Sindre Kummervold

Training of Reinforcement Learning Agents for Autonomous Driving in Simulated Environments

Master's thesis in Cybernetics and Robotics
Supervisor: Morten Breivik
Co-supervisor: Frank Lindseth and Gabriel Kiss
June 2023

Master's thesis

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

NTNU
Norwegian University of
Science and Technology

Sindre Kummervold

Training of Reinforcement Learning Agents for Autonomous Driving in Simulated Environments

Master’s thesis in Cybernetics and Robotics
Supervisor: Morten Breivik
Co-supervisor: Frank Lindseth and Gabriel Kiss
June 2023

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

Abstract

A system capable of autonomous driving needs to have several capabilities. Firstly, the vehicle needs to be able to sense its environment, and secondly, it needs to use the information about its surroundings to maneuver to its desired destination. This thesis is focused on the use of Reinforcement Learning (RL) to maneuver the vehicle in its environment - the open-source autonomous driving simulator CARLA. Previous RL systems in the CARLA simulator uses simple vision encoders to sense their surroundings, possibly limiting their performance.

This thesis investigates the utilization of more complex pre-trained vision encoders in the context of Reinforcement Learning (RL) for autonomous driving. Multiple agents were trained in environments within the CARLA simulator, varying in complexity. The baseline training using the Proximal Policy Optimization (PPO) algorithm with a complex transformer-based vision encoder produced suboptimal results, as the vehicle consistently veered off the road and encountered crashes. Incorporating expert demonstrations through the General Reinforced Imitation for Autonomous Driving (GRIAD) approach did not enhance performance, leaving the vehicle stationary on the road. To overcome these limitations, a simplified environment with reduced traffic complexity was created, resulting in notable advancements in subsequent training runs.

In the simplified environment, both the complex encoder and a simple Convolutional Neural Network (CNN) were employed. The CNN encoder demonstrated superior performance compared to the TransFuser encoder. Nevertheless, both approaches encountered challenges in avoiding collisions with other vehicles. It is important to note that the training duration only was 1 million steps, necessitating further investigation to draw definitive conclusions.

Future research should focus on assessing the impact of more complex vision encoders on the training of RL agents. Extending the training time and gradually increasing traffic complexity can lead to a more thorough understanding of the potential benefits and drawbacks associated with sophisticated vision encoders in RL scenarios. By addressing these areas, advancements can be made in the development of effective training methodologies for RL agents operating in complex real-world environments.

Sammendrag

Et system som er i stand til autonom kjøring, må ha flere funksjoner. For det første må kjøretøyet kunne sanse omgivelsene sine, og for det andre må det bruke informasjonen om omgivelsene for å manøvrere til ønsket destinasjon. Denne oppgaven er fokusert på bruken av Reinforcement Learning (RL) for å manøvrere kjøretøyet i dets miljø - den autonome kjøresimulatoren med åpen kildekode CARLA. Tidligere RL-systemer i CARLA-simulatoren bruker enkle syns-enkodere for å registrere omgivelsene, noe som muligens begrenser ytelsen.

Denne oppgaven undersøker bruken av mer komplekse forhåndstrente syns-enkodere i sammenheng med Reinforcement Learning (RL). Flere agenter ble opplært i miljøer i CARLA-simulatoren, med varierende kompleksitet og utfordringer. En ren RL agent ble trent for å gi ett sammenligninigsgrtunnlag, ved bruk av Proximal Policy Optimization (PPO)-algoritmen med Transfuser enkoderen og ga suboptimale resultater, ettersom kjøretøyet konsekvent svingte av veien og kolliderte. Inntroduksjon av ekspertdemonstrasjoner gjennom General Reinforced Imitation for Autonomous Driving (GRIAD)-tilnærmingen forbedret ikke ytelsen, og resulterte i at kjøretøyet sto stille på veien. For å overvinne disse begrensningene ble det laget et forenklet miljø med redusert trafikkkompleksitet, noe som resulterte i store fremskritt i påfølgende treninger.

I det forenklede miljøet ble både TransFuser-enkoderen og en enkel Convolutional Neural Network (CNN) brukt. CNN-enkoderen demonstrerte bedre ytelse sammenlignet med TransFuser. Likevel møtte begge tilnærmingene utfordringer med å unngå kollisjoner med andre kjøretøy. Det er viktig å merke seg at opplæringsvarigheten var relativt kort, noe som nødvendiggjorde ytterligere undersøkelser for å trekke definitive konklusjoner.

Fremtidig forskning bør fokusere på å vurdere virkningen av mer komplekse synskodere på opplæringen av RL agenter. Å forlenge treningstiden kan føre til en mer grundig forståelse av de potensielle fordelene og ulempene forbundet med sofistikerte syns-enkodere i RL-scenarier. Ved å adressere disse områdene kan det gjøres fremskritt i utviklingen av effektive opplæringsmetoder for RL-agenter som opererer i komplekse miljøer i den virkelige verden.

Preface

The topic of autonomous driving has always fascinated me, and I am grateful for the opportunity to delve deeper into this exciting and rapidly developing field as part of my studies. The insights and knowledge I have gained during the course of this project have been invaluable, and I am grateful to my supervisors for their guidance and support as I worked to complete this report.

This report represents the culmination of months of hard work and dedication. I am grateful to have had the support and guidance of my supervisors Morten Breivik, Frank Lindseth, Gabriel Kiss, and Florian Wintel throughout the process.

I would also like to extend thanks to Jan Christan Meyer, who helped me a lot with the simulator issues I encountered on Idun.

I hope that the information in this report will be useful to future students in the autonomous driving field.

Sindre B. Kummervold
Trondheim, 12. June, 2023

Table of Contents

List of Figures

List of Tables

Acronyms

AC Actor-Critic. x, xiii, 25, 42, 53

BEV Bird’s eye view. xiii, 28, 55, 80

CARLA Car Learning to Act. iii, v, x, xi, xiii, xvii, 1, 3–5, 17, 32, 35, 37, 38, 40, 41, 45, 47, 48, 53–57, 59, 60, 64, 67, 82, 87, 89

CNN Convolutional Neural Network. iii, v, ix, xi, xii, 8–10, 32, 45, 78, 79, 87, 104, 105

DDPG Deep Deterministic Policy Gradient. 43

FoV Field of View. 37

GNSS Global Navigation Satellite System. 38

GRIAD General Reinforced Imitation for Autonomous Driving. iii, v, x–xii, xiv, 3, 4, 32, 33, 45, 55, 64, 72–75, 81, 83, 87, 88, 102

GRU Gated Recurrent Unit. ix, 11, 12

HPC High Performance Computing. 35, 54, 72

IL Imitation Learning. x, 4, 7, 17, 32

IMU Inertial Measurement Unit. 38

KL Kullback-Leibler. x, 26, 27

LiDAR Light detection and ranging. xiii, 27, 28, 31, 37, 45, 46

LSTM Long Short-Term Memory. ix, xiii, 10–12

MDP Markov Decision Process. 19

MHA Multi Headed Attention. ix, 14–16

NAPLab NTNU Autonomous Perception Labratory. 36, 89

PID Proportional Integral Derivative Controller. 4, 31

PPO Proximal Policy Optimization. iii, v, x, xiv, xvii, 20, 25, 42, 43, 64, 65, 67, 72, 78, 83, 87

RL Reinforcement Learning. iii–vi, x, xi, 2–5, 7, 17, 19–21, 23, 25, 26, 32, 35, 37, 40, 41, 43, 45, 49, 53, 54, 57, 63, 65, 67, 69, 71, 72, 78–81, 83, 87–89

RNN Recurrent Neural Network. ix, 10, 11

SB3 Stable Baselines 3. x, 37, 39, 40, 53, 64, 78

SotA State-of-the-Art. 4, 45

SSA Smallest Signed Angle. 50

TD Temporal Difference. x, 21–23, 25

TD3 Twin Delayed DDPG. x, 25, 43, 44, 72, 73, 83

WoR World on Rails. 3

XRL Explainable Reinforcement Learning. xii, 89, 90

Chapter 1

Introduction

1.1 Motivation

In the whole world, 1.3 million people lose their lives due to traffic accidents each year, and it is the leading cause of death of people between 5-30 years old, according to a study by the world health organization [1]. While car safety has increased due to governmental regulations, that does little to protect vulnerable road users, who account for more than half of the world's road deaths. According to the U.S. Department of Transportation, 94% of all traffic accidents are caused by human error [2]. At the same time, another study shows that 55% of vehicle fatalities are caused by alcohol and speeding [3]. The WHO also lists several leading human causes of traffic accidents, such as driving under the influence and distracted drivers. Road accidents also have a significant economic impact, resulting in an estimated 3% reduction in GDP [1]. Autonomous vehicles and other advanced driving assistance tools have the potential to aid in reducing these numbers by removing the option for human error.

However, developing autonomous driving systems that are safe, reliable, and efficient is a challenging task. Real-world testing of these systems is expensive and risky and requires significant amounts of time and resources. Simulated environments, such as the Car Learning to Act (CARLA) simulator, have emerged as powerful tools for developing and testing autonomous driving systems. Simulations provide a safe and controlled environment to test new algorithms. They can be used to generate large amounts of training data that would be difficult or impossible to obtain in the real world.

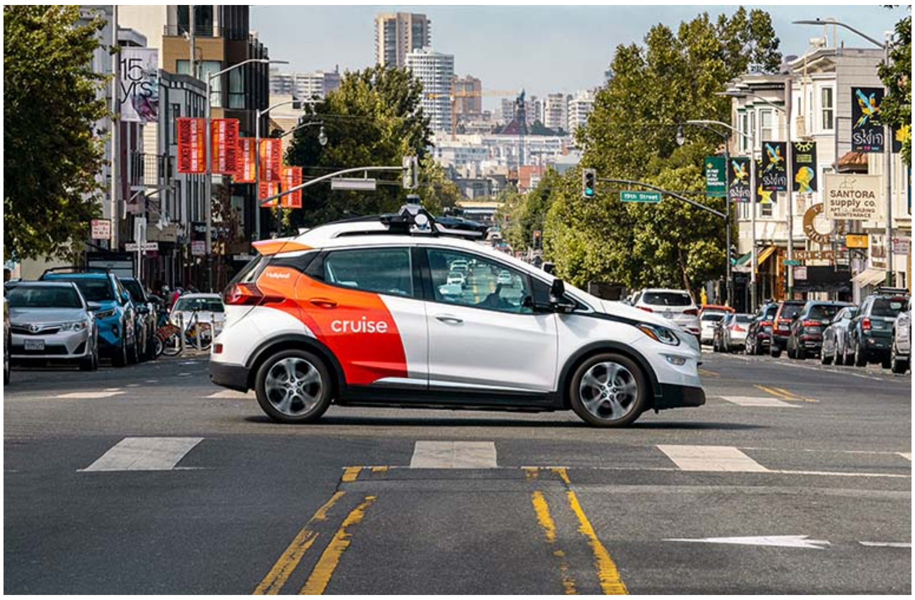


Figure 1.1: San Francisco already allows real-world testing of autonomous vehicles for personal transport. This image shows Cruise´s autonomous taxi on the road. Courtesy of Cruise [4].

Reinforcement Learning (RL) is a promising approach for developing autonomous driving systems in simulated environments. RL algorithms learn to make decisions by interacting with the environment and receiving feedback in the form of rewards. By repeatedly making decisions and receiving feedback, an RL agent can learn to navigate complex environments and perform complex tasks, such as driving a car. However, RL algorithms require large amounts of trial and error to learn effectively, which can be very time-consuming and computationally expensive.

One approach to addressing the data efficiency challenge in RL is to use pretraining and expert demonstrations. Pretraining involves training a neural network on a large, diverse dataset before fine-tuning it on a specific task. This can help the network learn useful features that generalize across different tasks and can reduce the amount of data needed for fine-tuning. Expert demonstrations involve using data from human or expert drivers to teach an RL agent how to perform a specific task. This can help the agent learn more quickly and effectively and can also provide a benchmark for evaluating the agent's performance.

By combining RL with pretraining and expert demonstrations, researchers can develop autonomous driving systems that are more data-efficient, more robust, and more effective. The use of these techniques can help accelerate the development and deployment of autonomous driving systems and bring the benefits of this technology to more people around the world.

1.2 Background and Previous Work

In recent years the open-source simulator CARLA has become popular for developing systems for autonomous driving. CARLA has support for simulating camera and lidar data and several other sensors. In addition, the simulator can run several predefined and custom scenarios and routes across several maps and weather conditions. When run, the agent receives a score for each scenario, allowing for comparisons between different systems. Researchers can submit these scores to a public leaderboard.

In their 2019 paper [5], Toromanoff et al. presented MaRLn, a model-free end-to-end approach to autonomous driving, marking the first successful RL entry on the leaderboard. MaRLn won the Camera-only CARLA challenge that year. They use a custom pre-trained ResNet-18 encoder. Pretraining is performed with segmentation and detection tasks. The detections are of traffic lights, intersections, and lane positions. The encoder weights are frozen while training the RL agent. Toromanoff also introduced implicit affordances, allowing training of replay memory-based RL with a much larger network and input size than most of the networks used in previous RL works.

Chen et al. introduced World on Rails (WoR) in [6], beating the previous top entry on the leaderboard - using 40 times less data. The central assumption of WoR is that " the world is on rails" - meaning that neither the agent nor its actions influence the environment. This assumption allows for decoupling the world model into the ego and static environment models. Firstly, a forward model of the agent is trained to simulate actions that the agent performs. Next, Chen et al. compute a table of action-value pairs for each training trajectory using a dynamic-programming evaluation of the Bellman equations. Finally, these tables are used to supervise the training of the final vision-based policy.

The highest-ranked Reinforcement Learning entry on the CARLA leaderboard was introduced by Chekorun et al. in 2021 [7]. GRIAD aims to combine expert demonstrations of IL with the exploration of RL, allowing the agent to learn faster than a pure RL agent. This is done by using an off-policy RL algorithm with a replay buffer. For each episode, a series of state-action pairs are collected by letting the agent explore the environment or sampling from a prerecorded dataset generated by an expert agent. The authors assume expert demonstrations can be seen as perfect data following an optimal policy.

In *Transfuser: Imitation with Transformer-Based Sensor Fusion for Autonomous Driving* - published in 2022 - Chitta et al. investigate how to combine information from different types of sensors [8]. The resulting network fuses an RGB image with a pseudo-image generated from a lidar scan using the self-attention mechanism, resulting in a 100% increase in driving score over GRIAD. The features output from the fusion encoder are passed to a waypoint generating network, and the control is performed via two PID controllers .

Interfuser is an end-to-end learning framework for autonomous driving developed by H. Shao et al. in 2022 [9] and currently ranks 2nd on the CARLA leaderboard [10]. Interfuser also utilizes transformer-based sensor fusion, but unlike Transfuser, it uses an encoder-decoder structure. The authors focused on creating a model with increased safety and interpretability. They do this by outputting the model's intermediate features and putting constraints on the actions to ensure they are safe.

In the previous two years, the introduction of transformers has revolutionized the CARLA leaderboard. At the time of writing, three out of the top five entries utilize transformers in their vision encoders, including the top two. A the same time, a shift has occurred from Reinforcement Learning (RL) towards Imitation Learning (IL), with the last RL entry being GRIAD in 2021.

1.3 Problem Description

The goal of this thesis is to investigate whether the improvements the vision encoders have gone through over the past two years have the possibility of improving previous state-of-the-art methods based on simpler vision systems. The method investigated will be General Reinforced Imitation for Autonomous Driving (GRIAD), which has an impressive performance despite having one of the simplest vision encoder and sensor setups out of the current top 10. Specifically, the following tasks will be performed:

- Perform a review of current State-of-the-Art (SotA) systems for the purpose of choosing a suitable vision encoder.
- Train a vision encoder on data collected from CARLA
- Develop the necessary software to train RL agents in CARLA, specifically, a Python environment based on the OpenAI interface.
- Train an agent in the same environment without expert demonstrations that can be used as a baseline for comparison purposes.
- Train an agent using GRIAD, using the same environment setup as the baseline agent.
- Evaluate the performance of the trained agents to each other and to other SotAmethods.

1.4 Contributions

The following are the contributions of this thesis to the research field of deep reinforcement learning for autonomous driving:

- An in-depth review of some state-of-the-art performing reinforcement learning and imitation learning approaches to autonomous driving.
- Creation of a Python wrapper that allows for easy training of RL agents in the CARLA simulator using the OpenAI gym interface.
- An investigation into how the performance of Reinforcement Learning agents are affected by using a more complex vision encoder.
- A look into how the complexity of traffic scenarios affects early stage training of Reinforcement learning agents in the CARLA simulator.
- Some issues were discovered with Transfuser's data generation pipeline if the encoder is used for Reinforcement Learning.

1.5 Thesis Outline

The thesis opens up with Chapter 2, which presents a detailed walkthrough of relevant machine learning theory. Chapter 3 present the used Reinforcement Learning algorithms, as well as the observation space, action space, and reward function, which will be used when interacting with the simulator. These will be used for training Reinforcement Learning agents in the simulator, and the results and discussions will be presented in Chapter 4. Finally, Chapter 5 concludes the thesis, along with some suggestions for future works.

Chapter 2

Background Theory

This chapter will cover the theoretical foundations and topics that are related to this thesis. Some important machine learning architectures will be introduced before a thorough description of Reinforcement Learning - including the current state-of-the-art approaches - and Imitation Learning. Finally, the previous work this thesis is based on will be covered in more detail than in Section 1.2. There will also be a short introduction to some statistical concepts used during the training evaluation.

2.1 Computer Vision

Computer vision tasks play a fundamental role in enabling machines to interpret and understand visual data. Some of the most common tasks in computer vision are object detection, classification, and segmentation. Each task has distinct objectives and real-world applications, which are summarized in Table 2.1.

Classification is a fundamental task in computer vision and aims to assign a single label to an image. It involves determining the class or category that best represents the content of the image. Some classification algorithms leverage deep learning architectures to extract high-level features from images and make predictions based on learned patterns. Object detection involves identifying and localizing multiple objects within an image. It surpasses simple classification by not only assigning labels to objects but also providing spatial information using bounding boxes. This task finds applications in autonomous driving, surveillance systems, and object tracking, enabling machines to perceive and interact with their environment effectively.

There are two main types of segmentation in modern computer vision - semantic and instance segmentation. Semantic segmentation is a pixel-level labeling task where the goal is to assign a label to each pixel in an image, indicating the category to which the pixel belongs. Instance segmentation is a more advanced computer vision task that combines object detection and pixel-level segmentation. It involves identifying and delineating each individual object instance in an image by providing pixel-level masks. Unlike object detection, which provides bounding boxes, instance segmentation provides detailed masks that precisely outline the boundaries of each object. It also differs from sematic segmentation in that it distinguishes individual object instances, while semantic segmentation focuses on labeling different areas or regions in an image based on their semantic meaning.

Aspect	**Object Detection**	**Classification**	**Instance Segmentation**	**Semantic Segmentation**
Objective	Identify and locate objects	Assign a label to an input	Identify and locate objects	Assign a label to each pixel
Output	Bounding boxes around objects	Class of object	Pixel-level masks for each object	Pixel-level masks for each class
Level of Detail	Object-level	Object-level	Object-level	Pixel-level
Number of Labels	Multiple	Single	Multiple	Multiple

Table 2.1: Common tasks in computer vision.

2.2 Convolutional Neural Network (CNN)

The CNN is the most common neural network used for vision tasks due to its ability to extract useful features from images. The basis of the CNN is the convolution operation, which is used to extract features from the input image. A visualization of the convolution operation is shown in Figure 2.1. It involves sliding a small matrix, called a kernel, over the input image. At each location, the filter is multiplied element-wise with the corresponding patch of the input image, and the resulting values are summed to produce a single output value. This operation is repeated for all possible locations in the image, producing a new output image - which is called a feature map. The kernels are usually 3x3 or 5x5 matrices of learnable parameters. By iteratively adjusting these parameters during training using gradient-based optimization techniques, the network learns to recognize patterns and make accurate predictions without requiring explicit feature engineering.

CNNs also typically include other layers, such as pooling and fully connected layers. Pooling layers are used to reduce the dimensionality of the output from convolutional layers, to achieve a more compressed representation and introduce translational invariance. The fully connected

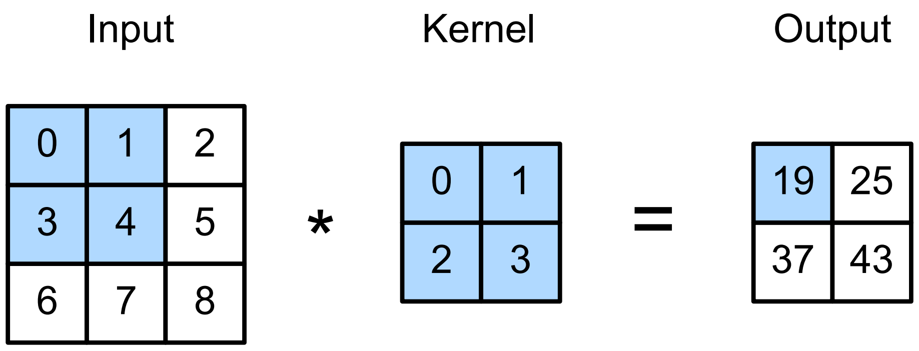


Figure 2.1: Visualization of the convolution operation.

layers can be used after the convolutional layers to perform classification or bounding box regression tasks.

2.2.1 Inductive Biases

The CNN architecture has some structural inductive biases. Introducing inductive biases into models can help them learn and generalize, especially with small amounts of data. However, with a large amount of data, it might be preferable to let the model be less constrained. In the early days of deep learning, data was more limited. Therefore, CNNs were created with some biases to aid learning.

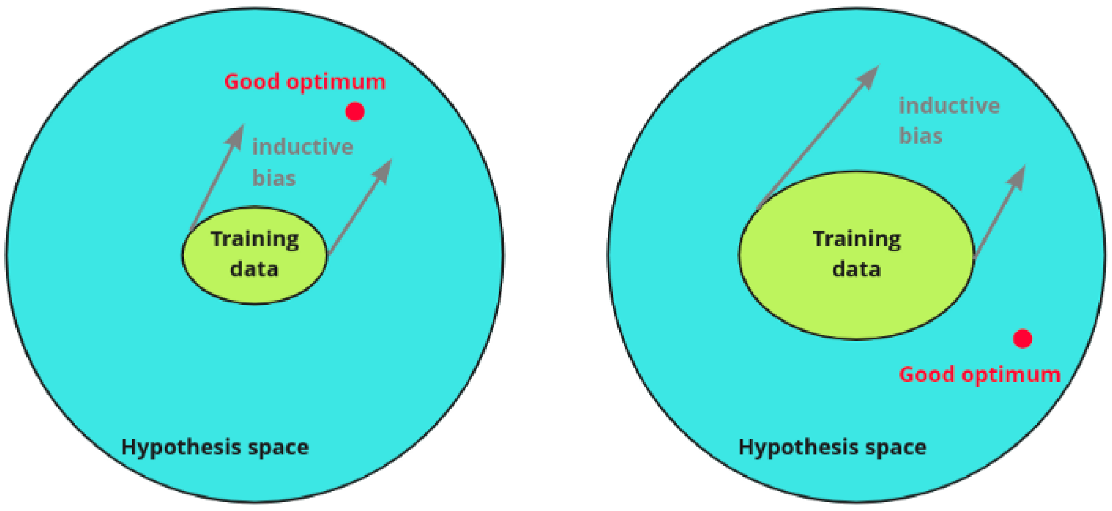


Figure 2.2: Inductive biases. Courtesy of [11]

The convolution operation is local; therefore, a CNN is guided towards relating local areas in images. This is reasonable for tasks involving finding what objects are in an image since the context of the object is less important. However, for tasks such as autonomous driving, the context in which objects appear in an image is vital, so plain CNN's might perform worse than other architectures such as Transformers, as explained in Section 2.4.

Weight sharing across the entire spatial dimensions of the feature maps is the source of the second inductive bias - translational invariance. This is a reasonable assumption in most applications since the appearance of an object usually will not change depending on where it is in an image. The exception is in images with a large amount of distortion in parts of the image. The appearance of objects may then vary depending on where in the image it is located.

2.3 Recurrent Neural Network (RNN)

Parts of this section are taken from the equivalent section in my project thesis [12].

RNNs are a particular type of neural network specially designed to be used on data that involve an ordered series of data points. In this case, a data point depends on the previous data points. RNNs are therefore adapted to incorporate this dependency. To do this, RNNs introduce the concept of memory so the previous data points' effect on the current can be modeled.

2.3.1 Long Short-Term Memory (LSTM)

Hochreiter and Schmidhuber proposed the LSTM architecture in 1997 [13]. One of the problems of previous architectures was that they overrode their memory at every time step, meaning the network was only able to remember things from the previous input. The LSTM tries to fix this by allowing the network to combine previous memory and the new information introduced at a time step. This is done through the input, output, and forget gates [14]. Another problem with traditional RNNs is that they suffer from the vanishing gradient problem - which means that the gradient for the early layers of the network becomes so small that the network is impossible to train. As a result of this problem, RNNs can be hard to train, especially for long input sequences. LSTMs also address this by allowing the gradients to flow unchanged [14]. The equations of the LSTM are

$$\boldsymbol{f}_t = \sigma\left(\boldsymbol{W}_f \boldsymbol{x}_t + \boldsymbol{U}_f \boldsymbol{h}_{t-1} + \boldsymbol{V}_o \boldsymbol{c}_{t-1}\right) \tag{2.1}$$

$$\boldsymbol{i}_t = \sigma\left(\boldsymbol{W}_i \boldsymbol{x}_t + \boldsymbol{U}_i \boldsymbol{h}_{t-1} + \boldsymbol{V}_i \boldsymbol{c}_{t-1}\right) \tag{2.2}$$

$$\boldsymbol{o}_t = \sigma\left(\boldsymbol{W}_o \boldsymbol{x}_t + \boldsymbol{U}_o \boldsymbol{h}_{t-1} + \boldsymbol{V}_o \boldsymbol{c}_t\right) \tag{2.3}$$

$$\bar{\boldsymbol{c}}_t = \tanh\left(\boldsymbol{W}_c \boldsymbol{x}_t + \boldsymbol{U}_c \boldsymbol{h}_{t-1}\right) \tag{2.4}$$

$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \bar{\boldsymbol{c}}_t \tag{2.5}$$

$$\boldsymbol{h}_t = \boldsymbol{o}_t \odot \tanh\left(\boldsymbol{c}_t\right) \tag{2.6}$$

where $\boldsymbol{x}_t \in \mathbb{R}^d$ is the input to the LSTM. The three vectors $\boldsymbol{f}_t \in [0,1]^h$, $\boldsymbol{i}_t \in [0,1]^h$, and $\boldsymbol{o}_t \in [0,1]^h$ are the forget, input, and output activation, respectively, where h is the dimension of the cell state. $\boldsymbol{c}_t \in \mathbb{R}^h$ is the cell state vector, or the long-term memory of the cell, $\boldsymbol{h}_t \in [-1,1]^h$ is the hidden state vector or the active memory of the cell and $\bar{c}_t \in \mathcal{R}^h$ is the input activation - which is the new information introduced. σ is the element-wise sigmoid function, while the $\odot$ operation is the element-wise multiplication. From the equations, it can be seen that $\boldsymbol{f}_t$ is the weight for how much of the previous cell state should be carried over to the memory of the new cell, while $\boldsymbol{i}_t$ is the weight for how much of the new information should be introduced into the memory of the new cell. There are 11 learnable matrices in each LSTM cell, $\boldsymbol{W} \in \mathbb{R}^{h \times d}$, $\boldsymbol{U} \in \mathbb{R}^{h \times h}$ and $\boldsymbol{V} \in \mathbb{R}^{h \times h}$ with their respective indices. The full architecture can be seen in Figure 2.3.

