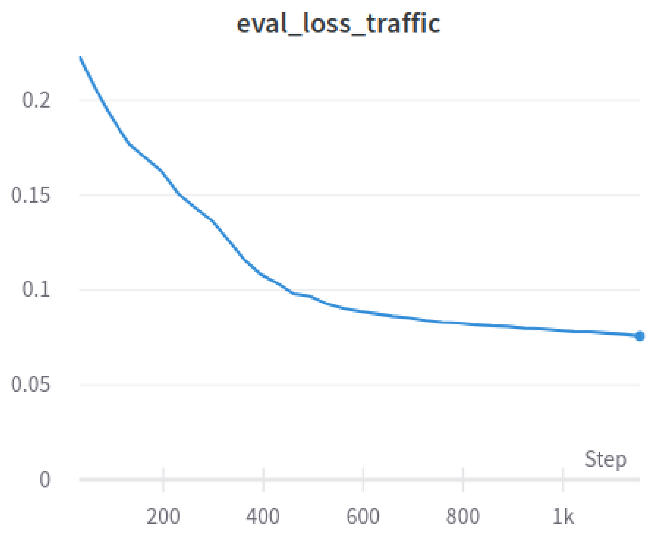


Figure 4.5: Object density prediction loss

Figure 4.5 shows the loss for the object density map prediction at each step in the training process. A lower loss reflects more accurate predictions of a combination of surrounding object existence, their heading, the offset in position from the center of the cell, and the size of the bounding box of the object.

During the training period, the loss keeps decreasing without bouncing back at any point, unlike the losses for traffic state and waypoint prediction. This part of the total loss is weighted more heavily than the output from the other prediction heads, and is the most important part of the model output. Since this loss is a combination of several L_1 losses across several explicit features describing the vehicles in the local environment, it is difficult to interpret whether the model is learning what it should. To empirically gauge the quality of the object density map predictions, the predictions from the frames used in the previous section for waypoints shown in figure 4.3 are compared to the ground truth.

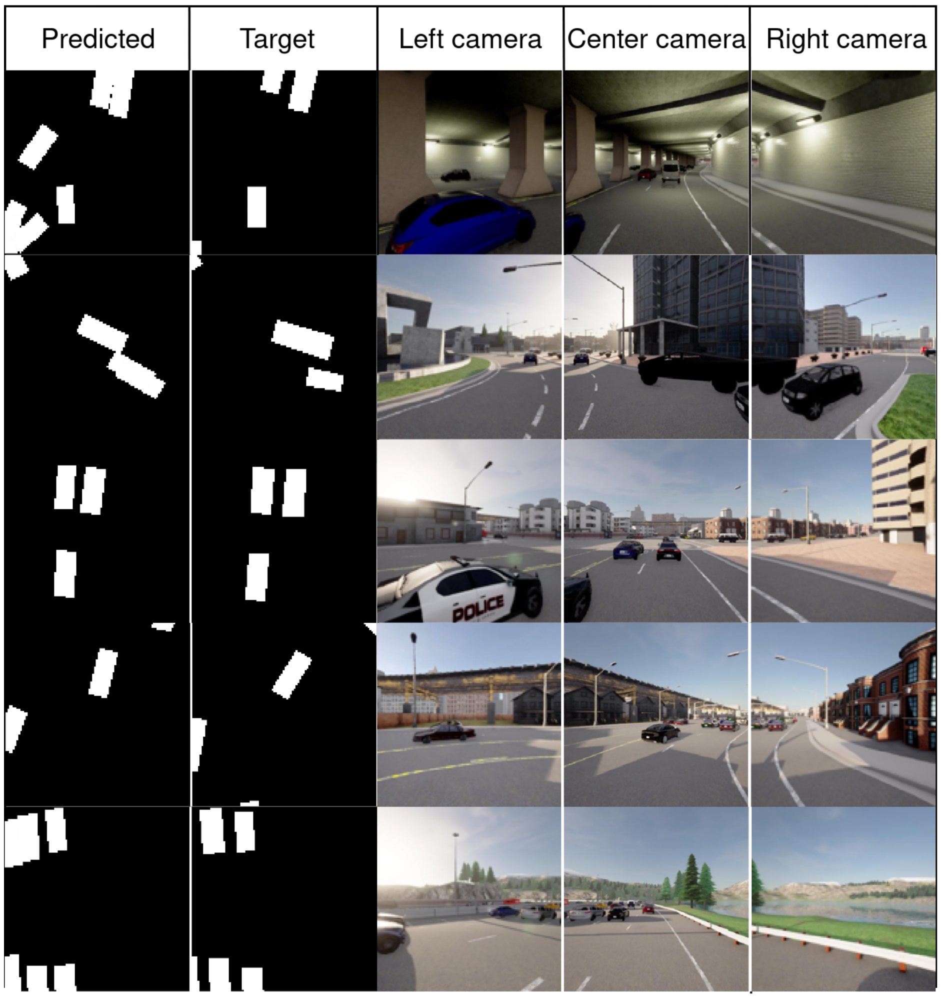


Figure 4.6: Traffic prediction examples

Figure 4.6 shows the predicted object density maps along with the ground truth and accompanying camera images. The object density map is a bird's eye view that represents a 20x20 grid of several predicted features for each cell. The cells represent a 1x1 meter area collectively covering 20 meters in front of the vehicle and 10 meters to each side of the vehicle, with the ego vehicle's position in the bottom center of the bird's eye view. This means the architecture can at most predict objects in 400 different square meter slots.

The examples demonstrate the predictions mostly show the vehicles in the scene, but often end up hallucinating vehicles that are not there. The first (top) example shows that the model might be estimating vehicles to be closer than they really are, as there is a vehicle present in the Left camera, but should be too far away to be visible in the object density map. It might also be the case that the poles are interpreted as moving objects, which is not really a problem. It is only when the model starts hallucinating objects present in the scene that stop it from driving when the lane is clear that this becomes an issue, as well as any phantom braking as a result from false positives.

4.2 RL agent training setup

The following experiments employ a set of different configurations for training RL agents, but all of them have some specific configurations in common to make the training process as similar and comparable as possible. These different agents are used to compare the performance of the Reinforced InterFuser method with baseline RL methods. The different types of RL agents include:

- **Blind:** A blind RL agent without visual sensor data available, operating only on common observations not found in visual sensors like RGB cameras or LiDAR.
- **Baseline:** A baseline RL agent trained directly on RGB camera sensor data from the environment.
- **RI:** The proposed Reinforced InterFuser method from section 3.1

This section goes over the common configurations for the different RL agents trained in this work, as well as the distinct differences in setup present for each category of RL agent. The configurations all revolve around the developed custom Carla environment described in 3.2.4 and its configurable modules.

4.2.1 Common configurations

Reward function

The reward function for each reinforcement learning agent is based on the reward function used in [18]. It is made up of three different components: a reward for the speed of the agent, a reward for the distance to the center of the lane, and a reward for difference in heading compared to the current closest waypoint.

The speed component is found by calculating a desired speed based on the distance to the closest hazard. The desired speed v_d is calculated as follows:

$$v_d = \begin{cases} speed\_limit, & \text{if } dist < 0 \\ \min\left(speed\_limit, speed\_limit \cdot \frac{\max(clipped\_dist - offset, 0)}{offset}\right), & \text{otherwise} \end{cases}$$

Where:

$$dist = \text{distance to closest hazard}$$
$$clipped\_dist = \min(dist, distance\_cutoff)$$

And the constants are defined as:

$$speed\_limit = 4.0, \quad distance\_cutoff = 20, \quad offset = 10$$

The desired speed aims to be at the speed limit when no hazards are present, and linearly go closer to 0 when closing in on a hazard. The reward for speed is then calculated as follows:

$$R_{speed} = \begin{cases} -1.0, & \text{if } speed \leq 0.5 \text{ and } desired\_speed > 2.0 \\ 0.0, & \text{if } speed\_diff > max\_speed\_diff \\ \frac{max\_speed\_diff - speed\_diff}{max\_speed\_diff}, & \text{otherwise} \end{cases}$$

Where:

$$speed\_diff = |speed - desired\_speed|,$$
$$max\_speed\_diff = \begin{cases} 0.5, & \text{if } speed > desired\_speed \\ 0.01, & \text{if } desired\_speed \leq 0.001 \\ 3.0, & \text{otherwise} \end{cases}$$

The reward is designed to punish the agent for standing still if desired speed is significantly higher than 0. Other than that, no reward is given if the agent has a speed outside the maximum allowed speed difference. This speed difference is reliant on whether the agent is going faster or slower than the desired speed.

