Zurich University of Applied Sciences
Bachelor Thesis in Computer Science

End-to-End Deep Reinforcement Learning for Autonomous Racing Dynamics

Fabian Ulrich Tobias Wehrli

Industrial Partner
Zurich UAS Racing

Supervisor
Prof. Dr. Jasmina Bogojeska

External Supervisor
Prof. Dr. Johannes Schneider

June 07, 2024

Declaration of Originality

I hereby declare that I have written this thesis independently or together with the listed group members.

I have only used the sources and aids (including websites and generative AI tools) specified in the text or appendix. I am responsible for the quality of the text and the selection of all content and have ensured that information and arguments are substantiated or supported by appropriate scientific sources. Generative AI tools have been summarized by name and purpose.

Any misconduct will be dealt with according to paragraphs 39 and 40 of the General Academic Regulations for Bachelor's and Master's Degree courses at the Zurich University of Applied Sciences (Rahmenprüfungsordnung ZHAW (RPO)) and subject to the provisions for disciplinary action stipulated in the University regulations.

Place, Date:
Winterthur, 07.06.2024
Thundorf, 07.06.2024

Student names:
Fabian Ulrich
Tobias Wehrli

Abstract

With the rise in computational power over the recent years, Reinforcement Learning has not only become more accessible but has also shown substantial improvements in performance and applicability. In the Formula Student racing environment, autonomous driving systems have relied primarily on modular pipelines incorporating perception, SLAM, trajectory planning, and controls. These approaches are limited by the technical complexity and significant manpower required to integrate multiple subsystems effectively. This thesis explores the integration of deep Reinforcement Learning with Proximal Policy Optimization and a one-stage object detector into a unified end-to-end system. The object detector processes input images and depth data to extract the positions of track-delimiting cones. These inputs, combined with other sensor data, are fed into the policy network that directly maps them to optimized driving actions. The open source driving simulator CARLA was modified to include a realistic model of the electric 2024 Zurich UAS Racing vehicle. Furthermore, tracks based on the disciplines outlined in the Formula Student Germany rules were modeled in a simulated environment, facilitating the training of a Reinforcement Learning agent. Although evaluated only in simulation, the results are promising. The agent successfully completed the acceleration race without crashing while performing on a level comparable to human performance. In the trackdrive event, the agent accomplished an average completion rate of 72%. This suggests that with continued development and enhancements, this research can be effectively employed by the Zurich UAS Racing team to enhance their competitive performance in future competitions.

Keywords: Reinforcement learning, Deep learning, Proximal Policy Optimization, Autonomous driving, Formula Student, CARLA simulator, Object detection, Racing

Acknowledgements

We would like to thank our supervisor, Prof. Dr. Jasmina Bogojeska, for her guidance throughout our entire development process. Her continued support and optimism helped us overcome the more challenging times of our project, ultimately leading to its success. We would also like to thank the Zurich UAS Racing team for making this project possible and welcoming us to the team with open arms.

Contents

1. Introduction

This chapter outlines the motivation behind this work. It introduces the Zurich UAS Racing team and their vehicles, followed by an explanation of the structure of the driverless team. Additionally, the requirements are presented, leading to the definition of three primary goals for this work.

1.1. Motivation

The domain of autonomous driving has been advancing rapidly over the past few years. Large, established industry leaders such as Tesla, Google, and Uber are competing to design and build the most reliable fully autonomous driving systems [1]. Traditionally, these systems have used a mix of machine learning to perceive and understand their environment, paired with a more hand-crafted approach to path planning and vehicle control [2]. Due to a substantial increase in computing power and the availability of large datasets, the field is shifting from using separate algorithms for specialized problems to more end-to-end (E2E) deep learning approaches, where a single deep neural network processes raw sensor data and directly controls the vehicle [2]–[4]. A prominent example includes Tesla's Full Self-Driving (FSD) Version 12, which reportedly replaced substantial portions of traditional coding with a deep learning model in 2024 [5]. Deep Neural Networks trained on large datasets promise a more robust and consistent driving performance and are more resilient to edge cases than hand-crafted systems [6].
This thesis aims to explore the viability and potential of a deep-learning-based E2E approach to autonomous driving in the domain of the Formula Student competition.

1.2. Zurich UAS Racing

This thesis was developed in close collaboration with Zurich UAS Racing (ZUR), the resident Formula Student team at ZHAW. After their launch in 2019, ZUR has since built and competed with three fully functional electric vehicles, with the fourth vehicle scheduled to compete in the 2024 season.

Figure 1.: The latest ZUR cars from 2023 (left) and 2024 (right)

Driverless Team

The Driverless Team at Zurich UAS Racing is dedicated to the driverless cup division of the Formula Student competitions and is responsible for all hardware and software required to implement such autonomous driving systems. The team is organized into the groups of Simulation, Simultaneous Localization and Mapping (SLAM), Trajectory Planning, Controls and the newly formed E2E group. To develop robust autonomous racing systems, continued improvement of the existing implementations, as well as the exploration of novel approaches, is required. The results and experiences gained from this thesis will be critical for deciding how the team will allocate resources for future development.

1.3. Requirements

The ZUR team currently utilizes an established, modular approach to self-driving. The objective of this thesis is to evaluate an E2E deep learning approach and acquire the necessary knowledge for its potential use in future competitions. It is crucial that this work be conducted in a manner that ensures the approach can be practically applied to the real car. This implies that the algorithm must not rely on unrealistic sensor data, such as the exact center of the road or other privileged simulation data.
Upon successful completion of the thesis, the results will be integrated into the driverless architecture and tested in real-world conditions.

1.4. Task definition

Given the official task description, the following three goals were established after careful consideration and discussions with the members of ZUR:

1. Build and test a simulated environment to perform end-to-end reinforcement training.
2. Create a working prototype agent that is conceptually applicable to the physical car.
3. Evaluate and determine if end-to-end neural networks are a viable approach for the ZUR team.

1.5. Thesis overview

The thesis begins with a brief overview of the Formula Student competition, followed by an introduction to Reinforcement Learning. Using this background, the methodology and implementation of the learning is introduced. The methodology covers the structure of the simulation, how the Reinforcement Learning is configured and how these components interact. The subsequent chapter on results presents the outcomes of the training. Following this, the discussion chapter evaluates the results across the three Formula Student disciplines, analyzes the implementation, and considers potential approaches for future performance improvement. The thesis concludes by defining future objectives and outlining the associated challenges.

To provide a visual representation of the training and evaluation process, a complementary video has been created and is available on YouTube [7]. The video provides a brief description of the thesis, followed by video recordings taken during the training and evaluation phases of the Reinforcement Learning agent.

2. Related Work

One of the initial endeavors in automated driving was the EUREKA Prometheus Project, conducted across Europe by leading automotive manufacturers in the 1990s. At the time, it was hypothesized that advancements in computing power over the following two decades would be adequate to realize autonomous driving capabilities [8]. As computational capacity has increased, different approaches were developed. In the following chapter, established methods of autonomous driving are detailed and challenges of related E2E projects are discussed.

2.1. Classical Approaches to Autonomous Driving

Historically, autonomous driving challenges have been addressed by evaluating sensor data using a modular setup of algorithms [3]. This approach, referred to in this work as the 'classical approach', typically uses a pipeline structure that decomposes the difficult task of driving into sequential phases: Perception, Localization and Mapping (SLAM), Planning and Decision Making, and Vehicle Control. The classical pipeline is illustrated on figure 2.

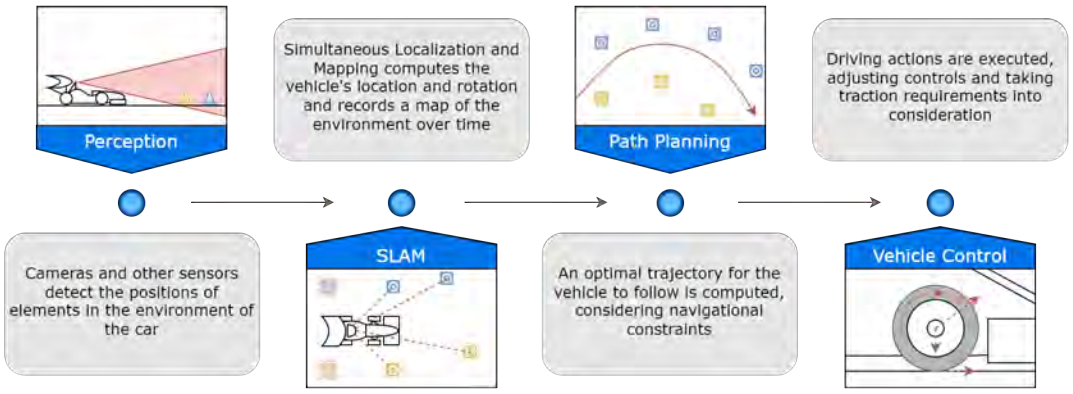


Figure 2.: Classical self-driving pipeline

Perception relies on technologies like cameras and LiDAR to gather information about the vehicle's surroundings. SLAM (Simultaneous Localization and Mapping) constructs a map of the captured environment while tracking the vehicle's location [9]. Path Planning and Decision Making involves algorithms to determine an optimal path and make decisions based on the output of the previous steps. The vehicle controller takes the computed values and translates them into instructions for the car. This includes calculating how much torque is needed so the tires do not lose traction and then sending this value to the motor controllers.

Despite its structured nature, modular systems face significant challenges. The implementation of such a system demands a substantial development effort and is generally designed to perform only one specific task. Due to their rule-based nature, these systems are often ineffective when presented with situations that were not considered during implementation [10]. Moreover, the pipeline structure is more sensitive to errors, as uncertainties and errors in one module are propagated through the entire system. Dynamic obstacles like moved cones also pose significant problem to these static systems, as eventualities have to be explicitly accounted for [11].

A substantial limitation encountered by ZUR's driverless team is the complexity of the current classical system, which complicates the transfer of knowledge among team members. The transient nature of university project teams, where members depart upon completing their studies, exacerbates this issue as it leads to loss of know-how. Additionally, computational constraints pose a critical challenge, as all algorithms are required to operate directly on the vehicle.

2.2. End-to-End Approaches

E2E driving systems generate driving outputs directly from sensor input. This is typically achieved by using neural networks that learn a representation of the driving task. This approach is detailed in figure 3.