2.3.2 Gated Recurrent Unit (GRU)

The Gated Recurrent Unit was introduced by Cho et al. in 2014 [14] and has the same goal as the LSTM - to combat the problem of short-term memory and the vanishing gradient - but in a more simplified form. The GRU only has an update and reset gate and removes the concept of the cell state. Because of this, it is now the hidden state that becomes the long-term memory of the network [14]. The equations of the GRU are simplified compared to the LSTM and are given by

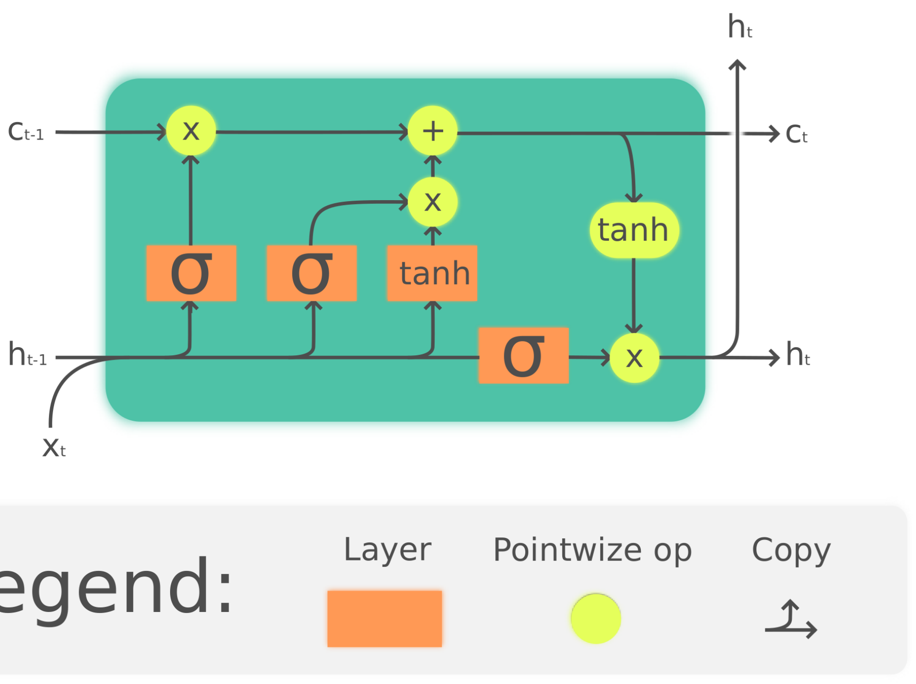


Figure 2.3: The LSTM architecture. Courtesy of [15] under the CC BY-SA 4.0 License via Wikimedia Commons.

$$\boldsymbol{z}_t = \sigma\left(\boldsymbol{W}_z\boldsymbol{x}_t + \boldsymbol{U}_z\boldsymbol{h}_{t-1}\right) \tag{2.7}$$

$$\boldsymbol{r}_t = \sigma\left(\boldsymbol{W}_r\boldsymbol{x}_t + \boldsymbol{U}_r\boldsymbol{h}_{t-1}\right) \tag{2.8}$$

$$\bar{\boldsymbol{h}}_t = \tanh\left(\boldsymbol{W}_h\boldsymbol{x}_t + \boldsymbol{U}_h(\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1})\right) \tag{2.9}$$

$$\boldsymbol{h}_t = \boldsymbol{z}_t \odot \boldsymbol{h}_{t-1} + (1 - \boldsymbol{z}_t) \odot \bar{\boldsymbol{h}}_t. \tag{2.10}$$

The vectors $\boldsymbol{x}_t \in \mathbb{R}^d$ and $\boldsymbol{h}_t \in \mathbb{R}^h$ are the input and output vectors for a given GRU cell, while $\boldsymbol{z}_t \in [0,1]^h$ and $\boldsymbol{r}_t \in [0,1]^d$ is the update and reset gate vectors. The reset gate vector chooses what parts of the previous cells memory to bring into the new cell, and this is combined with $\boldsymbol{x}_t$, both scaled my weight matrices to form the candidate activation vector $\bar{\boldsymbol{h}}_t \in [-1,1]^h$. The GRU has fewer learnable parameters than the LSTM, with a total of 8 matrices, $\boldsymbol{W} \in \mathbb{R}^{h\times d}$ and $\boldsymbol{U} \in \mathbb{R}^{h\times h}$, with their respective indices. The full GRU architecture is shown in Figure 2.4.

Since the GRU has fewer numerical operations than the LSTM, it is faster to train but allows it to model less complex dependencies. As a result, the LSTM tends to perform better on larger datasets, while the GRU performs better on smaller datasets. However, this is not a given, and

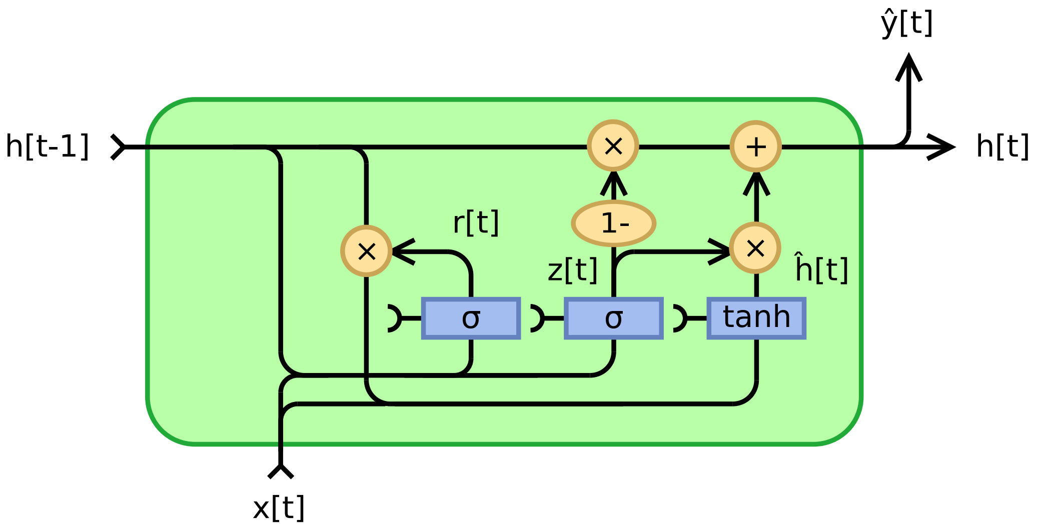


Figure 2.4: GRU architecture. Courtesy of [16] under the CC BY-SA 4.0 License via Wikimedia Commons.

researchers tend to experiment and figure out what works best.

2.4 Transformers

Parts of this section are taken from the equivalent section in my project thesis [12].

Transformers are a neural network architecture that has revolutionized Natural Language Processing and Computer Vision in the past couple of years. There are several key topics relating to Transformers, the foremost of which are self-attention, multi-head self-attention, and positional encoding. Since its introduction in 2017 as a model for neural machine translation [17], transformers have produced state-of-the-art performance in varying tasks such as language modeling, object detection, semantic segmentation, and sequence modeling [18–20]. The main component of the transformer is the attention mechanism. It allows the network to focus on different parts of the input sequence depending on the context of the sequence.

2.4.1 The Attention Mechanism

Since attention was first introduced in 2014, several score functions have been developed. However, the most commonly used today is the scaled dot-product attention, introduced by Vaswani et al. in 2017 [17]. In addition, they combined their new attention score with the self-attention mechanism introduced by Cheng et al. as intra-attention in 2016 [21]. Self-attention relates the different positions of the input sequence to each other. Self-attention has been shown to perform well in machine reading [21] and vision tasks [19]. The scaled dot-product score is the dot product of two vectors $\boldsymbol{s} \in \mathbb{R}^n$ and $\boldsymbol{h} \in \mathbb{R}^n$, scaled by the square root of their length;

$$\text{score}(\boldsymbol{s}, \boldsymbol{h}) = \frac{\boldsymbol{s} \cdot \boldsymbol{h}^T}{\sqrt{n}}. \tag{2.11}$$

The scaled dot-product attention of $\boldsymbol{Q} \in \mathbb{R}^{n \times d_k}$, $\boldsymbol{K} \in \mathbb{R}^{m \times d_k}$, and $\boldsymbol{V} \in \mathbb{R}^{n \times d_v}$ - also known as the query, key, and value matrices - is the scaled dot-product score of the key and query matrices passed through a softmax function, multiplied by the value matrix. Here d_v is the dimension of the value embeddings, while d_k is the dimension of the key and query embeddings. The scaled dot-product score between two matrices is a matrix containing the scores between all pairs of m vectors in $\boldsymbol{Q}$ and n vectors in $\boldsymbol{K}$. This can be expressed as

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{SoftMax}\left(\frac{\boldsymbol{Q} \cdot \boldsymbol{K}^T}{\sqrt{d_k}}\right) \cdot \boldsymbol{V}. \tag{2.12}$$

The attention mechanism between queries, keys, and values can be intuitively understood as a process of focusing on relevant information in the values based on the similarity between the queries and keys. Imagine a given query $\boldsymbol{q}$, a set of keys $\boldsymbol{K}$, and a corresponding set of values $\boldsymbol{V}$. Each key represents a piece of information, and its corresponding value contains the content associated with that information. The attention mechanism calculates the similarity between the query and each key. This similarity score determines the attention assigned to the corresponding value v. The higher the similarity between the query and a particular key; the more attention is placed on the corresponding value.

2.4.2 Multi Headed Attention (MHA)

Intuitively, multi-headed attention can be seen as having multiple heads, all looking at things from a different perspective. Each head captures different aspects or patterns in the data, allowing the model to gather a richer and more comprehensive understanding. For example, one head might focus on capturing the syntactic structure, another on capturing semantic relationships, and so on. By having multiple heads, the model can capture diverse patterns and dependencies in the input sentence simultaneously. Each head attends to different parts of the input sentence, learning different representations.

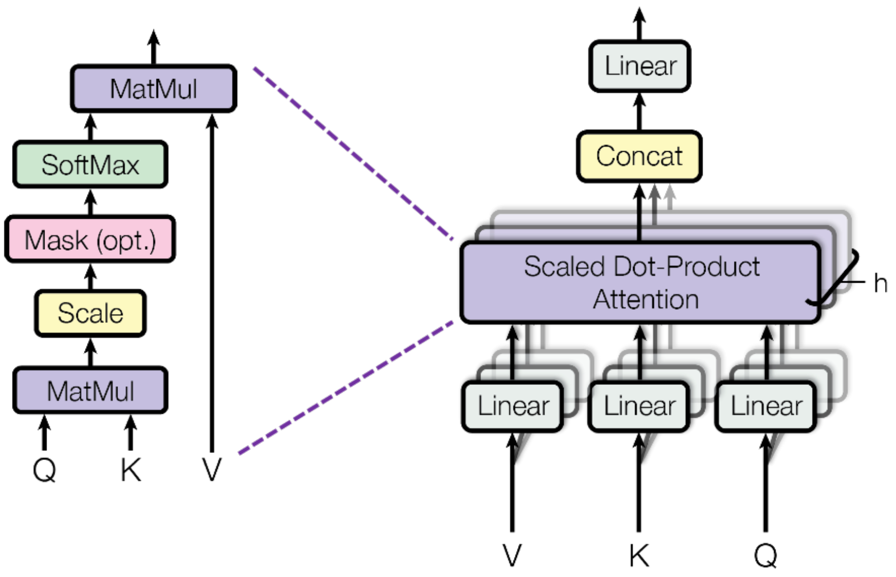


Figure 2.5: Scaled dot-product attention (left) and Multi-headed self-attention (right). Courtesy of [17].

Regular self-attention performs one computation with the query, key, and value vectors with dimensions d_m. Multi-headed attention projects the $\boldsymbol{K}$, $\boldsymbol{V}$, and $\boldsymbol{Q}$ matrices to lower dimensions, with different weight matrices for each head. The attention is computed for each of these projections in parallel. Finally, the output from each head is concatenated and transformed to the output dimensions using a linear transformation [17]. A visual representation of MHA can be seen in Figure 2.5, while the mathematical formulation is

$$\begin{aligned} \text{MultiHeadAttention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) &= \text{Concat}[\text{head}_1, \text{head}_2, \cdots]W^O \\ \text{head}_i &= \text{Attention}(\boldsymbol{Q}\boldsymbol{W}_i^Q, \boldsymbol{K}\boldsymbol{W}_i^K, \boldsymbol{V}\boldsymbol{W}_i^V) \end{aligned} \tag{2.13}$$

where $\boldsymbol{Q}$ and $\boldsymbol{K}$ are projected to a dimension of d_k with the weight matrices $\boldsymbol{W}_i^Q, \boldsymbol{W}_i^K \in \mathbb{R}^{d_m \times d_k}$. On the other hand, $\boldsymbol{V}$ is projected to a dimension d_k using the weight matrix $\boldsymbol{W}_i^V \in \mathbb{R}^{d_m \times d_v}$. $\boldsymbol{W}^O \in \mathbb{R}^{h \cdot d_v \times d_m}$ transforms the concatenated output from all the heads into the correct dimensions, where h is the number of heads and d_m is the size of the embedding input to the network [17].

2.4.3 Positional Encoding

Unlike some other machine learning methods, attention is invariant to the order of the input tokens. This invariance is problematic since the order of the inputs is essential to the meaning in many applications. For example, there is a difference between "I have to read this book" and "I have this book to read." Therefore positional encoding is introduced to give the model a sense of order in the input. The encoding has the same dimension as the embedding - d_m. There are several types of positional encoding, but one of the most common is a sinusoidal positional encoding which uses sin, and cos functions with different frequencies. For embedding number $k \in \{1, \cdots, L\}$ and the dimension of the embedding $i \in \{1, \cdots, \frac{d_m}{2}\}$, the positional encoding is

$$\begin{aligned} PE(k, 2i) &= \sin\left(\frac{k}{n^{\frac{2i}{d_m}}}\right) \\ PE(k, 2i+1) &= \cos\left(\frac{k}{n^{\frac{2i}{d_m}}}\right) \end{aligned} \tag{2.14}$$

where n is a user-defined scalar, set to $10,000$ by the authors of *Atention is All You Need* [17].

2.4.4 Encoder-Decoder Structure

The transformer uses an encoder-decoder architecture, which is designed for sequence-to-sequence tasks such as machine translation or text generation but can also be used for image-specific tasks. This structure is visualized in Figure 2.6.

The encoder takes an input sequence, represented as a sequence of tokens, and processes it to create a contextualized representation. The encoder is composed of multiple layers, where each layer contains two sub-layers: a multi-head self-attention mechanism and a fully connected network. The multi-head self-attention mechanism within the encoder allows each position in the input sequence to attend to all other positions, capturing the dependencies and relationships within the sequence. After the self-attention mechanism, a fully connected network is applied. This introduces non-linearity and further refines the representations.

The decoder takes the contextualized representation generated by the encoder and generates the output sequence step by step. The decoder also consists of multiple layers, each containing three sub-layers: a masked multi-head self-attention mechanism, a Multi Headed Attention layer, and a fully connected network. The masked multi-head self-attention mechanism within the decoder allows each position to attend to inputs that appear before it in the sequence while masking out later inputs. This ensures that the model maintains auto-regressive behavior during training, generating one output token at a time based on the previously generated tokens.

To generate the output sequence, the decoder employs an autoregressive strategy. At each decoding step, it takes into account the previously generated tokens using the masked multi-headed self-attention. The decoder then attends to the encoder's representations and uses the context information along with the previous outputs to predict the next token in the sequence.

2.5 Imitation Learning (IL)

Imitation Learning is a deep learning approach where the agent is trained to imitate the actions of an expert agent. The expert agent is often humans but can also be other neural networks or rule-based approaches. The advantage of imitation over reinforcement learning is that it can be trained offline. An RL agent needs an environment to interact with, while an imitation learning agent is trained using a pre-recorded dataset. The goal of the training process is to learn a mapping from observations to actions.

The downside of Imitation Learning originates from the expert data itself. Since the expert agent rarely ends up in dangerous situations, the IL agent will be unable to learn how to recover from them. The RL agents are likely to end up in these situations through trial and error and therefore learn how to deal with them. Despite these issues, recent advances have shown that IL agents can perform well in highly complex scenarios such as autonomous driving. The four highest entries on the CARLA leaderboard are IL agents at the time of writing [8, 9, 22, 23].

2.6 Reinforcement Learning (RL)

Parts of this section are inspired by my project thesis [12]. Specifically State, Action, Reward and Policy Representations.

Reinforcement Learning is a set of algorithms designed to reward desired behaviors and punish negative behaviors. These algorithms assign positive rewards to the desired actions and negative rewards to undesired behaviors. The agent is encouraged to seek long-term gain and maximize total discounted future rewards (called utility) to achieve an optimal policy [24]. The agent's policy - $\pi(\boldsymbol{s}, \boldsymbol{a})$ - decides the agent's action during training and inference. Generally, a reinforcement learning agent can perceive and interpret its environment, take actions, and

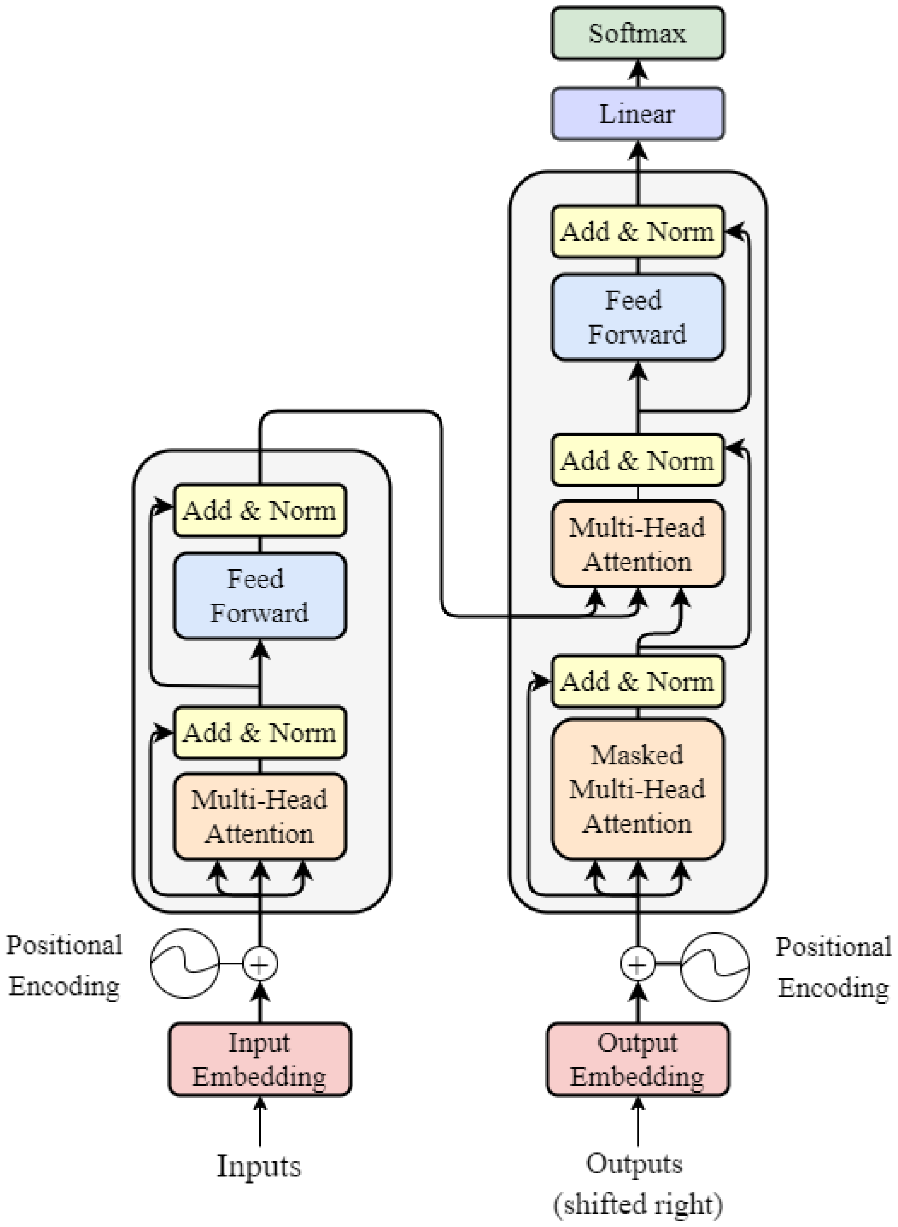


Figure 2.6: The encoder-decoder structure of the transformer. Courtesy of [17].

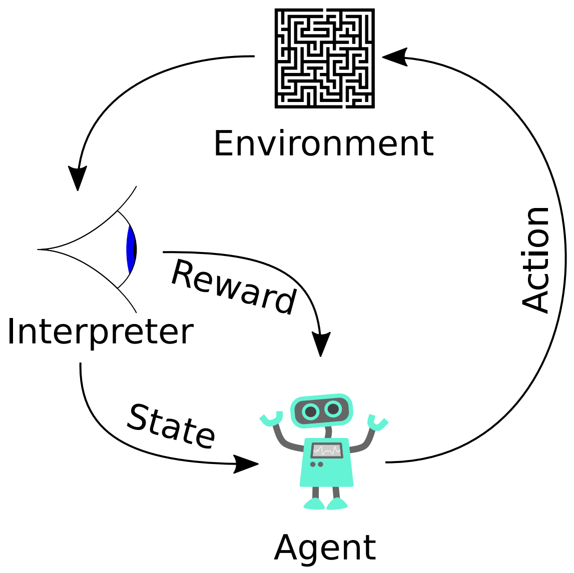


Figure 2.7: Reinforcement learning consists of an agent that perceives the world, chooses an action based on the observation, and then receives a reward based on the outcome of the action. Courtesy of [25] under the CC0 1.0 Universal License via Wikimedia Commons.

learn through trial and error. Reinforcement Learning differs from imitation learning, where the actor learns to act similarly to an expert agent. Imitation learning cannot perform better than the expert it imitates, but reinforcement learning can, in theory, exceed an expert's performance level. The agent's environment is often modeled as a Markov Decision Process (MDP) [24]. A MDP is a discrete-time stochastic control process. At each time step, the agent has to choose an action - $\boldsymbol{a} \in A$ - available in the current state - $\boldsymbol{s} \in S$. The world will then transition to a new state $\boldsymbol{s}^{'}$ with a probability given by a state transition function $P(\boldsymbol{s}, \boldsymbol{s}^{'})$. Finally, the agent receives a reward given by the reward function $R(\boldsymbol{s}, \boldsymbol{a})$ [24].

2.6.1 The State

The state represents the world, and the agent uses the state to decide what action to perform. The state can either be discrete or continuous and is represented by the variable $\boldsymbol{s} \in S_t$ where S_t is the set of all possible states at time step t [24]. There are multiple ways to parameterize the state, and the choice will impact the learning performance. For example, pong is a simple game where the goal is to score as many points as possible by forcing the ball to go past the opponent's paddle. A possible state representation is the position of the paddles and the velocity and position of the ball. Another representation for this game is to use all the pixel values directly. This would more closely resemble the way humans interact with the game but also create a much higher-dimensional and complex state space.

2.6.2 Actions

Actions are possible ways the agent can affect the environment. Like states, the actions can also be both discrete and continuous. The choice here depends on the chosen algorithm. For example, traditional Deep Q-Learning only supports discrete action spaces [24], while methods like PPO support continuous action spaces [26]. For the pong example, one choice of action space is move left, move right, stand still. This example is a discrete set of actions. Another option would be to have a continuous variable $\boldsymbol{a}_t \in [-1,1] = A_t$ representing how fast the paddle moves to the left or right.

2.6.3 Reward Function

As the central element of Reinforcement learning is to maximize future rewards, the design of a suitable reward function is an essential part of training a reinforcement learning agent. For the pong example, a reward function could be +1 for getting the ball past the opponent's paddle and -1 for the ball passing our paddle. Another option is also to reward each time the ball hits our paddle. Both of these reward functions are valid and would result in trainable agents but might result in different performances. As stated previously, the goal of RL is to maximize utility. The definition of utility is the estimated sum of discounted future rewards as shown in (2.15). Here, the discount factor $\gamma \in [0,1]$ is introduced because of the uncertainty of future rewards, while $R(\boldsymbol{s}) : S \longrightarrow \mathcal{R}$ is the reward function. The utility is associated with a policy - $U^{\pi}(\boldsymbol{s})$ - is the expected reward - $R(\boldsymbol{s})$ - for the current state - $\boldsymbol{s}$ - given a policy π [24]. This can be expressed mathematically as

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t) \mid S_t = \boldsymbol{s}\right]. \tag{2.15}$$

Another important quantity here is the action-value function $Q(\boldsymbol{s},\boldsymbol{a})$. It is similar to the utility function but gives the expected sum of discounted returns for a given state action pair $(\boldsymbol{s},\boldsymbol{a})$ [24], which, similarly to the utility, is expressed as

$$Q^{\pi}(\boldsymbol{s},\boldsymbol{a}) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t) \mid S_t = \boldsymbol{s}, A_t = \boldsymbol{a}\right] \tag{2.16}$$

where $\gamma \in [0,1]$ is the same discount factor as for the utility function.

2.6.4 Policy Representations

The policy is a mapping between states s and actions a. This mapping can either be deterministic or stochastic. Each state maps directly to action for a deterministic policy. In the wumpus world example in [24], each square would have an associated action $\pi(\boldsymbol{s}) = \boldsymbol{a}$. This policy is of the form $\pi : S \longrightarrow A$ and is an example of a deterministic policy - there is a direct mapping between states and actions. A stochastic policy - on the other hand - represents a probability distribution describing the probability that a certain action is optimal given the current state. The agent chooses an action by sampling this probability distribution. The policy function is then on the form $\pi : S \times A \longrightarrow [0, 1]$. Stochastic policies have some clear advantages over deterministic policies. When the environment is stochastic, a deterministic policy might fail since it chooses the same action every time. For example, in rock, paper, and scissors, the optimal policy is to choose randomly between the three actions. If someone always chooses rock, it is easy to create a strategy to beat them. Stochastic environments are common, so deterministic policies often fall flat. Another problem with stochastic policies appears when the world is not fully observable. In this case, stochastic policies can consider the uncertainty of inferring unknown states.

Table 2.2: Example of deterministic policy $\pi(\boldsymbol{s})$.

State	Action
S1	A2
S2	A1
S3	A3

Table 2.3: Example of stochastic policy $\pi(\boldsymbol{a}|\boldsymbol{s})$.

S\A	A1	A2	A3
S1	0.15	0.8	0.05
S2	0.3	0.1	0.6
S3	0.1	0.0	0.9

For discrete action spaces, the policy can be represented as a table. This would be a table with entries for each state for a deterministic policy. For a stochastic policy, it will be a table with a probability for each state-action pair. With continuous state and action spaces, this is not possible to do. The policy function is then approximated using a neural network - called a policy network - which is trained using gradient-based optimization techniques [24]. Examples of both stochastic and deterministic policy tables are shown in Table 2.3 and Table 2.2.

2.6.5 Policy Gradient Methods

There are many approaches to reinforcement learning. Most methods learn a state-action or utility function, like Temporal Difference (e.g., Q-learning) and Monte Carlo methods. Policy-Gradient methods differ from these by directly learning the policy using gradient-based optimization. The policy is represented with trainable parameters - $\boldsymbol{\theta}$ - and is depicted as $\pi_\theta(\boldsymbol{a} \mid \boldsymbol{s})$.

The policy gradient is computed based on the collected trajectories, which are pairs of states - $\boldsymbol{s}_t$ - the chosen action - $\boldsymbol{a}_t$ - the reward received - r_t- and the state the agent ended up in - $\boldsymbol{s}_{t+1}$. The gradient represents the direction and magnitude of adjustment to the policy's parameters that would maximize the utility.

Using the gradient derived for a given policy gradient algorithm, the parameters can be updated using any gradient-based optimization technique with a learning rate of α - which is a weighting term determining how far along the gradient direction each step should be.

2.6.6 Temporal Difference Learning

Temporal Difference learning is a reinforcement learning technique used to estimate the value function of a state or state-action pair, given the sequence of rewards and states. The basis of Temporal Difference is the Bellman equation

$$U^{\pi}(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi(s))U^{\pi}(s') \tag{2.17}$$

which describes the relationship between a function of a state and its neighboring states as shown in (2.17). It states that the utility of a state - $s \in S_t$ - is equal to the immediate reward obtained in that state - $R(s)$ - plus the discounted expected utility of the neighboring states - $\gamma * \sum P(s'|s,a)U(s')$ - weighted by the probability of ending up in that state given the chosen action, where $s' \in S_{t+1}$ is the neighboring states, which is the set of possible states for the next time step.

The utility function obeys the Bellman equation, and when a transition happens from state s to s', Temporal Difference learning updates the utility function using

$$U^{\pi}(s) \leftarrow U^{\pi}(s) + \alpha(R(s) + \gamma U^{\pi}(s') - U^{\pi}(s)). \tag{2.18}$$

TD learning works by calculating the Temporal Difference error, which is the difference between the actual utility of the current state - represented by $R(s) + \gamma U^{\pi}(s')$ - and the estimated value of the current state - $U^{\pi}(s)$. The TD error is the discrepancy between the predicted value and the observed reward, and it quantifies how much the current estimate needs to be updated.

An example of a TD learning method is Q-learning, where instead of learning the utility function, the Q function is learned. Since the Q function also obeys the Bellman equation, (2.18) can be updated to use the $Q(s,a)$ instead of $U(s)$ [24].