The reward for distance to the center of the lane is based on the distance to the closest waypoint in the dense privileged global plan of the current route. It is calculated as follows:

$$R_{distance} = \begin{cases} -10, & \text{if } |distance| > max\_distance \\ 1 - \frac{distance}{max\_distance}, & \text{otherwise} \end{cases}$$

Where:

$$max\_distance = 2$$
$$distance = \text{Distance from vehicle to the closest waypoint}$$

The reward for the heading of the vehicle compared to the heading of the lane is calculated as follows:

$$R_{heading} = \begin{cases} -10, & \text{if } |diff| > max\_diff \\ \frac{max\_diff - diff}{max\_diff}, & \text{otherwise} \end{cases}$$

Where:

$$max\_diff = \frac{\pi}{2}$$
$$diff = \text{Difference in heading between agent and two closest waypoints}$$

The two closest waypoints are found from the privileged dense global plan of the route. Additionally, the agent is given a reward of -50 if it collides with any obstacle. The episode is stopped if any reward component is below -1. The final collective reward is:

$$R = \frac{R_{speed} + R_{distance} + R_{heading}}{3}$$

Action space

For the action space of each agent, the environment is configured to use 3x31 discrete actions, where there are three different goal speeds of $A_{speed} = [0,0,2.0,4.0]m/s$ and 31 different steering values of linearly spaced values between $[-1.0, 1.0]$, Where -1.0 is a full left turn, and 1.0 is a full right turn.

Observation space

The configured observation space for each agent includes the following, as described in 3.2.4:

- **Speed:** The current speed of the agent
- **Directional vector:** A directional vector pointing towards the next goal waypoint relative to the orientation of the ego vehicle.
- **Command:** The current high-level command for the agent, (e.g *Turn left*, *Turn right*, *Follow lane*, *Change lane left*, etc.)

PID controller

A PID controller with K_p(Proportional gain) $= 0.5$, K_i(Integral gain) $= 0.1$, and K_d(Derivative gain) $= 0.2$, is included in the environment used for each type of RL agent. These are employed to convert the current speed and selected a_{speed} into control actions applied to the ego vehicle.

Carla episode manager

The episode manager employed when training each type of agent is configured to use the training routes from the Carla offline leaderboard in appendix 6.2 during training, as well as the evaluation routes during evaluation.

Sensors

The ego vehicle is employed with an IMU sensor that provides orientation of the vehicle, a privileged collision sensor, a speedometer and a GNSS sensor that provides the current rough position of the vehicle.

Algorithm

For training the agents, the on-policy algorithm PPO by OpenAI [32] is used, The reason for this over other algorithms, is that it has a simple implementation as well as being stable without much hyperparameter tuning needed. Since PPO is an on-policy algorithm, the trained policy is the same as the one used for collecting the state transitions. The algorithm is not very sample efficient, as each sample is only used to train the model once, but this does give a more stable training process, allowing for a more fair comparison between the learning capabilities of different agents. Specifically the asynchronous version of PPO, APPO provided by Ray RLlib [6] is used, which is an asynchronous version of PPO.

The used hyperparameters for all agents during training are:

$$\alpha = 3 \cdot 10^{-5}$$
$$\gamma = 0.95$$

Where α is the learning rate and γ is the discount factor. Initial testing showed that having too high of a discount factor, like approaching 0.99, resulted in the agents not picking up any useful strategies that improved reward accumulation. Lower discount factors like 0.9 resulted in the agent learning to approach the correct speed quickly, but not learning much beyond quickly driving outside the lane or crashing. The rest of the hyperparameters are set to the default values provided by Ray RL lib. This includes an entropy coefficient of 0.01, which is used to encourage exploration and proved to be sufficient for balancing exploration and exploitation during training.

Policy and value networks

"All agents utilize a common conditional, fully connected network for the policy and value functions implemented in the chosen RL algorithm. The policy predicts the action probability distribution, while the value function estimates the value of the current state. The network architecture is an MLP with hidden layer sizes of 1024, 512, and 256 , followed by a separate final linear layer for value and policy predictions. The ReLU activation function is used for all layers. The network is made conditional based on the high-level command (e.g., turn left, turn right, follow lane, etc.) derived from the observation space. This results in six parallel fully connected networks, in which outputs from all networks unrelated to the given high-level command are masked to 0. This approach mirrors the method employed in 'implicit affordances', as described in section 2.4.1

4.2.2 Blind configuration

The "blind" agent uses all the same configurations as described in the previous section 4.2.1, and provides no additional observations. This means that the agent only relies on the current speed, directional vector and high-level command to navigate the environment.

4.2.3 Baseline configuration

The baseline agent adheres to all the configurations specified in section 4.2.1. However, it is equipped with additional visual sensors:

- **Front-facing camera:** A 600x300 (width x height) resolution RGB camera, angled 0 degrees relative to the ego vehicle's heading.
- **Left-facing camera:** A 600x300 (width x height) resolution RGB camera, angled -60 degrees relative to the ego vehicle's heading.
- **Right-facing camera:** A 600x300 (width x height) resolution RGB camera, angled 60 degrees relative to the ego vehicle's heading.

At each step, all images are resized to 200x100 pixels and normalized to match the mean and standard deviation of the ImageNet dataset [33]. The LiDAR sensor is not used, as it proved to be too computationally expensive to combine with the RGB cameras. This is because the resulting model and batched frame samples would become too large to fit in memory.

Regarding the architecture, each image from the RGB cameras at each frame is processed through its own 3-layer convolutional neural network.

	filters	**Kernel size**	**Stride**
$layer_0$	16	6x8	[3, 4]
$layer_1$	32	6x6	4
$layer_2$	256	9x9	1

Table 4.1: Baseline agent convolutional network configuration

The configuration for each layer is specified in table 4.3. Every layer employs a ReLU activation function. These are components of the RL policy and value networks and are optimized during RL training. No weights are frozen for this baseline agent.

Three of these convolutional feature extractors are utilized, one for each camera input. The resulting flattened vectors from each CNN are of size 256 and are concatenated along with other observations, such as speed, command,

and directional vector. This concatenated vector is then passed to the common conditional fully connected network described in section 4.2.1.

4.2.4 Reinforced InterFuser configuration

The Reinforced InterFuser agent adheres to all the configurations specified in section 4.2.1. However, it is equipped with three RGB cameras and a LiDAR sensor. The additional visual sensors are as follows:

- **Front-facing camera:** An 800x600 (width x height) resolution RGB camera, angled 0 degrees relative to the ego vehicle's heading.
- **Left-facing camera:** A 400x300 (width x height) resolution RGB camera, angled -60 degrees relative to the ego vehicle's heading.
- **Right-facing camera:** A 400x300 (width x height) resolution RGB camera, angled 60 degrees relative to the ego vehicle's heading.
- **LiDAR sensor:** A 64-channel LiDAR sensor with a range of 85 meters. The sensor is positioned at the center of the ego vehicle, angled 90 degrees relative to the vehicle's heading, and collects 300,000 points per second.

The agent's environment incorporates an InterFuser vision module, as described in 3.2.4. This module employs a pre-trained InterFuser model with frozen weights. It processes sensor data and outputs between 1 and n vision_encoding vectors. These vectors, together with default observations like speed, command, and the directional vector constitute the observation space.

The architecture of the RL policy is consistent with the common configurations, yet accommodates any additional vision_encoding vectors from the InterFuser vision module.

4.3 Experiment 2a: Reinforced InterFuser with "eos" feature

To address **RQ1**, the proposed Reinforced InterFuser architecture, outlined in 3.1 and depicted in figure 3.1, is deployed. The "end of sentence" feature vector, derived from the "eos" query token added during the training of the custom-trained InterFuser encoder, serves as a single vision encoding for the **RI** agent. This method's potential for effective policy learning is evaluated by comparison with the "blind" and "baseline" agents, with accumulated rewards monitored during training and evaluation. Additionally, metrics such as route completion and red light infractions are collected during evaluation phases.

4.3.1 Setup

The vision encoding used for the Reinforced InterFuser method in this experiment is a compact vector of size 256, aligning with figure 3.1. All other observations adhere to the default specifications of the environment described in 3.2.4. Given the minimal context provided by this approach from a lean custom-trained InterFuser model, the experiments are initially conducted in an environment devoid of traffic. Thus, the agents navigate all Carla offline leaderboard training routes without encountering traffic or other challenging scenarios.

Agents This experiment employ the following agent configurations:

- **Blind:** The blind agent devoid of visual sensors, relying solely on "speed", "command" and "directional vector" observations from the environment.
- **Baseline:** The baseline agent as described in 4.2.3, equipped with three cameras and three CNNs forming part of the policy and value networks.
- RI_{eos}**:** The Reinforced InterFuser agent incorporating the "eos" feature vector into the observation space for the RL head.

All agents utilized the reward function outlined in 4.2.1. The action space comprises the discrete 93 actions described in the common configuration section 4.2.1. The simulator operates at 10 fps, translating to each time step being equivalent to 0.1 seconds in the simulator. Each agent is trained for 1 million time steps, using batches of 2048 environment time steps for each training iteration. The agents are evaluated every 10th training iteration, or approximately every 20k time steps.