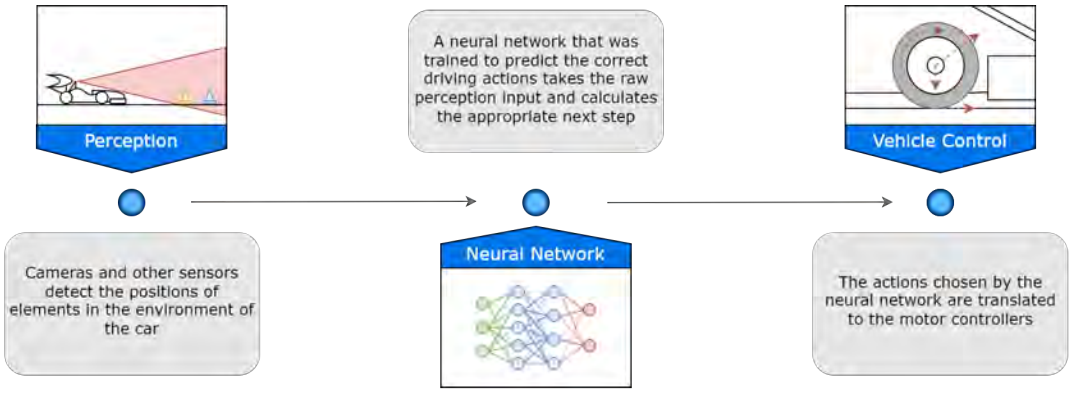


Figure 3.: End-to-End pipeline

The earliest of these systems is ALVINN, a 3-layer neural network that was trained to follow roads by utilizing a video camera with a range finder [12]. Over the last few years, more research on the topic started to emerge [13], [14]. This can be attributed to the high computational requirements to train and operate such a system and the rapid advancements in artificial intelligence over the last decade.

2.2.1. Imitation Learning

Imitation Learning (IL) is subset of machine learning where a model is trained to imitate an expert driver, typically a human.

In 2016, researchers of the NVIDIA Corporation presented PilotNet, a single Convolutional Neural Network (CNN) capable of mapping camera sensor data directly to vehicle steering controls [15]. A dataset containing 72 hours of videos, with corresponding steering control inputs for each image, was used to perform supervised learning. Various augmentation techniques were used to improve generalization, which allowed the model to perform effectively on previously unseen road sections. PilotNet demonstrated the viability of the end-to-end approach using a prototype that was tested both in a simulated environment and on a physical car. Over the next four years, the system was refined further and the various experiments were documented in [16]. Using a single camera, the system was able to drive an average of 500km on highways before human intervention was required, highlighting the potential of end-to-end systems over modular systems. However, the researchers also emphasize the importance of diagnostic tools, given how sensitive the learning process is to properly labeled data [16].

In [17], researchers of UC Berkley presented a modified Imitation Learning approach that introduces a safety controller to improve collision avoidance. Similar to PilotNet, the model was trained on a previously gathered dataset of driving experiences with corresponding controls. They conclude that the driving performance improves significantly and further describe that Reinforcement Learning can be used to further fine tune the model. By combining these paradigms, training time can be improved significantly, while also allowing the model to be trained on more varied experiences [18].

The mentioned projects and conducted surveys indicate that IL has become a viable alternative to the modular approach for complex domains [10]. The primary challenge of these approaches is their need for extensive, labeled datasets that contain a large variety of driving experiences.

2.2.2. Reinforcement Learning

Reinforcement Learning (RL) algorithms are also used to learn autonomous driving tasks. Instead of having to provide a dataset of collected experiences, the model is able to explore the environment independently, where its only feedback is the reward function. In [14], the authors trained an agent to drive a realistic rally game using an asynchronous actor-critic framework with promising results. A similar approach was used in [19] to evaluate multiple RL algorithms for racing a F:SAE (Society of Automotive Engineers) vehicle, which was subsequently tested on real world tracks with an average completion rate of 60%. Both projects highlight the difficulty of more complex perception when training using RL. In [10] over 250 papers regarding E2E autonomous driving were analyzed. The authors concluded that the gradients obtained in RL are not sufficient to train the perception layers of deeper networks. This hurdle is often addressed using a mediated perception approach, where the perception task is separated from the driving task by means of object detection or other techniques [10]. While this can add complexity and introduce new errors, network size can be reduced drastically, making the approach possible without significant computational capacity. Nevertheless, RL is very sample-intensive and, due to the trial-and-error strategy of learning, requires realistic simulated environments. However, they appear to generally transfer better into real environments (sim2real) than other E2E approaches [20].

3. Background

This chapter introduces the necessary theoretical background knowledge for the presented implementation and discussion. The relevant Formula Student disciplines are detailed, the 2024 ZUR vehicle and its sensors are described, and an introduction to reinforcement learning and object detection is provided.

3.1. Formula Student

Formula Student is the world's largest engineering competition, involving university teams across the globe [21]. The participating teams design, build and compete with custom formula-style race cars. The competition is divided into three categories: Internal Combustion Vehicles, Electric Vehicles and the Driverless Cup. This work specifically focuses on the Electric Vehicle and the associated Driverless Cup, as these are the only categories ZUR competes in. The following descriptions of disciplines are based on the rules published by Formula Student Germany (FSG) [22] as they have become the de facto standard for teams competing in Europe.
The competitions involve a series of static and dynamic events designed to test the performance of the competing teams and vehicles. Static events include engineering design, cost planning and a business plan presentation. The dynamic events focus on different aspects of the car's capabilities and are explained in the following sections. Each event can earn points based on formulas in the FSG rules.

3.1.1. Acceleration

The acceleration event is a test of the car's ability to quickly gain speed. Competitors race along a straight course, aiming to achieve the fastest acceleration from a standing start and thus reaching the finish line quickest. After crossing the finish line, the vehicle must come to a complete stop in the stop area. The orange triangle contains timekeeping equipment.

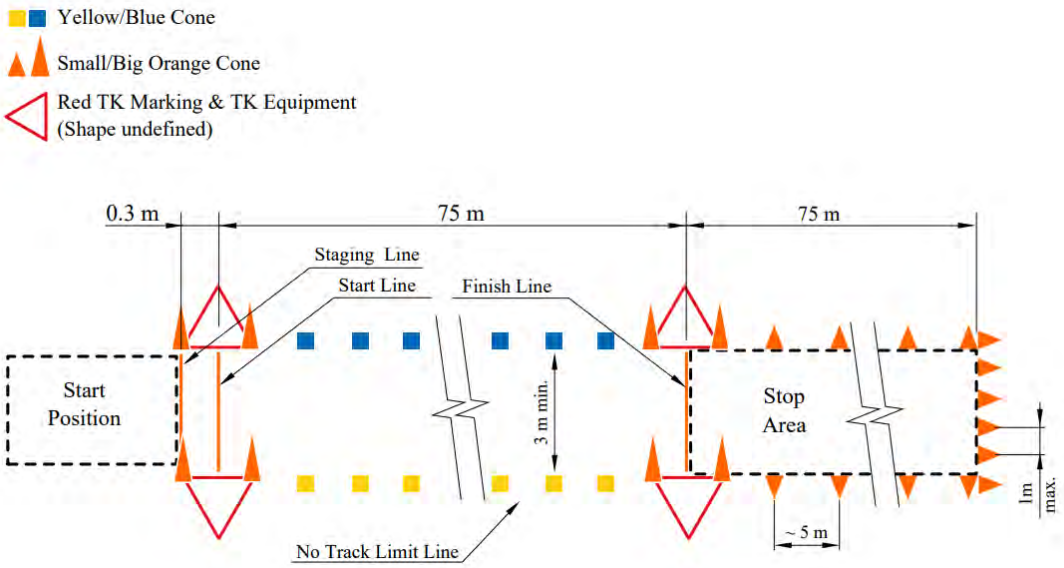


Figure 4.: Illustration of the acceleration event[1]

3.1.2. Skidpad

The skidpad event evaluates the vehicle's cornering ability on a tight figure-eight course. This event challenges the car's handling and stability in cornering conditions. The vehicle starts in the middle and completes two loops of the right hand part, followed by two loops of the left hand part. After finishing the figure-eight, the vehicle must come to a stop in the stop area.

[1]Image Source: FSG, 2024 Competition Handbook v1.1. Retrieved from: `www.formulastudent.de/fileadmin/user_upload/all/2024/important_docs/FSG24_Competition_Handbook_v1.1.pdf`

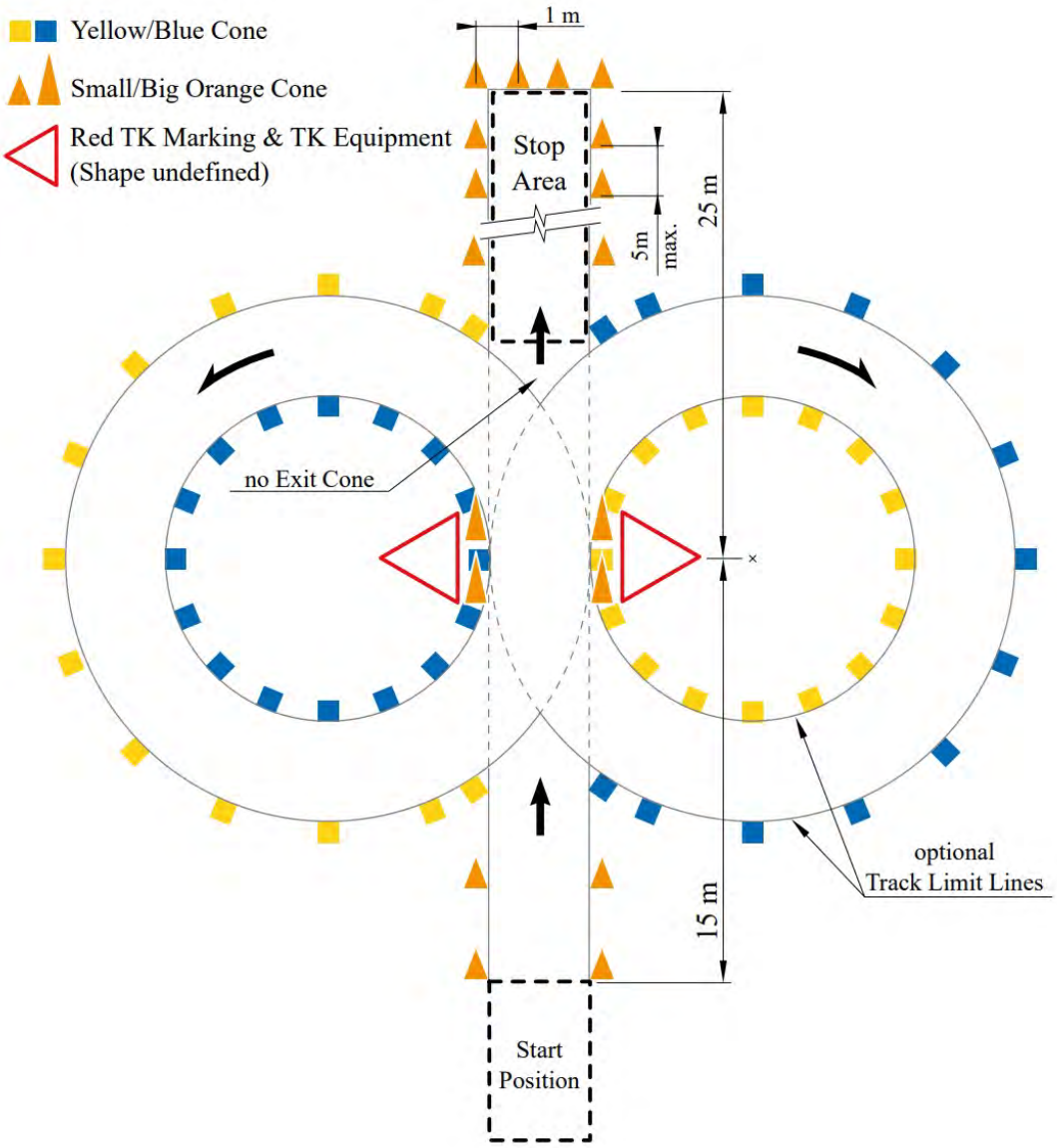


Figure 5.: Illustration of the skidpad event[1]

3.1.3. Trackdrive

The trackdrive event is a ten lap race around a circuit. This event tests not only the vehicle's durability and efficiency but also the driver's ability to remain attentive over an extended race period. It is important to note that according to the FSG rules [22], the track is unknown to the competing teams until the competition is held. Traditionally, the vehicles dedicate the first lap to map out the track and use the subsequent laps to gradually increase speed and improve their times.