If a neural network represents the utility function, the weights of the network have to be updated using a gradient-based optimizer. The equation

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\left[R(s) + \gamma \hat{U}_{\boldsymbol{\theta}}(s') - \hat{U}_{\boldsymbol{\theta}}(s)\right] \frac{\partial \hat{U}_{\boldsymbol{\theta}}(s)}{\partial \boldsymbol{\theta}} \tag{2.19}$$

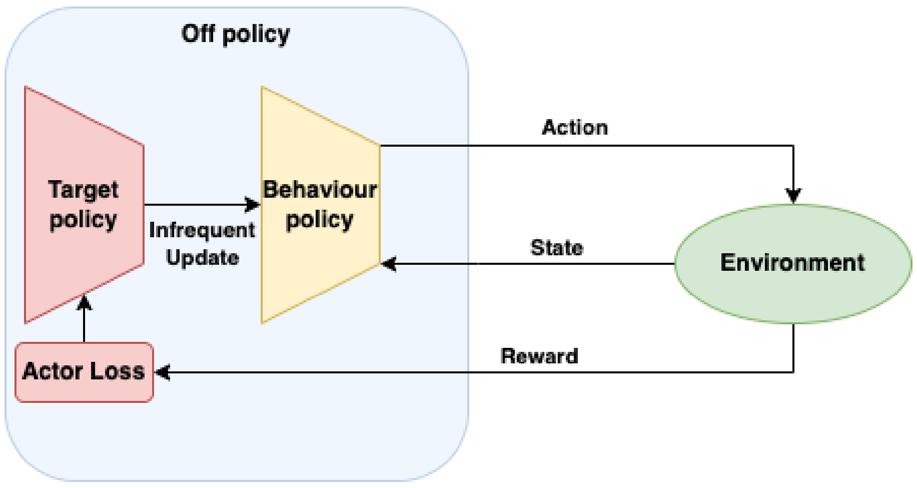
shows this when using regular gradient descent, but the gradient can also be used in other optimizers. The TD error is here used as a scaling factor for the gradient

2.6.7 Off-policy vs On-policy

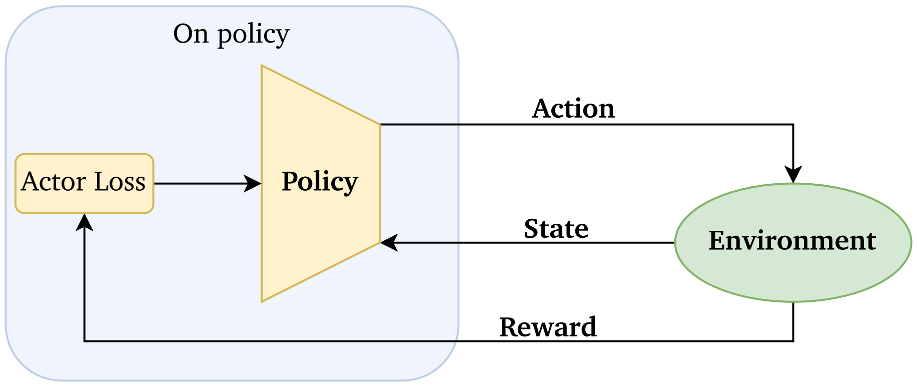
There are two kinds of policies - behavior and target policies. The agent uses the behavior policy to choose what action to perform, i.e., it is the policy it uses to interact with the environment. The target policy is the policy the actor is learning, i.e., it is the policy that is continuously updated during training. These two policies might be different, or they might be the same [24].

Off-policy algorithms in RL have different target and behavior policies. This can be advantageous when exploration is difficult or expensive, as the agent can learn from a pre-existing dataset of expert demonstrations or a different exploration strategy. The difference between the behavior policy and the target policy is often referred to as the *off-policy* nature of the algorithm. One of the challenges of off-policy algorithms is the *importance sampling* problem. When the behavior policy differs from the target policy, the data generated by the behavior policy has a different distribution than the data the target policy is interested in. Importance sampling is a technique used to adjust for this difference in distribution by weighting the updates based on the probability of generating the data under the target policy. Off-policy algorithms can be more sample-efficient than on-policy algorithms because they can learn from a wider range of data. Still, they require careful handling of importance sampling to ensure the updates are consistent with the target policy. Off-policy algorithms can also suffer from unstable updates because they rely on estimating the value of a different policy, leading to instability in the learning process [24]. A visual representation of off-policy algorithms can be found in Figure 2.8a.

On-policy algorithms in RL, however, have the same target and behavior policy. This means that the agent is actively exploring the environment while updating its policy, which can be



(a) Off policy



(b) On policy algorithm.

Figure 2.8: A visual representation of on-policy and off-policy algorithms

advantageous when the environment changes or the optimal policy is not well-defined. The difference between the behavior and target policies is minimal in on-policy algorithms, making them more stable but less sample-efficient than off-policy algorithms [24]. A visual representation of on-policy algorithms can be found in Figure 2.8b.

One of the challenges of on-policy algorithms is the "exploration-exploitation" trade-off. Since the agent actively explores the environment, it may get stuck in suboptimal policies and fail to converge to the optimal policy. On-policy algorithms often use exploration strategies such as epsilon-greedy to address this.

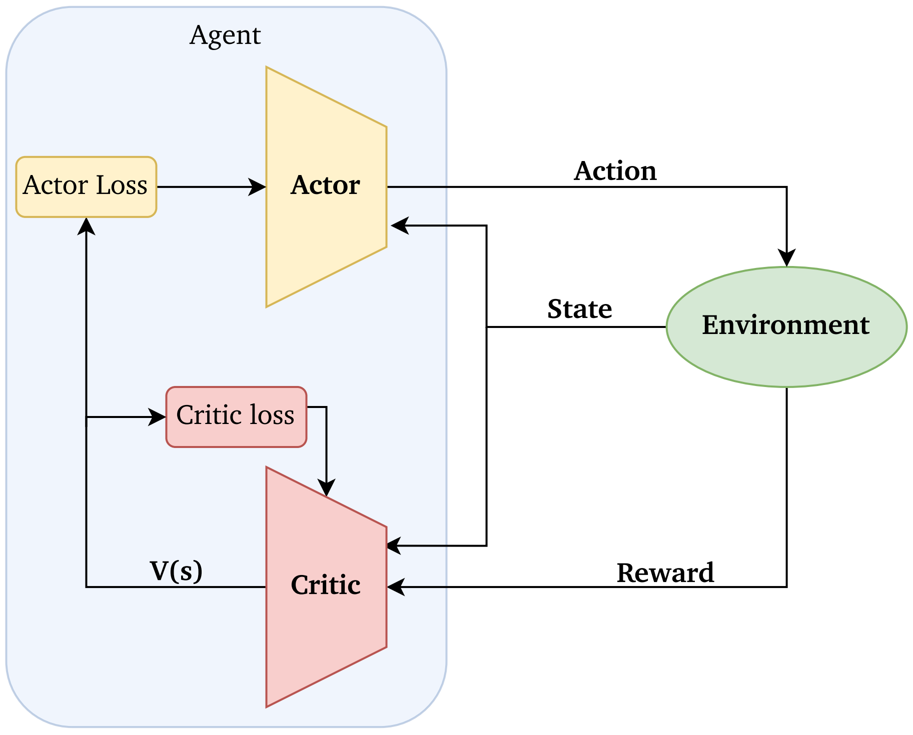


Figure 2.9: Actor-Critic architecture.

2.6.8 Actor-Critic (AC)

Actor-Critic is a Reinforcement Learning approach that uses two different neural networks; an actor and a critic network. It combines several traits and advantages of both policy-based and value-based methods. The actor network takes in the state and predicts the best action. This can either be a stochastic output as in PPO [26] or deterministic as in TD3 [27]. The critic network takes in the state and the chosen action by the critic and predicts the value function of the state-action pair - an estimate of the future discounted reward. During training, the actor and critic networks are updated using a combination of policy gradient and TD learning. The policy gradient is used to update the actor-network, while TD learning is used to update the critic network. Actor-Critic can be used in both on- and off-policy algorithms, for example, PPO and TD3, respectively. A visualization of the Actor-Critic architecture can be seen in Figure 2.9.

2.7 Information Theory and Statistics

Since the policy of an RL, algorithm can be stochastic, it can be useful to employ some statistical tools to evaluate the training progression. This can be particularly useful when the stochastic policy is used as a tool for managing Exploration-Exploitation. Statistics can also be useful when evaluating the quality of the policy or value network predictions. This thesis utilizes the Shannon entropy, Kullback-Leibler (KL) divergence, and explained variance, and these will be explained in more detail in this section.

2.7.1 Shannon Entropy

In information theory, the Shannon entropy is a measure of the amount of uncertainty or information in a probability distribution by taking into account the probabilities of all possible outcomes [24]. A distribution with higher entropy is more random, and it requires more information to describe a sample from the distribution. The Shannon entropy of a distribution - $H(X)$ - is given as the expected value of the information content - $I(X)$ - of a sample of a random variable. In other words, it is how much information you expect to get about the distribution given a random sample from it. The full expression is

$$H(X) = \mathbb{E}[I(X)] = \mathbb{E}[-\log_2 p(x)] = -\sum_{x \in X} p(x) \log_2 p(x) \tag{2.20}$$

for discrete random variables and for continuous distributions it is expressed as

$$H(X) = -\int_{-\infty}^{\infty} p(x) \log_2 (p(x))\, dx. \tag{2.21}$$

In Reinforcement Learning algorithms, entropy is often used to encourage exploration and prevent convergence to a suboptimal policy. In this case, entropy refers to the randomness or unpredictability of the policy, which can be measured using the Shannon entropy formula described earlier.

2.7.2 Kullback-Leibler (KL) Divergence

KL-divergence, also called relative entropy, is a measure of how different a probability distribution P is from distribution Q. Its equation is similar to that of Shannon entropy. The KL-divergence is the expected value of the logarithmic difference $\log\left(\frac{P(x)}{Q(x)}\right)$:

$$D_{KL}(P \parallel Q) = \int_{-\infty}^{\infty} P(x) \log_2 \left(\frac{P(x)}{Q(x)}\right) dx. \tag{2.22}$$

The KL divergence quantifies the extra information needed when using one distribution to approximate another. A low KL divergence means a good approximation, while a high KL divergence indicates a larger mismatch. It is commonly used in machine learning to optimize models and compare distributions.

2.7.3 Fraction of Explained Variance

The fraction of explained variance - also known as the coefficient of determination or R-squared - measures the proportion of the variance in some estimated values from a mathematical model explained by the actual dataset.

For a dataset with n values ,$\boldsymbol{y} = [y_1, \cdots, y_n]$, each with an estimate made by a mathematical model , $\hat{\boldsymbol{y}} = [\hat{y}_1, \cdots, \hat{y}_n]$, the fraction of explained variance is

$$R^2 = 1 - \frac{\operatorname{Var}[\boldsymbol{y} - \hat{\boldsymbol{y}}]}{\operatorname{Var}[\boldsymbol{y}]}. \tag{2.23}$$

The value of the numerator of (2.23) is the mean-squared error of the estimate, while the denominator is the variance of the actual values. A value close to 1 means that the model predictions are accurate, while a large negative value indicates that the model is worse than just predicting zero.

2.8 Transfuser

This section is taken from the equivalent section in my project thesis [12].

Consider a scenario where the ego vehicle is about to enter an intersection. To safely navigate the intersection, the ego vehicle needs to capture the global context of the 3d scene and model the dependencies between the traffic lights on the right and the vehicles on the left. The ego vehicle perceives the environment through different sensors, with cameras and lidar being the most prevalent. A camera can provide dense semantic information about the scene, but it lacks reliable 3d information and is highly sensitive to weather variations. On the other hand, LiDAR consists of 3d information, but measurements are typically sparse and do not contain important information such as traffic-light states; hence image-only and lidar-only methods are likely to fail in complex scenarios.

One way to mitigate this is to use fusion-based approaches, which aim to combine the high-density information from the cameras with the 3d information from the lidars. Prior literature

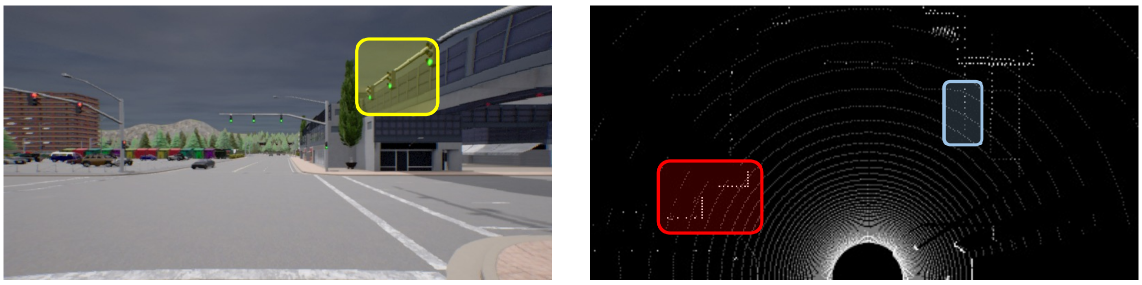


Figure 2.10: Image and LiDAR BEV from intersection.

on multi-modal sensor fusion has primarily focused on using geometric feature projections from 3d spaced to image space and vice versa. Chitta et al. observed that geometric fusion underperforms in complex traffic scenarios [8]. They hypothesize that this happens due to the lack of global context. Feature aggregation happens between local regions in the 2d-image space and 3d-lidar space. However, to fully understand the context of a traffic scenario, it is essential to look at the local features in the global setting of the scene. Chitta et al. exemplifies this with a scenario from an intersection. In the illustration shown here, geometric fusion would aggregate features from the yellow and blue areas since the blue region in 3D space projects to the yellow region in the image. It is essential to consider features from the vehicles on the left side to navigate the intersection safely. The red area in the lidar data should therefore be considered alongside the yellow and blue areas.

2.8.1 Attention-Based Fusion

Transfuser's key idea is to use attention-based feature fusion to capture the global context of the 3d scene. Transfuser uses transformers for this purpose. An RGB image and LiDAR Bird's eye view (BEV) projection are the inputs to the model. The inputs pass through two branches of one convolutional backbone interconnected using transformers. After each stage of the backbones, the features from each branch are downsampled. The resulting intermediate features have dimensions $22 \times 5 \times C$ for the image branch and $8 \times 8 \times C$ for the BEV branch. The intermediate features are divided into one C-dimensional token per spatial coordinate - $\boldsymbol{F}_{in}$ -, and positional encoding is added. The output features of the model are computed as a non-linear transformation

$$\boldsymbol{F}_{out} = \mathrm{MLP}(\boldsymbol{A}) + \boldsymbol{F}_{in} \tag{2.24}$$

where $\boldsymbol{A}$ is the result of a series of L multiheaded-attention layers, as described in Section 2.4.

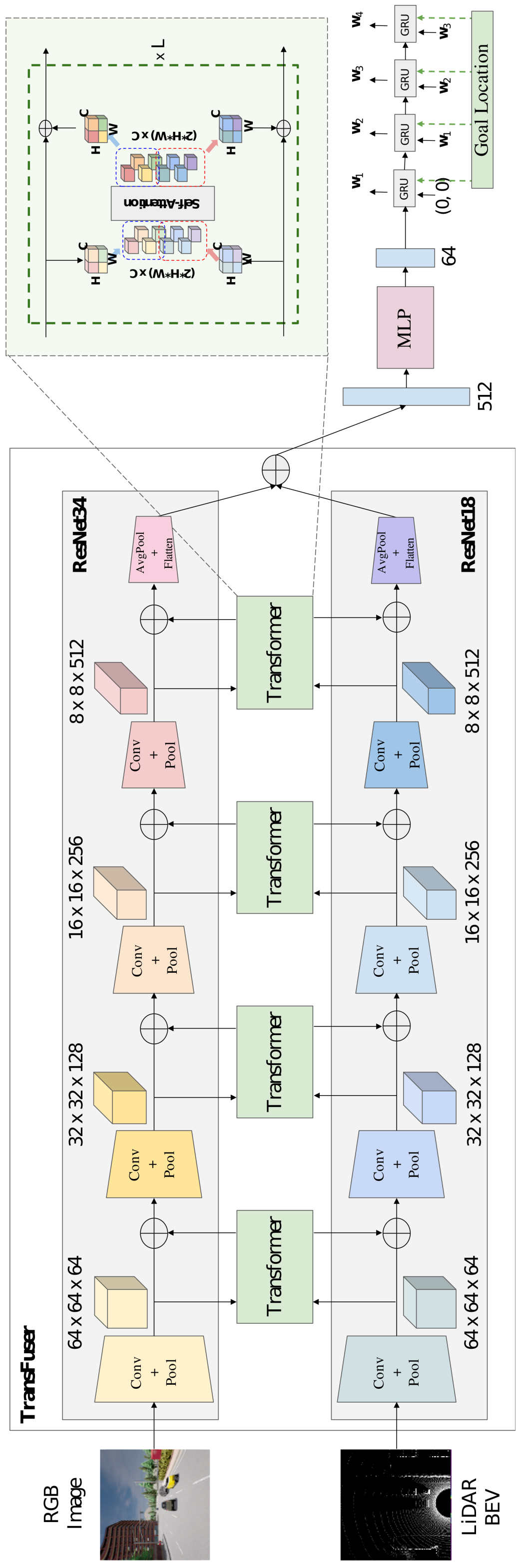


Figure 2.11: Transfuser model. Courtesy of [8].

The fused output - $\boldsymbol{F}_{out}$ - is split into the image and lidar features and returned to the individual branches. The resulting network gives a fusion of features at multiple scales throughout the network. After the final fusion stage, the features from both branches are flattened and reduced to a 512-dimensional feature vector via one fully connected layer. The resulting feature vectors are combined via element-wise addition and are then passed to the waypoint prediction network.

2.8.2 Waypoint Prediction

The waypoints are predicted using an autoregressive GRU network. The feature vector is first reduced to a dimension of 64 using an MLP. Then, this reduced representation is used as the initial hidden state of the GRU network. The inputs are the GPS position of the goal location as well as the current position of the ego. Next, the output from each GRU block is passed through a linear layer that predicts the differential waypoints. The absolute waypoints are then found by cumulative summation. Finally, the output of one block is used as the new hidden state to predict the next waypoint. In total, T=4 waypoints are predicted at each timestep. This gives the waypoints

$$\boldsymbol{w}_t = \boldsymbol{w}_{t-1} + \boldsymbol{\delta w}_t \tag{2.25}$$

where w_t is the t^{th} waypoint and δw_t is the t^{th} waypoint differential from the GRU network.

The loss of the waypoints is computed as an L1 loss between the predicted waypoints and the ground-truth waypoints -

$$\mathcal{L}_w = \sum_{t=1}^{T} \|\boldsymbol{w}_t - \boldsymbol{w}_t^{gt}\|_1. \tag{2.26}$$

2.8.3 Auxiliary Tasks

In addition to the waypoint loss, the training process includes several additional tasks. These include semantic segmentation, image depth estimation, and object detection in the LiDAR BEV. This has been shown to give a more robust model in complex temporal and spacial scene structures [8]. The auxiliary tasks add extra entries to the total loss function. For example, the semantic segmentation task generates semantic masks with seven classes. It is supervised with a cross-entropy loss function $\mathcal{L}_{seg}$, while the depth estimation uses an L1 loss - $\mathcal{L}_d$. The classes

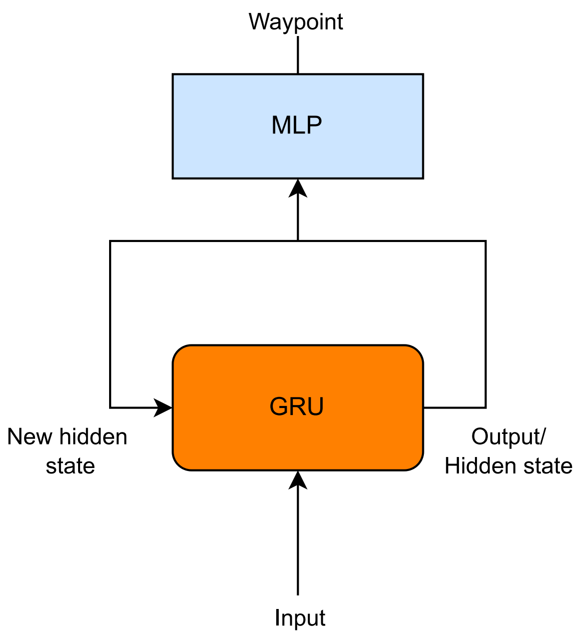


Figure 2.12: Waypoint prediction network in Transfuser.

predicted by the segmentation decoder are 1) unlabeled, 2) vehicle, 3) road, 4) red light, 5) pedestrian, 6) lane markings, and 7) sidewalk. The BEV semantic segmentation generates a downsampled version of the HD map with three classes - road, lane markings, and other - and uses a cross-entropy loss $\mathcal{L}_{map}$. Finally, object detection of other vehicles in the BEV features is done with a CenterNet decoder. The network predicts a position map $\hat{\boldsymbol{P}} \in [0,1]^{64\times64}$ and an Orientation map $\hat{\boldsymbol{O}} \in [0,1]^{64\times64\times12}$. It also generates a regression map $\hat{\boldsymbol{R}} \in [0,1]^{64\times64\times5}$ with a vehicle size estimate, an orientation offset, and a position offset. They use a focal, cross-entropy, and F1 loss, respectively.

2.8.4 Control

The predicted waypoints are fed to PID controllers that output steering, throttle, and brake values. A reoccurring problem for imitation learning agents is the Inertia problem, where the agents get stuck for prolonged periods. To combat this, the Transfuser implements a creeping when no movement has happened for 55 seconds. However, creeping can lead to dangerous situations. Therefore, a safety check overrides the creeping mechanism by checking if any LiDAR detections are close to the front of the car.

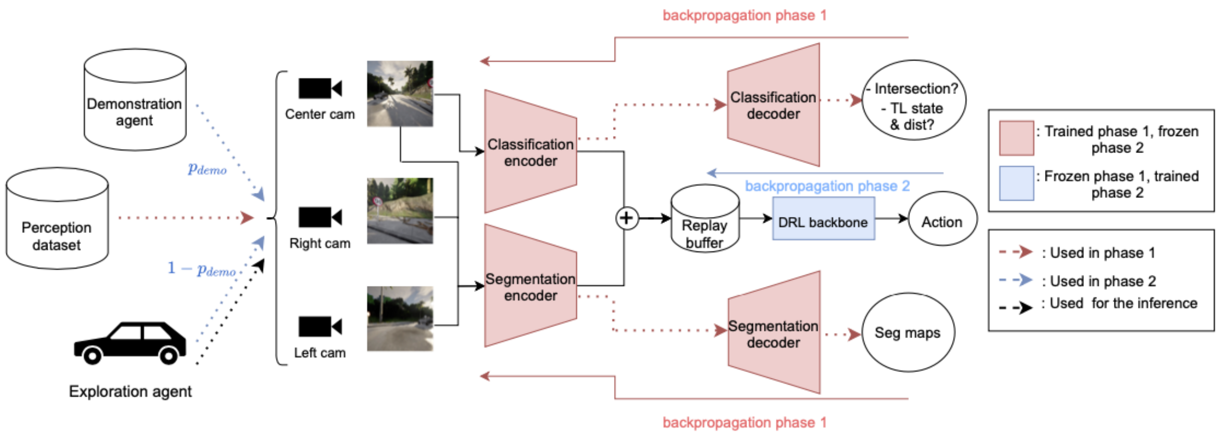


Figure 2.13: Griad architecture and training stages [7].

2.9 General Reinforced Imitation for Autonomous Driving (GRIAD)

Chekorun et al. introduced GRIAD in 2021, with top placement on the CARLA leaderboard [10]. At the time of writing, it is still the highest-ranked entry utilizing RL on the leaderboard. GRIAD has the aim of utilizing demonstrations from an expert dataset to improve the sample efficiency of the system while maintaining the exploration element that separates RL from IL. Another aim was to create a flexible method that can easily be used on other off-policy algorithms. [7]

To accomplish this, Chekorun et al. make the assumption that the demonstrations are perfect actions following an optimal policy. The transitions in the expert dataset can therefore be given a constant high reward and can then be inserted directly into the replay buffer. This is done with a probability of $p_{demo} = 0.25$, while the remaining entries are exploration transitions. Training is performed in two stages. The first stage pre-trains two visual encoders on segmentation and classification tasks. This encoder is then used in stage two to create a latent representation of the agents' surroundings, which is used as the world state for training the RL model. The second phase of the GRIAD training process involved the use of $200,000$ expert transitions and was trained for approximately 60 million steps. The training comprised 45 million exploration steps and 15 million samples taken from the expert transition dataset.

GRIAD is also notable for being the best-performing camera-only entry on the leaderboard and for having a relatively simple encoder compared to later entries like Transfuser and Interfuser. While Transfuser uses several transformers to fuse the information from separate CNNs, GRIAD simply concatenates the output from each CNN.

Algorithm 1 GRIAD training stage 2.

```
Require: $r_{demo}$ demonstration reward value
Require: $p_{demo}$ probability to use demonstration agent
  Initialize empty buffer $\mathcal{B}$
  while not converged do
    if len($\mathcal{B}$) > min_buffer then
      Do a DRL network update
    end if
    if random.random() > $p_{demo}$ then
      Collect episode ($s_t^{online}$, $a_t$, $r_t$, $s_{t+1}^{online}$ ) in buffer $\mathcal{B}$ with exploration agent
    else
      Add episode ($s_t^{offline}$, $a_t$, $r_t$, $s_{t+1}^{offline}$ ) in buffer $\mathcal{B}$ with demonstration agent
    end if
  end while
```

Chapter 3

Software and Algorithms

This chapter presents the design of all components needed for training an RL agent, as well as the setup required for interacting with the simulator. The design of the observation and action spaces will be put forth, as well as the design of the reward function. Some focus will also be put on the hardware and software tools that were used throughout the project.

3.1 Computational Resources

Both the CARLA simulator and training of the machine learning algorithms require heavy computational resources, more specifically GPUs. Several computational resources have been available during the completion of this project - Idun, NAP02, and VCXR12 - all of which are described in this section. The focus is mainly on the hardware specifications, but some tools for scheduling jobs are mentioned - specifically for Idun.

3.1.1 Idun

The Idun cluster is a High Performance Computing (HPC) platform designed to provide a high-availability and professionally administrated computing platform for the Norwegian University of Science and Technology (NTNU). The project was initiated as a collaboration between the IT division and various faculties with the aim of combining the computing resources of individual shareholders to create a cluster for rapid testing and prototyping of HPC software [28].

The cluster has a diverse range of compute nodes, including Intel Xeon CPUs with varying numbers of cores and RAM, as well as NVIDIA P100, V100, or A100 GPUs. The high-speed interconnect network with Mellanox passive FDR, EDR, and HDR switches for interconnect/storage on different nodes ensures that data transfer is efficient and fast [28].

The Slurm Workload Manager is used to manage the provided resources and to schedule jobs on the resources. Users can submit jobs to the cluster using Slurm job scripts, which specify the resources required for the job, such as the number of compute nodes, CPUs, GPUs, and memory. Users can also test their scripts and jobs on the "short" partition, which has servers with P100 GPUs, before submitting their jobs to the main cluster. With a maximum wall time of 7 days or 167 hours, the Idun cluster provides a reliable and flexible platform for machine learning research and other HPC workloads [28].

Table 3.1: Hardware specifications of all computing resources available during the project.

Hardware	CPU Cores	GPU	Memory	VRAM
IDUN	Up to 128	P100, V100 or A100	Up to 2 TB	Up to 80 GB
NAP02	96	2 x A100	512 GB	2 x 89 GB
VCXR12	32	RTX 4090	32 GB	24 GB

3.1.2 NAP02

NAP02 is a compute server primarily used by the NAPLab for large-scale machine learning training. It has two A100 data center GPUs, each boasting 84 GB of VRAM. One of its main benefits over the IDUN cluster is that it is exclusively available to NAPLabresearchers, so jobs rarely experience significant queueing times. Its powerful hardware and ample memory make it an excellent resource for handling large datasets and complex computations. Despite this, the A100 GPU is designed for machine learning and is, therefore, not optimal for rendering graphics. The simulator will therefore run significantly slower on NAP02 compared to a PC with an RTX-class GPU.

3.1.3 VCXR12

VCXR12 is a computer owned by NAPLab equipped with an NVIDIA RTX4090 GPU. Although this GPU is slower than the dataset GPUs of NAP02 and Idun for machine learning computations and has less VRAM, it is specifically designed for rendering video games. As a result, it offers faster rendering performance and can potentially lead to faster training times. The reason for this is that rendering is often the largest bottleneck in the training pipeline. Training on the RTX4090 can be faster than an A100, depending on the number of environments run in parallel. It is important to note, however, that the efficacy of using an RTX4090 for training will depend on the specific use case and the particular requirements of the task at hand. While the A100 has a higher theoretical peak performance than the RTX4090, it may not always be the best choice for training. This highlights the importance of selecting hardware based on the specific needs of the task rather than relying solely on peak performance metrics.

3.2 Libraries and Tools

The choice of RL framework and driving simulator is the first thing that should be established since the choice will impact all future decisions. There are several good options for RL framework, but one, in particular, stands out - Stable Baselines 3 (SB3). NAP Lab does not have access to NVIDIA drive sim, so there is really one main choice for the simulator - CARLA.

3.2.1 CARLA Simulator

The CARLA simulator is an open-source urban driving simulator with the primary goal of democratizing research into autonomous driving. CARLA provides a flexible and easily customizable platform that users can adapt to their needs. It consists of a client-server architecture, where the server performs the rendering and runs the simulation while the client is responsible for the agent logic. An accompanying Python API performs the communication between the client and the server. [29]

Agents in the simulator observe the world through the use of sensors. The sensors can be divided into two groups - exteroceptive sensors and interoceptive sensors - which again can be divided into raw and ground truth sensors.

Exteroceptive sensors:

- **LiDAR**: Provides a distance measurement to points surrounding the sensor. The number of channels and rotational frequency are customizable.
- **RGB camera**: Provides an RGB image of the surroundings. The user can customize most camera parameters, most importantly, the FoV and the image dimensions.
- **Radar**: Provides information about objects' positions and movement.
- **Semantic LiDAR**: Classifies each point in the pointcloud into one of 23 classes.
- **Semantic Camera**: Classifies each pixel of the image into one of 23 classes.
- **Depth camera**: Provide an image where each pixel represents the depth of the scene.

Interoceptive sensors:

- **Lane invasion detector**: Signals if the agent has crossed line markings, and what markings were crossed.
- **GNSS**: Most commonly known as GPS. Provides positional measurements such as latitude, longitude, and height measurements.
- **Collision sensor**: Signals if the agent has collided with an object, and what type of object it is.
- **IMU**: A combination of an accelerometer, a gyroscope, and a compass. Gives accelerations, rotational rates, and heading.