4.3.2 Results

For the 1 million steps all agents are trained, these are the resulting average reward gathered every 2048 steps in the training environment, as well as the mean reward gathered during each evaluation:

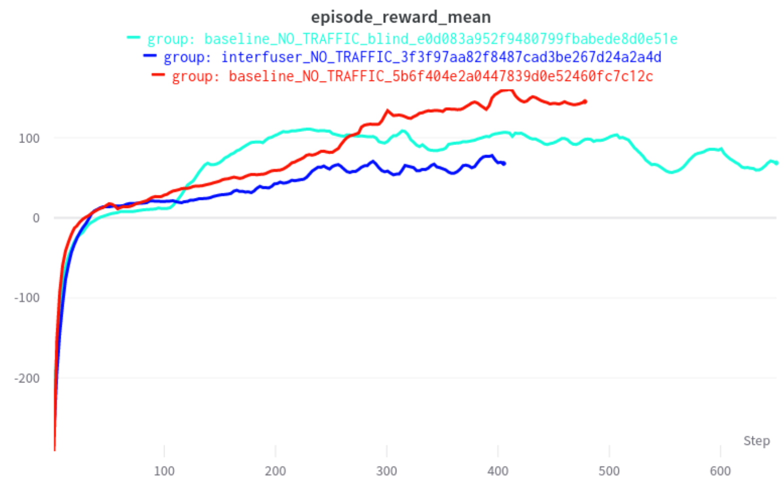


Figure 4.7: Average reward per 2048 steps on training rollouts. Reinforced InterFuser with "eos" as vision encoding (dark blue), Blind agent (cyan), Baseline agent (red)

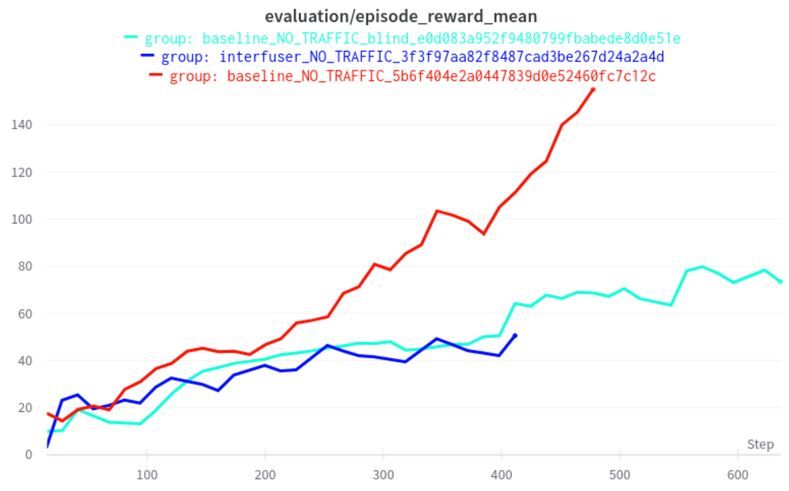


Figure 4.8: Average reward first 10 evaluation routes, every 10th training iteration. Reinforced InterFuser with "eos" as vision encoding (dark blue), Blind agent (cyan), Baseline agent (red)

Figure 4.7 shows that the reinforced InterFuser agent (dark blue) performs even worse than the agent deployed without visual sensors. It was stopped early due to consistently performing worse than the blind agent.

To properly gauge the resulting performance of the agents after finished training, the top metrics during evaluation for several categories are gathered:

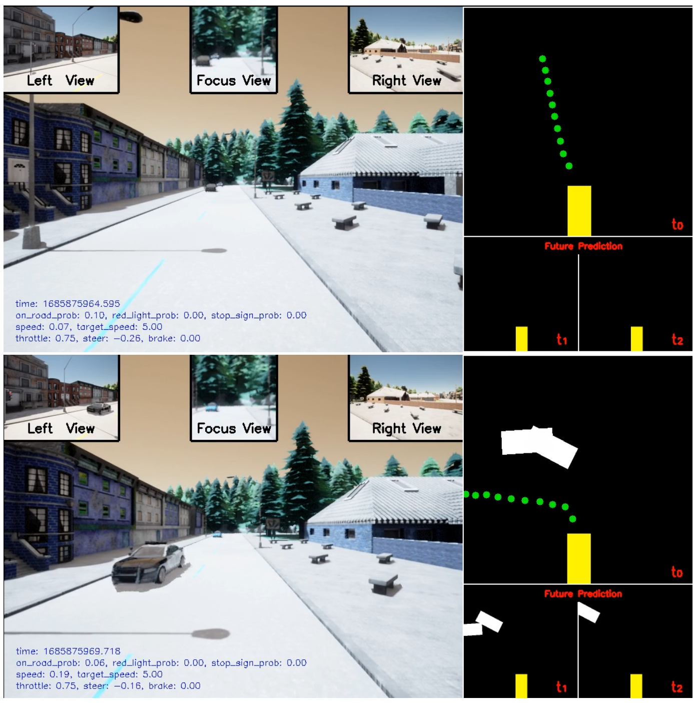
	RC_{mean} **(%)**	RC_{max} **(%)**	r_{mean}	r_{max}	**TLI**
Blind	4.69	26.08	222.89	1355.301	3
Baseline	**6.361**	**37.7**	**332.201**	**2058**	2
RI_{eos}	3.367	11.08	155.565	615.296	**1**

Table 4.2: Evaluation metrics for experiment 2a

4.3.3 Discussion

The baseline agent achieves rewards marginally higher than the blind agent during training, exhibiting strong performance during evaluation. Early in training rollouts, the blind agent surpasses the other agents, potentially attributable to the increased likelihood of adopting a strategy that merely involves slow forward motion. This increased probability might stem from the decreased noise or nuance in the agent's observations, enabling a more rapid alignment with a specific strategy.

Table 4.2 presents the maximum reward found during any episode of evaluation (r_{max}), as well as the mean reward r_{mean}, route completion percentage RC_{mean}, maximum route completion percentage RC_{max}, and the number of red light infractions **TLI**. All these metrics suggest the baseline agent outperforms both the blind and Reinforced InterFuser agents. Although the baseline agent incurs more red light infractions, this occurs because it encounters more red lights. However, none of the agents have achieved the proficiency needed to correctly respond to red lights during the evaluation routes.



```
time: 1685875964.595
on_road_prob: 0.10, red_light_prob: 0.00, stop_sign_prob: 0.00
speed: 0.07, target_speed: 5.00
throttle: 0.75, steer: −0.26, brake: 0.00
```

```
time: 1685875969.718
on_road_prob: 0.06, red_light_prob: 0.00, stop_sign_prob: 0.00
speed: 0.19, target_speed: 5.00
throttle: 0.75, steer: −0.16, brake: 0.00
```

Figure 4.9: Object density and traffic state predictions

Figure 4.9 presents explicit prediction examples from the custom-trained InterFuser model, set in an evaluation environment with reserved routes and towns. These highlight the model's limitations, particularly when vehicles are present in the scene. Predictions often indicate multiple vehicles where there are either only one or none. Furthermore, the predicted vehicle directions frequently misalign with actual headings.

This discrepancy may be attributable to the distribution shift, as the reinforcement learning (RL) evaluation environment utilizes different weather conditions and routes than those present in the dataset used for training the encoder. Despite these prediction inaccuracies, they should offer the agent some environmental information.

Empirical observations suggest that object density maps adequately predict the presence and location of vehicles, and the traffic state predictions appear consistently correct. These observations further support the result that the compact "eos" vector in this experiment fails to provide any meaningful context.

4.4 Experiment 2b: Reinforced InterFuser with waypoint feature

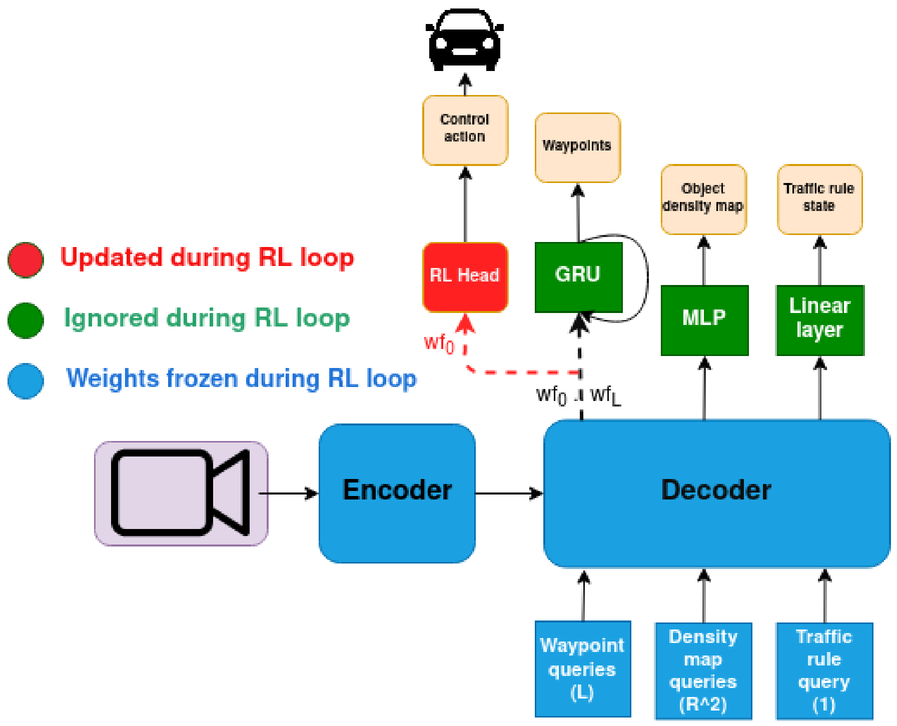


Figure 4.10: Reinforced InterFuser with waypoint feature

To further investigate **RQ1**, we also explore using the waypoint feature intended for predicting the next waypoint w_0 during training of the encoder. On top of this, the InterFuser model is replaced with the pretrained model from Shao et al. [22].