[1]Image Source: FSG, 2024 Competition Handbook v1.1. Retrieved from: `www.formulastudent.de/fileadmin/user_upload/all/2024/important_docs/FSG24_Competition_Handbook_v1.1.pdf`

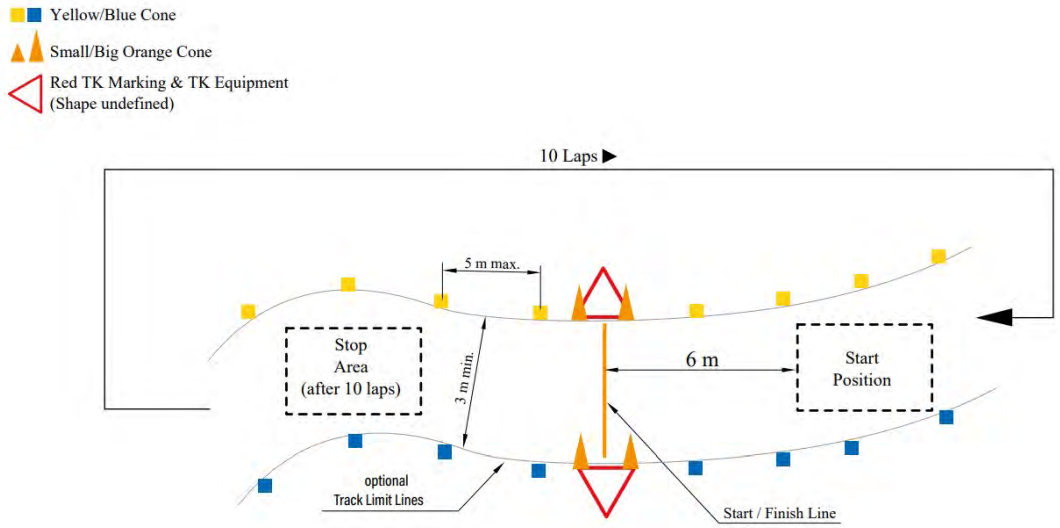


Figure 6.: Illustration of the trackdrive event[1]

The rules [22] also reference an autocross event, but for driverless purposes it can be viewed as a longer version of the trackdrive event. The main distinction to the trackdrive event is that the teams do not have the opportunity to map an accurate representation of the track to refine their timings in subsequent runs.

[1]Image Source: FSG, 2024 Competition Handbook v1.1. Retrieved from: `www.formulastudent.de/fileadmin/user_upload/all/2024/important_docs/FSG24_Competition_Handbook_v1.1.pdf`

3.2. The 2024 ZUR race car

For the 2024 season of Formula Student, the ZUR team built a new racing vehicle, as illustrated on figure 7. The various components and sensors relevant to this work are detailed in the sections below.

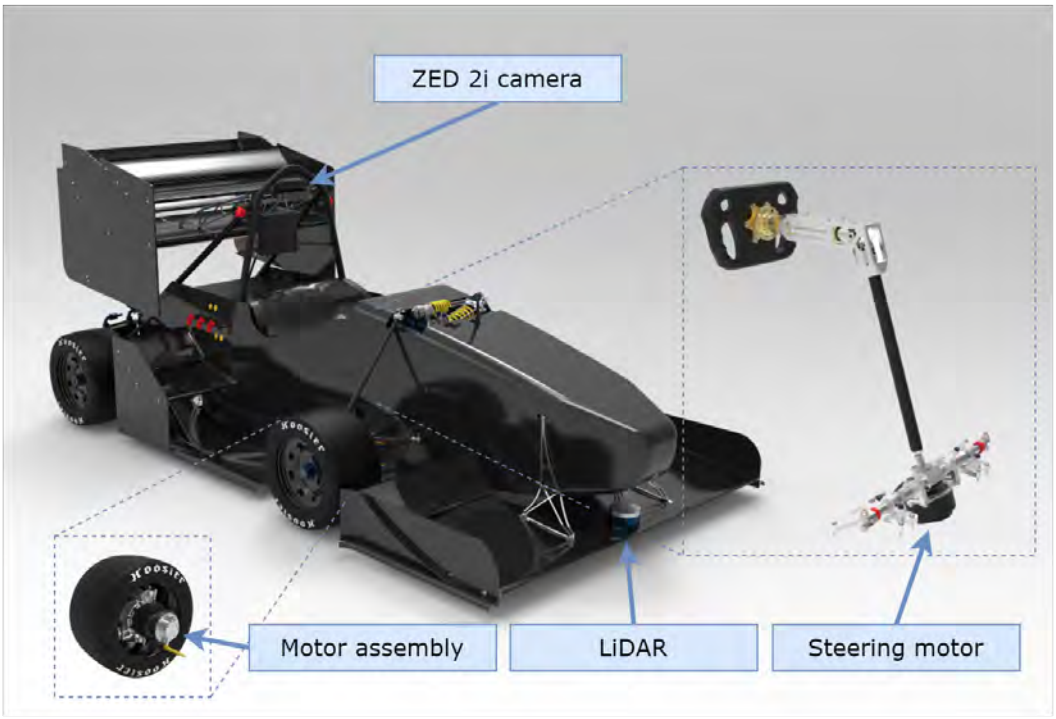


Figure 7.: Illustration of the components in the 2024 ZUR race car

3.2.1. NVIDIA Jetson

The vehicle is equipped with an NVIDIA Jetson Xavier, a powerful AI computing platform capable of executing 30 trillion operations per second (TOPS) [23], which handles on-board computations. Nonetheless, compared to larger computing platforms, the processing power available is limited, making the efficiency of the algorithms running on this device crucial for optimal performance. During races, the vehicles are not allowed to communicate with external sensors or computing platforms [22].

3.2.2. ZED stereo camera

The primary sensor for autonomous driving is the ZED 2i stereo camera. This wide-angle 3D camera is designed for outdoor applications and includes multiple built-in sensors, such as an inertial measurement unit, barometer, and magnetometer. It senses depth using stereo vision and computes the resulting grayscale image using neural networks [24]. A key advantage of this self-contained system for the ZUR team is that these computations are performed on the device itself, reducing the computational load on the Jetson. The camera is mounted within the roll cage, directly above the driver's head, pointing forward.

3.2.3. Actuators

The car is equipped with four 80 kW motors, each housed within the wheels and connected to a planetary gearbox. The motors are water cooled for maximum efficiency. For steering, a separate, smaller motor is mounted below the steering assembly. The exact communication method between Jetson and motor control inputs is outside of the scope of this work.

3.2.4. LiDAR

The front wing of the car has a slot for a LiDAR sensor. This sensor contains an array of lasers that spin, measuring accurate distances to objects in a 360 degree field of view. However, at the time of writing this thesis, it was uncertain whether this sensor would be available. Therefore, this sensor will be excluded in this work. Nevertheless, it is important to note that CARLA has the capability to emulate a LiDAR sensor, making a future integration possible.

3.3. Reinforcement Learning

In supervised learning, a model learns to predict values by comparing predictions to data that was labeled by a supervisor, from which a loss gradient can be calculated to update the model. More specialized domains, such as autonomous driving for a Formula Student race car, often lack freely available, labeled datasets that would allow for supervised learning. Furthermore, while supervised models can potentially exceed the performance of the human supervisors in some aspects, they are fundamentally constrained by the diversity and range of the labeled experiences available to them [25]. Additionally, it is crucial to develop decision-making that optimizes for both immediate and long-term rewards. This is highlighted especially by the domain of autonomous driving, as actions, such as increasing speed, may only later reveal to be detrimental. Because of these constraints, supervised learning is not suitable for achieving robust performance in this dynamic and highly sequential environment without labeled data [26][27].

3.3.1. Introduction

Reinforcement Learning (RL) is a machine learning paradigm where a decision-maker, called an agent, learns to interact with an environment by choosing what actions to take in a particular state. For each action it receives feedback in the form of a scalar value, indicating how good the chosen action is, called the reward. Using this trial-and-error learning strategy, the agent iteratively refines its actions based on the rewards received to maximize the total reward over time, as displayed in figure 8. Thus, an optimal strategy for decision-making can be developed without the need for labeled data [27].

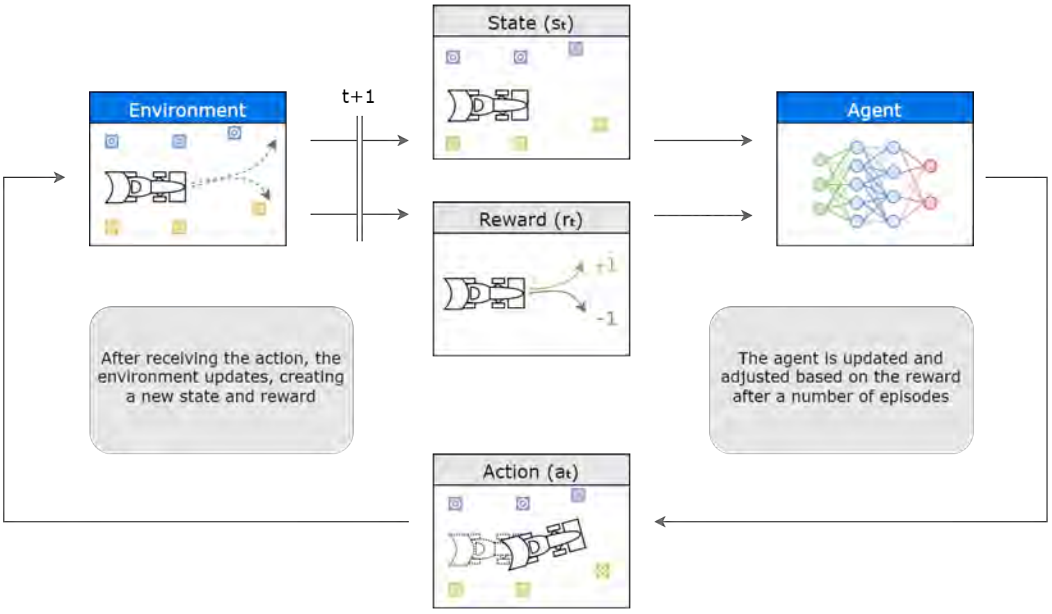


Figure 8.: Illustration of the RL loop

3.3.2. Concepts and Terminology

To gain a foundational understanding of Reinforcement Learning, it is necessary to know the following definitions and concepts.