Benchmarking and the CARLA Leaderboard The CARLA leaderboard is a collaboration between the CARLA team and alpha drive - a cloud-based testing and validation platform for autonomous driving - that provides standardized and reproducible benchmarking for agents in the simulator. The CARLA scenario-runner provides the ability to define routes the agent should follow and scenarios it will be exposed to on the route. A route consists of waypoints with corresponding high-level commands. High-level commands include what path to take in an intersection and when to change lanes on the highway. Scenarios are events the agent has no control over, like a pedestrian crossing the road or a car driving at a red light. During the evaluation, the agent will be subjected to multiple instances of 10 scenarios on multiple routes. Teams who submit their agents for evaluation on the leaderboard will not have access to the routes and scenarios to eliminate bias. Finally, the team receives a driving score based on the agent's performance. The score breaks down into an infraction score and a route completion score.

Route completion (RC) The route completion is a percentage of the route distance completed - R_i - averaged across N driven routes;

$$RC = \frac{1}{N}\sum_{i}^{N} R_i. \tag{3.1}$$

If the agent drives across the lanes for parts of the route, the score is reduced by a multiplier.

The infraction score (IS) is a geometric series of infraction penalty coefficients - p^j - for every instance of infraction j. Agents start with an ideal score of 1.0. A penalty coefficient reduces the score for every infraction. This is given as

$$IS = \prod_{j}^{\text{Ped, Veh, Stat, Red , stop}} (p^j)^{\#\text{ infraction}^j} \tag{3.2}$$

where the penalty conefficients are:

- Collision with static layout = 0.65
- Collision with pedestrian = 0.5
- Collision with vehicle = 0.6
- Red light violation = 0.7
- Stop sign violation = 0.8

The driving score (DS) is the average route completion - R_i - weighted by the infraction score IS_i across all the routes. The score is computed by

$$DS = \frac{1}{N} \sum_{i}^{N} R_i IS_i \tag{3.3}$$

where R_i and IS_I is the route completion and infraction score for route i.

3.2.2 Stable Baselines 3 (SB3)

Stable Baselines 3 is an open-source library for reinforcement learning that provides a collection of state-of-the-art RL algorithms that are efficient and easy to use. The library is implemented in PyTorch, a popular deep learning framework, and is designed to be compatible with other popular machine learning frameworks like TensorFlow. Stable Baselines 3 provides a straightforward and easy-to-use API, which makes it easy to implement, train, and evaluate RL algorithms. The library is a fork of OpenAIs Baselines, aiming to be more flexible and user-friendly while keeping high performance [30].

One of the main advantages of using Stable Baselines 3 is its efficiency. The library provides a set of tools to speed up the training process, such as parallelized environments, which can lead to significant time savings. The library also supports GPU acceleration, further speeding up the training process for larger and more complex models. This can be particularly important for RL applications in domains such as robotics and autonomous driving, where real-time performance is critical. Additionally, Stable Baselines 3 provides a set of advanced features that can be particularly useful for complex RL applications, such as support for distributed training [30].

Another advantage of using Stable Baselines 3 is its flexibility. The library is designed to be modular and extendable, which makes it easy to customize the learning process to the specific application requirements. Stable Baselines 3 provides a set of advanced features that can be used to tailor the learning process, such as custom observation and action spaces and support for multiple agents and environments [30].

3.2.3 OpenAI Gym Interface

Like most other RL libraries, SB3 uses the OpenAI Gym interface for their environments [30]. Carla does not natively support this. However, some environment wrappers have been created that are open-source. The issue with these is that they do not support the use of CARLAs scenario runner, which allows the user to expose the vehicle to several traffic scenarios. To address this issue, an extension to scenario runner was created that utilizes the OpenAI gym interface.

The issue with scenario runner is the order in which steps are performed. It runs a loop, where it sends new observations to a carla Agent class - which computes the next action - and then performs a step of the simulation. This looping prevents an external policy from giving commands to the agent. The OpenAI gym interface requires that the environment takes in an action, performs a step, and then returns the next observation, along with the rewards.

The created extension allows for the stepwise execution of a scenario by decoupling the steps performed in the loop. It also allows for external actions to be passed to the runner by rearranging the order of steps the scenario runner performs. This makes it compatible with the majority of Reinforcement Learning libraries.

3.3 Sensor Setup

The CARLA simulator is built upon Unreal Engine 4, which uses a left-handed coordinate system with the z-axis pointing upwards, as shown in Figure 3.1.

The sensors used are the same as in Transfuser [8]. To make the comparison to Transfuser as close as possible, the location of the sensors is also the same. Three cameras are used, oriented with a yaw of -60, 0, and 60 degrees in relation to the car. They are placed on the front windscreen at a height of 2.3m above the ground and 1.3m ahead of the center of the vehicle. The lidar is placed directly above the cameras at a height of 2.5m. The lidar is also rotated with a yaw of -90 degrees to align its coordinate system with the car.

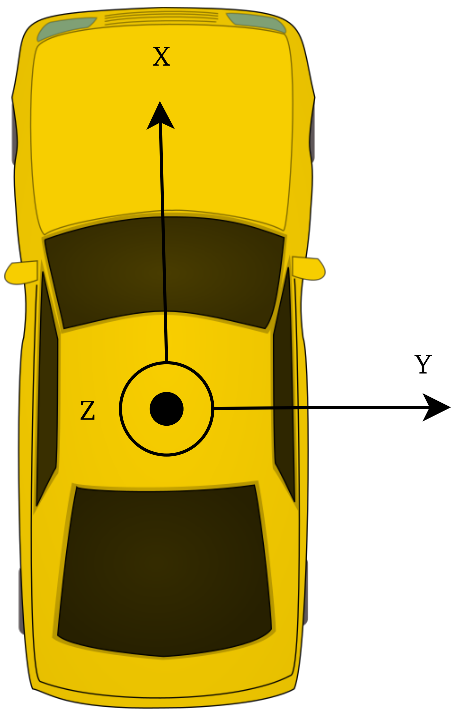


Figure 3.1: The z-up left-handed coordinate system of the CARLA vehicles.

Sensor	x [m]	y [m]	z [m]	yaw [deg]
Lidar	1.3	0	2.5	-90
Camera 1	1.3	0	2.3	60
Camera 2	1.3	0	2.3	0
Camera 3	1.3	0	2.3	-60

Table 3.2: Position of exteroceptive sensors on the vehicle.

3.4 Choice of RL Algorithms

To conduct a valid comparison between different setups, it is useful that they are all evaluated using the same RL algorithm. This avoids any potential biases that may arise from using different algorithms and enables a clear and meaningful assessment of the relative strengths and weaknesses of each setup. However, some expert demonstration approaches require the use of off-policy algorithms. This is because off-policy algorithms can learn from trajectories generated by a different policy than the one being evaluated, which is often necessary when using expert demonstrations. On the other hand, on-policy algorithms can be more stable and easier to train in complex environments, making them a good choice for autonomous driving.

To balance these considerations, one off-policy and one on-policy algorithm have been chosen for the experiments in this thesis. This approach will enable comparison of the performance of different methods consistently and fairly while also considering the specific requirements of expert demonstration techniques and the complexity of the environments being studied.

3.4.1 Proximal Policy Optimization (PPO)

Algorithm 2 PPO

```
Initialize policy network $\pi_\theta$ with random weights $\boldsymbol{\theta}$
Initialize value network $V_\phi$ with random weights $\boldsymbol{\phi}$
for each iteration $k$ do
    Collect trajectories $\mathcal{D}_k = \{\boldsymbol{s}_i, \boldsymbol{a}_i, r_i, \boldsymbol{s}_{i+t}\}$ using policy $\pi_{\theta_k}$
    Compute the reward-to-go $\hat{R}_t$
    Compute the advantage estimates $\hat{A}_t$ using the current value function $V_{\phi_k}$
    Compute the policy loss using the PPO-Clip objective $\mathcal{L}_{\pi_\theta}$
    Compute the value loss $\mathcal{L}_{V_\theta}$
    update $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$ using total loss $\mathcal{L} = \mathcal{L}_{V_\theta} + \mathcal{L}_{\pi_\theta}$
end for
```

Proximal Policy Optimization (PPO) is a reinforcement learning algorithm that aims to optimize the policy of an agent in a way that balances exploration and exploitation. PPO is a model-free, on-policy AC algorithm that can handle continuous and discrete action spaces. It utilizes a clipped surrogate objective function that prevents the new policy from deviating too far from the old policy during each update. This makes PPO more stable than other policy gradient methods [26]. This objective function is

$$\mathcal{L}_{\pi_\theta} = \frac{1}{N}\sum_{i=0}^{N} \min(r(\pi)A_i, \text{clip}(r(\pi), 1-\epsilon, 1+\epsilon)A_i) \tag{3.4}$$

where

$$r(\pi) = \frac{\pi_\theta(\boldsymbol{a}|\boldsymbol{s})}{\pi_{\theta_k}(\boldsymbol{a}|\boldsymbol{s})} \tag{3.5}$$

is the policy ratio, A_i is the advantage estimate, and ϵ is the clipping parameter. The policy ratio is the ratio of the probability of selecting an action under the new policy to the probability of selecting the same action under the old policy.

PPO also includes a value function that estimates the expected return of the agent, which is used to compute the advantage function and further improve the policy update [31]. This is updated using an MSE loss between the predicted and actual value:

$$\mathcal{L}_{V_\phi} = MSE(V_\phi(\boldsymbol{s}_t), \hat{R}_t) \tag{3.6}$$

where V_ϕ is the value-networkm $\boldsymbol{s}_t$ is the state, and R_t is the reward received.

PPO has gained popularity due to its simplicity, effectiveness, and ability to scale to large-scale problems. PPO has achieved state-of-the-art results in various challenging environments, such as Atari games, robotic control tasks, and continuous control tasks. Moreover, PPO has been extended and modified in various ways, such as incorporating parallelism, importance sampling, and adaptive learning rates, to improve its performance and stability further [31].

PPO is a practical choice of on-policy algorithm for this thesis, given its simplicity, stable policy updates, and ease of use. OpenAI has also chosen PPO as their method for solving many challenging RL problems, highlighting the practical relevance of PPO [31]. Therefore, PPO is a robust and versatile choice and a well-established and widely used approach.

3.4.2 Twin Delayed DDPG (TD3)

Twin Delayed DDPG (TD3) is a state-of-the-art algorithm for Reinforcement Learning that addresses some of the limitations of the Deep Deterministic Policy Gradient (DDPG) method. TD3 builds on the actor-critic architecture, where two deep neural networks are used to represent the policy and the value function. The policy network, also known as the actor, maps the state of the environment to an action. The value network, or the critic, estimates the value of the current state-action pair.

TD3 introduces several innovations that improve the performance and stability of the learning process. One of the key contributions is the use of twin critics, which estimate the value function independently and reduce the overestimation bias. In addition, TD3 employs delayed policy updates, which improve the stability of the learning process by updating the policy network less frequently than the value network.

TD3 is a great choice for an off-policy algorithm, primarily due to its stability compared to other off-policy algorithms. It addresses the problem of overestimation of action values, resulting in more reliable and accurate learning. The twin network structure reduces the variance in value estimates, leading to more consistent training. Moreover, the delayed updates for the actor and critic networks help to reduce the variance of the value estimates, which leads to a smoother gradient and faster learning speed. This enhanced stability ensures that TD3 can consistently learn optimal policies even in complex and challenging RL environments.

Algorithm 3 TD3

```
Initialize actor network $\pi_\theta$ and critic networks $Q_{\phi_1}, Q_{\phi_2}$
Initialize target networks $\pi_{\theta'}, Q_{\phi'_1}, Q_{\phi'_2}$
Initialize replay buffer $\mathcal{B}$ with capacity $N$
for $episode = 1$ to $M$ do
    Reset environment to initial state $\boldsymbol{s}_1$
    for $j = 1$ to $T$ do
        Sample action $\boldsymbol{a}_t = \pi_\theta(\boldsymbol{s}_t) + \epsilon_t$, where $\epsilon_t \sim \mathcal{N}(0, \sigma)$
        Store transition $(\boldsymbol{s}_t, \boldsymbol{a}_t, r_t, \boldsymbol{s}_{t+1})$ in replay buffer $\mathcal{B}$
        if time to update then
            for $i = 1$ to $N_{\text{updates}}$ do
                Sample transitions $(\boldsymbol{s}, \boldsymbol{a}, r, \boldsymbol{s}') \sim \mathcal{B}$
                Compute target actions $a' = \pi_{\theta'}(s') + \epsilon'$, where $\epsilon' \sim \mathcal{N}(0, \sigma')$
                Compute target values $y = r + \gamma \min_{i=1,2} Q_{\phi'_i}(\boldsymbol{s}', \boldsymbol{a}')$
                Update critic networks using mean squared Bellman error
                if $j$ mod $poilcy\_delay = 0$ then
                    Update actor network using policy gradient:
                    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_\theta \nabla_{\boldsymbol{\theta}} J(f_\theta)$, where
                    $J(\pi_\theta) = -\frac{1}{|B|} \sum_{s \in B} Q_{\phi_1}(s, \pi_\theta(s))$
                    Update target networks:
                    $\boldsymbol{\theta}' \leftarrow \rho \boldsymbol{\theta} + (1 - \rho) \boldsymbol{\theta}'$
                    $\phi'_i \leftarrow \rho \boldsymbol{\phi}_i + (1 - \rho) \boldsymbol{\phi}'_i$
                end if
            end for
        end if
        $s_t \leftarrow s_{t+1}$
    end for
end for
```

3.5 Base Algorithm

The main components of a reinforcement learning system are the observation space, action space, and reward function. These are some of the most important choices for the performance of the agent. The architecture of the actor and critic networks is also important. To make comparisons between different methods viable, these should preferably be the same across all experiments. This section presents the choices that went into designing these components.

3.5.1 Vision Encooder

In autonomous driving, the perception of the surrounding environment is vital. This is obtained through visual inputs, such as camera images and LiDAR. RL algorithms do not easily interpret raw image data. The sample inefficiency of RL training would make training an RL algorithm directly on the visual input a bad choice. A low-dimensional representation, therefore, needs to be extracted from the raw sensor data, which can then be used in the observation for the RL policy. To address this issue, a vision encoder can convert the raw image data into a lower-dimensional representation that is more easily interpretable by the RL algorithm. Additionally, the lower-dimensional representation obtained from the vision encoder can help reduce noise and irrelevant input information, making the learning process more effective.

Several factors need to be considered when choosing a visual encoder. Sensor choice is one of the most significant factors. The simplest choice for a vision encoder would be a classic CNN architecture, which was the choice in GRIAD [7]. A simple architecture would result in a low runtime for the system, which would be important when deployed in the real world. The disadvantage of a plain CNN is that there are few good ways to introduce lidar data into the mix. The resulting latent representation would therefore contain more uncertain depth information. CNNs has also been used before in RL approaches like GRIAD [7].

One alternative approach is to leverage encoders from SotA systems on the CARLA leaderboard. Specifically, two encoders stand out: Transfuser and Interfuser. Both utilize transformers to fuse lidar and camera data. Transfuser employs multiple transformer blocks to fuse data at multiple scales, whereas Interfuser fuses the outputs of separate CNN encoders for each image and LiDAR scan. While Interfuser delivers better performance on the CARLA leaderboard and has a lower runtime - having almost a third of the runtime compared to Transfuser. Despite this, the latter was published earlier, and I have greater familiarity with its codebase. Consequently, I opted for Transfuser despite its lower performance. The encoder can then be seen in Figure 3.2, giving an output vector of size 512.

3.5.2 Observation Space

Choosing the appropriate observation space is crucial in RL as it directly impacts the agent's ability to learn and generalize its behavior. The observation space determines what information the agent can perceive about the environment and make decisions based on. If the observation space is too narrow, the agent may fail to capture important features of the environment, resulting in suboptimal performance. On the other hand, if the observation space is too broad, the agent may struggle to identify relevant information and become overwhelmed with irrel-

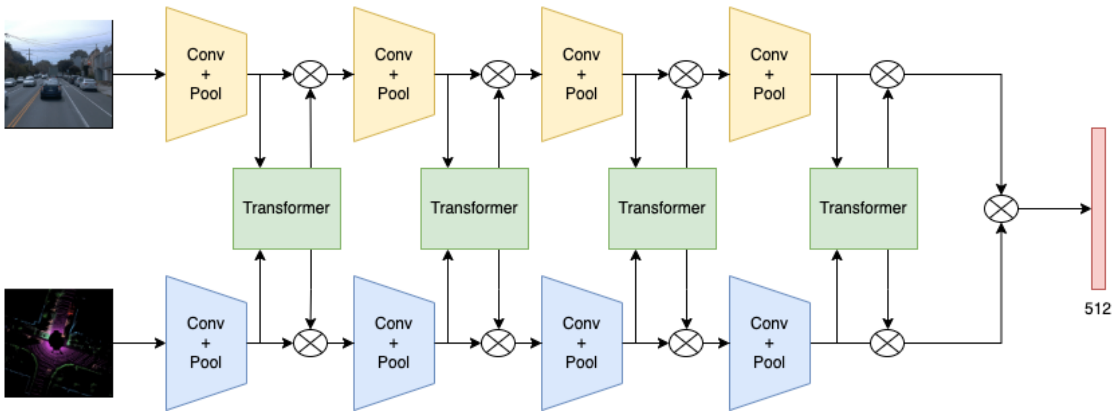


Figure 3.2: Transfuser backbone with four transformer blocks.

evant details, leading to slow learning and poor performance. Therefore, carefully selecting the observation space is critical to ensure the agent can effectively perceive and learn from the environment.

The 512-dimensional vector from the vision encoder forms the basis of the observation vector. This gives the policy the necessary information to maneuver without colliding with objects in the environment. This vector can be expressed as

$$X_{vision} = ENCODE(X_{RGB}, X_{Lidar}) \in \mathbb{R}^{512} \tag{3.7}$$

where X_{RGB} and X_{Lidar} is the Image and LiDAR inputs respectively. The image has dimensions $160 \times 704 \times 3$, while the LiDAR pseudo-image has dimensions $256 \times 256 \times 3$.

Another important point to add to the observation is the goal locations. These are points related to the overarching goal and are given along with a high-level command (i.e., take a right turn, lane shift left). These points are transformed into the local coordinate frame of the ego vehicle using the transformation

$$X_{T,local} = \begin{bmatrix} \cos\theta_{ego} & -\sin\theta_{ego} \\ \sin\theta_{ego} & \cos\theta_{ego} \end{bmatrix} \cdot (X_{T,GPS} - X_{ego}) \tag{3.8}$$

to make them more interpretable. With this, the agent gets the required information to maneuver around the world to complete its global objectives correctly. The goal location - $X_{T,local} \in \mathbb{R}^2$

- is given as 2D coordinates transformed into the coordinate frame of the ego vehicle using its heading angle θ_{ego} and global position $\boldsymbol{X}_{ego}$. The command values in CARLA are given as a number from 0 to 5. Using this directly would make it difficult for the policy to interpret. A better option is to one-hot-encode the command value, giving a vector $\boldsymbol{C}_{HLC} \in \{0,1\}^6 : ||\boldsymbol{C}_{HLC}|| = 1$. This gives the target's contribution to the observation space

$$\boldsymbol{X}_{target} = \begin{bmatrix} \boldsymbol{X}_{T,local} \\ \boldsymbol{C}_{HLC} \end{bmatrix} \in \mathbb{R}^8. \tag{3.9}$$

Finally, we want to add information about the vehicle's velocity to the observation vector. This is essential information to include in the observation. Since the encoded vector only is based on sensor data from one timestep, there is no other way to get information about the vehicle's movement. The velocity is given as a scalar value given by the speedometer, resulting in:

$$X_{vel} = V \in \mathbb{R}^+. \tag{3.10}$$

There are several different ways to combine these elements together to form the final observation. The simplest is to concatenate them into a single vector;

$$\boldsymbol{X}_{obs} = \begin{bmatrix} \boldsymbol{X}_{vision} \\ \boldsymbol{X}_{target} \\ \boldsymbol{X}_{vel} \end{bmatrix} \in S_t \subset \mathbb{R}^{516}. \tag{3.11}$$

3.5.3 Action Space

There are three main ways to formulate the action space. The policy can predict waypoints just like Transfuser [8] and Interfuser [9] does, or it can predict the control outputs directly. It can also predict the steering control directly and predict the desired velocity and let the throttle and brake be determined using a normal controller.

Waypoint prediction This method predicts the waypoints the vehicle wants to follow and then uses these in a regular controller to get control inputs in the same manner as Transfuser. Doing this is much more explainable than directly predicting the controls. Having the waypoints makes it possible to visualize the decision the vehicle is making in real-time. It also makes it easier to implement safety checks for the agent similar to Transfuser [8] and Interfuser

[9].

Direct control prediction This is the simplest of the two options, as there is no need for a separate controller. There are a total of seven controllable values for the vehicles in CARLA, all shown in Table 3.3.

Table 3.3: CARLA control parameters.

Control	Values
throttle	$[0.0, 1.0]$
steer	$[-1.0, 1.0]$
brake	$[0.0, 1.0]$
hand_brake	boolean
reverse	boolean
manual_gear_shift	boolean
gear	$\mathbb{N}$

The vehicle will use an automatic transmission and will never reverse. To make control easier, we also assume there will be no need for the hand brake. The only outputs from the policy, therefore, are the throttle, steering, and braking. In addition, since the vehicle never will brake and accelerate at the same time, the throttle and brake can be combined into one action value - a_1 - with negative values indicating braking and positive indicating acceleration. The steering is then controled by a second action value - a_2.

The action space is, therefore, two output variables $a_1, a_2 \in [-1, 1]$. The individual control inputs are computed as follows:

$$\text{throttle} = clip(a_1, 0, 1) \tag{3.12}$$

$$\text{brake} = clip(-a_1, 0, 1) \tag{3.13}$$

$$\text{steer} = a_2. \tag{3.14}$$

where $clip$ limits the value of the first parameter to a range determined by the second and third parameters. If the value of a_1 is negative, the brake value will be given a positive value, while the throttle will be clipped to 0.

Discrete Action Space One disadvantage of the direct control prediction action space described above is that randomly selecting a value gives an equal probability for braking and accelerating. This might make training an agent without expert demonstrations very slow in the beginning since the agent will have difficulty building up speed due to the inertia of the vehicle and will, therefore, rarely receive a positive reward signal for driving closer to the desired speed.

Let's consider an example where the agent needs to drive a car at a specific speed. In the direct control prediction action space, the agent outputs the throttle value, which could be any real number between -1 and 1, with -1 indicating full braking, 0 indicating no acceleration or braking, and 1 indicating full acceleration. However, since the agent chooses a value randomly, it has an equal chance of choosing any value between -1 and 1, including negative values that result in braking. As a result, the agent may not accelerate fast enough to reach the desired speed, and it may even slow down by randomly choosing a negative throttle value.

One solution to this problem is for the policy to use a discrete action space, where one action is the predicted desired speed of the vehicle, while the other is discretized steering angles. For example, $a_1 = 0 \Rightarrow v_{d,p} = 0km/h$, $a_1 = 0.5 \Rightarrow v_{d,p} = 30km/h$ and $a_1 = 1 \Rightarrow v_{d,p} = 60km/h$. By comparing the predicted desired velocity $v_{d,p}$ to the one computed from (3.20), a positive reward signal can be given every time the agent predicts the correct velocity, bypassing the need to build up speed. The control inputs can then be computed using

$$[\text{throttle}, \text{brake}]^T = \text{LongditudinalController}(v_{d,p}, v_{ego}). \tag{3.15}$$

3.5.4 Reward Function

The reward function is probably the most important design feature for a RL system. It decides what's good and bad for the agent to do in the environment and, therefore, directly guides the training of the networks. Several elements could be included in a reward function for driving.

The most basic thing the autonomous vehicle needs to do is follow the road and drive close to the center of the lane. It, therefore, makes sense to add a term to the reward function punishing driving away from the center of the lane and another term punishing being misaligned with the direction of the road. These terms are called R_{dist} and R_θ in:

$$R_{lane} = R_{dist} + R_\theta \in \mathbb{R}. \tag{3.16}$$

Driving perfectly in the center of the road is a too strict requirement. For small deviations from the centerline, the negative reward should be negligible, allowing for some deviations without conferring large negative rewards. Taking the square of the distance could have this effect but also greatly increase the punishment for larger deviations. Dividing the current deviation by D_{max} - the maximum centerline deviation allowed before a terminal state - would reduce the scaled distance to the interval [-1, 0]. This results in

$$R_{dist} = -\left(\frac{||\boldsymbol{x}_{lane} - \boldsymbol{x}_{ego}||}{D_{max}}\right)^2, \tag{3.17}$$

where $\boldsymbol{x}_{ego}$ is the position of the ego vehicle and $\boldsymbol{x}_{lane}$ is the closest point in the center of the lane.

Simply taking the difference between the heading angle of the ego vehicle and the road is not a good solution for R_θ. The largest possible angle error is π since a yaw rotation of more than π is equivalent to a yaw rotation of less than π but in the opposite direction. The Smallest Signed Angle (SSA) of the heading difference is used here - as described in Fossen [32] - giving an output angle in the range $[-\pi, \pi]$. For the same reason as R_{dist}, the SSA is scaled by π and squared, giving a reward in the range $[-1, 0]$. This gives

$$R_\theta = -\left(\frac{\text{SSA}(\theta_{ego} - \theta_{lane})}{\pi}\right)^2, \tag{3.18}$$

where θ_{ego} is the heading angle of the ego vehicle and θ_{lane} is the heading direction of the lane.

Another important factor in driving is speed, where there are three cases to consider. Firstly, the ego vehicle might be in a scenario where it has to stand still, for example, if it is stuck in traffic or at a red light. In these scenarios, there should be no negative reward for the vehicle to stand still. The second case is when the ego vehicle drives faster than the desired speed. A negative reward should be given - increasing the higher the breach is - with a minimum reward of -1. This is achieved by using the ratio of v_{ego} - the speed of the ego vehicle - and v_d - the current desired speed. This ratio is scaled, subtracted from 1, and capped using a max function. The final case is if the agent drives slower than the desired speed but is not in traffic. The reward should max out on $v_{ego} = v_d$ with a value of 1. Formulating this mathematically gives the equation

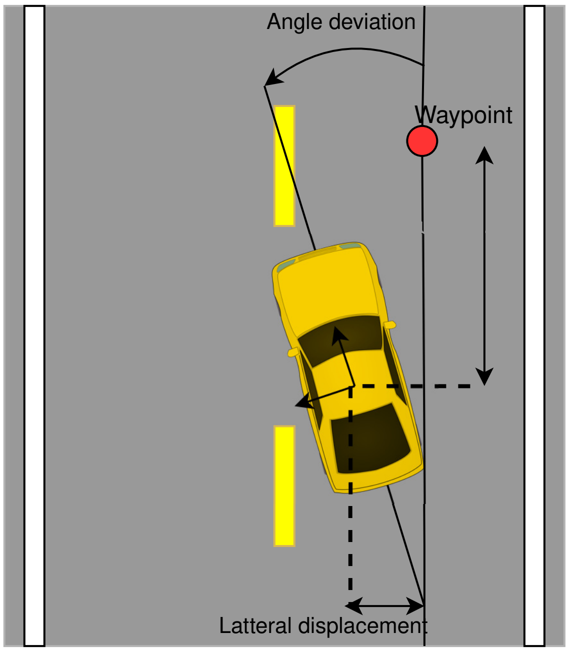


Figure 3.3: Values used for computing rewards.

$$R_{vel} = \begin{cases} \max(-1, 1 - c_v \frac{v_{ego}}{v_d}) & v_{ego} > v_d \\ \frac{v_{ego}}{v_d} & v_{ego} \leq v_d \\ 0 & v_d < 1m/s \end{cases} . \tag{3.19}$$

The desired speed of the ego vehicle depends on the current situation. If it is at a red light or has another vehicle standing still directly in front of it, the desired speed should be zero. If there are no obstructions, the ego vehicle should drive at the speed limit. In this thesis, it is set to 40km/h for all parts of the map since the setup has no memory elements. This is done to simplify the environment and increase the likelihood of successful training runs. An abrupt change in v_d is not desired, and it should gradually be decreased to zero after a certain threshold. This is shown in

$$v_d = \begin{cases} 0km/h & \text{if closest} \leq \text{5m} \\ 40km/h \cdot (\frac{closest}{10} - \frac{1}{2}) & \text{if 15m} < \text{closest} < \text{5m} \\ 40km/h & otherwise \end{cases} , \tag{3.20}$$

where *closest* is the closest vehicle, pedestrian, or red light ahead of the ego vehicle. This is

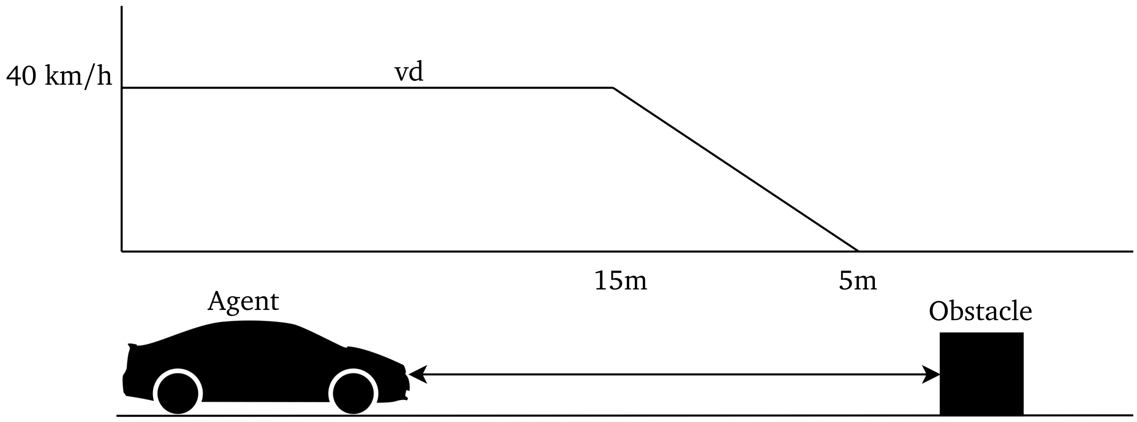


Figure 3.4: Visualisation of the desired velocity as a function of distance.

also visualized in Figure 3.4.