4.4.1 Setup

The environments for this experiment includes normal background traffic as described in 3.2.4, instead of no traffic as in experiment 2a. This is to investigate whether the compact waypoint feature is sufficient for the agent to brake to avoid collisions, or if there is a need for additional context from the other features extracted from InterFuser. This way, we can gauge if the waypoint feature helps the agent learn following the lane and route, and if it is sufficient for the agent to learn to stop at the correct time. All other aspects of the experiment are equal to experiment 2a. The agents trained and compared for this experiment are:

- **Blind:** The blind agent devoid of visual sensors, relying solely on "speed", "command" and "directional vector" observations from the environment.

- **Baseline:** The baseline agent as described in 4.2.3, equipped with three cameras and three CNNs forming part of the policy and value networks.
- RI_{wp}**:** The Reinforced InterFuser agent incorporating the first "waypoint" feature vector into the observation space for the RL head, as illustrated in figure 4.10

All RL agent configurations follow the specifications given in section 4.2. Since evaluation performance compared to training rollout performance showed a correlating consistent improvement, the evaluation performance is not included during training for this experiment. Instead, the agents are evaluated on each evaluation route once after training the full 1M steps while collecting statistics for the accumulated rewards, route completion and traffic infractions. The simulator is set to 10 fps, meaning each time step is 0.1 seconds in the simulator.

4.4.2 Results

The agents are instead evaluated after training the full 1M steps. The resulting average reward gathered every 2048 steps in the training environment for the baseline, blind and Reinforced InterFuser agent:

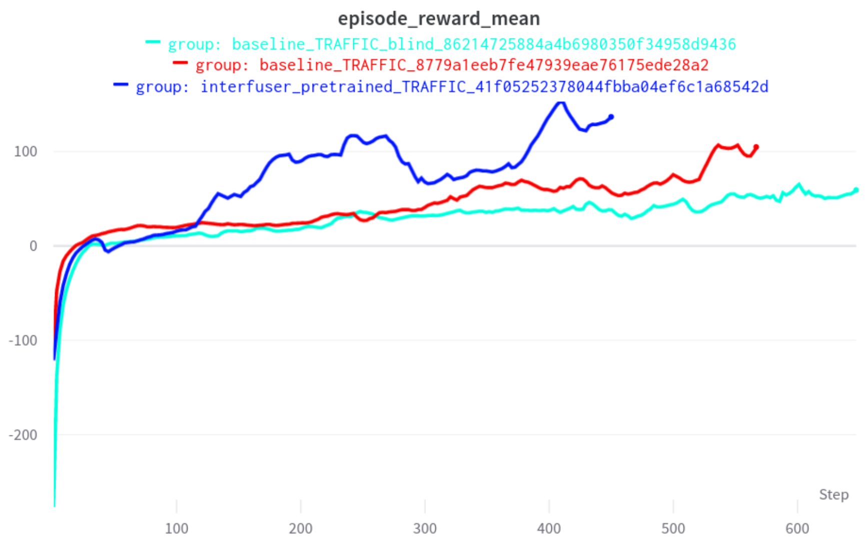


Figure 4.11: Average reward per 2048 steps, blind(cyan), baseline(red), and RI_{wp} (dark blue)

Some of the reinforced InterFuser's rewards are missing due to technical issues during rollouts, and thus is somewhat offset from the other agents. However, the reinforced InterFuser agent shows a clear improvement over the baseline and blind agents. With the maximum average reward at **180**, and the baseline agent at **144**. Empirical observations of the driving behavior show that the baseline and RI agent learns to follow the lanes and route, but fail to cross any intersections. Both seem to incorporate stopping to avoid colliding into the preceding vehicle in its lane, while the blind agent simply learns to slowly drive forward. Sample videos of the driving behavior can be found in appendix 6.2.

	RC_{mean} **(%)**	RC_{max} **(%)**	r_{mean}	r_{max}	**light infr.**
Blind	3.856	21.8	60.091	508.27	1
Baseline	4.187	19.15	124.79	562.01	1
RI_{wp}	**6.03**	**37.70**	**289.23**	**2595.53**	**1**

Table 4.3: Evaluation metrics for experiment 2a

Table 4.3 shows the metrics for each agent running on the 25 evaluation routes after training for 1M steps.

4.4.3 Discussion

While using the waypoint feature seems to be a vast improvement from using the "end of sentence" feature, beating out the baseline agent in all metrics, The agent still does not learn to stop consistently and struggles to cross intersections.

From observations of the driving behavior, there are some cases where the RI_{wp} agent ends up driving outside the lane to avoid colliding, which indicates that the compact vector to some extent encodes information about the surrounding vehicles, but does not seem to be sufficient for the agent to learn to consistently stop to avoid colliding into vehicles in its lane. This might be because the output from the feature during pre-training is only exposed to a loss function that tries to predict the following waypoints in the privileged dense global plan, disregarding any traffic. This trajectory still includes having to change lanes and avoid other vehicles this way, but is never about stopping. For InterFuser, stopping is mostly handled by the explicit rules employed in the safety controller described in 2.4.4. Which uses the explicit predictions from the traffic state and object density map to determine a desired speed.

4.5 Experiment 3: Reinforced InterFuser with all features as input

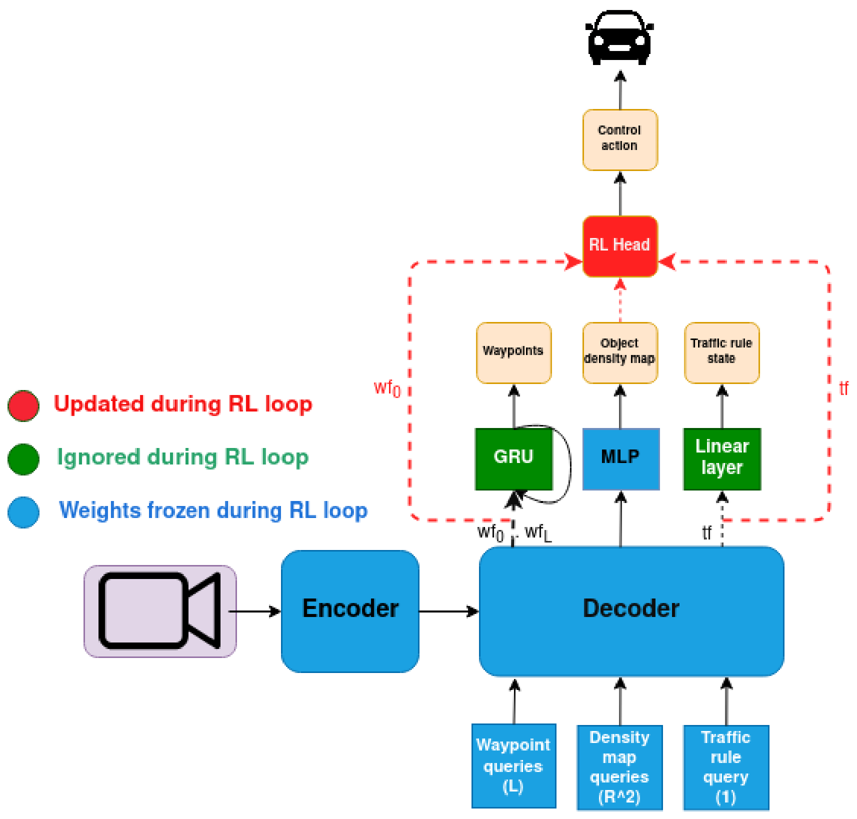


Figure 4.12: Reinforced InterFuser with all features as input

This approach uses the waypoint feature, as well as the feature for traffic state and the object density map. This differs from experiment 2, as the object density map is a 20x20x7 array with explicitly predicted information about the surrounding environment, and therefore is not a single feature vector. Having this multidimensional array of information requires a convolutional feature-extraction layer for the RL policy network to be used alongside the rest of the compact features.

4.5.1 Setup

The architecture for extracting features from the object density map is set up equally to the one in the baseline agent, as explained in section 4.2.3. This means a 3-layer CNN that outputs a flattened vector of size 256, which is concatenated with the rest of the features before being passed to the conditional fully connected network. The convolution filters are however configured differently, as the object density map is a 20x20x7 array, compared to the 200x100 pixel RGB images used in the baseline agent.

	filters	**Kernel size**	**Stride**
$layer_0$	16	5x5	2
$layer_1$	32	5x5	2
$layer_2$	256	5x5	2

Table 4.4: Convolutional network configuration

Agents For this experiment, the blind agent has not been included, and the reinforced InterFuser approach is only compared to the baseline agent. The compared agents in this experiment are therefore:

- **Baseline:** Baseline agent with the configuration described in section 4.2.3
- **RI(all features):** The proposed method as illustrated in figure 4.14 with a feature extractor trained during RL for the object density map as outlined in table 4.4.