Agent

An agent refers to an entity that makes decisions by interacting with its environment. The agent's goal is to learn the best strategy, or policy, to maximize the reward it receives over time. In autonomous driving, this would be the car driving through the world. [27]

Environment

The environment represents the world through which the agent moves. By interacting with the environment, the agent receives a scalar reward for every step that is taken, indicating how good the executed action was in that state. For the task of autonomous driving this might be a racetrack or road. [27]

Episodes and Trajectories

An episode is the time frame in which the agent performs a sequence of actions, moves to new states, and receives rewards. It starts in an initial state and ends in a terminal state. The concept of an episode is central to episodic tasks where the interaction between the agent and the environment is naturally broken down into segments of experiences that end in a specific event, such as failing the task or reaching a goal. For the autonomous racing task an episode might start with the agent at the starting line and ends when the agent reaches the finish line or crashes. [27]

The trajectory is the specific sequence of states, actions, and rewards experienced by the agent from the start to the end of an episode [27]:

$$S_0, A_0, R_0, S_1, A_1, R_1, S_2, A_2, R_2... \tag{3.1}$$

Reward and Return

The reward r is a scalar feedback signal given to the agent to evaluate the effectiveness of an action taken in a particular state.

The cumulative reward that an agents receives starting with reward r_t from the current state s_t at time t until the end of the episode is called *return* G_t.
It is formally defined as

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \ldots + \gamma^{N-1} r_{N-1} \tag{3.2}$$

where $\gamma \in [0, 1]$ is the discount factor and N is the total number of steps in an episode. As shown, the hyperparameter γ is used to scale future rewards to determine their importance for the current decision-making process. A higher γ therefore indicates that the action a_t at time t has a high impact on future rewards. A lower γ indicates that long-term rewards are less important than those in the immediate future. Since the agent optimizes for return, γ impacts how much the agent considers future consequences of its current action. [27]

Policy

The policy π is a function that maps a given state of the environment to an action that is to be taken. Formally, a policy π is defined as a function $\pi : S \to A$, where S is the set of all possible states and A is the set of all possible actions. Policies may also be stochastic, where instead of an action, the probability of taking a given action a in state s is calculated instead. In that case the policy takes the form $\pi(a|s)$.
The primary goal of reinforcement learning is to find an optimal policy π^* that maximizes expected return for every given state. [27]

Value functions

Value functions estimate the expected return for an agent in a particular state. They can be divided into state-value functions V and action-value functions Q. The state-value function $V^\pi(s)$ represents the expected return when starting from state s and following the policy π. The action-value function $Q^\pi(s, a)$ represents the expected return when starting from state s, taking action a and then following the policy. [27]

3.3.3. Partial and Full Observability

In RL environments, the extent to which an agent can observe the underlying state of the environment significantly influences the complexity of the decision-making process.

Fully Observable Environments

In a fully observable environment, the agent has access to all the relevant information about the current state of the environment. For example, in the game of chess, both

players can see the entire board and all the pieces on it. Since there is no hidden information, this makes the chess board a fully observable environment [27].

Partially Observable Environments

In partially observable environments, the agent does not have complete information about the current state. Instead, it has access only to observations that provide partial information about the state [27]. A practical example of this is autonomous racing, where the vehicle must make decisions based on limited and sometimes uncertain sensory data. This makes the decision-making process more difficult, as the agent may not always have all the information to choose the optimal action. To address these challenges, incorporating a history of past observations to form a pseudo-state has shown a performance improvement in partially observable environments [28].

3.3.4. Markov Decision Processes

To encapsulate a reinforcement learning problem formally, the prevalent method is to define it as a Markov Decision Process (MDP). This is done in order to capture the dynamics of the environment and to systematically identify all possible states and actions [27]. An MDP provides a mathematical framework to model decision-making in an environment where outcomes are partially under the control of the agent and partially random.
An MDP is defined as a 4-Tuple (S, A, P_a, R_a) as follows:

- S is the set of all possible states in which an agent or environment can exist, called the state space. The state space may be infinite and also continuous (e.g. when sensor data is part of the state).
- A is the set of actions that an agent can perform. This may depend on the current state, in which case A_s is used instead.
- $P_a(s, s') = \mathbb{P}(s_{t+1} = s' | s_t = s, a_t = a)$ describes the probability that, when in state s, the action a will lead to the new state s'.
- $R_a(s, s')$ is the reward that an agent receives when transitioning from state s to s'

The MDP further assumes the Markov property: Given a state s where action a is taken, the resulting state is independent of any previous states or actions taken. Because of this, decision-making can occur without any memory of previous states, simplifying the process [27].

In partially observable environments, the MDP becomes a **PO**MDP (Partially Observable Markov Decision Process), where the agent only has access to a partial observation of the state instead of the state itself. Consequently, the classic Markov property is modified: optimal decision-making may depend on a history of observations to maintain and update a belief state [27]. While the specifics of POMDPs are outside the scope of the thesis, the relevance of an observation history for the purposes of this project are considered in section 4.3.2.

3.3.5. Exploration-Exploitation Trade-off

As discussed, RL is a trial-and-error approach. During training, the agent has to explore new strategies (known as exploration) and simultaneously refine those that have shown to earn the highest rewards (exploitation). To explore new strategies the agent has to intentionally choose actions that, to its best knowledge, are not optimal, in hopes of finding a strategy that yields even higher return in the long-term. If no new strategies are explored, the agent will likely get stuck in a local optimum. However, if a promising strategy is not allowed to be refined, the agent will never achieve optimal performance. Therefore, there is a direct trade-off when choosing to favour one objective over the other, known as the exploration-exploitation trade-off. It's crucial to find a suitable balance between exploring new, random actions and allowing the agent to follow its learned policy [27].

For discrete action spaces, a simple and popular method is "epsilon-greedy", where $\epsilon \in [0, 1]$ denotes the probability of choosing a random action instead of the one suggested by the policy. To train the agent, the ϵ parameter starts with a high value and is decayed down near 0 towards the end of the training, gradually moving from favouring exploration to only exploitation [27].
Epsilon-greedy merely serves as a simple example for exploration. The specific method of exploration used in this project is described in section 4.3.4.

3.3.6. Value optimization and policy optimization

Reinforcement learning can be further subdivided into value optimization and policy optimization. The two concepts are later combined to form the class of actor-critic algorithms.

Value estimation using Monte Carlo methods

In value optimization the optimal policy is computed by estimating the return that can be obtained in a given state. The state-value function $V^\pi(s)$ is the expected return when starting from state s and following the policy π. It can therefore be represented recursively using the Bellman Expectation Equation:

$$V^\pi(s) = \mathbb{E}_\pi[r_{t+1} + \gamma V^\pi(S_{t+1})|S_t = s] \tag{3.3}$$

However, this function is unknown and must be learned during training. The goal is to find the optimal function $V^*(s)$ that best approximates $V(s)$.

Starting with $V^\pi(s) = 0$ for every state, the estimation of the value function can then be updated after every episode by calculating the difference between the expected return $V^\pi(s)$ and the actual observed return G for every state in the trajectory[27].

$$V^\pi_{k+1}(s) \leftarrow \mathbb{E}[V^\pi_k(s) + \alpha(G - V^\pi_k(s))] \tag{3.4}$$

Where k is the current iteration of the estimated value function and α the learning rate. This method of updating the state-value function, where an entire trajectory must be completed first, is known as Monte Carlo Estimation. Note that other methods like Dynamic Programming or Temporal Difference Learning are not covered here.

Policy optimization

In policy optimization the step of estimating the expected return to decide on an action is skipped. Instead, the policy function is updated and optimized directly. The goal is to find the optimal parameters (weights and biases of a neural network) θ that maximize the expected return. In the context of policy gradient methods, this is called the policy objective function $J(\theta)$ [27].

$$J(\theta) = \mathbb{E}_{\pi_\theta}[\sum_{t=0}^{\infty} \gamma^t r_t] \tag{3.5}$$

In supervised learning this is where the loss function would be used to compute a gradient and perform a policy update. Since in RL there is no labeled data to compute a loss, a different method of computing the gradient must be used. The policy gradient theorem provides a way to compute the gradient of the expected return with respect to the policy parameters ∇_θ, meaning gradient ascent methods can be used to optimize the policy.

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) \cdot Q^{\pi_\theta}(s,a)] \tag{3.6}$$

This gradient is estimated by collecting samples from the trajectories that the policy generates. Similar to supervised learning, a batch of trajectories is collected, which then allows the gradient of the policy objective function to be estimated. Using the gradient, this leads to the update rule of the policy parameters θ as follows:

$$\theta_{k+1} \leftarrow \theta_k + \alpha \nabla_\theta J(\theta_k) \tag{3.7}$$

Where α is the learning rate and k the current iteration of the policy.

3.3.7. Actor-Critic Methods and Advantage

Actor-critic approaches combine both value estimation and policy optimization and optimize both the expected return of a policy (the actor) and an estimation of the value function (the critic). The actor updates the policy parameters using the gradient provided by the critic, which has shown to be more stable and efficient than pure value or policy optimization approaches on their own [29][27].

To achieve this, actor-critic methods introduce the concept of advantage A [30]. The advantage function $A^\pi(s,a)$ measures the benefit of taking the action a in comparison to the average of all possible actions in the state s. It therefore provides an estimate of what the extra benefit of performing an action is, compared to the already learned behaviour. It is formally defined as the difference between the action-value and the state-value function:

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s). \tag{3.8}$$

Using the calculated advantage, the updates to the policy can be scaled in proportion. This stabilizes the learning of the policy and softens disruptive and destructive changes [30].

3.3.8. Overview of RL Algorithms

A wide variety of RL algorithms exist, most of which are designed to tackle a specific type of problem based on the characteristics of the action and state spaces involved. The

suitability of any given algorithm depends greatly on whether these spaces are discrete or continuous. Figure 9 displays an overview of several common RL algorithms, grouped by their environment types [31].