To be able to perform meaningful progress throughout the map, the agent also needs to be able to follow its given waypoints. The observation specified in (3.8) contains the target points as coordinates in the agent's local coordinate system. The norm of this vector will be a measure of how far away from the waypoint the agent is. Since the waypoints can be quite far away from the agent, this value has to be scaled if it is to be used in the reward function. The agent should also be rewarded positively if it gets close to the target position. The final waypoint reward looks like

$$R_{wpt} = \begin{cases} -\frac{||X_{T,local}||}{D_{T,max}} & ||X_{T,local}|| > 2 \\ 10 & otherwise \end{cases}. \tag{3.21}$$

Finally, punishment needs to be given for collisions with vehicles and pedestrians. This can be given as a singular negative value if there have been any collisions and a zero if no collisions have occurred,

$$R_{col} = \begin{cases} -1 & \text{collision occured} \\ 0 & \text{otherwise} \end{cases}. \tag{3.22}$$

All these possible reward components can be combined to form the final reward function

$$R = c_{lane} \cdot R_{lane} + c_{vel} \cdot R_{vel} + c_{wpt} \cdot R_{wpt} + c_{col} \cdot R_{col}, \tag{3.23}$$

where each element is scaled by a constant to signify the importance of each element.

3.5.5 Network Architecture

Stable Baselines 3 allows the user to customize the architecture of the policy and/or value networks for all its algorithms. Parts of the network can also be shared between the policy and value nets in an Actor-Critic algorithm like PPO. This can be useful since it allows the early shared layers to pick up on patterns in the observation while the policy and value heads convert this into an action and value estimate, respectively. The shared layers also prevent the representations in the policy and value nets to diverge too far from one another.

According to a study by M. Andrychowicz, it is recommended to use policy and value nets with no shared layers in less complex environments [33]. The vision encoder does reduce the complexity of the CARLA environment substantially; however, the observation is still quite complex compared to the environments used in the study. Because of this, using shared layers was chosen over the alternative.

A network architecture with two shared layers is chosen, with a width of 1024 and 512, respectively. The policy and value networks then have a head, each with one hidden layer of 256 nodes.

3.6 Parallel Training Environments

A big issue with doing RL in the CARLA simulator is the runtime of the simulator itself. Rendering one scene and computing the latent representation of the scene takes a lot of GPU power. Collecting trajectories can therefore take a long time. To combat this, several environments can be run in parallel. By doing this, experiences can be collected from different instances simultaneously, accelerating the training process. This can be especially beneficial in complex environments like CARLA.

One way of doing this is to run a vectorized environment. In a vectorized environment, several environments are run as separate processes, and the same agent instance controls all environments. The state of each environment instance is stored in a tensor, with the batch size being the number of environments, allowing multiple instances to be processed simultaneously. The agent receives a batch of states as input and outputs a batch of actions, one for each state. The actions are then sent to the corresponding environment instances, which perform an update and return a reward, and the updated state [34].

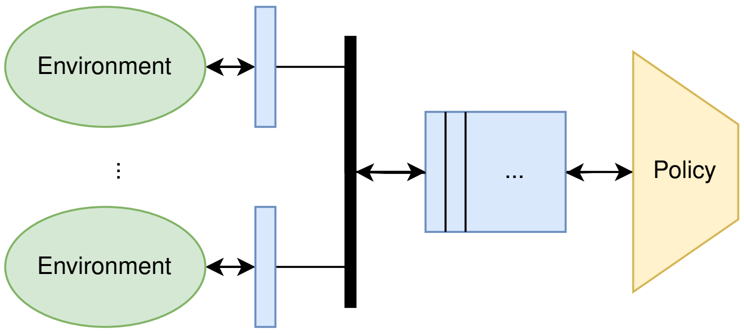


Figure 3.5: Vectorized environment. The policy takes as input a batch of observations. One from each environment, and choose one action per observation.

However, there are also challenges associated with using vectorized environments. One challenge is coordinating the interactions between the different instances of the environment. This requires careful synchronization and communication between the different instances, which can be complex to implement. Additionally, using a vectorized environment can require more computational resources, which may not be feasible on all systems.

After weighing the pros and cons of using vectorized environments in training RL agents in the CARLA simulator, it seems beneficial to use vectorization in this context. The benefits of vectorized environments are numerous, including faster training times. Furthermore, with access to a High Performance Computing (HPC) cluster in IDUN, the computational burden of using vectorization can be easily managed. Therefore, using vectorized environments can lead to significant improvements in training efficiency and agent performance, making it a worthwhile time investment when training RL agents in the CARLA simulator.

3.7 Vision Pre-training

Training a large neural network purely from reward signals might be infeasible and would definitely dramatically increase the training time due to the large increase in parameter count and the low sample efficiency of RL. This is why most RL approaches utilize a multi-stage training approach, where a vision encoder is pre-trained on regular computer vision tasks such as segmentation and object detection. The weights of the encoder are frozen in the second stage, where the agent interacts with the environment. This also makes the entire system more explainable since the output from the encoder can be decoded and visualized to the user.

3.7.1 New Dataset Generation

The newest version of the CARLA simulator - 0.9.14 - has some graphical changes from 0.9.10, which is used in Transfuser, so a new dataset needs to be generated on the new version. There have also been some changes to the available spawn points for vehicles. Luckily Chitta et al. have included code for route and dataset generation in their GitHub repository [1]. Since the data for each route are independent, the generation can easily be parallelized on Idun by creating one job per route.

3.7.2 Training of Encoder and Expert Agent

The vision pretraining step is performed similarly to the Transfuser. Five loss functions are used in training - one for semantic segmentation, depth estimation, HD map, and BEV object detection - and the waypoint prediction loss as described in Section 2.8.3. The waypoint loss is included since the waypoint prediction network will be used in the expert agent for GRIAD.

The depth and semantic segmentation are inferred from the output features from the image encoder using the same decoder as Transfuser [8]. The depth prediction uses an L_1 loss,

$$\mathcal{L}_{depth} = L_1(X,Y) = \sum_{i}^{H}\sum_{j}^{W} \|\hat{y}_{i,j} - y_{i,j}\|, \tag{3.24}$$

where $y_{i,j}$ and $\hat{y}_{i,j}$ are the real and predicted depth for pixel (i,j). On the other hand, semantic segmentation uses a cross-entropy loss,

$$\mathcal{L}_{seg} = -\frac{1}{H \cdot W}\sum_{i=1}^{H}\sum_{j=1}^{W}\sum_{k=1}^{C} y_{i,j,k}\log(\hat{y}_{i,j,k}). \tag{3.25}$$

where (H, W, C) is the dimensions of the image, $\hat{y}_{i,j,k}$ is the output prediction for pixel (i,j) for class k, while $y_{i,j,k}$ is a value which is 1 if the class of pixel (i,j) is k, and 0 otherwise.

The HD map and BEV bounding boxes are predicted from the output from the lidar encoder. The HD-map loss $\mathcal{L}_{map}$ uses the same cross-entropy loss function as the segmentation loss,

[1] `https://github.com/autonomousvision/transfuser`

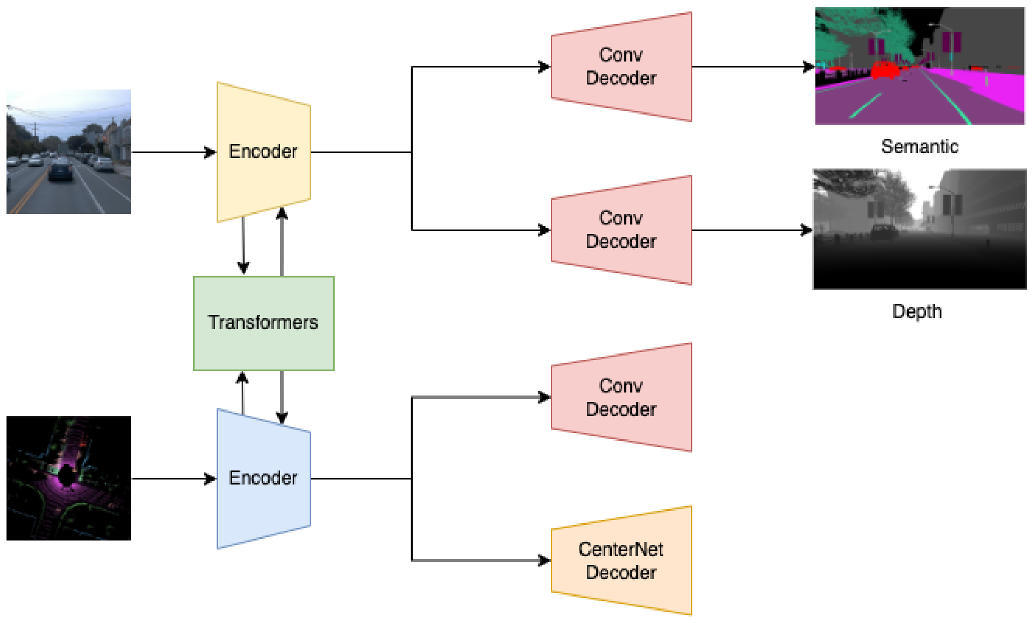


Figure 3.6: Vision pre-training architecture.

while the bounding box loss is a combination of losses for the position map $\hat{\boldsymbol{P}} \in [0,1]^{64,64}$, orientation map $\hat{\boldsymbol{O}} \in [0,1]^{64,64,12}$ and regression map $\hat{\boldsymbol{R}} \in [0,1]^{64,64,5}$ which are focal, cross-entropy and L_1 losses respectively.

3.8 Benchmarking

The official evaluation leaderboard in the CARLA simulator has 100 secret routes, but teams are limited to only 200 hours of evaluation time per month. Due to the time required for a single evaluation (over 100 hours), the official leaderboard is unsuitable for ablation studies or obtaining detailed statistics involving multiple evaluations of each model. To address this issue, Chitta et al. propose the Longest6 Benchmark, which is similar to the official leaderboard but can be used to evaluate locally [8].

The CARLA leaderboard repository provides a set of 76 routes for training and evaluating agents, but there is a significant imbalance in the number of routes per town. To balance the Longest6 driving benchmark across all available towns, the six longest routes per town from the 76 routes were chosen, resulting in 36 routes. To ensure a high density of dynamic agents during evaluation, vehicles are spawned at every possible spawn point allowed by the simulator. Each route also has a unique environmental condition obtained by combining one of 6 weather conditions with one of 6 daylight conditions. CARLA's adversarial scenarios are included in the evaluation, spawned at predefined positions along the route. The Longest6 benchmark also includes CARLA's scenarios 1, 3-4, and 7-10 [8].

To obtain a representative benchmark for each experiment, the agent is evaluated on the Longest6 benchmark three times to get a representative benchmark for each experiment, and the results are averaged. This helps to smooth out any randomness in the evaluation procedure and provides a more reliable estimate of the agent's performance. Using the Longest6 Benchmark, more detailed statistics on the performance of the RL agents under a broader range of driving scenarios and environmental conditions can be obtained without the time constraints of the official evaluation leaderboard.

Chapter 4

Simulation and Results

In this chapter, all the performed training runs will be introduced with a short description of their purpose, along with hyperparameter choices for the given run and a discussion of the results. The chapter will be concluded with a closing discussion in the context of all the previous results.

For all the runs, videos were created at intervals throughout the training process. These are stored on OneDrive; a link can be found in Chapter A. The videos are stored in folders according to the experiments and training run number, with the video file names indicating during which steps of training the videos were recorded.

There were also lots of data plotted during training. Some of these plot are included in this chapter. The ones not discussed here can be found in Chapter B.

4.1 Issues With Training on Idun

Whether A100 or RTX4090 GPUs provide the fastest training depends on the number of environments that can be run in parallel. Each CARLA server requires around 4GB of VRAM to run, while the CARLA Python client with a Transfuser backbone requires 3.5 GB. Since the A100s have 80GB of VRAM, a maximum of 11 environments can be run at the same time. It is, therefore, likely beneficial to train using A100s over RTX 4090s.

When training on Idun, some issues were quickly encountered. When running with one environment, everything worked as normal, but introducing a second caused the Python client to exit with a CUDA_INTERNAL_ERROR. After a thorough investigation, it was discovered that the root of the issue was something on the Idun nodes having A100 GPUs. Using the nodes with P100 or V100 GPUs did not result in this issue.

Since the V100 and P100 only have 16 and 12 GB, respectively, there is no longer any advantage of training on Idun. The choice, therefore, fell on VCXR12 since it was less competition for computing resources compared to NAP02.

4.2 Vision Pre-training

4.2.1 Graphical Differences Between CARLA 0.9.10 and 0.9.14

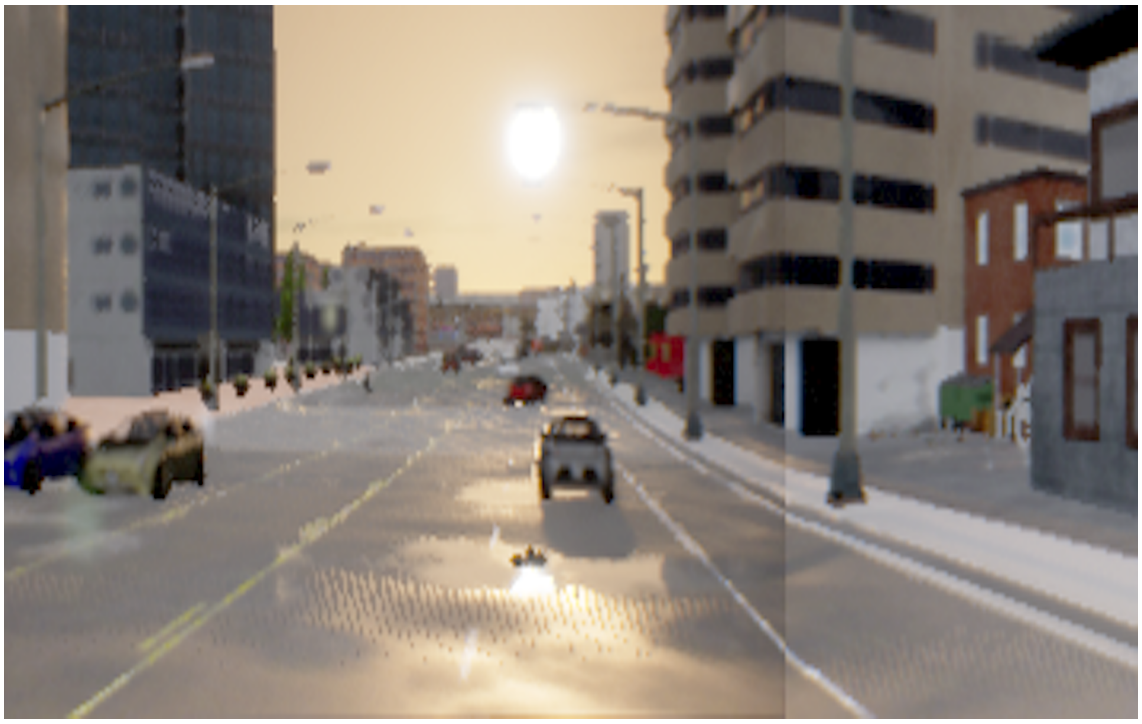
The newest version of the CARLA simulator has some graphical and lighting changes compared to the version Transfuser was trained on. These changes are especially noticeable when it comes to lighting. An example is shown in Figure 4.1 and Figure 4.2. The images from the two versions are taken using different resolution cameras, but the differences in lighting quality are still evident. These changes cause the performance of the Transfuser agent trained on the 0.9.10 dataset to drop on the new version.

The Transfuser repository contains all the scripts for generating the original dataset, so regenerating a dataset on the newer version should be simple. Since the data from different routes are independent, this process can easily be parallelized on Idun, running multiple jobs. Despite this, there were some parts of the dataset from the original authors that were not recreated while using their scripts. It is unknown where this data comes from, but as a result, the new dataset is smaller compared to the original.

4.2.2 Training Results

The training script in the Transfuser repository uses a learning rate of 10^{-4} for the entire training process by default. In the first training run, the loss started to increase again after around 7 epochs, this can be seen in Figure 4.3. This is caused by too high of a learning rate. To flatten out this curve, the learning rate can be reduced at different points of the training. This can lead to better training results.

In the second training run, the learning rate was reduced by a factor of 10 after 10 and 15 timesteps. This resulted in significant improvements compared to the run with a constant learning rate, completely flattening the bump in the loss around epoch 10-15.


(a) Version 0.9.10.

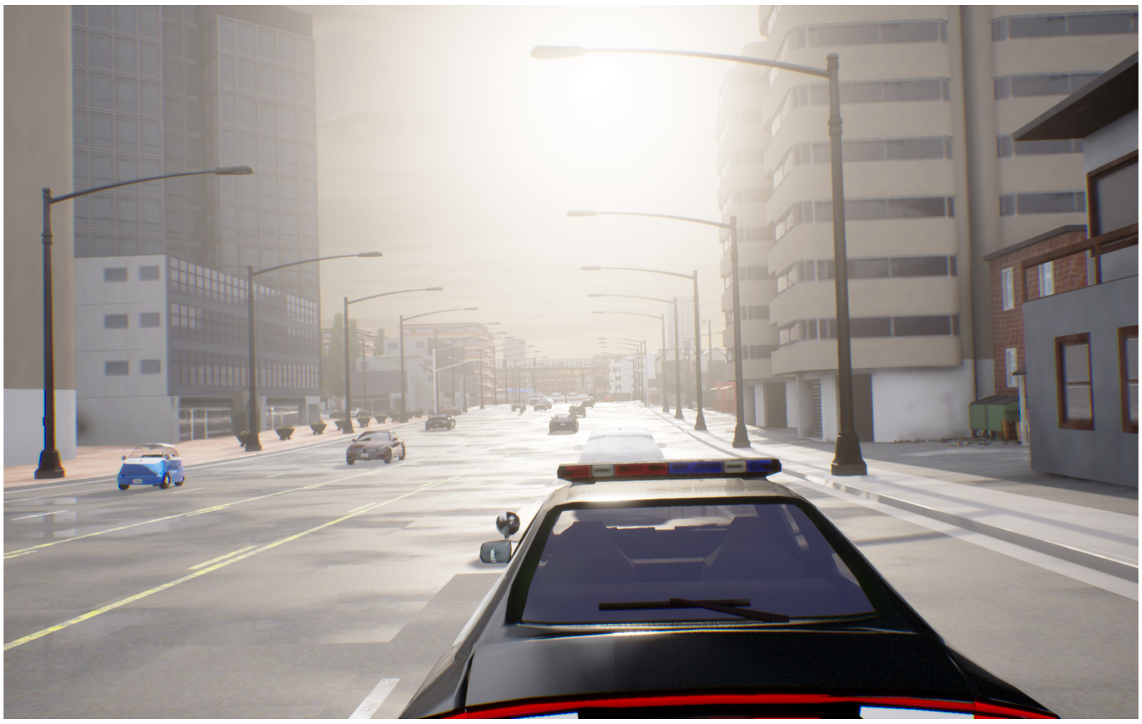
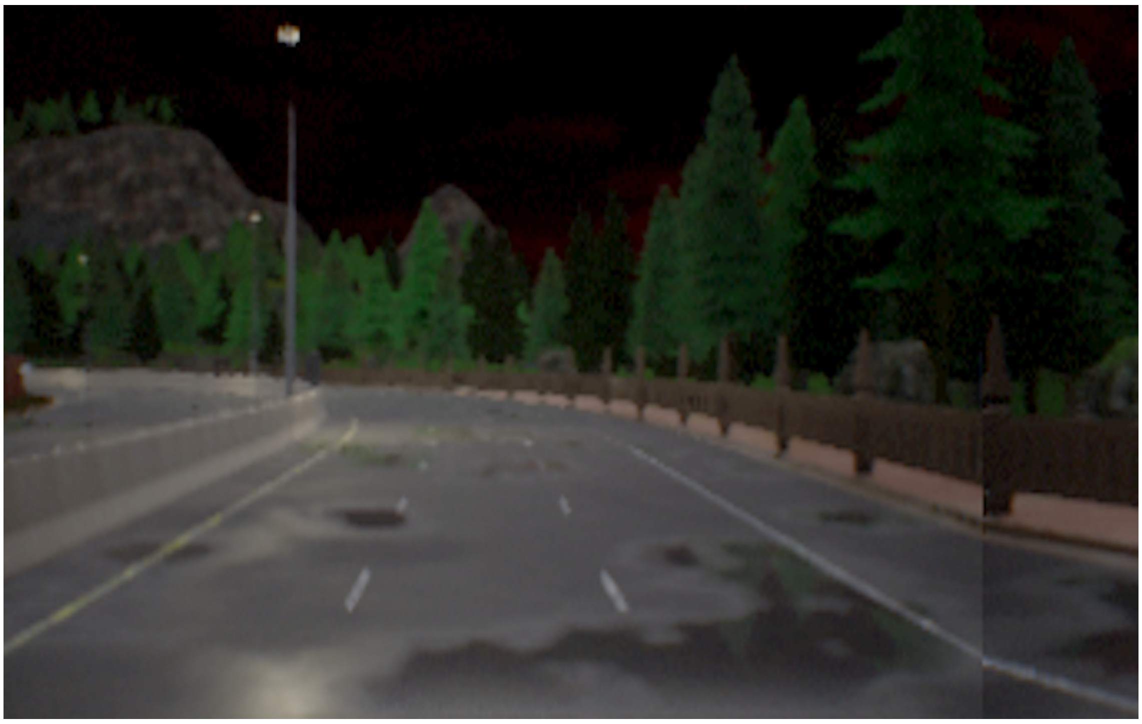
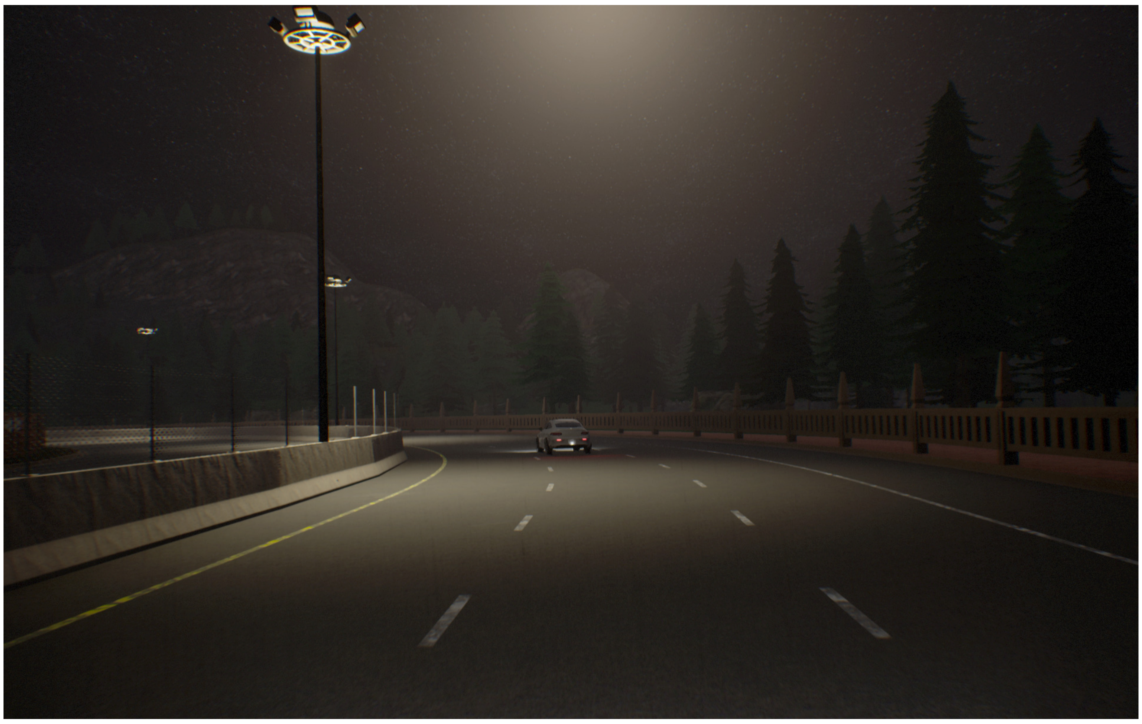

(b) Version 0.9.14

Figure 4.1: Lighting differences during the day.



(a) Version 0.9.10.



(b) Version 0.9.14.

Figure 4.2: Lighting differences during the night.

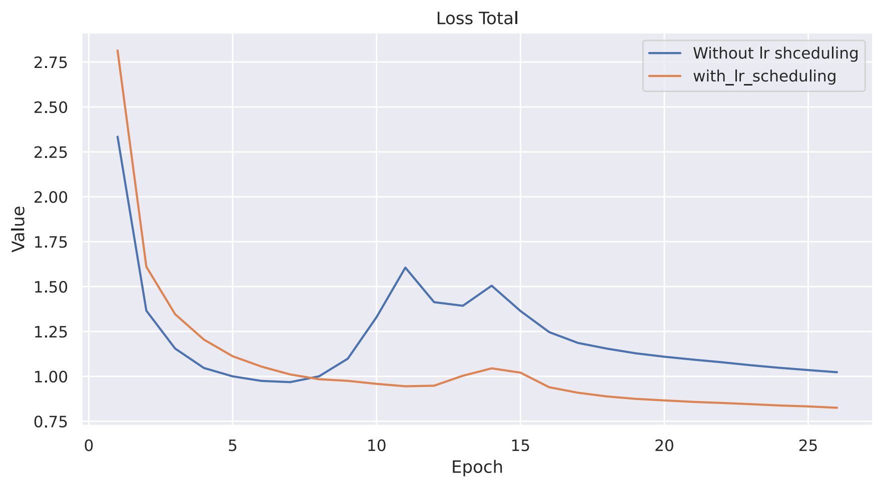


Figure 4.3: Plot of total loss for the vision pertaining step.

4.3 Experiment 1 - RL Baseline Using PPO

Before introducing any external factors or techniques to influence the training process of an agent, it is essential to establish a baseline performance level. One of the primary benefits of training a baseline agent is that it enables a comparison of the performance of future agent models to this starting point. By comparing the performance of these models to the baseline, the effectiveness of any additional techniques or external factors introduced during the training process can be established. This comparison also helps to measure the overall change in performance due to additional techniques.

In addition to providing a benchmark for evaluating future agent models, training a baseline agent also helps identify potential issues that may arise during the training process. Since the baseline agent learns only from the environment without any external assistance, it may highlight any challenges in the problem domain that might require additional attention or adjustments. Furthermore, the knowledge gained from training a baseline agent using RL can be used to improve the overall training process. Understanding how the agent interacts with the environment, the challenges it faces, and the decisions it makes during training can inform the design of future training strategies, leading to more efficient and effective training of the agent.

4.3.1 Setup

The reward function is chosen to be as in (3.23), with $c_{lane} = c_{vel} = c_{wpt} = 1$, and $c_{col} = 200$. The observation space described in Section 3.5.2 is used, along with the direct control action space in Section 3.5.3. The training was done on vcxr12, running 3 exploration environments in parallel. To better understand the performance progression, a video recorder wrapper is used around the standard environment, which records a video of the agent during training at a given interval and video length. For all the runs, an interval of 10,000 steps is used, with a video length of 1000 steps.

Due to crashes of the CARLA server, checkpoints have to be stored frequently. This allows training to be resumed automatically when a crash is detected. All the code is run in docker containers. For the CARLA server, the premade CARLA 0.9.14 image [1] is used while the client runs in a custom container. Communication between containers is handled using docker-compose. This also allows for the use of Docker composes *restart on failure* setting, which restarts the containers if the program running inside crashes. If the server crashes, the client will time out, causing both containers to restart.

Considering the time and computational resources available, the training goal has been set at $1,000,000$ steps. Although this is considerably less than what was done in the GRIAD paper [7], it is hoped that this should be sufficient to demonstrate the potential of the approach within reasonable limits and yield valuable insights into the system's performance.

The PPO implementation in SB3 allows for several hyperparameters to be set. The default optimizer in SB3 is Adam. The most common learning rates used for this optimizer are in the range $[10^{-3}, 10^{-5}]$, but the default parameter set in sb3 is $4 \cdot 10^{-4}$. The number of steps performed in the simulator before each policy update was set to 128. During each update, 10 epochs are run with the collected data, with a batch size of 64. For the discount factor γ and clip range ϵ, the values used in the original paper were chosen [26]. All the chosen hyperparameters are shown in Table 4.1.