The baseline with no frozen weights and the proposed method of using All features from the reinforced interfuser output as a *vision_encoding* to the RL policy head are trained for 1 million steps each on the 50 offline Carla leaderboards training routes with challenging scenarios while collecting average reward accumulations during rollouts, then evaluated on 26 offline leaderboard evaluation routes.

4.5.2 Results

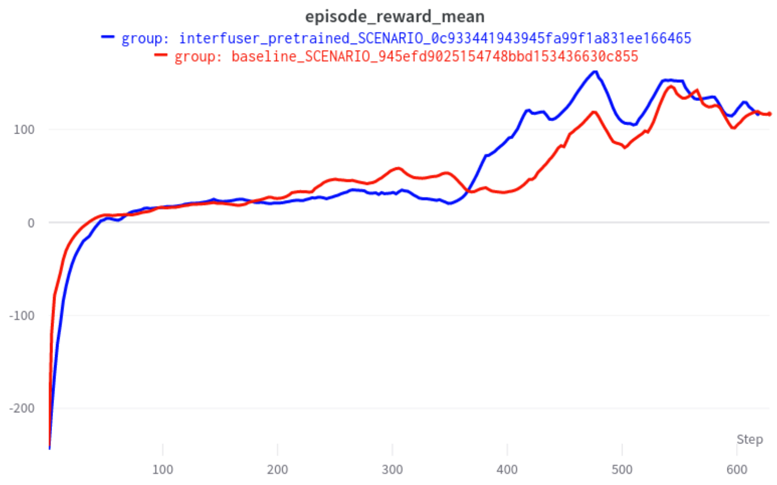


Figure 4.13: Average reward per 2048 steps

	RC_{mean} (%)	RC_{max} (%)	r_{mean}	r_{max}	light infr.
Baseline	**5.05**	**42.05**	**268.75**	**1592.52**	0
RI(all features)	2.24	6.65	89.37	233.95	0

Table 4.5: Evaluation metrics for experiment 3

4.5.3 Discussion

From figure 4.13 it seems that the reinforced InterFuser agent with all features as input is able to learn a policy that performs very similarly to the baseline agent. However, when inspecting the driving behavior, the agent seems to not be able to learn to follow the lane. It does well on the routes where it is able to drive straight, but fails to follow the lane when it has to turn. The agent perfectly learns to stop to avoid colliding into the car in front of it, and therefore ends up with a decent average reward overall. Both agents fail to cross any intersections, or handle any of the challenging scenarios set up in the routes. Table 4.5 shows that the difference in performance between the two agents is vastly different on the evaluation set compared to the training rollouts. The baseline agent even manages to complete 42 % of a route, with the reinforced InterFuser agent only completing maximum 6.65 % of a route in the 25 evaluation routes. There could be numerous reasons as to why the agent behaves seemingly worse when using all the features as input, but one reason could be that the combination of the features is too noisy for the agent to learn a good policy, potentially drowning the features from the waypoint vector. Another reason could be that the checkpoint model used when evaluating is in a state where it has recently learned some strategies that perform worse on the evaluation set than on the training routes. Since the average route completion is heavily lifted by one route in the evaluation set, the difference indicated by the evaluation metrics might not be as significant as it seems. It could be that the baseline agent happens to perform well on that specific route, and that the reinforced InterFuser agent has a policy that happens to perform poorly on that route.

Given the observations from the driving behavior of the agent it is however clear that the policy for the reinforced InterFuser agent is better at avoiding collisions, but almost completely inept at following the lane.

4.6 Experiment 4: Reinforced InterFuser with safety controller

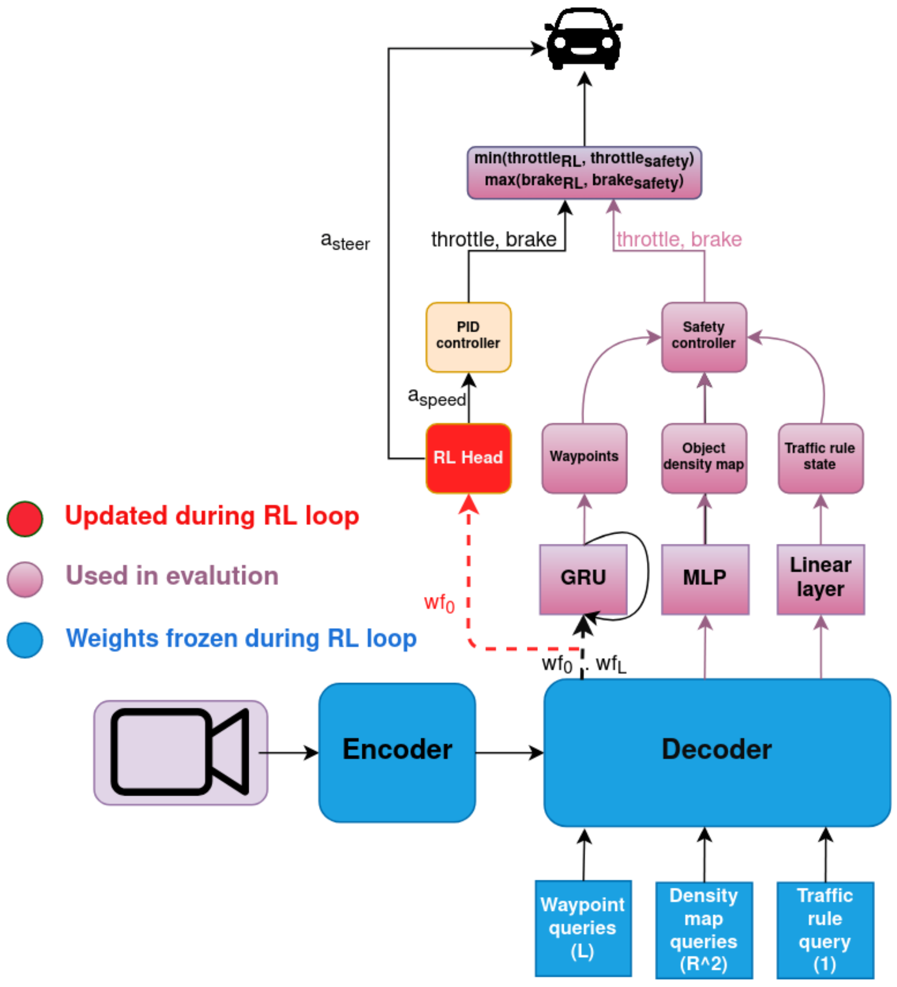


Figure 4.14: Reinforced InterFuser with all features as input

To address **RQ2**, the proposed Reinforced InterFuser architecture is utilized, consistent with the approaches used in Experiments 2b and 3, with the InterFuser safety controller integrated into the action selection process.

4.6.1 Setup

The safety controller, as described in Section 2.4.4, calculates a constrained "safe" action and overrides the selected throttle if

$$throttle_{safety} < throttle_{RL}$$

and the selected brake if

$$brake_{safety} > brake_{RL}$$

Given that this safety controller's potential braking predictions are not aligned with the reward function utilized in the RL policy network, rewards are not used as performance indicators. Instead, metrics such as the number of collisions, route completion rate, and red light violations are employed to assess the agent's performance, which are then compared with those of the baseline agent.

The baseline agent used for comparison is the best performing agent from Experiment 3 in section 4.5, trained on the most challenging traffic environment with scenarios. The Reinforced InterFuser agent is the best performing agent from Experiment 2b, trained on a normal traffic environment without scenarios. In this experiment, all agents are evaluated in both normal traffic and difficult traffic environments with scenarios.

Due to certain scenarios causing the simulator to crash, 8 of the evaluation routes are disabled when scenarios are involved in the environment. The objective of this experiment is to investigate whether the safety controller contributes to improved avoidance of collisions and other traffic violations.

Agents The agents used in this experiment are:

- **Blind:** This is the blind agent without visual sensors, trained on training routes with traffic and without challenging scenarios.
- **Baseline:** This is the best performing agent from experiment 3, trained on training routes with traffic and with challenging scenarios.
- RI_{wp}**:** The Reinforced InterFuser agent with waypoint feature, trained on training routes with traffic and without challenging scenarios.
- RI_{wp} **+ safety:** The same RL agent as RI_{wp}, but with the safety controller from InterFuser included in the action selection process.
- **InterFuser**: The InterFuser agent, not trained in an RL process, but uses the actual control output from the pretrained model from Shao et al. [22].

Evaluation metrics

In this experiment we use similar metrics to those used in experiment 2b and 3, and additionally include the number of collisions. Reward metrics are excluded as the safety controller has a different configuration to when the agent should be stopping, and thus conflicting with the reward function employed during training. Since there is a conflict, the controller is also not included during training, but used only during this evaluation experiment.