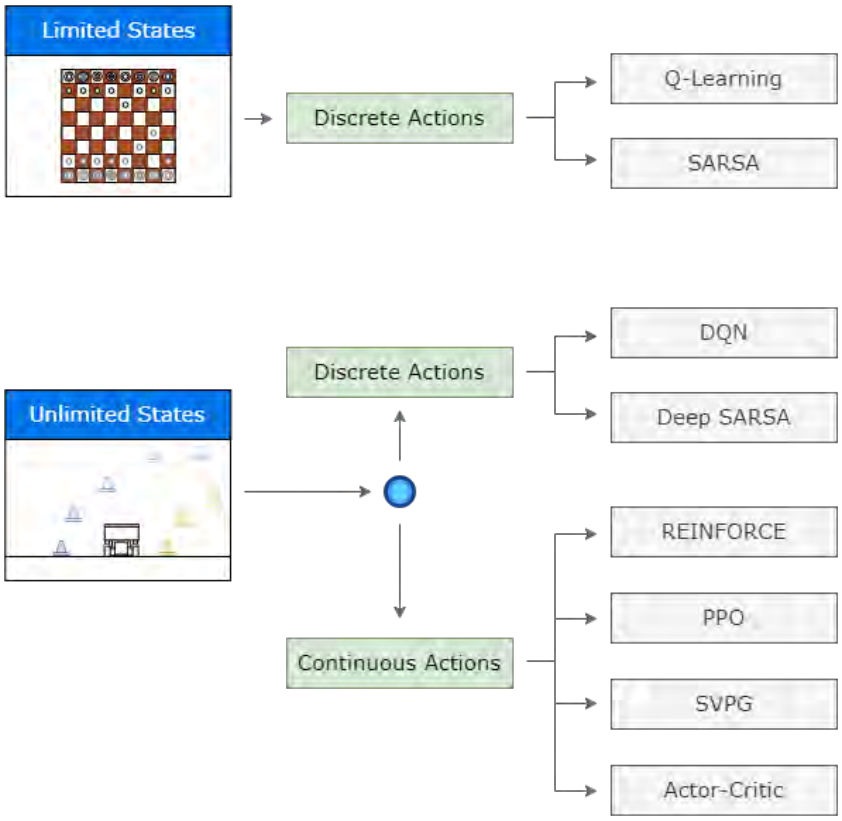


Figure 9.: Graph of RL algorithms

3.3.9. Proximal Policy Optimization

Proximal Policy Optimization (PPO) is a popular RL algorithm that aims to address several challenges in learning stability. It can be categorized as both an actor-critic and a policy gradient algorithm and is suitable for continuous and discrete action spaces. PPO is less sensitive to hyperparameters, which allows for a more stable and predictable training process. This is achieved by clipping the objective function, which allows avoiding the potential for large, destructive updates to the policy. It is also considered easier to implement and has a lower computational overhead when compared to similarly performing algorithms like Trust Region Policy Optimization (TRPO). [27]

Clipped objective function

To implement these improvements to the learning stability, PPO's objective function adds a clipping term to the probability ratio, which is calculated as follows:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{3.9}$$

This leads to the clipped surrogate objective function:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]. \tag{3.10}$$

where $\hat{A}_t$ is the estimated advantage at time t and ϵ is a hyperparameter, usually a small positive number (e.g. 0.2) that determines how much the probability ratio is allowed to deviate from 1 before being clipped. Using this clipping term, policy updates are kept within a reasonable range, while still allowing for advantage to scale the step size of the updates. [27]

3.4. Object detection

Object detection is the task of identifying the location and type of objects within an image. The outputs of an object detection algorithm are bounding box coordinates and dimensions (x, y, h, w) of the detected objects with corresponding class labels for each box, illustrated in figure 10. In this example all cones in the image are detected and assigned either class "blue cone" or "yellow cone".

Figure 10.: Object detection example with cones used in formula student races[1]

3.4.1. One-stage vs Two-stage object detectors

The object detection task can be split into two separate machine learning tasks:

1. Regression - finding the object instances and their bounding box coordinates.
2. Classification -s assigning a class to each previously detected box.

Two-stage detectors

Two-stage detectors, such as R-CNN and its variants (Fast R-CNN, Faster R-CNN), split the object detection task into two stages. In the first stage, proposals for potential bounding boxes are generated, typically using a region proposal network (RPN), which can be trained separately. In the second stage these proposals are refined by performing bounding box regression and classifying each refined region using a separate network [32].

[1]Image Source: fsoco-dataset.com

One-stage detectors

One-stage detectors, like YOLO (short for "You Only Look Once") and SSD (Single Shot Multibox Detector), shorten the process by combining the bounding box prediction and object classification into a single stage. Bounding box coordinates and class probabilities are predicted directly from the full images in one evaluation of the network. This approach tends to be faster as it eliminates the need for a separate proposal generation stage [33].

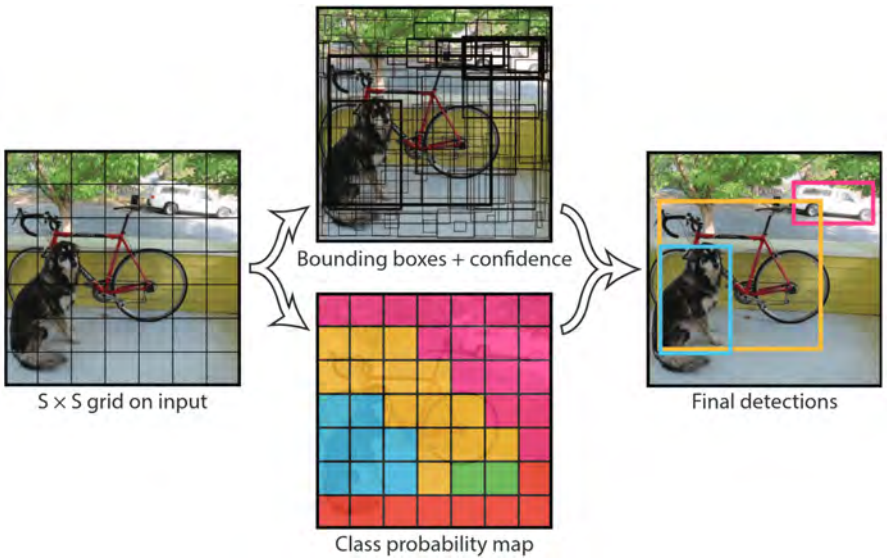


Figure 11.: Illustration of one-stage object detection as demonstrated by YOLO 2016 [33]

3.4.2. YOLOv8

YOLOv8 is the latest, stable iteration of YOLO model family. It provides state-of-the-art performance for various computer vision tasks, including a one-stage object detection model. This model is particularly well-suited for real-time detection tasks due to its ability to maintain high frame rates while still achieving competitive accuracy rates. A comprehensive command line interface (CLI) is also provided, which allows users to train and tune models for various dataset formats. [34]

4. Methodology

In the following chapter, the considerations and structure of the implementation is detailed. The simulation setup and its interaction with the training process is described, followed by the configuration details of the reinforcement learning process.

4.1. Simulator Setup

Training the agent on a physical vehicle in a real-world environment is impractical for obvious reasons. Consequently, creating a virtual equivalent of the real-world competition environment is an essential prerequisite for this thesis and the execution of RL. Creating a simulation environment from scratch is not feasible. Due to this, after evaluating multiple simulators, CARLA was chosen due to its extensive documentation, high fidelity in simulating real-world conditions, and robust support for autonomous driving research [35].
CARLA is an open-source simulator for autonomous driving research which is based on Unreal Engine. It is a computationally intensive simulation environment, with a focus on realism and accurate physics. CARLA exposes a python API that can be used to spawn and control vehicles. Furthermore, the API allows for creating sensors like an RGB camera or LiDAR and reading the values produced by those sensors in python. It is important to note that the unreal physics engine is non-deterministic, which means that the exact same actions can lead to a different outcome. The simulator was configured to run at exactly 60 frames per second as this provides a robust and repeatable frame rate, ensuring consistent performance and accurate evaluation of the agent's actions. [35]
To train the agent, an equivalent to the real-world competition environment is needed. The CARLA engine was built from source and assets and maps were tailored to mimic the Formula Student races. It is crucial that the tracks honor the FS regulations [22], which is why special attention was given to ensure they conform to the rules. For each of the events, a map was created that allowed the training of multiple agents simultaneously.

4.1.1. Acceleration Map

The simplest of the environments, acceleration, supports 20 simultaneous agents. The map that was built consists of a track for each agent. To simulate real-world errors, some of these tracks include minor inconsistencies, such as moved or rotated cones. Two of these tracks are depicted in figure 12.

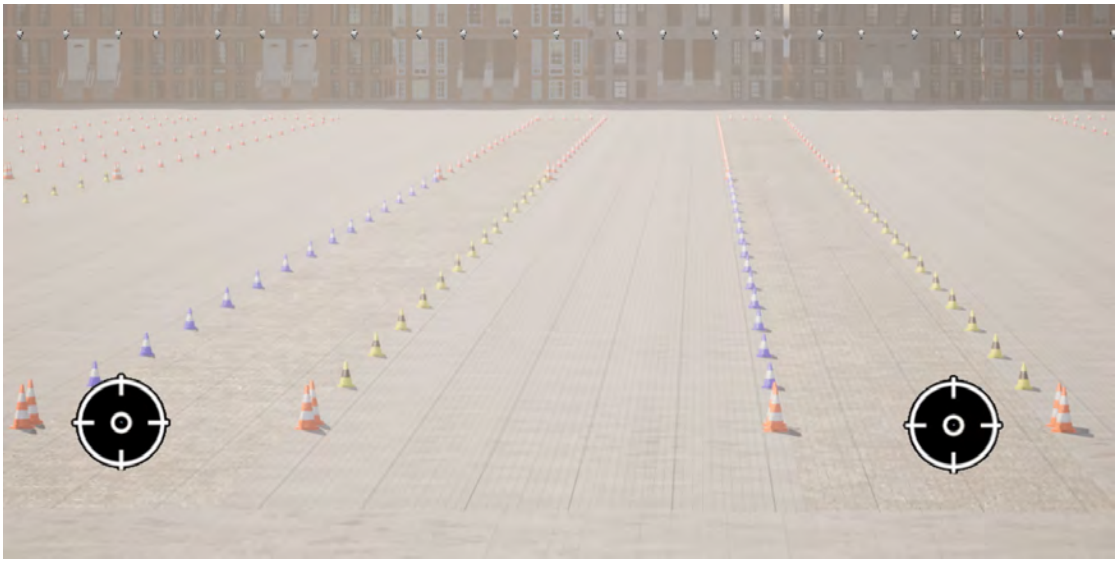


Figure 12.: Acceleration event in CARLA with two agent spawn points

4.1.2. Skidpad Map

The skidpad environment also has 20 different tracks on a single map. It is presented in figure 13.

Figure 13.: Skidpad event in CARLA

4.1.3. Trackdrive Map

The tracks for the trackdrive map are specifically designed to be challenging. For instance, they feature a higher number of hairpin turns and closely spaced tracks than would be expected at an official event. Eight different tracks were manually crafted to simulate real-world inconsistencies, with some cones deliberately moved or rotated. One of these maps is depicted in figure 14.