To gain more insights into the training of the PPO algorithm, a custom callback was created to log the weight and bias histograms of the action and value networks. These distributions can provide insights into how the parameters are changing and whether there are any issues. By logging these histograms at regular intervals, it is possible to monitor the progress of the training process and detect any abnormalities or inefficiencies. In addition to the histograms, several other parameters were logged. These are explained in Table 4.2.

[1] `https://hub.docker.com/r/carlasim/carla`

Parameter	Value	Description
Learning rate	0.003	Scale factor for gradient update
gamma	0.99	Discount factor for future rewards
clip range (ϵ)	0.2	Probability ratio for importance sampling clipped to $[1-\epsilon, 1+\epsilon]$
n steps	128	Number of steps performed before gradient update
batch size	64	Batch size for gradient update
n epochs	10	Number of epochs to perform for each n steps

Table 4.1: Hyperparameters used for PPO training.

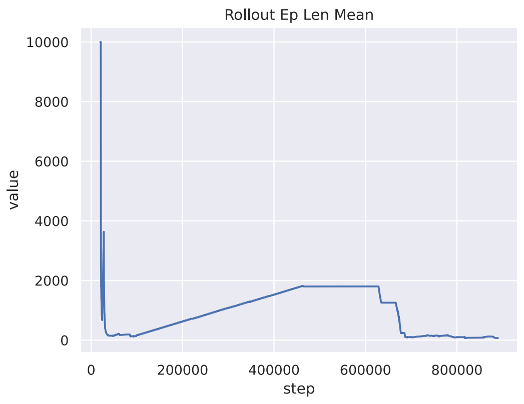
Value	Description
Entropy loss	The negative of the entropy. A measure of how spread out the distribution is. A low value indicates more randomness, while a higher value results from a more deterministic policy
KL_divergence	An approximation of the KL-divergence of the polity between each update
clip fraction	The fraction of the policy loss contributions being clipped
standard deviation	The standard deviation of the policy
Value loss	The loss of the value function given in (3.6)
Policy loss	The loss value for the policy network given in (3.4)

Table 4.2: Logged scalars from training PPO.

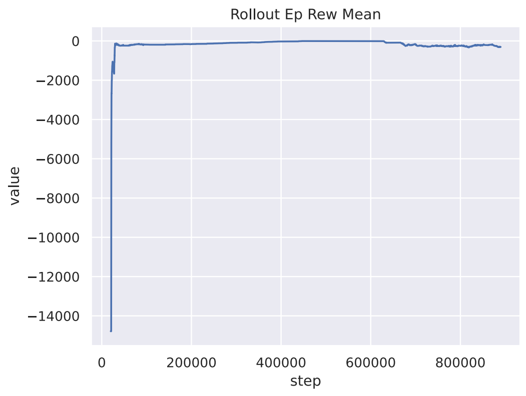
4.3.2 Results for Run 1 - Initial Setup

Looking at the mean episode reward and length, it is clear that the training stagnates quite quickly. After about 40,000 steps, there is little change in the average reward received. Looking at the video recordings - which can be found in Chapter A - clearly show the cause of this. The agent has learned to accelerate forward and turn to the right. This causes the vehicle to quickly move further away from the centerline than D_{max}, causing the episode to terminate early. This is likely the case since the punishment for crashing is a lot higher than the punishment for deviating too far. The action of driving off the right side of the road is the simplest way to drive forward, thus keeping the desired speed but also not colliding with something.

Another thing to notice about this training run is the clip fractions in Figure 4.5. This represents the fraction of updates where the surrogate objective function was clipped, which indicates that the policy update was constrained to a certain range to prevent too-large policy changes. The clip fraction should ideally be small, indicating that the policy updates are mostly within the desired range. In this run, the clip fraction often exceeds 0.8. This suggests that the policy was changing too rapidly, and that can make it difficult to converge on a good solution. The slow increase in the clip fraction could be counteracted by using a learning rate scheduler to slowly decrease the clip fraction throughout training. This might not be enough, though, and it might



(a) Mean episode length.



(b) Mean episode reward.

Figure 4.4: Rollout statistics during training PPO run 1.

be necessary to reduce the initial learning rate as well.

The explained variance plot also shows parts of the problem. The explained variance will be close to one if the value network accurately predicts the advantages, while a negative explained variance shows that it is making poor predictions. Figure 4.6 shows a negative explained variance for most of the training period, with occasional massive negative spikes. It also seems to get worse as the training progress. The value network was, therefore, not accurately predicting the advantages, which made it difficult for the policy to learn effective behavior. This, in combination with the fast-changing policy, explain why the training quickly diverged to a suboptimal policy.

The plots of entropy and standard deviation can provide valuable insight into the training progression of the PPO algorithm. As shown in Figure 4.7, the entropy and standard deviation decrease sharply at the beginning of training. This leads to the policy becoming more deterministic, and hence less exploration is performed by the actor, which might be the result of the stagnation seen in Figure 4.4. Since the starting state of every episode is quite similar, with the ego vehicle centered on the road. A more deterministic policy might therefore cause every episode to follow similar trajectories, limiting the exposure to different parts of the observation space and hence impeding learning.

One way to counteract this is to perform entropy regularisation. This is done by adding another component to the loss function being minimized - called the entropy loss. This is simply just the negative of the total entropy and is shown in

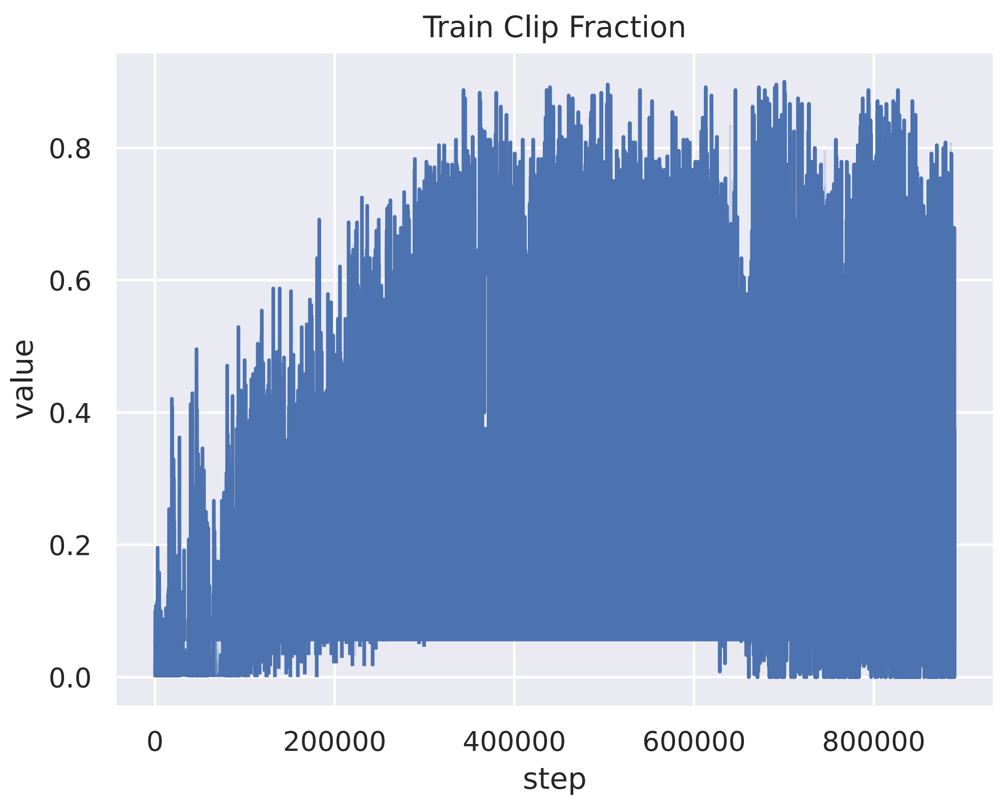


Figure 4.5: Clip fraction from training run 1 using PPO.

$$\mathcal{L}_{entropy} = -c_{entropy} H(X), \tag{4.1}$$

where $H(X)$ is given in (2.21) and $c_{entropy}$ is a scaling coefficient.

By incorporating the entropy loss term into the loss function, the algorithm is also incentivized to maximize entropy. This encourages exploration and can help to prevent the actor from getting stuck in local optima.

Choosing a good value for the entropy coefficient for the loss function is a hyperparameter tuning problem that can be solved using hyperparameter tuning. Due to the long training times in CARLA, this is not a viable choice. Values between 0 and 0.01 are shown to work well [26], so 0.005 was chosen - right in the middle of this range.

The final graphs to mention are the policy and value losses. In Figure 4.8, it is difficult to

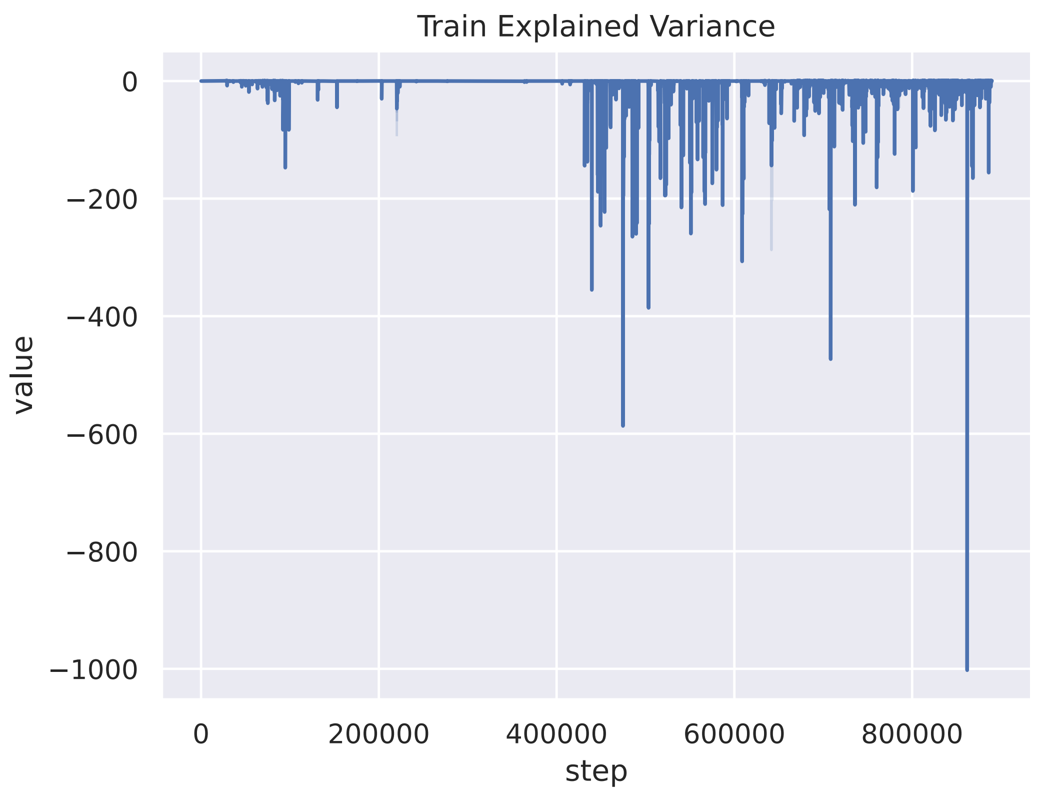
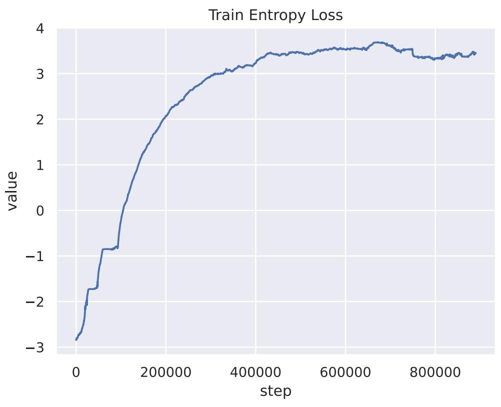


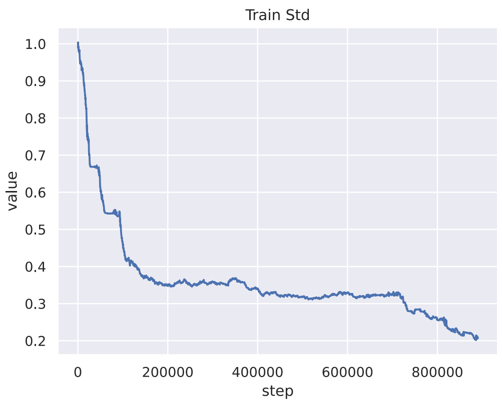
Figure 4.6: The expected variance of the value function.

make out the trajectory of the losses because of very noisy plots. The high noise level of the loss might partially be because of the previously discussed high learning rate. However, the biggest contributor is likely that the parameter n steps is low. With a small number of steps performed per update, the collection of transitions is unlikely to be very representative of the environment or consistent with one another. As a result, significant noise is introduced into the loss values. By increasing the number of steps per update, a more representative sample of transitions is obtained each time the network weights are updated.

Despite the noisy loss, it is clear that the value loss has a much higher magnitude than the policy loss. This can be an issue since the networks share some weights, making the value loss dominate the weight update in the shared layers. The reason for the high loss value is that the rewards can be quite large in magnitude. A way to mitigate this is to perform normalization and clipping of the rewards. In the study by M. Andrychowicz, it is recommended to test out reward normalization to see if it improves performance [33].

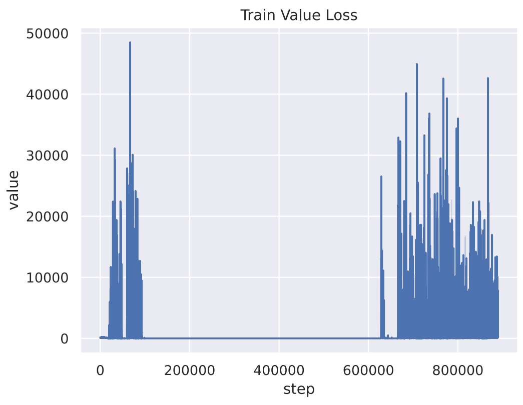


(a) Entropy loss.

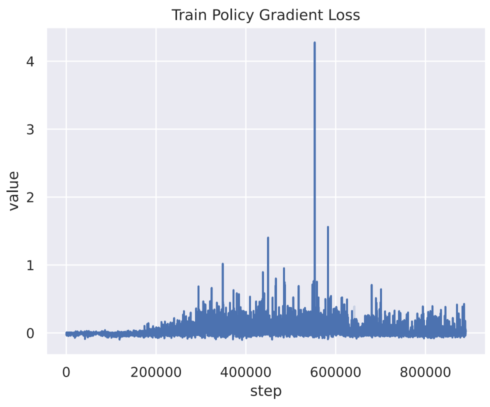


(b) Standard deviation.

Figure 4.7: Statistical information about the policy in run 1.



(a) Value loss.



(b) Policy loss.

Figure 4.8: Losses for PPO training run 1.

4.3.3 Results for Run 2 - Reward Normalization and Learning Rate Scheduling

Normalizing the rewards can be done by estimating the moving mean and variance of the discounted rewards using an estimator for the mean,

$$\bar{\mu}_k = \bar{\mu}_{k-1} + \frac{x_k - \bar{\mu}_{k-1}}{k}, \tag{4.2}$$

where $\bar{\mu}_{k-1}$ is the estimate for the mean of the k-1 first samples, x_k is the k^{th} sample, and $\bar{\mu}_k$ is the updated estimate using the new sample. The variance can be estimated using Welford's online algorithm [35] for the variance,

$$s_k^2 = \frac{k-2}{k-1}s_{k-1}^2 + \frac{(x_k - \bar{\mu}_{k-1})^2}{k}, \tag{4.3}$$

where s_k^2 is the updated estimate for the variance, while s_{k-1}^2 is the old estimate. The x_k is the new sample, while $\bar{\mu}_k$ is an estimate of the mean computed using (4.2). The normalized rewards can then be computed with

$$R_{k,norm} = \frac{R_k - \bar{\mu}_k}{s_k}, \tag{4.4}$$

where the mean of all received rewards up until timestep k - μ_k - is subtracted from the reward and the difference is divided by the standard deviation of the received rewards - σ_k [33].

The initial learning rate in run 1 was probably too high at $3 \cdot 10^{-4}$. Reducing the initial learning rate slightly and then reducing it again throughout training should improve performance. For run 2, the learning rate schedule

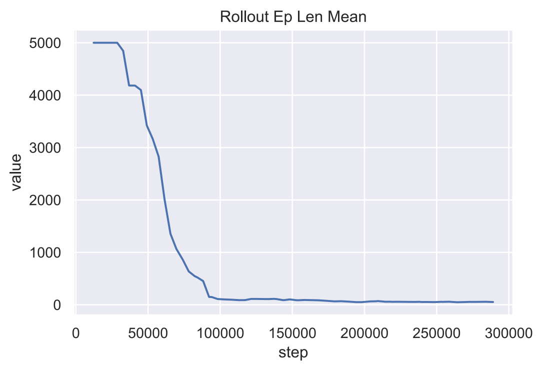
$$\text{learning rate} = \begin{cases} 10^{-4} & step < 100,000 \\ 5 \cdot 10^{-5} & step < 300,000 \\ 10^{-5} & step > 300,000 \end{cases} \tag{4.5}$$

was used. Using a higher learning rate in the beginning, allows the agent to explore and learn quickly. As training progresses and the agent becomes more knowledgeable about the environment, reducing the learning rate helps fine-tune the policy with more precision.

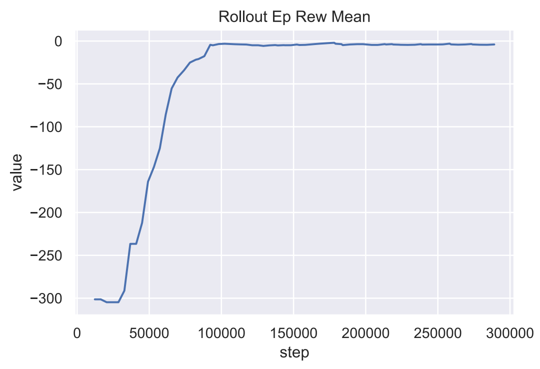
The number of steps before each update was also increased to 2048, while the number of epochs trained for each update remained at 10. This should help reduce the high noise levels observed in the losses during run 1, which again should increase learning speed, as noisy gradients for the policy update are known to reduce learning speed [27]. An entropy loss was also added with a scaling coefficient $c_{entropy} = 0.005$.

These changes resulted in way smoother loss curves, shown in Figure 4.10. However, the performance of the agent did not improve significantly. The mean reward again flattens out just below zero, and again the result of this is quick termination with the agent turning to the right.

Looking at the losses in Figure 4.10, there are clear signs of improvement over Figure 4.8. The

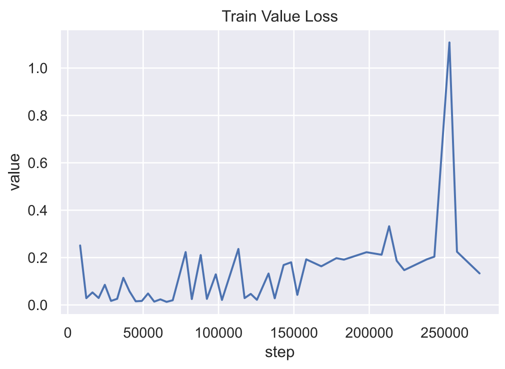


(a) Mean episode length.

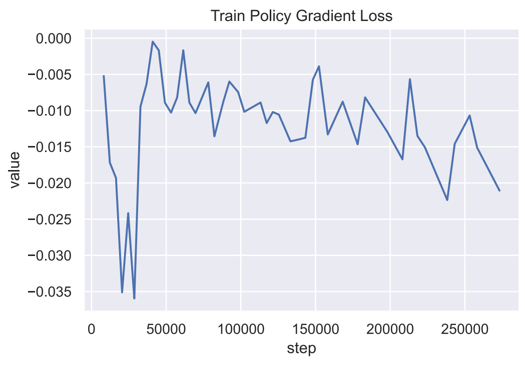


(b) Mean episode reward.

Figure 4.9: Rollout statistics for PPO run 2.



(a) Value loss.



(b) Policy loss.

Figure 4.10: Losses for PPO run 2.

value and policy losses are now much closer in scale, giving a better potential for learning. The losses do not decrease much, but there is a small downward trend on the policy loss. Since this run had to be cut short at 250k steps, this is to be expected.

Despite these apparent improvements, the results did not change much from run 1. After 125k steps, the agent again started to terminate by veering off the road. The policy is, however, more random at the end of this run, with an entropy loss still in the negative at 250k steps - seen in Figure B.2e, as opposed to over 2 in Figure 4.7, giving a possibility of improvements with further training. However, this is unlikely as it so quickly converged to the same policy as in run 1.

4.4 Experiment 2 - GRIAD Variation

GRIAD is a method introduced by Chekorun et al. to introduce expert data from a demonstration dataset into the training process of any off-policy RL algorithm. Since it require an off-policy algorithm TD3 is used here instead of PPO.

4.4.1 Setup

The implementation used here differs slightly from the original paper. Instead of using prerecorded data, the expert agent interacts with the simulator in the loop. This should reduce the preparation needed before training, at the cost of slightly more computation at training time. Using a HPC, this computation can be performed in parallel with the exploration agents, making the payoff worth it. The overall setup is shown in Figure 4.11

Again, the reward function is chosen to be (3.23), with $c_{lane} = c_{vel} = c_{wpt} = 1$, and $c_{col} = 200$. The training goal is also set to 1,000,000 steps, and the same setup for observation space, action space, docker, and video recording is used as in Section 4.3.

The chosen off-policy algorithm - TD3 - has several hyperparameters that need to be set. For most of these, the ones chosen in the original paper are used [27]. The exception is the train frequency, which is set to 1000 steps instead of at the end of every episode. The rest of the hyperparameters are shown in Table 4.3.

Hyperparameter	Value	Description
Learning rate	10^{-3}	gradient scaling factor
Discount factor γ	0.99	Discount for future reward
Replay buffer size	10^6	Capacity of the replay buffer
Train frequency	1000 steps	After how many environment steps the networks are trained
Policy delay	2	Policy network is updated every policy delay training steps
Action noise	$\mathcal{N}(0, 0.1)$	Noise applied to the actions from the policy

Table 4.3: Hyperparameters for TD3.

4.4.2 Results

The training videos for GRIAD can be found in Chapter A and have two components. On the top is a video of the expert agent driving, while on the bottom, there is a video of one agent driving using the learned policy. It does not take a long time before the agent just starts standing still. The cause of this might again be the size of the negative rewards for collisions. The policy might first try to imitate the expert agent, but since there are 3 agents running on the trained policy, the constant big negative rewards for colliding might lead the agent just to stand still.

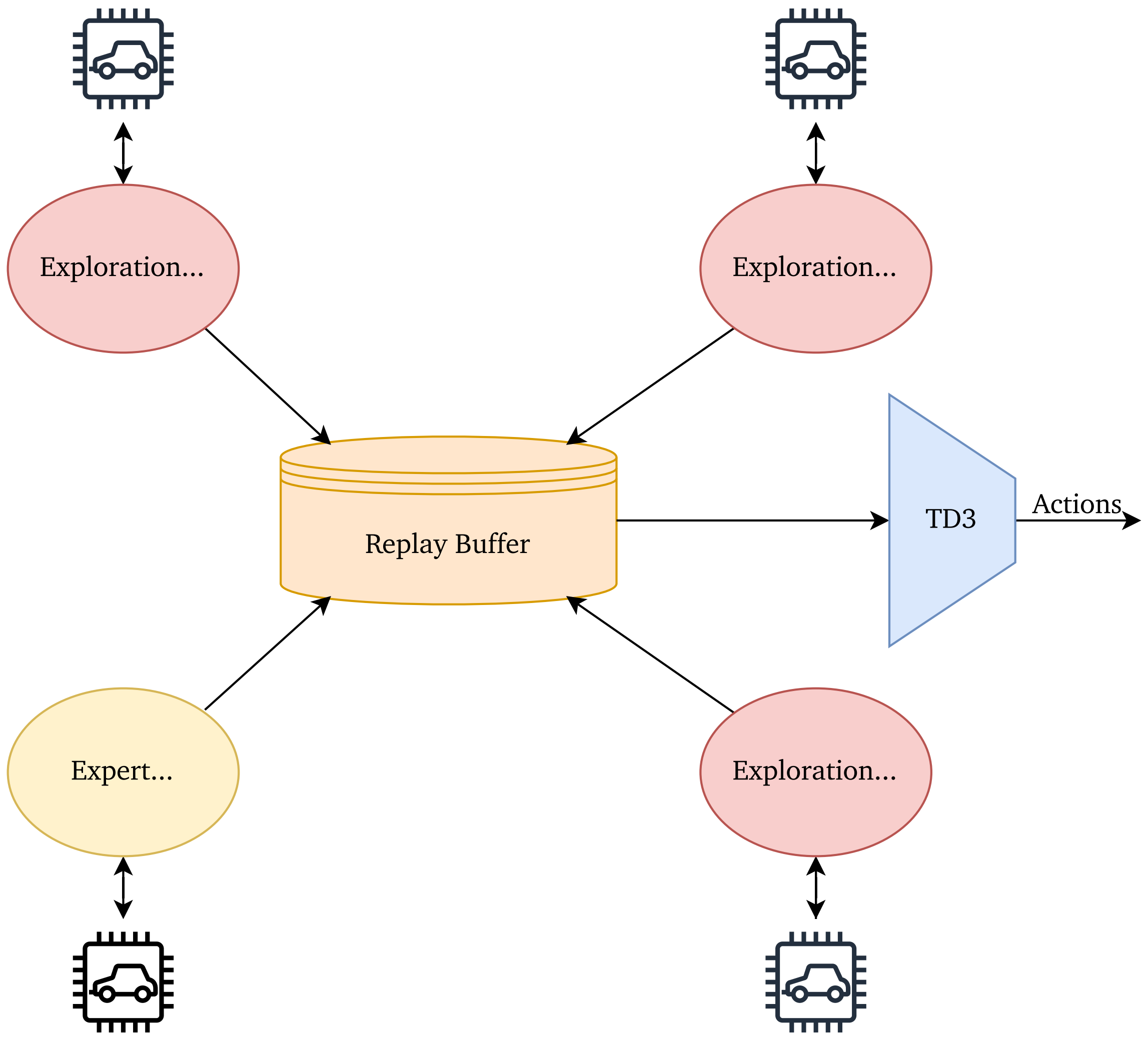
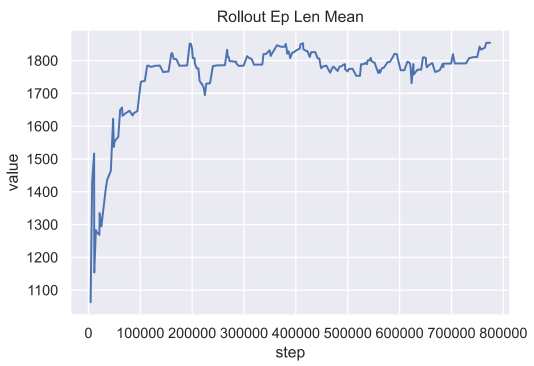


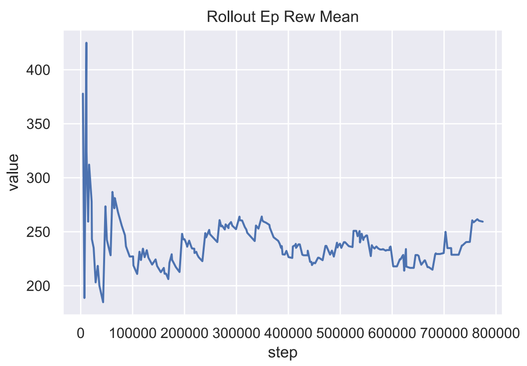
Figure 4.11: Variation of GRIAD with the expert interfacing with the simulator.

The rewards in Figure 4.12 are not comparable to that of Figure 4.4 because of the constant positive rewards received from the expert agent. These rewards will help smoothen t. What can be seen is that the mean episode length flattens out at about 1800 steps. This is close to the termination criteria at 2000 steps, which was the most common cause for termination of episodes. The rewards also quickly flatten out.

The TD3 actor and critic losses for GRIAD is shown in Figure 4.13. There is no real progress being made, with both losses being higher at the end of training than at the start. Learning rate issues might be the cause of this. Since TD3 does not perform clipping of losses, a lower learning rate might have to be used to stabilize the training.

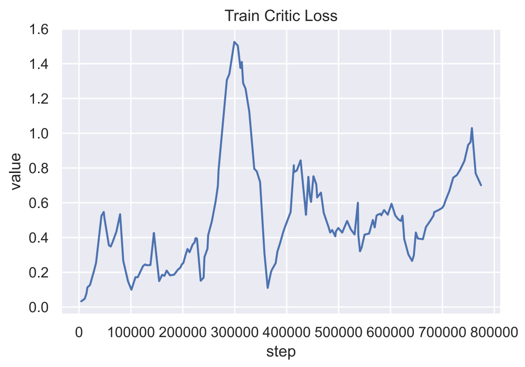


(a) Mean episode length.