To gauge how well a "good" agent should be doing with the current evaluation setup, InterFuser's imitation learning agent is used for comparison. The metrics used are:

- RC_{mean}: The average percentage route completion across all 25 routes in the evaluation set.
- RC_{max}: The highest percentage route completion across all 25 routes in the evaluation set.
- **C**: Number of collisions during the evaluation routes. Any evaluation route ends whenever an agent collides with something, meaning that the amount of collision reflects how many of the routes ended in a collision.
- **TLI**: Traffic light infractions. The number of times the agent rand a red light during the evaluation routes. The route does not end whenever the agent runs a red light, so multiple infractions in one route is possible.

4.6.2 Results

The resulting metrics after running each agent on the evaluation routes once are shown in Table 4.6.

Traffic evaluation routes

	RC_{mean} **(%)**	RC_{max} **(%)**	**C**	**TLI**
Blind	3.856	21.8	7	1
Baseline	5.46	40.89	3	3
RI_{wp}	6.03	37.70	8	1
RI_{wp} **+ safety**	6.5	37.58	**2**	**0**
$InterFuser$	**18.40**	**43.91**	5	2

Table 4.6: Evaluation metrics for environment with normal traffic and no scenarios

Challenging scenario evaluation routes

	RC_{mean} **(%)**	RC_{max} **(%)**	**C**	**TLI**
Blind	2.56	8.97	8	0
Baseline	5.05	42.05	3	1
RI_{wp}	3.22	10.78	10	0
RI_{wp} **+ safety**	5.73	37.5	2	0
$InterFuser$	**23.25**	**43.89**	3	0

Table 4.7: Evaluation metrics for environment with challenging scenarios

Recorded driving behavior for the RI_{wp} **+ safety** agent and baseline agent can be found in appendix 6.2.

4.6.3 Discussion

The results presented in Table 4.6 demonstrate that the RI_{wp} **+ safety** agent delivers superior performance in terms of average route completion and collision metrics when compared to all other agents, excluding the **InterFuser** agent. In normal traffic conditions, the performance of the RI_{wp} agent does not yet match that of the InterFuser agent, particularly in the context of route completion. Furthermore, the performance improvement relative to the baseline agent is minimal. This becomes particularly evident when comparing the performance of the baseline agent after training on routes with challenging scenarios (as shown in Experiment 2b, Section 4.4, Table 4.3) to its performance when trained in an environment with normal traffic conditions.

The inclusion of the safety controller notably impacts the performance during normal traffic evaluation, substantially reducing the number of collisions from 8 to 2. In fact, the RI_{wp} agent shows an improved average route completion, increasing the mean from **6.03%** to **6.50%**.

Despite these improvements, a close observation of the RI_{wp} + safety agent's behavior reveals certain inefficiencies. The agent frequently halts in the middle of the road, a phenomenon not seen when the safety controller is not employed. This typically occurs when the ego vehicle's heading points towards the crossing lane while the front of the vehicle encroaches on the lane.

Furthermore, the autonomous actors in the simulation environment stop by default when they encounter any obstruction, including the ego vehicle. This can result in a deadlock, with the ego vehicle and other actors waiting for each other to move until the scenario times out or, for an undetermined reason, the ego vehicle begins to move.

All agents have some form of infraction with the traffic light, except for the $RI_{waypoints}$ + safety agent, where there are no cases of running a red light during evaluation.

Considering that the pre-trained InterFuser agent still manages to collide and run red lights, as well as not being able to fully complete the routes might indicate that some of the evaluation routes or exit conditions might be misconfigured or have some issue that might prematurely end the scenario evaluation. The InterFuser agent is employed in an environment where each time step is 0.1 seconds, while the intended time step for InterFuser is 0.5 seconds. This might cause some problems with the predicted control actions in the InterFuser agent in some cases. However, the huge route completion difference between the RL agents and the InterFuser agent indicate that the agent is still mostly correctly configured, as it on average completes 18.4% of the evaluation routes, crossing many intersections, following the lane and avoiding collision with other actors. The difference in performance also indicate that the RL agents are not performing particularly well in general, but does show that the evaluation routes and exit configurations are reasonably indicative of the driving performance, as the InterFuser agent is expected to perform a lot better than the RL agents given the limited training time.

For the evaluation of the same agents on the challenging scenarios as seen in table 4.7, the Reinforced InterFuser without the safety controller RI_{wp} performs much worse than on the normal traffic evaluation. It performs even worse than the baseline agent. This might be due to the Reinforced InterFuser agent not being trained on challenging scenarios, while the baseline agent is. The routes used in the evaluation from table 4.7 are however equal to the previous evaluation from table 4.6, meaning that the agent should be able to follow the lanes just as well as on the previous evaluation. It does seem that the addition of the scenarios along the route is enough to completely ruin the route completion of the RI_{wp} agent. However, with the safety controller employed, the same agent performs much better. The collisions are reduced from 10 to 2, the mean route completion is almost doubled, and the RC_{max} is quadrupled. This huge leap in performance shows that the challenges RI_{wp} has on the novel scenarios not seen during training, are mostly handled by the safety controller, resulting in a model not even trained on routes with scenarios employed having a massive increase in collision avoidance, traffic light handling, and route completion. Although the route completion of all RL agents is not yet very good, the limited training time is still able to indicate that the safety controller can handle most unexpected scenarios that the agent may encounter.

Chapter 5

Discussion

This chapter discusses the results obtained from the experiments carried out with variations of the proposed reinforced InterFuser approach. The discussion is organized to broadly address the results of the experiments in light of the proposed research questions.

5.1 Research question 1

In pursuit of an answer to the first research question mentioned in section 1.2:

Can Reinforcement learning approaches in autonomous driving benefit from sensor fusion neural network outputs as state representations?

a series of experiments were conducted. These experiments incorporated various methods of utilizing intermediate output from the InterFuser architecture. The results point towards potential advantages of integrating the InterFuser architecture within a reinforcement learning setting, relative to a naive RL approach. Experiment 2b, in particular, demonstrates that using the compact feature from just a single waypoint query is adequate to outperform a traditional RL approach that is given direct access to all camera images during training.

One notable challenge lies in the fact that the experiments, despite covering 1 million transitions, account for only approximately 28 hours of driving experience. This duration seems sufficient for agents to learn fundamental behaviors such as lane following and avoidance of forward collisions to a certain extent. However, it falls short in providing the agents with ample opportunity to learn more complex behaviors like managing intersections or adhering to traffic signals. Due to this limited training time, the results of the experiments can only provide a partial assessment of the learning potential of the proposed Reinforced InterFuser method compared to a naive RL approach. A peculiar result is that the agent trained with additional context of the environment from experiment 3 in section 4.5 performs worse than the baseline agent with no pre-trained visual encoder on the evaluation routes. Even though during training rollouts, the agent performs better than the baseline agent. The driving behavior does show that the $RI(allfeatures)$ agent with access to all output types from the InterFuser architecture is effectively blind to the trajectory it should be following, but very aware of the vehicles in front of it. While it has access to the same waypoint feature as the agent outperforming it, it does not seem to utilize it to the same extent. Potentially the full-feature agent is overfitting using the explicit predictions of the object density map and consequently neglecting the information encoded in the waypoint feature. Additional training time might be required to allow the agent to learn to utilize the waypoint feature to a greater extent. Different configurations of learning rate and architecture for the CNN receiving the object density map features might also have been useful to further analyze the performance of the full-feature agent.

Missing from these findings is an indication of whether the architecture delivers a direct improvement in comparison to a similar encoder pre-training strategy with a standard CNN architecture. An example of this would be a strategy similar to the one employed by Toromanoff et al. [18]. This gap in understanding is a limitation of the conducted experiments, a result of both the limited open-source code availability and the time constraints inherent to the thesis work. Nonetheless, the results hint at the potential utility of different features extracted from the InterFuser architecture in a reinforcement learning setting.

5.2 Research question 2

In pursuit of an answer to the second research question mentioned in section 1.2:

Can Reinforcement learning approaches in autonomous driving benefit from safety mechanisms deduced from explicit predictions of the environment?

For this research question, experiment 4 from section 4.6 is used to investigate whether additional safety mechanisms from the InterFuser model can be used to improve the performance and safety of the agent. In this experiment, the safety controller of the pre-trained InterFuser model was used as a post-processing step for the reinforced InterFuser agent from experiment 2b in section 4.4. This agent was only trained on normal traffic with no challenging scenarios. Results indicate that the agent, even with the safety controller employed, still is far away from the performance of the InterFuser agent itself.

However, using the safety controller in addition to the reinforced InterFuser improved the agents average route completion and reduced the number of infractions. When evaluating the agent on challenging scenarios, in which it was not exposed to during training, the agent performed worse than the baseline agent (which has no pre-trained visual encoder available relying on using raw image data as observations). However, with the safety controller attached, the agent performed better than the baseline, even managing to complete up to 37% of a single route, while without the controller, it could only complete maximum 10% of a route. These results strongly indicate that InterFuser's safety controller improves the safety of pure RL agents when it comes to unseen and unexpected scenarios.