Figure 14.: Trackdrive map in CARLA

4.1.4. Simulated 2024 ZUR Vehicle

For the vehicle, the new 2024 model was used and implemented to closely resemble reality. The model and physical setup is depicted in figure 15. The CARLA editor provides access to parameters like mechanical setup, steering curve, mass and more. To approximate real-world characteristics, the parameters in CARLA were adjusted to a close estimate of the real vehicle. The exact parameters can be found in the appendix A.2.3.

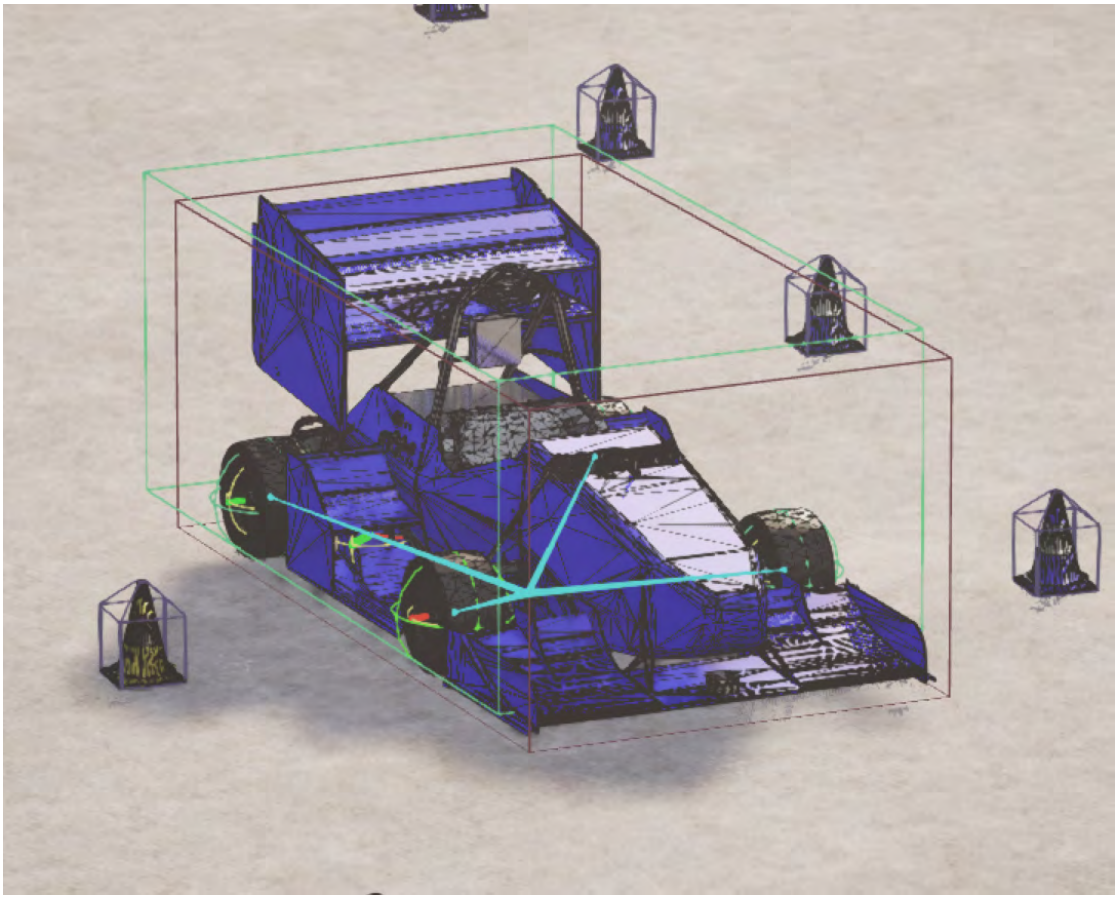


Figure 15.: The 2024 ZUR race car with wheel setup in CARLA

4.1.5. Debug screen

To facilitate easier debugging and monitoring of the simulation, a debug screen was implemented that displays the most important properties of the learning process. This screen visually presents all of the necessary values to verify the correctness of the implementation. The camera output and explanations of the relevant properties are shown in figure 16.

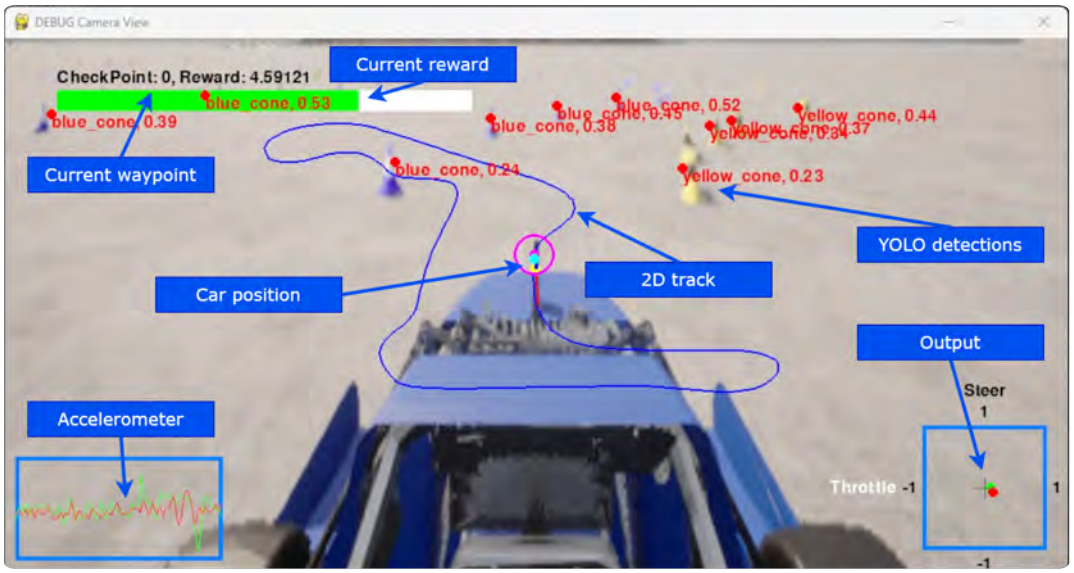


Figure 16.: The screen for visualizing debug data

4.2. Gym-like interface to CARLA

To perform Reinforcement Learning in a structured and familiar environment, the different track types and their reward functions are implemented in the style of Gymnasium environments [36]. Gymnasium has become a de facto standard for RL, as it provides a clear and concise separation of the interaction between the environment and the learning process.

Using this setup, an abstraction for the underlying interaction with the CARLA Python API can be provided, which is also an important requirement for the ZUR team. Future implementations need to be able to easily swap environment implementations, such as when the physical car is used instead of the CARLA simulator.

A Gymnasium environment is implemented as a python class. The documentation describes the following main API methods [36]:

- `step(a) -> (observation, reward, terminated, truncated)` takes a single step in the environment and returns a 4-tuple of the next `observation`, the `reward` for taking action `a`, and two booleans that indicate if the episode has `terminated` or was `truncated`.
- `reset() -> (observation)` - Resets the environment to its initial state and returns the first observation.
- `close() -> void` - Closes the environment and disposes any used resources.
- `render() -> void` - Renders the environment to visualize what the agent sees. This method is not used or implemented for the purposes of this thesis.

Structure

Figure 17 shows the class diagram of the environment implementation used for the trackdrive discipline, chosen due to its complexity compared to the other disciplines. For the other disciplines, an implementation in a similar structure is available. The `MutliAgentEnvAcceleration` and `MultiAgentEnvSkidpad` class handle up to 20 agents in parallel for the acceleration and skidpad disciplines respectively.

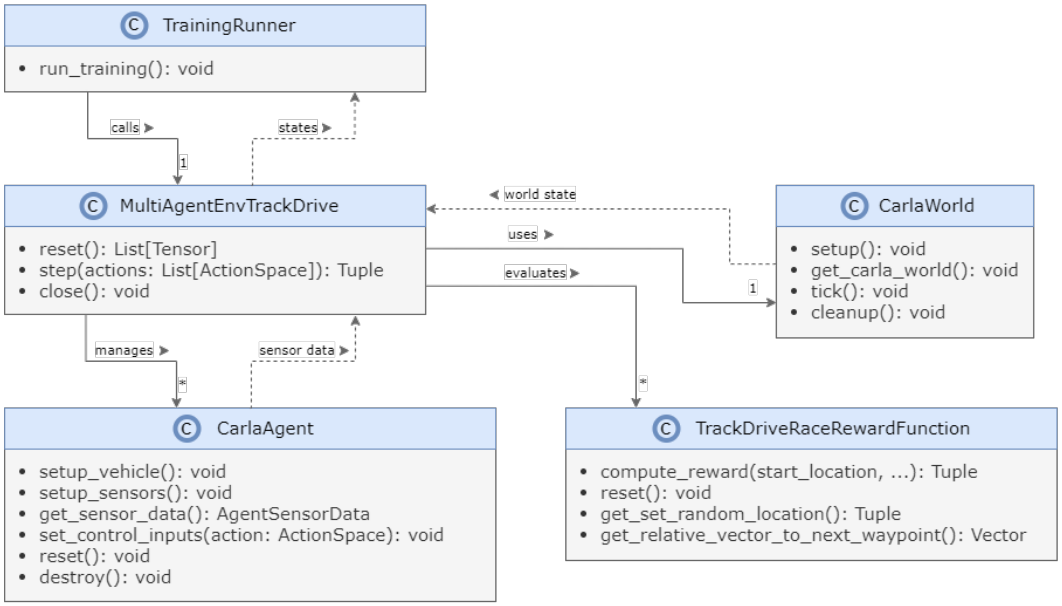


Figure 17.: Class diagram overview of the environment implementation and its dependencies

The behaviour and purpose of the classes is as follows:

- `TrainingRunner` - Contains the training loop and handles logging.
- `MultiAgentEnvTrackDrive` - The Gymnasium-like representation for all the tracks in the in the Trackdrive event. It contains the previously mentioned `reset`, `step` and `close` methods. The `render` method is not implemented, as the visualisation is not handled by the environment. Unlike classic environments, this implementation is able to handle multiple agents in parallel. Therefore, the `step` method takes an array of actions, one for each agent, instead of just one. It is comparable to a Gymnasium *Vectorized Environment* [37], although the different agents are not completely independent, as they spawn in the same world on separate tracks.
- `CarlaAgent` - Is a representation of the simulated car and serves as an abstraction of the Carla PythonAPI calls regarding said car. It configures the necessary sensors and cameras that are used and provides their sensor data to the environment. The vehicle's throttle, braking and steering is set via `set_control_inputs`.
- `CarlaWorld` - A representation of the simulated world in which the agent drives. It is used to set the necessary settings for the simulation environment and manages when the next simulation step is executed.

- `TrackDriveRaceRewardFunction` - Computes the reward that an agent receives for a single step.

The actual implementation of the project involves many more classes and interactions. However, for the sake of brevity and clarity, these details were not included in this document. The full implementation can be viewed on GitHub [38].

4.3. Reinforcement learning setup

This section outlines the configuration and components of the reinforcement learning setup, detailing action space, states, observations, and algorithm choice.