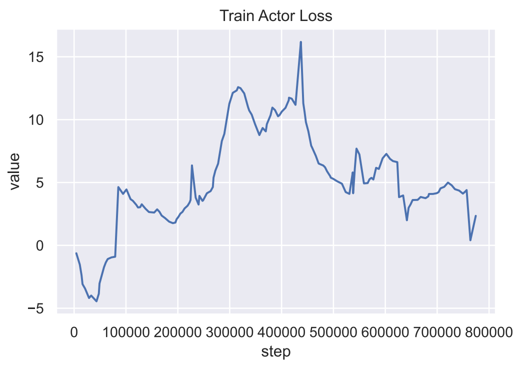


(b) Mean episode reward.

Figure 4.12: Rollout statistics for GRIAD training.



(a) Mean episode length.



(b) Mean episode reward.

Figure 4.13: Training losses for GRIAD.

The weight histogram from one of the Q functions in the TD3 critic network shows that there are some serious issues with the training. Most of the weights remain around zero, while some of the weights grow extremely large. It is difficult to see, but in Figure 4.14, some weights have grown to be larger than -25. This again indicates that the learning rate might be part of the problem.

An important factor to take into account here is the frequent crashes of the simulator. When this happens, the training is restarted from the latest checkpoint. The checkpoints do, however, not contain the contents of the replay buffer. Clearing the replay buffer entirely can result in the loss of valuable past experiences, leading to a loss of learned knowledge and potentially slower convergence. The replay buffer serves as a valuable source of diverse training data, allowing the agent to learn from a variety of past experiences. Cleaning up in the replay buffer can be beneficial, but this should be done in a structured way. Removing all the experiences at

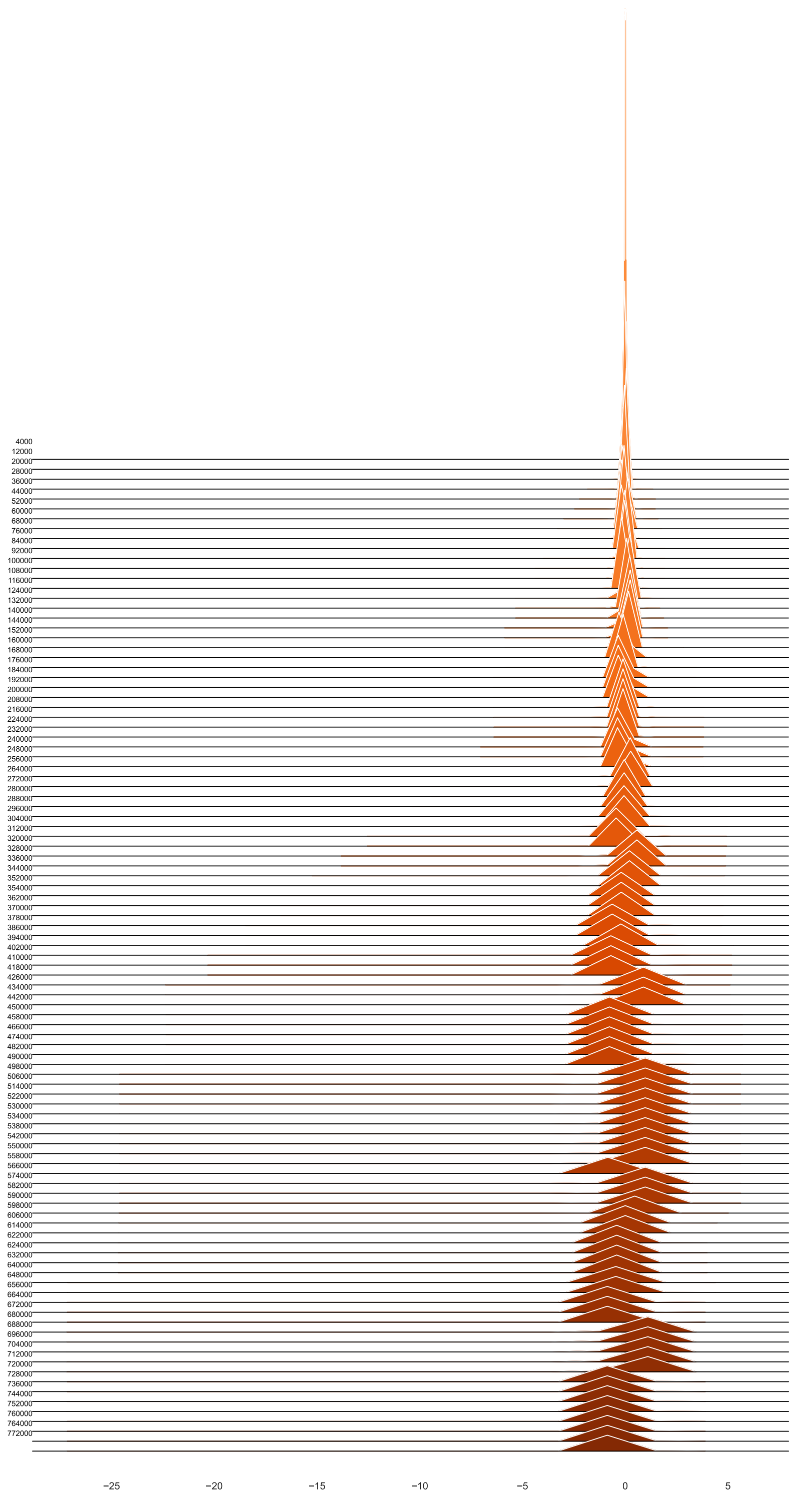


Figure 4.14: Weight histogram of layer 1 of Q_1 in the critic network. Here the depth axis is the time step, and each line is the weight histogram for a given timestep. Earlier steps are further back.

once eliminates this valuable data source, which might hinder the agent's ability to generalize and learn robust policies.

4.5 Simplifying the Environment

Due to the poor performance of the agent in Section 4.3 and Section 4.4, a simplified environment was created. Simplifying the environment can make it easier for the agent to learn. Since the scenarios trained previously have dense traffic, giving a high probability that the agent will collide during exploration and receive big negative rewards. This might lead the agent towards just avoiding collisions instead of trying to maximize the other rewards.

The simplified environment only has waypoints for the agent to follow, with no scenarios. There are also significantly fewer cars, with only a couple of other cars spread around the map. In addition, only Town 3, with the time set to noon, is used during training. This is to make all the episodes as like one another as possible. This will also hopefully aid the learning process.

4.6 Experiment 3a - Transfuser in the Simplified Environment

4.6.1 Setup

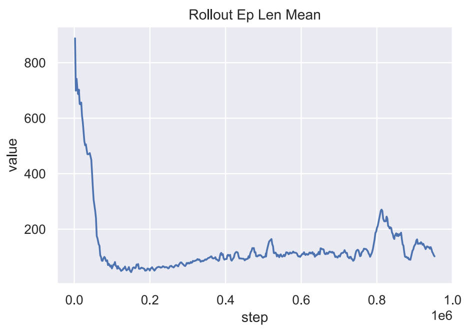
To give the closest comparison to Experiment 1, the first run in the simplified environment is performed with the Transfuser backbone. The setup is the same as in Section 4.3, both with observation and action space. Learning rate scheduling was performed using (4.5), alongside reward normalization and an added entropy loss, with a scaling coefficient $c_{entropy} = 0.01$. Every other hyperparameter is the same as in Section 4.3.3.

4.6.2 Results

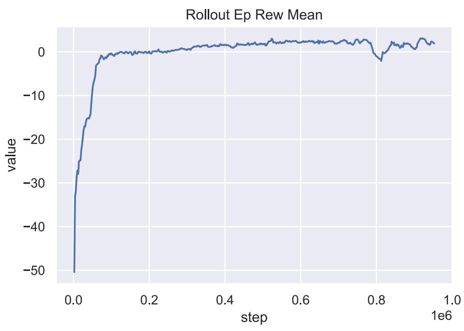
Even with the simplified environment, the agent is incapable of learning some parts of the environment while using the Transfuser encoder. Just like previously, the mean reward quickly rises to around zero while the episode length falls. There are, however, some big improvements over the more complex environment. The mean episode length does slowly increase, along with the mean reward. Learning is happening, but 1 million steps seem to be insufficient to get reasonable performance.

The agent is able to follow the road, although with large amounts of swerving. It does also drive a lot faster than the desired speed. This is likely in an attempt to receive the positive rewards for getting to a checkpoint as fast as possible. In addition, to follow the road lines on a straight road, it is also able to do so while the road is turning. This is important, as it shows the policy does not just choose random turning angles, causing the agent to drive in a somewhat straight line.

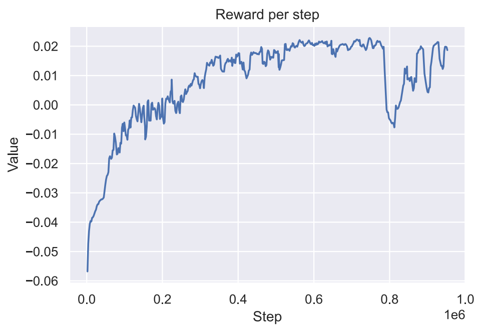
Looking at the rollout statistics in Figure 4.15, there is clearly still progress being made, and the bump in rewards and episode length at around 800k steps tells us that the policy and value function is still changing. The reward per timestep does seem to flatten out, but it also has a slump at around the same time. The fact that the reward goes down, but the episode length goes up is a good sign. It shows that it wasn't a series of bad events that caused the drop in rewards, as that would have terminated the episode and also reduced the episode length.



(a) Mean episode length.



(b) Mean episode reward.

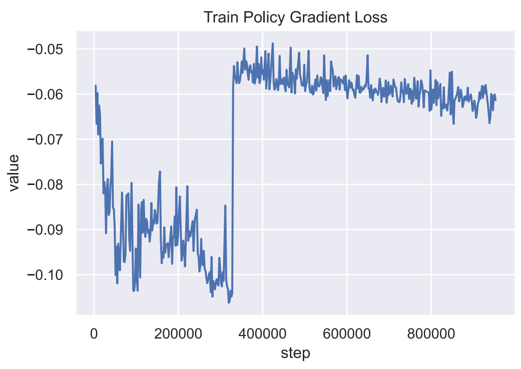


(c) Reward per step.

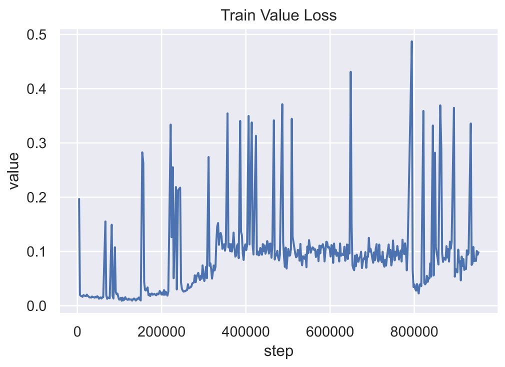
Figure 4.15: Rollout statistics during training in the simple environment using Transfuser encoder.

The losses in Figure 4.16 all end up higher than when they started. The value loss seems to have a consistent rise until it flattens out. This is not necessarily a bad thing since a high loss indicates that the agent encounters parts of the observation space where it is unable to correctly predict the value function. This in combination with the entropy loss flattening out -

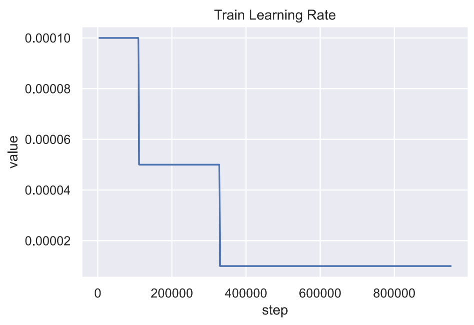
as seen in Figure B.4e - shows that there is still a lot of exploration going on.



(a) Policy loss.



(b) Value loss.



(c) Learning rate

Figure 4.16: Training losses in the simple environment using Transfuser encoder.

4.7 Experiment 3b - RL Training of CNN in the Simplified Environment

4.7.1 Setup

SB3 allow images to be used as observations directly. The observation will then be passed through a feature extractor before being passed to the value and policy networks. If the observation space is specified as a dictionary, both images and vector values can be used in the same observation. Images are passed through a CNN before all the values in the dictionary are concatenated together. This allows the CNN to be trained alongside the action and value networks.

PPO was also used for this. Since this is very different from the previous experiments, this approach is tried both with and without learning rate scheduling. It is also attempted without reward normalization, as suggested in [33].

The CNN architecture used in the paper *Human-level control through deep reinforcement learning* by V. Mnih et al. was chosen [36]. The architecture consists of 3 Convolutional layers, with 32, 64, and 64 kernels, respectively, each followed by a ReLU activation function. This architecture is quite simple and will likely have problems capturing complex visual inputs. It will, however, be faster to train because of fewer parameters. It has also been shown to work in RL applications. Since the main goal of this experiment is to get the vehicle to follow the road correctly, the issue with complex visual inputs hopefully won't be an issue since road markings are geometrically quite simple.

4.7.2 Results for Run 1

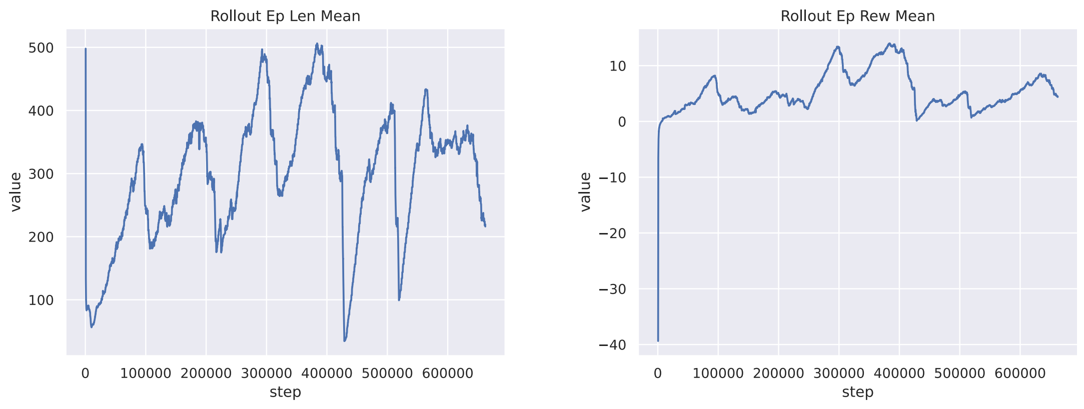
There is clear evidence of learning after training for 600,000 steps. The mean episodic reward quickly becomes positive, while the mean episodic length increases to around 300 steps, as seen in Figure 4.17. At this point, both of them start to oscillate, which might be caused by a learning rate being too high.

Looking at the videos in Chapter A, the agent learns to follow the lane markings, although with significant oscillatory motion. It is, however, not yet able to detect when there are other vehicles in front, and almost every episode terminates with a collision. With the heavily reduced number of vehicles in the simplified environment, learning to detect them by training an encoder in the loop alongside the policies probably would take a lot more time than detecting the lanes. Cars are complex shapes, while lane lines are very simple. This is probably also affected by the use of a very simple feature extractor for the images, as discussed in Section 4.7.

4.7.3 Results for Run 2

Introducing the same learning rate schedule as in (4.5) results in a smoother convergence compared to run 1. After 800k steps, the mean reward is still increasing, along with the episode length. The reward per timestep, which can be seen in Figure 4.19c, has flattened out after about 200k steps. Most of the gains are, therefore, from the agent lasting longer and not the agent performing better for each step. Despite this, the performance is slightly higher than that achieved without the learning rate schedule, and there is still progress being made after 800k steps.

The Policy loss in Figure 4.19 seems to start increasing again fairly quickly before starting



(a) Mean episode length. **(b)** Mean episode reward.

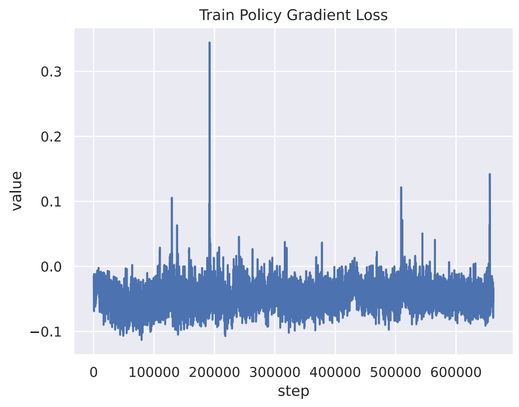
Figure 4.17: Rollout statistics during training run 1 in the simple environment using CNN encoder.

to decrease after the final learning rate reduction at 300k steps. This coincides well with the increase in episode length and the moment the agent starts driving forwards. It's common for the policy loss to increase during certain stages of training, especially when the agent encounters complex or challenging situations. It is, therefore, likely that the increase in policy loss comes as a result of the agent encountering unexplored areas of the observation space.

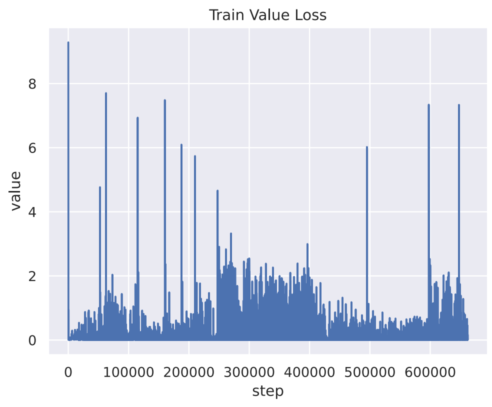
Looking at the videos, there is still a large amount of swerving, and the agent is still unable to avoid crashing into cars, just like in run 1, but the performance is significantly better than with the Transfuser encoder. The convergence is also significantly faster, likely due to the lower complexity of the encoder and the fact that the encoder parameters are trained, allowing the agent to quickly pick up on simple patterns in the image data. It is uncertain if this simple encoder could capture enough information - given enough training - to avoid colliding with other vehicles. A more complex CNN would likely be able to do this, but it is uncertain if training it with RL is feasible, and it would likely take much more time.

4.8 Discussion

In all the experiments where the Transfuser encoder was used, the agent failed to learn sufficient information about the environment. One possible contributing factor is the data used to train the encoder. The Transfuser training pipeline performs some data augmentation to simulate different car placements on the road. This does not seem to be sufficient, as the BEV estimate deteriorates when the vehicle position is slightly off the optimal line. Because of this, the RL agent may not be able to learn how to recover from mistakes and adjust its behav-



(a) Policy loss.



(b) Value loss.

Figure 4.18: Training losses for run 1 in the simple environment using CNN encoder.

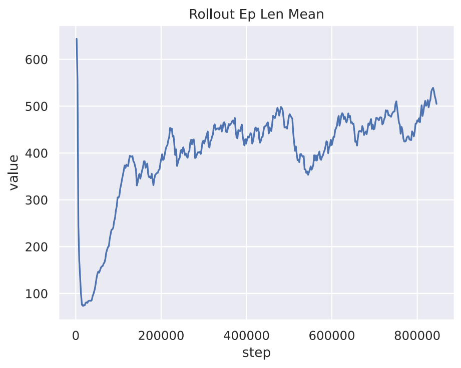
ior accordingly. This can severely impact the agent's performance, as it may not be able to learn properly in the early stages of training. Furthermore, the lack of diverse examples in the pretraining data can also limit the ability of the network to generalize.

4.8.1 Importance of Training Duration

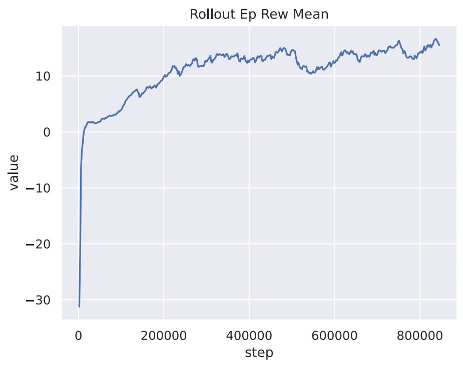
The relatively short training duration of one million steps might have also contributed to the suboptimal performance observed in the experiments. GRIAD was trained for 70 million steps, and this highlights the potential benefits of longer training durations in capturing the complexity of the Transfuser encoding and for learning complex behaviors. The training might not have been long enough for the policy to capture the complexities in the encoded vector. What has to be noted is the tendency of all the training runs to stagnate fairly quickly, which is aided by the needed reduction of the learning rate. It is uncertain if the continuation of the training while using the same setup would result in improved performance. Further investigation with extended training periods and different hyperparameters could provide valuable insights into the performance of RL agents using the Transfuser encoder.

4.8.2 Simplified Environment and Encoder

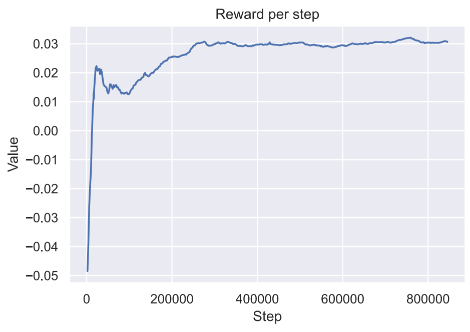
The most successful training runs were performed in a simplified environment, even when using the same setup and reward function as in the other runs. The biggest difference in complexity is the traffic density, which might indicate that the reward function is to punishing for collisions. In the simple environment, the agent is capable of gaining enough positive rewards, from getting to checkpoints, to prevent convergence to a suboptimal policy. Based on this, it



(a) Mean episode length.



(b) Mean episode reward.

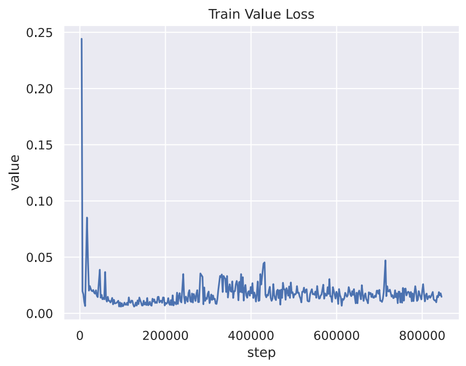


(c) Reward per step.

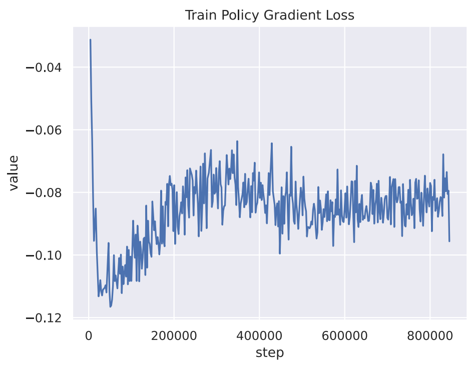
Figure 4.19: Rollout statistics for run 2 in the simplified environment using CNN encoder.

might be necessary to perform training in the early stages using simpler traffic scenarios than the ones on the CARLA leaderboard. Complexity could then be introduced gradually, with more complex traffic patterns and scenarios, until the agent is able to handle the full complexity of the CARLA leaderboard. This might help the agent learn the simple parts of the environment before being exposed to scenarios with very dense traffic. Other forms of agent pretraining might also be deployed, such as curiosity-based exploration, as discussed in [37].

Furthermore, given the relative success of the simpler encoder trained alongside the policy and value networks in the simplified environment, it raises the possibility of training a more complex encoder in a similar manner. By allowing the encoder to learn relevant features and representations directly from the environment, it may be able to capture more nuanced information and improve the agent's ability to detect and interact with other vehicles effectively. This would, however, decrease the interpretability of the entire system, as there would be no way to visualize what the encoder sees. This is a massive drawback, as interpretability for autonomous vehicles is essential for them to gain the trust of the general public. It also makes it



(a) Value loss.



(b) Policy loss.

Figure 4.20: Losses for run 2 in the simplified environment using CNN encoder.

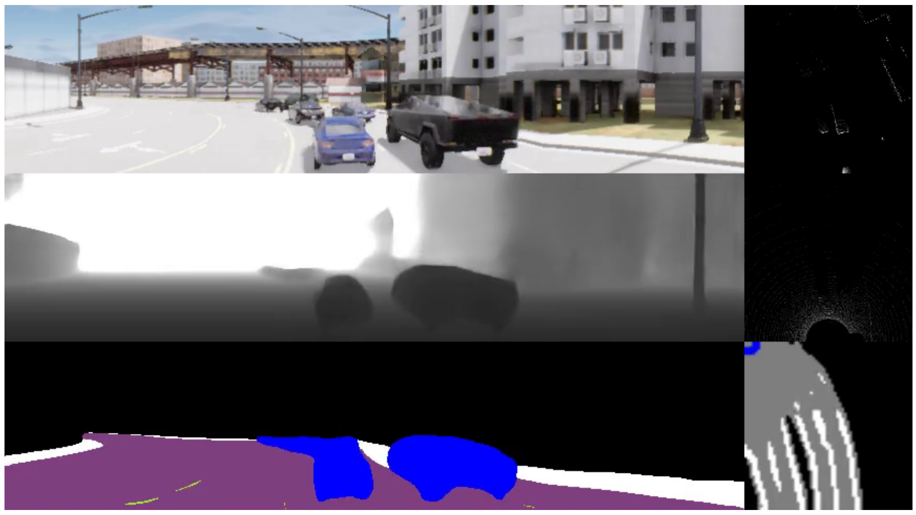
easier to run diagnostics to determine what goes wrong when something happens.

4.8.3 Impact of Expert Demonstrations

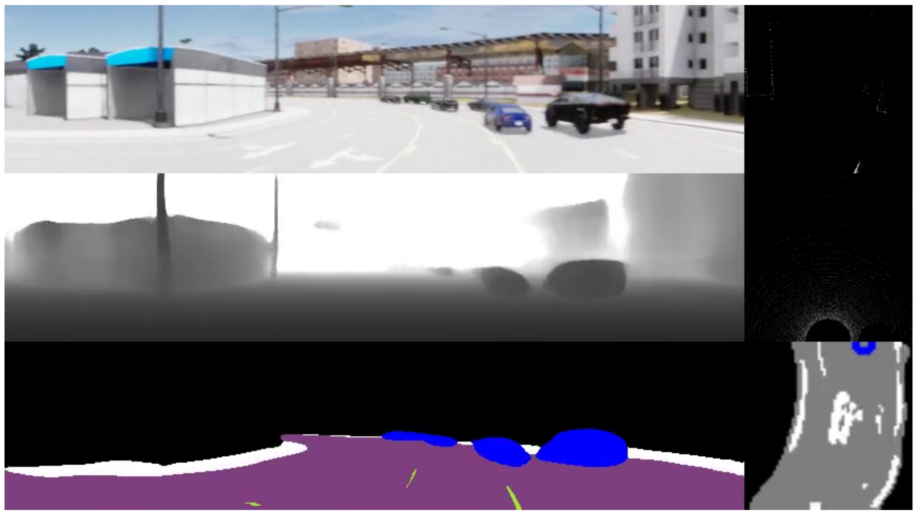
The results from training were insufficient to draw any conclusions regarding expert demonstrations. Standing still on the road is a better policy than intentionally driving off the road and shows more promise for improvement with more training if it had not been for the diverging weights in Figure 4.14. There was unfortunately not enough time to train GRIAD in the simplified environment, which might have given valuable insight. The time constraint again prevented further exploration of hyperparameters, which might have given better results, as the lessons learned from PPO might not be directly transferable to TD3.

4.8.4 Issues with the Transfuser Dataset Generation

Looking at the HD-map prediction from the Transfuser encoder, it is evident that the current method for generating data for pretraining the encoder has some issues for use in RL. In Section 4.8.4, the network is incapable of predicting a reasonable HD map when the agent is off-center and misaligned with the lane. Since all the data in Transfuser is generated using an expert agent, there are no example images where the vehicle is not perfectly positioned in the lane. The encoder network is therefore not exposed to images where the vehicle is angled weirdly in the lane, causing inconsistencies when the RL agent drifts off course. The training pipeline of Transfuser does perform some augmentation of the training images to try to mitigate this, but it is apparently not enough for the larger deviations seen during RL training. The issue becomes especially apparent during the early stages of training when the agent is still exploring the environment and has not yet learned how to navigate it properly. There might therefore be necessary to perform improvements to the Transfuser data generation pipeline if future experiments are performed using the Transfuser encoder.



(a) Good HD map predictions.



(b) Poor HD map predictions.

Figure 4.21: Bad and good predictions of the HD-map depending on the vehicle's location. When the images are taken from a vehicle not in an optimal position, the HD map (bottom right) prediction is way off, and the agent, therefore, has no idea about the road layout.

Chapter 5

Conclusions and Future Works

5.1 Conclusions

The focus of this thesis has been on the use of Reinforcement Learning (RL) in autonomous driving in the CARLA simulator, an open-source simulator for autonomous driving research. Other Reinforcement Learning systems in CARLA have used simple vision encoders. This thesis has investigated if introducing more complex vision encoders and sensor suites has the possibility of improving the performance of agents trained using Reinforcement Learning.

Several training setups for Reinforcement Learning have been tested using a transformer-based vision encoder as a basis for observations. The baseline training using Proximal Policy Optimization (PPO) gave unsatisfactory results, with the vehicle driving off the road and crashing straight away. At the same time, training showed signs of stagnation, showing no further progress was being made. Introducing expert demonstrations using General Reinforced Imitation for Autonomous Driving (GRIAD) did not show any significant improvements in performance. Instead, the vehicle ended up remaining stationary on the road without making any substantial progress.