Training an agent that can handle crossing intersections is still a challenge for the results from the experiments in this thesis. Employing the safety controller, as well as comparing with other strategies for safety, on an RL agent that more robustly handles intersections and lane following could lead to a stronger indication on the usefulness of safety mechanisms. Experiment 4 does not indicate whether this type of safety mechanism is useful for state-of-the-art performing agents, which is a shortcoming of the experiment. Additional training time, hyperparameter tuning and experimentation with other visual encoders could also be useful to set a stronger indication of which types of safety mechanisms are useful. The experiment does however show strong indication specifically for the InterFuser safety controller on lower performing agents, giving huge performance boosts when deployed in environments with unseen and unexpected scenarios.

5.3 Shortcomings

The conducted experiments show the potential of the various approaches proposed for leveraging a state-of-the-art sensor fusion model. Nonetheless, the execution of the thesis is marked by several limitations.

Reward Function The reward function in use is intended to model optimal driving behavior, a task fraught with inherent complexities. As explained by Lu et al. [23], modelling driving behavior with a reward function is a challenging task, and is evident from the behaviors of the agents across the experiments. The employed reward function does not effectively incentivize maintaining a practical driving speed. Rather, agents frequently resort to significantly reducing their speed to avoid collisions. Despite the existence of penalties for non-alignment with the lane, the agents—most notably the baseline—tend to oscillate within the lane. The root cause of such behavior could be attributed to the reduced likelihood of collisions or encountering an unmanageable scenario when the travelled distance is artificially shortened.

Consequently, oscillation while slowly progressing along the route emerges as a seemingly viable strategy, as it allows the agent to accumulate higher rewards for the same total route completion in a safer and more achievable manner.

Baseline agent Worth noting is that the evaluation results of the baseline agent (with no frozen weights and standard raw camera data as observations for the RL head) are worse when training on the environment with background traffic, than when training on the environment on the same routes with challenging scenarios. This might be an indication that the chosen training structure might not be fully suited for comparing performance. The inherent randomness for this type of training might not be sufficient to provide a fair comparison between the set of agents, especially given the low number of training steps. This is however only when it comes to the evaluation routes and the baseline agent specifically. Performance of reward gathering during training rollouts are still consistent for the baseline agent across the different types of training environments.

Training time The number of training steps used in this thesis to train each agent within the experiments is significantly lower than those suggested in related works. For instance, the experiments in [18] and [9] train for 60 million steps. Given the available hardware and training configuration, training for 1 million steps takes approximately 28 hours, despite the parallelization strategy described in section 3.2.5. This represents a constraint of this study, as the possibility of extending the training time was not explored. Such an extension could have provided insights into whether the agents would learn to navigate intersections—a metric that could be compared with the performance of already well-trained models, such as the InterFuser model itself. One way to expedite the training process while still limiting the number of training steps might involve the inclusion of imitation learning, either as a preliminary step or through the use of workers during the reinforcement learning process. This method aligns with the approach employed by Chekroun et al. [9]. Alternatively, the strategy proposed by Lu et al. [23], as described in section 2.4.5, could be utilized. Given that the custom-trained InterFuser architecture, as used in Experiment 2a, is trained to predict a probability distribution of future waypoints, this prediction could be applied to modify the action selection within the utilized RL algorithm. Consequently, this would incorporate the strategy of blending imitation learning with reinforcement learning.

The focus of the experiments lies in evaluating whether the incorporated sensor fusion architecture can enhance the performance of a reinforcement learning agent within a pure reinforcement learning setting. Although there are numerous methods to incorporate imitation learning into the reinforcement learning process, this thesis prioritizes exploring the sensor fusion architecture as a visual encoder. Therefore, the integration of imitation learning was not considered a primary necessity in the experiments.

Hardware Most of the experiments were run on a setup with dual A100 GPUs, capable of running many parallel workers for the reinforcement learning algorithm. Between the available choices for hardware usable for this thesis, this was the most powerful setup available and reduced training time of 1 million steps from several days to around 28 hours. However, the A100 GPUs produce some graphical glitches in the Carla simulator when collecting images from the camera sensors of the ego vehicle.

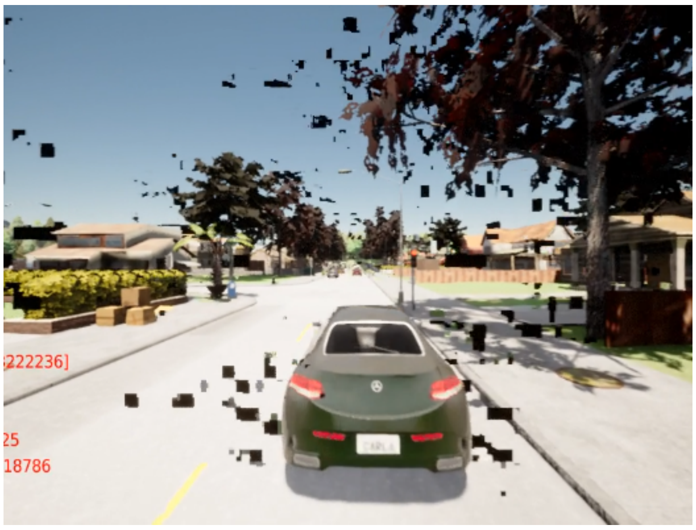


Figure 5.1: Camera artifacts

These artifacts, as shown in figure 5.1 on the images might have affected the performance of the predictions from the InterFuser models, as the training data used to train the InterFuser models did not include images with these artifacts.

Technical issues A significant amount of time has been spent getting around various technical issues arising from basing much of the experiments on using the CARLA scenario runner as a core for running reinforcement learning episodes. With the Carla simulator, running many routes in sequence often results in crashes or potential memory leaks and illegal memory access. This resulted in episodes ran with the episode manager being unpredictably terminated or losing connection to the simulator. Debugging the frequent crashing of the simulator was a time-consuming task, and the specific cause of the crashes was not found, but instead many workarounds were employed and tested. The goal of having a reinforcement learning process with scenarios at the core of the training process with a process robust enough to handle crashes, keeping checkpoints, resuming training and getting enough information from the simulator and reinforcement learning statistics proved to be a much more difficult task than originally anticipated. Optimally, more time should have been spent on testing additional strategies and tuning the reward function based on results from the experiments, than on creating various workarounds for technical issues.

5.4 Reflection

If this work were to be repeated, more time should have been allocated to developing more robust and diverse baseline agents. It would have been beneficial to define and implement an agent with pre-trained visual encoders, rather than relying on a naive deep reinforcement learning agent with raw camera data as part of the observation space. This would have allowed for a more comprehensive comparison between sensor fusion transformer architectures and classical visual encoder architectures. Unfortunately, a significant amount of time was spent on developing tools for running RL in the CARLA simulator, particularly with the scenario runner at the core. Unexpected technical issues arose, resulting in this process taking much longer than anticipated.

During the experiments, the performance of the agents did not reach a level that would justify the use of challenging scenarios and specific sets of routes through various map variations. Given the achieved performance, using two similar towns, one for training and another for evaluation, might have been sufficient. By settling with a simpler environment implementation, more time could have been allocated to developing more robust baseline comparisons. Additionally, this would have allowed for the inclusion of performance evaluation with other types of training regimes, such as GRIAD or the approach used in "Imitation is not enough" by Lu et al. [23]. Moreover, it would have been beneficial to compare the results with other RL algorithms instead of limiting the experiments to using only PPO.

Chapter 6

Conclusion and Future Work

In this chapter, a conclusion based on the findings of the experiments are drawn, as well as considerations for future work.

6.1 Conclusion

The advancement of self-driving technology is imminent, and crucial research is required to ensure its safe and reliable implementation. This thesis explored the potential of leveraging sensor fusion for deep reinforcement learning in autonomous driving within simulated environments, with a particular focus on the use of the InterFuser architecture as a visual encoder. The findings revealed promising potential in applying the InterFuser architecture within a reinforcement learning setting, outperforming a naive RL approach when using a compact feature vector initially meant for waypoint prediction. Nevertheless, the thesis's results also highlighted certain limitations such as insufficient training time and a lack of comparison to other pre-trained encoder approaches. The reward function, which sought to model desired driving behavior, often led to agents driving at exceptionally slow speeds to avoid collisions, revealing a potential weakness in its design. Despite the flaws of the reinforcement learning process within the experiments conducted in this thesis, the experiments still produce agents that can follow lanes and stop to avoid collisions with vehicles in front of the ego vehicle. Effectively, the proposed Reinforced InterFuser architecture shows that reinforcement learning can benefit from sensor fusion approaches, although the approach is not yet fully realized and comparable to state-of-the-art approaches.

Deploying safety mechanisms to an agent trained with the proposed Reinforced InterFuser method proved to be especially useful for avoiding collisions when new unseen challenging scenarios are involved in the environment, vastly improving the agent's performance. In the simple test cases used in this work, RL agents seem to benefit from additional safety mechanisms deduced from explicit predictions of the environments.

6.2 Future work

Future research might benefit from a more precise design of the reward function to promote optimal driving behavior more effectively. Crafting a reward function that encourages optimal driving is a complex challenge, and reward shaping strategies that align with this goal are critical if pure reinforcement learning is to outshine imitation. Furthermore, the potential integration of imitation learning into the reinforcement learning process warrants exploration. While this integration wasn't deemed essential for the experiments conducted, it could present a fruitful area for future research. Most successful approaches in the CARLA challenge have leveraged imitation learning; however, imitation is inherently restricted by the data it's trained on. Strategies that combine the benefits of imitation learning for standard driving behavior and reinforcement learning for managing unseen, challenging scenarios may represent a promising future research path, with the current work serving as a basis.