4.3.1. Action space definition

Electric FS vehicles usually have simple control interfaces for the driver. In comparison to internal combustion engines, there is no mechanical transmission that needs to be manipulated. Due to this, the race car is equipped with only three controls: a brake pedal, an acceleration pedal and a steering wheel. For the purposes of the agent, these three controls were simplified to two linear outputs. This facilitates the neural network's ability to correlate braking and throttle actions. Additionally, the physical pedals of the race car should not be engaged simultaneously, as this could potentially damage the vehicle.
Thus, the action space of the agent is defined as follows:

- **Steering** $[-1, 1]$: Controls the direction of the vehicle, where -1 represents full steering to the left and $+1$ represents full steering to the right.
- **Drive** $[-1, 1]$: Manages the vehicle's speed, where $+1$ corresponds to full throttle, -1 to full braking, and 0 indicates maintaining the current speed (cruise).

It is important to note that both values are continuous, meaning they can take on any value within the specified range.

4.3.2. States and Observations

To perform Reinforcement Learning, the state needs to be defined such that the agent is able to choose the optimal action at every time step, given the available sensor data. The following section describes the considerations and issues faced during the implementation and how they were resolved to ensure the effectiveness and efficiency of the agent. The structure of the state representation, the rationale behind its design, and the impact of these choices on the agent's learning and decision-making process is detailed.

In principle, the agent needs to perform two separate tasks:

- **Perception**: Detect and classify cones to perceive the dimensions of the race track.
- **Racing Dynamics**: Using the perceived cones to accurately navigate the vehicle towards its goal.

The separation of these tasks is relevant to the final definitions of observations and states.

Simulated Vehicle Sensors

Given the requirement for all system components to be on the vehicle and considering the available sensors, the simulated vehicle sensors are defined as follows:

- **Camera RGB image**: Captures a color image of the vehicle's surroundings, providing visual context for navigation and obstacle detection.
- **Camera depth image**: Offers a depth map generated by the camera, which helps in determining the distance to objects in the environment.
- **Accelerometer data**: Measures the acceleration forces acting on the vehicle in three dimensions.
- **Speed**: The current absolute speed of the vehicle. While the physical ZUR car has no ground speed sensor, it could be estimated using the current rotation speed of the wheels or using the inertial measurement unit of the ZED 2i camera.
- **Wheel angle**: Records the current angle of the front left and right wheels.

A considerable challenge in the implementation is processing the complex, high-dimensional sensor data. Specifically, the camera's RGB and depth images require substantial computational power and larger neural networks to be utilized effectively.

Initial Setup

Multiple attempts were made to train a convolutional neural network capable of mapping input images directly to driving actions. This involved feeding camera data directly into a pre-trained convolutional network such as MobileNet or VGG16, from which the final, fully connected classification layers were removed to only extract features. These features were then combined with sensor data and fed through the final fully connected layers. This approach is detailed in figure 18, with the depth data being processed in a similar manner, as this multimodal architecture has been demonstrated to offer advantages over using just RGB data [39].

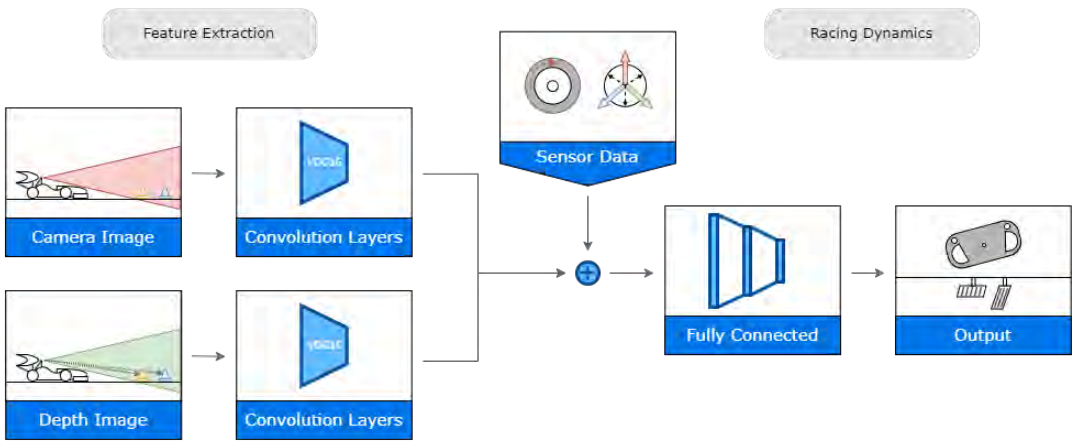


Figure 18.: Initial architecture of the network that processes the raw image data using convolution layers, later combined with the sensor data

However, this proved to be impractical as the step of the perception was non-transparent. This made training more difficult, as it was hard to understand when to attribute poor performance to perception of the environment or the decision-making regarding the actions. Because the network required more trainable parameters to process raw images, training time increased considerably and was more unstable, in some experiments even leading to complete collapse of the policy. This necessitated a more streamlined approach to handle the perception tasks separately.

Cone Detection

The decision was made to exclude the complexity of object detection by utilizing a separate YOLOv8 network, as figure 19 displays. This approach effectively treats the detected cones as another sensor input within the E2E framework. To add to this, this practice is utilized by other similar works [19], where the perception task is separated. The utilized YOLO network was trained on the fsoco dataset [40]. Because the NVIDIA Jetson platform offers limited computational power, the nano version was chosen, because it is the smallest version of the network provided by Ultralytics [34]. As displayed in results 5.5, this model proved to be sufficient for the task.

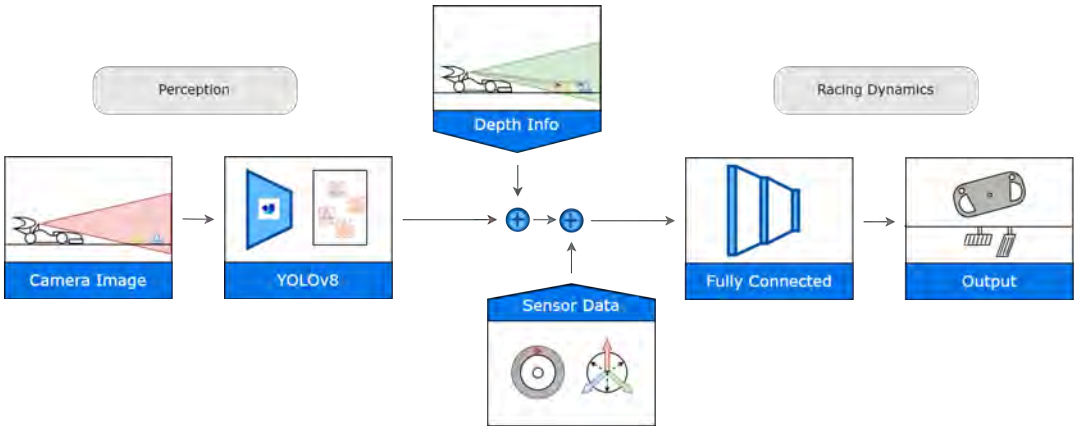


Figure 19.: Architecture using perception with YOLO

The YOLO network only identifies the positions and classes of the cones without an estimate of the depth. To give the actor model maximum information, the depth image of the camera is used to compute the distance at the center of each cone. Then, the closest ten detections are forwarded into the actor network, with the tensor being padded with zeroes in cases of fewer detections. This approach allows for more visibility while separating the fundamentally different tasks of perception and racing dynamics.

Final Observation Definition

Having addressed the issue of image perception, the resulting observations that are processed by the agent are as follows, where all values are normalized using min-max feature scaling:

1. **Detected cones**: The 10 closest cones, as determined by the depth map. Every cone has the data points:

- x/y-position on screen, normalized to $[0, 1]$. The exact position of each cone is determined from the center of the YOLO output box.
- Distance from the camera, normalized to $[0, 1]$. The minimum and maximum distances are 0.3m and 20m respectively.
- Cone classification: Integer that represents the kind of cone that is detected (large orange cone = 1, orange cone = 2, blue cone = 3, yellow cone = 4).

2. **Accelerometer**: Acceleration values in all three dimensions, normalized to $[0, 1]$. The accelerometer data is expected to range between -5g and 5g.
3. **Vehicle Speed**: The current speed of the vehicle, normalized to $[0, 1]$, where the vehicles maximum speed is expected to be 1. The maximum speed is expected to be around 50km/h.
4. **Wheel angles**: The angle of the front left and right wheels, normalized to $[-1, 1]$, with 1 representing the maximum steering angle of 50°.

The resulting observations are therefore represented as a one-dimensional tensor with $(10 * 4) + 3 + 1 + 2 = 46$ entries.

Temporal Aspects and Resulting State

Temporal information is critical in dynamic environments like racing, where conditions change rapidly. The agent's state is defined by stacking the last five observations as shown in figure 20. This allows the agent to discern trends such as velocity and trajectory over time and can also allow the agent to make optimal decisions in situations where no cones are visible. For example, this situation can occur in hairpin turns. If the outer cones are spaced too far apart, there are frames where no cones are visible. In such cases, the agent can, to an extent, rely on the previous observations.

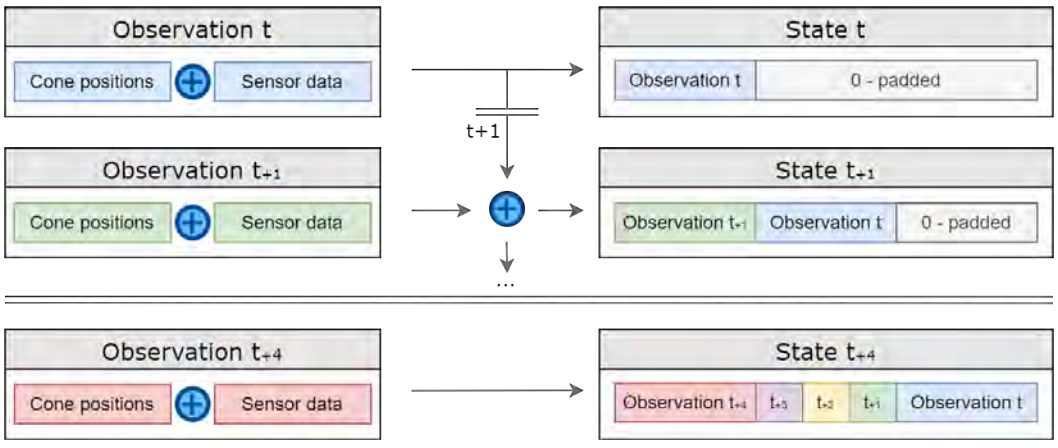


Figure 20.: Illustrates the concatenation of previous observations to form the state that the agent uses to make decisions

This method captures both the current environmental conditions and their progression, which implicitly allows the agent to plan a trajectory. Five past observations were chosen

to balance between sufficient temporal resolution and manageable computational load, although further experimentation will be performed on this.