Because of training stagnation and insufficient results, a hypothesis was made that complex scenarios caused the training to stagnate. A simplified environment was created to address these issues, with fewer cars and pedestrians to increase the performance potential of the agents. Two training runs were performed in this environment - one using Transfuser and one a simple Convolutional Neural Network (CNN) - both of which performed significantly better than previously. Using the Transfuser encoder, the vehicle was able to drive and follow the road. Neither agent was, however, able to avoid collisions with other vehicles. The improvement confirmed the hypothesis that the complexity of the other environment hindered the training process. Reducing the complexity of scenarios in early training could help improve training convergence.

While the simple encoder provided better results than the Transfuser encoder in the simple environment, there is not enough information to conclude if a more complex vision encoder could significantly impact the performance of a trained Reinforcement Learning (RL) agent with only one million training steps. Both agents are improving at 1 million steps, and using a more complex encoder would likely result in longer convergence times for the policy. This is also the case for GRIAD, where the performance difference from the baseline agent is too small.

Based on the results gathered from training, a definitive conclusion can not be drawn on the hypothesis that improving the vision encoder would improve the performance of autonomous driving agents trained using RL since one million training steps was not enough to show the potential of any of the methods attempted. Therefore, longer training runs need to be performed for any definitive conclusions to be drawn. However, there were discovered some issues with the current training setup that has to be addressed in future training runs.

5.2 Reflections on the Project Execution

Looking back at the project, it is clear that the complexity of the problem was underestimated. If the simplified environment had been used from the start, there would have been a higher probability of getting good results. In this case, longer training runs could have been performed, which would have given valuable insight. It would also have saved a lot of time by not implementing and testing the OpenAI gym interface for the scenario runner, which could have been used more productively. Generally speaking, it is useful to test things out in a simplified setting first. It is unfortunate that the realization that the high traffic density would negatively impact training to such a degree did not come earlier.

The original scope of the thesis was also too wide. Introducing GRIAD was just a distraction, taking away time that could have been spent elsewhere. More time should have been spent at the beginning of the project, discussing with supervisors and setting a more reasonable scope.

The biggest area of learning during this project was how quickly the difficulty of training Reinforcement Learning algorithms increased with environment complexity. The difficulty seems to increase faster than for other machine learning tasks such as computer vision. Because of this. a better understanding of how training Reinforcement Learning algorithms should be performed and what factors have the biggest impact on the performance has been acquired.

5.3 Future Work

Future work of other students should be focussed on furthering the final goals of NAPLab, deploying their algorithms on a physical car. Some effort should also be invested into adressing issues presented in this thesis. Based on this, the following areas should be focused on:

- Start utilizing sensor calibration in the simulator
- Focus on explainability
- Improve synthetic data generation

These points are elaborated upon below.

5.3.1 Camera Calibration

The overall goal of NAPLab is to apply the algorithms to a real vehicle. Domain adaptation can be a challenging problem, and camera calibration in a simulator plays a crucial role in achieving domain adaptation to the real world. Simulators are widely used for training and testing autonomous systems before deploying them in real-world environments. However, there is often a significant domain gap between simulated and real-world data, leading to poor performance when deploying models trained solely in simulation. Camera calibration, in this context, refers to the process of aligning virtual camera properties in the simulator with the characteristics of the physical cameras used in the real world. This calibration step is important for achieving accurate perception and reliable decision-making in real-world scenarios. Camera calibration is unfortunately not supported natively in CARLA, and the way the simulator distorts the images have no easy translation to regular distortion models. In July 2022, Soliman et al. [38] released an extension to CARLA to support camera calibration. This should be explored by NAPLab and integrated into future systems.

5.3.2 Explainable Reinforcement Learning (XRL)

Reinforcement Learning (RL) systems are by nature not very explainable. With the goal of deploying the system on NAPLab's own vehicle, some work should be put into explainability, especially if an RL approach is used. It provides transparency and interpretability, allowing humans to understand the system's decision-making process [39]. This transparency enhances safety, accountability, and collaboration between humans and autonomous vehicles. XRL enables effective intervention, investigation of accidents, compliance with regulations, and smooth coordination on the road, making autonomous driving more reliable and trustworthy. Several surveys of XRL methods have been performed [39, 40], which can be used as a starting point for an investigation.

5.3.3 Improvement of Dataset Generation

As discussed in Section 4.8.3, some improvements should be made to the Transfuser data generation pipeline. Perturbations should be introduced to the position and orientation which the sensor data is captured from.

A simple solution would be to randomly sample a transformation consisting of a lateral translation and yaw rotation applied to the camera and lidar. Since the lidar and cameras in Transfuser have the same coordinate along the length of the car, this transformation applied to both sensors is equivalent to a transformation being applied to the vehicle, except that the vehicle can now be positioned normally, making control easier.

If recovery trajectories are desired, this is not sufficient. There would, in this case, need to be perturbations of the vehicle's position - both laterally on the road and in the vehicle's yaw. It is uncertain if the current expert agent would be able to recover from these kinds of perturbations, so a study would first have to be done into this, and improvements made if necessary.

Bibliography

[1] T. Pietrasik, *Road traffic injuries*, Jun. 2022. [Online]. Available: `https://www.who.int/news-room/fact-sheets/detail/road-traffic-injuries`.

[2] National Traffic and Safety Administration, *Critical reasons for crashes investigated in the national motor vehicle crash causation survey*, Feb. 2015. [Online]. Available: `https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812115`.

[3] National Traffic and Safety Administration, *2016 fatal motor vehicle crashes: Overview*, Oct. 2017. [Online]. Available: `https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812456`.

[4] [Online]. Available: `https://getcruise.com/`.

[5] M. Toromanoff, E. Wirbel, and F. Moutarde, "End-to-end model-free reinforcement learning for urban driving using implicit affordances," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2020.

[6] D. Chen, V. Koltun, and P. Krähenbühl, "Learning to drive from a world on rails," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, Oct. 2021, pp. 15 590–15 599.

[7] R. Chekroun, M. Toromanoff, S. Hornauer, and F. Moutarde, *Gri: General reinforced imitation and its application to vision-based autonomous driving*, 2021. DOI: `10.48550/ARXIV.2111.08575`. [Online]. Available: `https://arxiv.org/abs/2111.08575`.

[8] K. Chitta, A. Prakash, B. Jaeger, Z. Yu, K. Renz, and A. Geiger, "Transfuser: Imitation with transformer-based sensor fusion for autonomous driving," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–18, 2022. DOI: `10.1109/TPAMI.2022.3200245`.

[9] H. Shao, L. Wang, R. Chen, H. Li, and Y. Liu, "Safety-enhanced autonomous driving using interpretable sensor fusion transformer," in *6th Annual Conference on Robot Learning*, 2022. [Online]. Available: `https://openreview.net/forum?id=qzMY915hCYX`.

[10] *Carla leaderboard*. [Online]. Available: `https://leaderboard.carla.org/leaderboard/`.

[11] G. Kumichev, *The inductive bias of ml models, and why you should care about it*. [Online]. Available: `https://towardsdatascience.com/the-inductive-bias-of-ml-models-and-why-you-should-care-about-it-979fe02a1a56`.

[12] S. Kummervold, *Training of autonomous driving agents in simulated environments*, 2022.

[13] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–80, Dec. 1997. DOI: `10.1162/neco.1997.9.8.1735`.

[14] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, *Empirical evaluation of gated recurrent neural networks on sequence modeling*, 2014. DOI: `10.48550/ARXIV.1412.3555`. [Online]. Available: `https://arxiv.org/abs/1412.3555`.

[15] Jeblad. "The lstm cell." (2018), [Online]. Available: `https://commons.wikimedia.org/wiki/File:The_LSTM_Cell.svg`.

[16] G. Chevalier. "Gated recurrent unit." (2018), [Online]. Available: `https://commons.wikimedia.org/wiki/File:Gated_Recurrent_Unit,_base_type.svg`.

[17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30, Curran Associates, Inc., 2017. [Online]. Available: `https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf`.

[18] W. Wang, H. Bao, L. Dong, J. Bjorck, Z. Peng, Q. Liu, K. Aggarwal, O. K. Mohammed, S. Singhal, S. Som, and F. Wei, *Image as a foreign language: Beit pretraining for all vision and vision-language tasks*, 2022. DOI: `10.48550/ARXIV.2208.10442`. [Online]. Available: `https://arxiv.org/abs/2208.10442`.

[19] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," in *International Conference on Learning Representations*, 2021. [Online]. Available: `https://openreview.net/forum?id=YicbFdNTTy`.

[20] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 1877–1901. [Online]. Available: `https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf`.

[21] J. Cheng, L. Dong, and M. Lapata, "Long short-term memory-networks for machine reading," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, Austin, Texas: Association for Computational Linguistics, Nov. 2016, pp. 551–561. DOI: `10.18653/v1/D16-1053`. [Online]. Available: `https://aclanthology.org/D16-1053`.

[22] D. Chen and P. Krähenbühl, "Learning from all vehicles," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2022, pp. 17 222–17 231.

[23] P. Wu, X. Jia, L. Chen, J. Yan, H. Li, and Y. Qiao, *Trajectory-guided control prediction for end-to-end autonomous driving: A simple yet strong baseline*, 2022. DOI: `10.48550/ARXIV.2206.08129`. [Online]. Available: `https://arxiv.org/abs/2206.08129`.

[24] S. J. Russell and P. Norvig, *Artificial Intelligence: A modern approach*. Pearson Education Limited, 2022.

[25] Megajuice. "Gated recurrent unit." (2017), [Online]. Available: `https://commons.wikimedia.org/wiki/File:Reinforcement_learning_diagram.svg`.

[26] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, *Proximal policy optimization algorithms*, 2017. arXiv: `1707.06347 [cs.LG]`.

[27] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *Proceedings of the 35th International Conference on Machine Learning*, J. Dy and A. Krause, Eds., ser. Proceedings of Machine Learning Research, vol. 80, PMLR, Oct. 2018, pp. 1587–1596. [Online]. Available: `https://proceedings.mlr.press/v80/fujimoto18a.html`.

[28] *Idun documentation*. [Online]. Available: `https://www.hpc.ntnu.no/idun/`.

[29] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 1–16.

[30] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, “Stable-baselines3: Reliable reinforcement learning implementations,” *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021. [Online]. Available: `http://jmlr.org/papers/v22/20-1364.html`.

[31] J. Schulman, O. Klimov, F. Wolski, P. Dhariwal, and A. Radford, *Proximal policy optimization*, Jul. 2017. [Online]. Available: `https://openai.com/research/openai-baselines-ppo`.

[32] T. Fossen, *Handbook of Marine Craft Hydrodynamics and Motion Control*. Wiley, 2021, ISBN: 9781119575030. [Online]. Available: `https://books.google.no/books?id=tCQqEAAAQBAJ`.

[33] M. Andrychowicz, A. Raichuk, P. Stańczyk, M. Orsini, S. Girgin, R. Marinier, L. Hussenot, M. Geist, O. Pietquin, M. Michalski, S. Gelly, and O. Bachem, “What matters for on-policy deep actor-critic methods? a large-scale study,” in *International Conference on Learning Representations*, 2021. [Online]. Available: `https://openreview.net/forum?id=nIAxjsniDzg`.

[34] *Vectorized environments¶*, 2022. [Online]. Available: `https://stable-baselines3.readthedocs.io/en/master/guide/vec_envs.html`.

[35] D. E. Knuth, *The Art of Computer Programming Volume 2: Seminumerical Algorithms*. Addison-Wesley, 1998.

[36] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, and et al., “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, no. 7540, pp. 529–533, 2015. DOI: `10.1038/nature14236`.

[37] Z. Xie, Z. Lin, J. Li, S. Li, and D. Ye, *Pretraining in deep reinforcement learning: A survey*, 2022. arXiv: `2211.03959 [cs.LG]`.

[38] A. Soliman, F. Bonardi, D. Sidibé, and S. Bouchafa, “IBISCape: A simulated benchmark for multi-modal SLAM systems evaluation in large-scale dynamic environments,” *Journal of Intelligent & Robotic Systems*, vol. 106, no. 3, p. 53, Oct. 2022, ISSN: 1573-0409. DOI: `10.1007/s10846-022-01753-7`. [Online]. Available: `https://doi.org/10.1007/s10846-022-01753-7`.

[39] E. Puiutta and E. M. S. P. Veith, “Explainable reinforcement learning: A survey,” in *Machine Learning and Knowledge Extraction*, A. Holzinger, P. Kieseberg, A. M. Tjoa, and E. Weippl, Eds., Cham: Springer International Publishing, 2020, pp. 77–95, ISBN: 978-3-030-57321-8. [Online]. Available: `https://arxiv.org/abs/2005.06247`.

[40] S. Milani, N. Topin, M. Veloso, and F. Fang, *A survey of explainable reinforcement learning*, 2022. arXiv: 2202.08434 [cs.LG].

Appendix A

Videos and Images From Simulator

A.1 Videos From Training Runs

Videos from training can be found here:
`https://drive.google.com/drive/folders/1KCh2P4Hs9xtiPO6PZnE78eVqTWNLY4s_?usp=sharing`

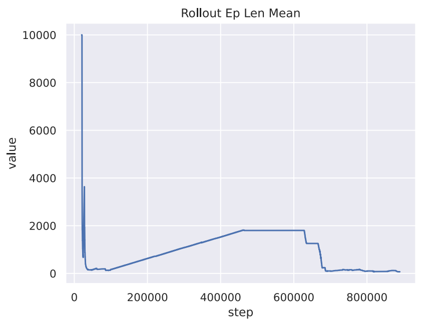
Appendix B

RL Training Graphs

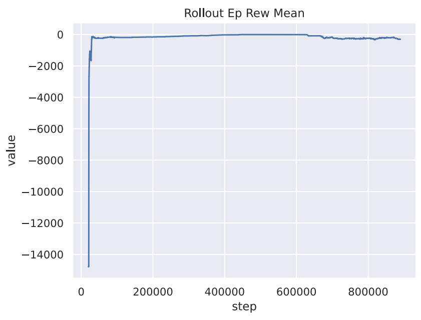
This section contains the training plots for all the reinforcement learning runs performed, even the ones not mentioned in the report. This is for readers who want to look at the plots not discussed earlier

B.1 PPO

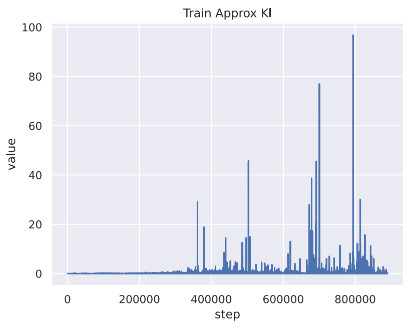
B.1.1 Run 1



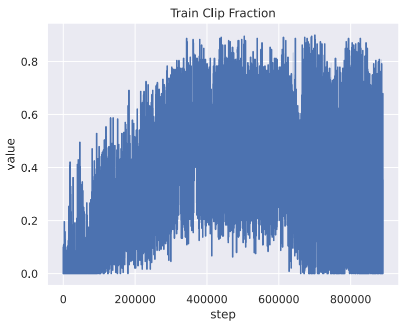
(a) Episode length.



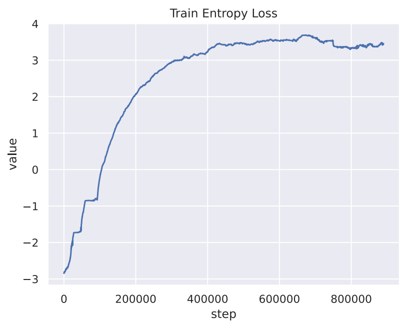
(b) Episode rewards.



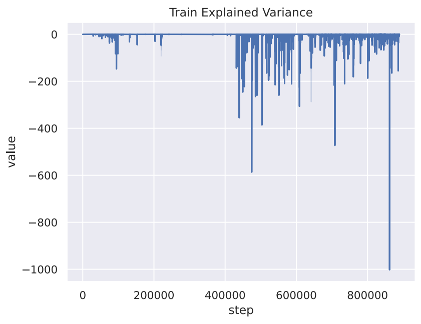
(c) KL divergence.



(d) Clip fraction.



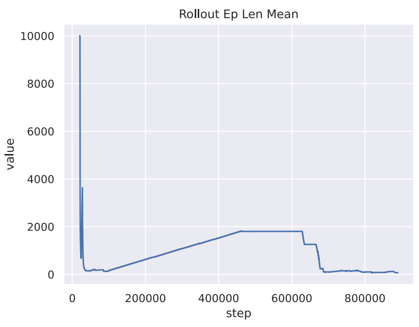
(e) Entropy loss.



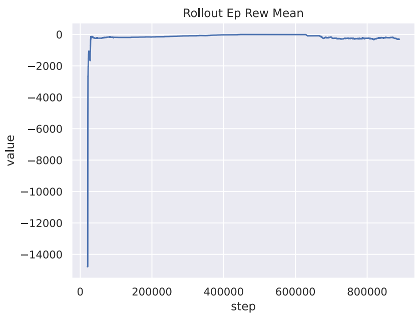
(f) Explained variance.

Figure B.1: Scalar values for run1 using PPO.

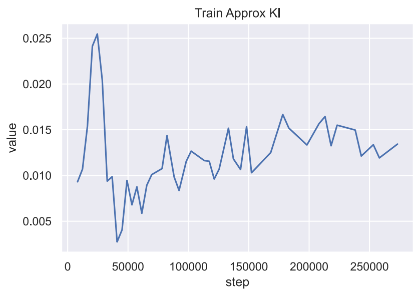
B.1.2 Run 2



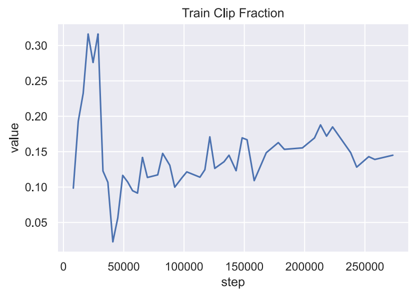
(a) Episode length.



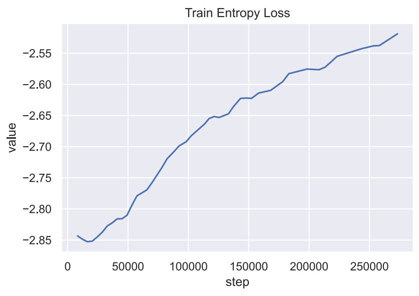
(b) Episode rewards.



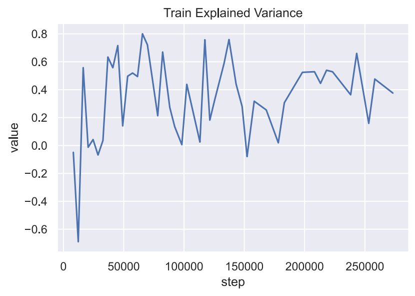
(c) KL divergence.



(d) Clip fraction.



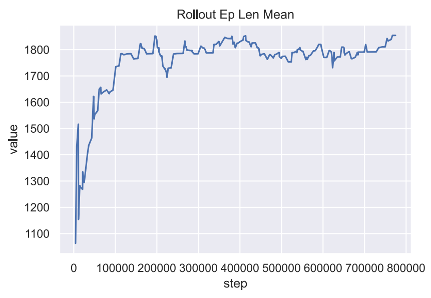
(e) Entropy loss.



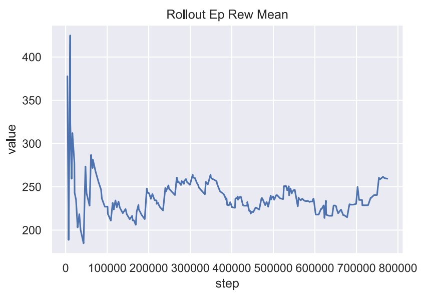
(f) Explained variance.

Figure B.2: Scalar values for run2 using PPO.

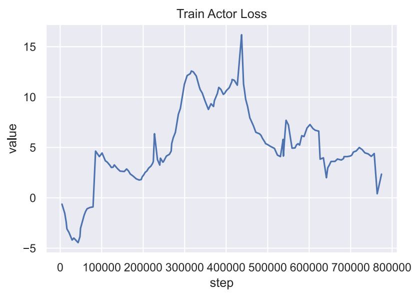
B.2 General Reinforced Imitation for Autonomous Driving (GRIAD)



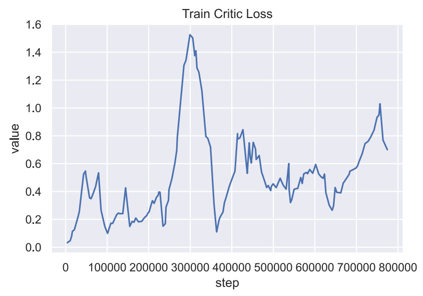
(a) Episode length.



(b) Episode rewards.



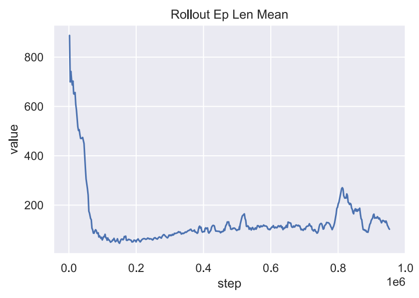
(c) Actor loss.



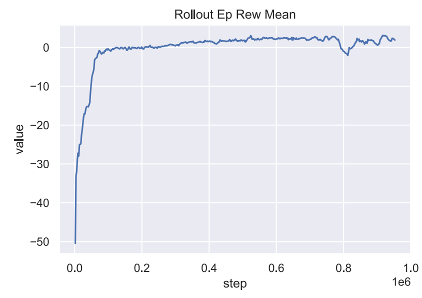
(d) Critic loss.

Figure B.3: Scalar values for run1 using PPO.

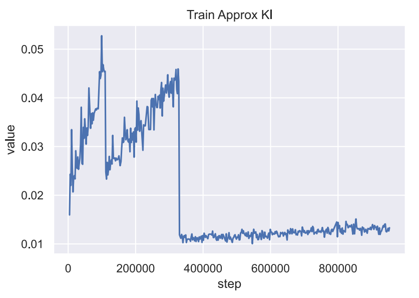
B.3 Simple environment Transfuser



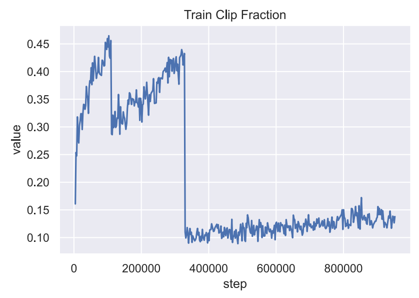
(a) Episode length



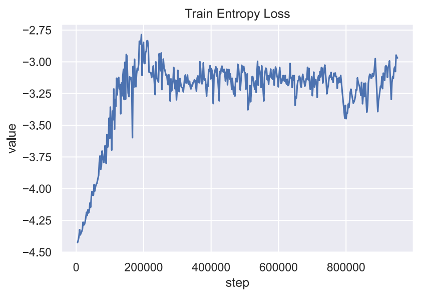
(b) Episode rewards



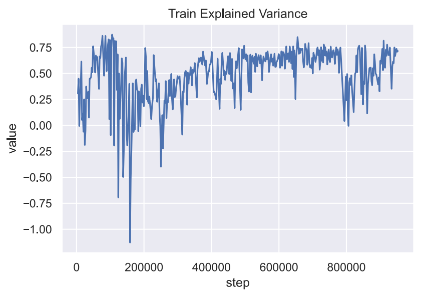
(c) KL divergence.



(d) Clip fraction.



(e) Entropy loss.

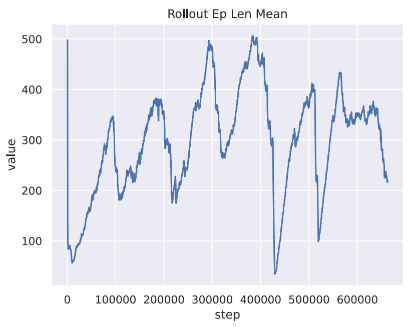


(f) Explained variance.

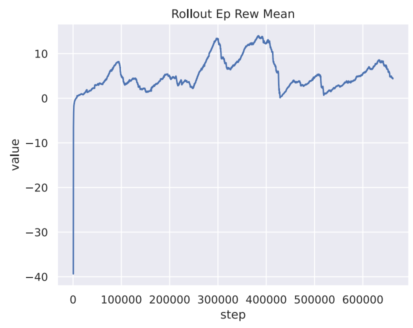
Figure B.4: Scalar values for the simple environment using Transfuser encoder.

B.4 Simple environment CNN

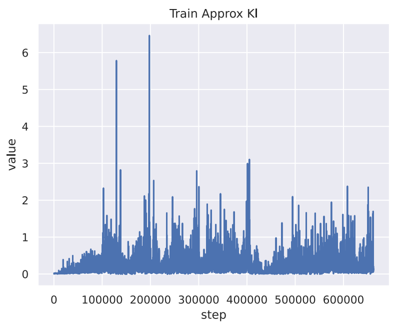
B.4.1 Run 1



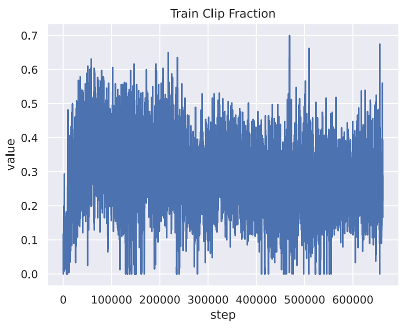
(a) Episode length.



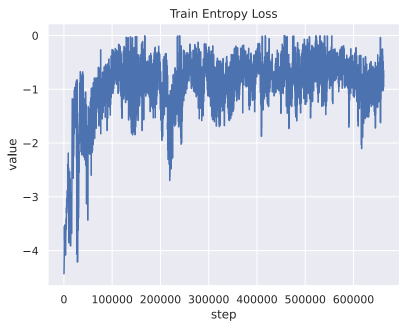
(b) Episode rewards.



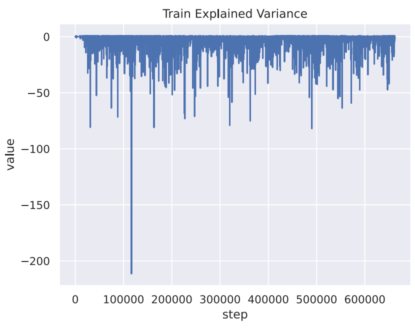
(c) KL divergence.



(d) Clip fraction.



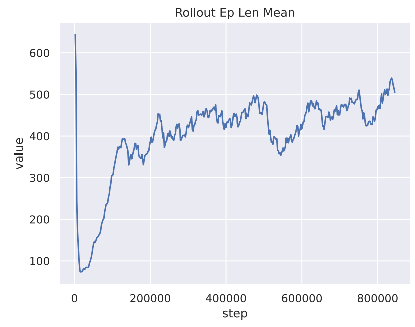
(e) Entropy loss.



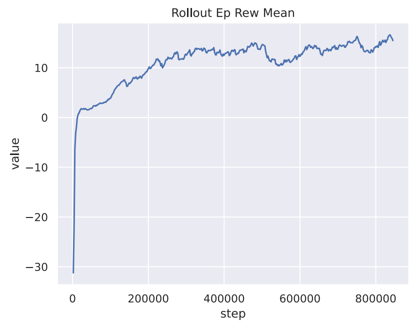
(f) Explained variance.

Figure B.5: Scalar values for run1 in the simple environment using CNN encoder.

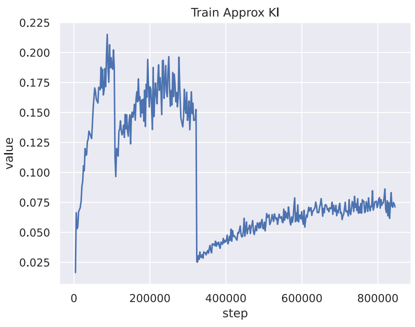
B.4.2 Run 2



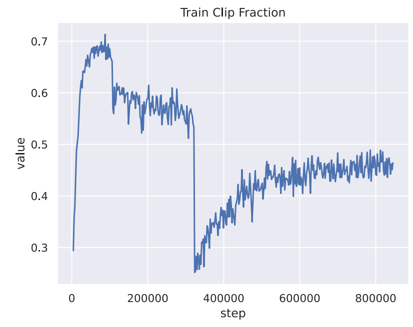
(a) Episode length.



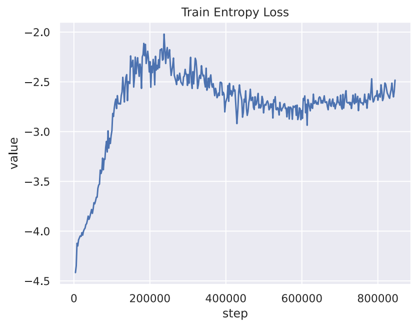
(b) Episode rewards.



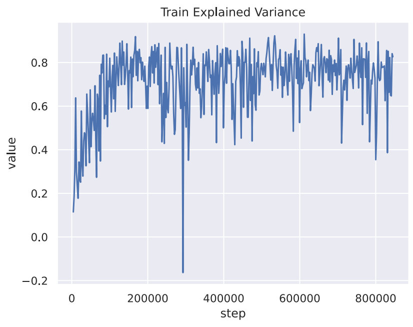
(c) KL divergence.



(d) Clip fraction.



(e) Entropy loss.



(f) Explained variance.

Figure B.6: Scalar values for run 2 in the simple environment using CNN encoder.