The custom-trained InterFuser model used in this thesis could be adapted to use waypoint prediction probabilities in conjunction with a GRU unit as a waypoint-prediction RL head, as opposed to a simpler MLP action selection head. Other potential avenues for exploration include comparing other pre-trained encoder models trained on implicit affordances and extending the training duration to assess whether the approach can learn more complex driving behaviors, such as navigating challenging scenarios and properly crossing intersections. Approaches like UniAD [25], discussed in section 2.4.6, appear promising for future research, as they build upon the concept of sensor fusion using transformer architectures and centralize planning in the architecture. Like all imitation learning approaches, this could also be combined with reinforcement learning to manage edge cases where expert demonstrations might be unavailable. Other works, like ReasonNet [27], which include a temporal reasoning module, could have significant potential for reinforcement learning. An exploration of such a module in an RL setting could be instrumental in handling situations where other actors are occluded but were previously visible in the agent's field of view.

Bibliography

[1] S. International, *SAE Levels of Driving Automation*, https://www.sae.org/blog/sae-j3016-update, [Online; accessed 5-December-2022], 2022.

[2] Waymo LLC, *Waymo*, n.d. [Online]. Available: https://waymo.com/.

[3] Carla, *Carla - Open-source simulator for autonomous driving research*, https://carla.org/, [Online; accessed 5-December-2022], 2022.

[4] Nvidia, *Nvidia Drive sim*, https://developer.nvidia.com/drive/simulation, [Online; accessed 5-December-2022], 2022.

[5] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor and Y. Wu, *Stable baselines*, https://github.com/hill-a/stable-baselines, 2018.

[6] *RLlib: Industry-Grade Reinforcement Learning x2014; Ray 2.4.0 — docs.ray.io*, https://docs.ray.io/en/latest/rllib/index.html, [Accessed 08-Jun-2023].

[7] J. Ho and S. Ermon, 'Generative adversarial imitation learning,' *CoRR*, vol. abs/1606.03476, 2016. arXiv: 1606.03476. [Online]. Available: http://arxiv.org/abs/1606.03476.

[8] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot and et al., *Mastering the game of go with deep neural networks and tree search*, Jan. 2016. [Online]. Available: https://www.nature.com/articles/nature16961.

[9] R. Chekroun, M. Toromanoff, S. Hornauer and F. Moutarde, 'GRI: general reinforced imitation and its application to vision-based autonomous driving,' *CoRR*, vol. abs/2111.08575, 2021. arXiv: 2111.08575. [Online]. Available: https://arxiv.org/abs/2111.08575.

[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser and I. Polosukhin, 'Attention is all you need,' *CoRR*, vol. abs/1706.03762, 2017. arXiv: 1706.03762. [Online]. Available: http://arxiv.org/abs/1706.03762.

[11] IBM, *Recurrent Neural Networks*, `https://www.ibm.com/cloud/learn/recurrent-neural-networks`, [Online; accessed 5-December-2022], 2022.

[12] R. C. Staudemeyer and E. R. Morris, 'Understanding LSTM - a tutorial into long short-term memory recurrent neural networks,' *CoRR*, vol. abs/1909.09586, 2019. arXiv: `1909.09586`. [Online]. Available: `http://arxiv.org/abs/1909.09586`.

[13] OpenAI, *Gpt-4 technical report*, 2023. arXiv: `2303.08774 [cs.CL]`.

[14] J. Devlin, M. Chang, K. Lee and K. Toutanova, 'BERT: pre-training of deep bidirectional transformers for language understanding,' *CoRR*, vol. abs/1810.04805, 2018. arXiv: `1810.04805`. [Online]. Available: `http://arxiv.org/abs/1810.04805`.

[15] G. Sharir, A. Noy and L. Zelnik-Manor, 'An image is worth 16x16 words, what is a video worth?' *CoRR*, vol. abs/2103.13915, 2021. arXiv: `2103.13915`. [Online]. Available: `https://arxiv.org/abs/2103.13915`.

[16] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Doll'a r and C. L. Zitnick, 'Microsoft COCO: common objects in context,' *CoRR*, vol. abs/1405.0312, 2014. arXiv: `1405.0312`. [Online]. Available: `http://arxiv.org/abs/1405.0312`.

[17] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin and B. Guo, 'Swin transformer: Hierarchical vision transformer using shifted windows,' *CoRR*, vol. abs/2103.14030, 2021. arXiv: `2103.14030`. [Online]. Available: `https://arxiv.org/abs/2103.14030`.

[18] M. Toromanoff, É. Wirbel and F. Moutarde, 'End-to-end model-free reinforcement learning for urban driving using implicit affordances,' *CoRR*, vol. abs/1911.10868, 2019. arXiv: `1911.10868`. [Online]. Available: `http://arxiv.org/abs/1911.10868`.

[19] valeoai, *Rainbow-IQN-APE-X*, `https://github.com/valeoai/rainbow-iqn-apex`, [Online; accessed 5-December-2022], 2022.

[20] K. Chitta, A. Prakash, B. Jaeger, Z. Yu, K. Renz and A. Geiger, *Transfuser: Imitation with transformer-based sensor fusion for autonomous driving*, 2022. doi: `10.48550/ARXIV.2205.15997`. [Online]. Available: `https://arxiv.org/abs/2205.15997`.

[21] K. Chitta, A. Prakash and A. Geiger, 'NEAT: neural attention fields for end-to-end autonomous driving,' *CoRR*, vol. abs/2109.04456, 2021. arXiv: `2109.04456`. [Online]. Available: `https://arxiv.org/abs/2109.04456`.

[22] H. Shao, L. Wang, R. Chen, H. Li and Y. Liu, *Safety-enhanced autonomous driving using interpretable sensor fusion transformer*, 2022. doi: `10.48550/ARXIV.2207.14024`. [Online]. Available: `https://arxiv.org/abs/2207.14024`.

[23] Y. Lu, J. Fu, G. Tucker, X. Pan, E. Bronstein, B. Roelofs, B. Sapp, B. White, A. Faust, S. Whiteson, D. Anguelov and S. Levine, *Imitation is not enough: Robustifying imitation with reinforcement learning for challenging driving scenarios*, 2022. arXiv: `2212.11419 [cs.AI]`.

[24] E. Bronstein, S. Srinivasan, S. Paul, A. Sinha, M. O'Kelly, P. Nikdel and S. Whiteson, 'Embedding synthetic off-policy experience for autonomous driving via zero-shot curricula,' in *6th Annual Conference on Robot Learning*, 2022. [Online]. Available: `https://openreview.net/forum?id=cF1dxVGxic-`.

[25] 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. arXiv: `2212.10156 [cs.CV]`.

[26] Z. Li, W. Wang, H. Li, E. Xie, C. Sima, T. Lu, Q. Yu and J. Dai, *Bevformer: Learning bird's-eye-view representation from multi-camera images via spatiotemporal transformers*, 2022. arXiv: `2203.17270 [cs.CV]`.

[27] 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. arXiv: `2305.10507 [cs.CV]`.

[28] *TORCS - The Open Racing Car Simulator — sourceforge.net*, `https://sourceforge.net/projects/torcs/`, [Accessed 12-Jun-2023].

[29] *TensorBoard | TensorFlow — tensorflow.org*, `https://www.tensorflow.org/tensorboard`, [Accessed 08-Jun-2023].

[30] *Weights Biases – Developer tools for ML — wandb.ai*, `https://wandb.ai/site`, [Accessed 08-Jun-2023].

[31] *Homepage — comet.com*, `https://www.comet.com/site/`, [Accessed 08-Jun-2023].

[32] J. Schulman, F. Wolski, P. Dhariwal, A. Radford and O. Klimov, 'Proximal policy optimization algorithms,' *CoRR*, vol. abs/1707.06347, 2017. arXiv: `1707.06347`. [Online]. Available: `http://arxiv.org/abs/1707.06347`.

[33] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. S. Bernstein, A. C. Berg and L. Fei-Fei, 'Imagenet large scale visual recognition challenge,' *CoRR*, vol. abs/1409.0575, 2014. arXiv: `1409.0575`. [Online]. Available: `http://arxiv.org/abs/1409.0575`.

Appendix

Code

- Reinforced InterFuser: `https://github.com/Haavasma/reinforced_interfuser`
- Episode manager: `https://github.com/Haavasma/episode_manager`

Links

- Carla Online leaderboard: `https://leaderboard.carla.org/leaderboard/`

Data download links

- Expert dataset
- Reinforced InterFuser with waypoint features weights
- Baseline agent weights
- Custom InterFuser weights
- Carla routes

Driving behavior videos

- Reinforced interfuser with waypoint feature and safety controller, challenging scenario environment: `https://youtu.be/DBnEtQ85d7Q`
- Baseline agent, challenging scenario environment: `https://youtu.be/94lPLkKEMRg`

NTNU
Norwegian University of
Science and Technology