4.3.3. Algorithm Choice and Implementation

Several deep reinforcement learning algorithms like Soft Actor-Critic (SAC), Deep Deterministic Policy Gradient (DDPG), Deep Q-Network (DQN) and Proximal Policy Optimization (PPO) were successfully tested for simulated autonomous driving in related projects. All of them showed promising driving performance, although with varied sample efficiency [41] [42]. Given the expensive simulation and limited time frame, sample efficiency and convergence time were crucial aspects for us to consider when choosing a training algorithm. A more popular algorithm was also preferred since the project will be handed over to other members of the ZUR team for the next Formula Student season. Ultimately, PPO was chosen as it is easy to understand and implement [43] and promises faster convergence than other methods [44].

Implementation

A minimal implementation of PPO by Nikhil Barhate [45] was used. The implementation provides a fully functional python class with support for continuous action spaces and buffering of trajectories. The implementation was integrated into the code base and adjusted for the outlined configuration structure, which facilitated running experiments early into the development process.

Adjustments for Multiple Agents

In the early stages of the project it became clear that collecting experiences using only a single agent would be too inefficient. The simulation has a significant performance overhead to simulate a single vehicle, as a substantial portion of the computational power is used to render the map and operating the physics engine. This means that simulating multiple vehicles on a single map only requires marginally more resources than a single vehicle. Therefore, the approach was scaled by deploying multiple agents in parallel, each navigating a separate race track within a single CARLA world, as detailed in section 4.1. To facilitate this, the environment was implemented to handle multiple agents concurrently and the PPO implementation was also adjusted. Specifically, when new actions are selected from the policy, each agent's actions are designated and stored in individual buffers. After running n agents in parallel for m episodes, a total of $n * m$ trajectories can be collected. Before updating the policy, the separate experience buffers are merged into a single buffer, making it appear as if the trajectories had been collected sequentially.

4.3.4. Exploration

New strategies and actions are explored by interpreting the outputs of the policy network as the means μ of a multivariate normal distribution, where the components are steering and throttle values, as described in section 4.3.1. The standard deviation σ of this distribution is controlled by the hyperparameter `action_std`. At every step, the vehicle control inputs are then sampled from this distribution. Initially, `action_std` is set to a higher value to force the agent to explore a wide range of actions, potentially deviating significantly from those suggested by the policy's mean output. As training progresses,

`action_std` is gradually reduced through a linear decay process. This reduction aims to decrease the extent of exploration as the policy network learns and optimizes the action choices. Ultimately, the standard deviation approaches 0, allowing the policy to deterministically dictate the control inputs with minimal to no randomness.

4.4. Reward Functions

Reward function design is crucial in training a reinforcement learning algorithm. Implementing effective reward functions required numerous iterations. Initial attempts included:

- Giving rewards for driving forward, solely relying on the negative collision reward for learning.
- Creating waypoints that the agent needs to drive towards to gain rewards.
- Allowing the agent to ignore collisions with cones.

From these iterations and attempts, several important constraints and knowledge points were gathered that apply to this specific problem:

1. A good action should return a positive reward as soon as possible.
2. The agent needs a balanced mix of short-term and long-term rewards.
3. Any reward for velocity needs to be clipped.
4. The discount factor γ has a significant influence on how safety-conscious the car operates.

The reward functions chosen for the three disciplines differ. Those differences are detailed in the following sections.

Acceleration

The reward function for the acceleration discipline is the simplest of the three. It consists of the starting position, from which the y-axis (driving direction) is used to calculate the reward for the agent. The distance from the x-axis is used to discount that reward for driving on the y-axis, creating a band where the agent receives positive rewards, as depicted in figure 21. The calculation during the acceleration discipline is detailed in table 1.
The reward for braking, $\text{Reward}_{\text{brake}}$, is applied once the agent passes the finish line. Once the agent is stationary, the reward for stopping, $\text{Reward}_{\text{stop}}$, is applied once. Similarly, the negative reward for collision, $\text{Reward}_{\text{collision}}$, is applied if the agent collides with anything. Combining all these components, the total reward function encourages the agent to move forward, maintain its lane, avoid unnecessary steering and braking, and reward safe stopping behavior.

Table 1.: Reward function calculations for acceleration discipline

Reward Component	**Calculation**
$\text{Reward}_{\text{move}}$	$10 \times (y_{\text{curr}} - y_{\text{prev}})$
$\text{Reward}_{\text{lateral}}$	$(0.5 - \lvert x_{\text{curr}} - x_{\text{start}} \rvert) \times \lvert \mathbf{v} \rvert \times 3$
$\text{Reward}_{\text{stability}}$	$-(\lvert \text{current\_steer} - \text{last\_steer} \rvert + \lvert \text{current\_drive} - \text{last\_drive} \rvert)$
Total Reward	$\text{Reward}_{\text{move}} + \text{Reward}_{\text{lateral}} + \text{Reward}_{\text{stability}}$
$\text{Reward}_{\text{collision}}$	-100
$\text{Reward}_{\text{stop}}$	500
$\text{Reward}_{\text{brake}}$	$-20 \times \text{throttle}$

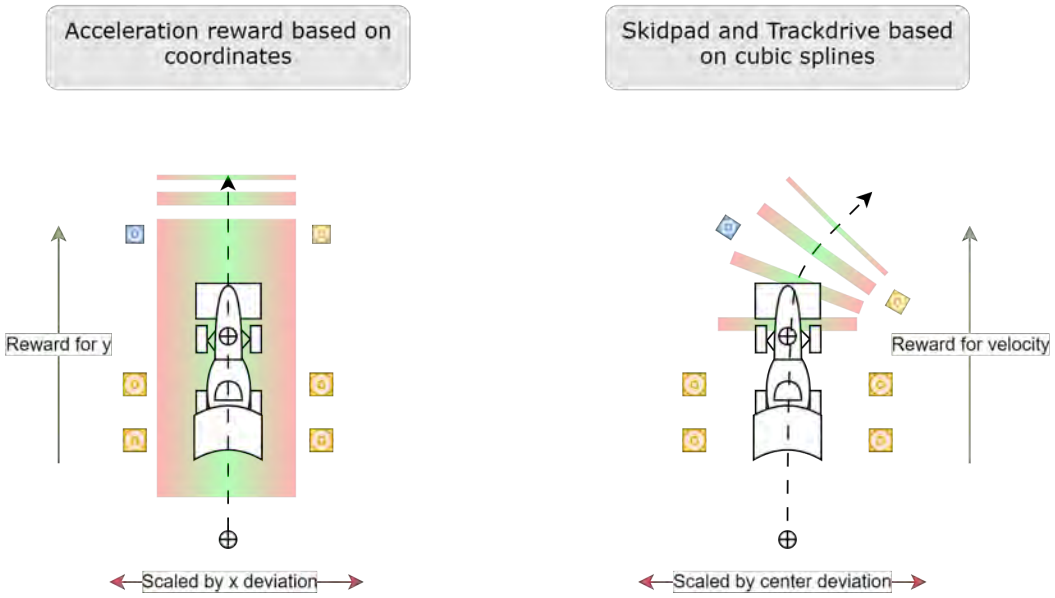


Figure 21.: Visual explanation of reward calculation

Skidpad and Trackdrive

The reward functions for the skidpad and trackdrive disciplines are similar to acceleration. To encourage the agent to drive in the center of the track, a system was implemented where manually gathered waypoints on the track are connected using cubic splines. This allows for the calculation of the distance from the center of the car to the track's center line, as shown in figure 21. The reward is then computed by scaling the car's velocity with the distance from the track's center line, following exactly the same method used in the acceleration discipline. The proportional reward for velocity is capped at 10m/s to discourage the vehicle from accelerating excessively.

For the trackdrive discipline, the proportional reward for distance from the center line is not calculated from the center of the vehicle like in the other two disciplines but from the front of the vehicle, as this yielded better results.

In the skidpad discipline, the agent needs to decide which direction to take each time it crosses the start line, as described in 6.7.1. Because the lap counter component does not exist yet, the waypoints used in the reward function were used to calculate the correct direction. This information is then passed to the network in the sensor data.

4.5. Neural Network architecture

After extensive iterative experimentation and a gradual increase in network size, adequate dimensions for the dynamics network were selected, as displayed in figure 22. The combined past observations are passed to the actor and critic networks, which consist of three hidden layers of decreasing width.

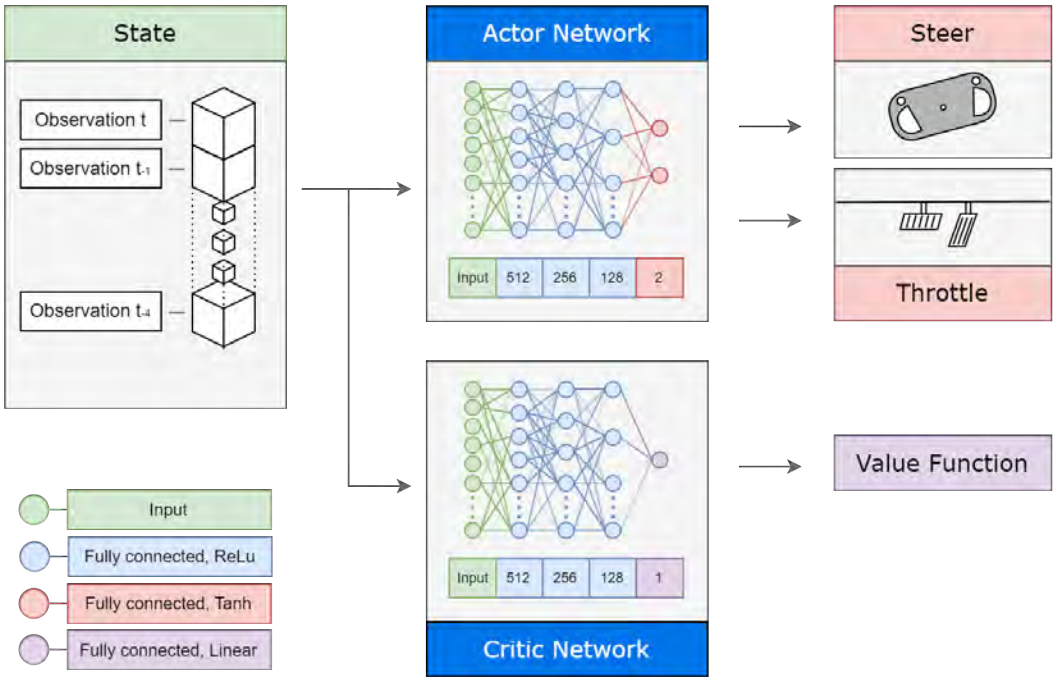


Figure 22.: Architecture of the actor and critic networks

4.6. Training loop

The following diagram describes a high-level view of the training loop. It's important to note that there are several `CarlaAgent` objects that are managed by the environment. For the sake of brevity, the sequence diagram only shows a single instance.

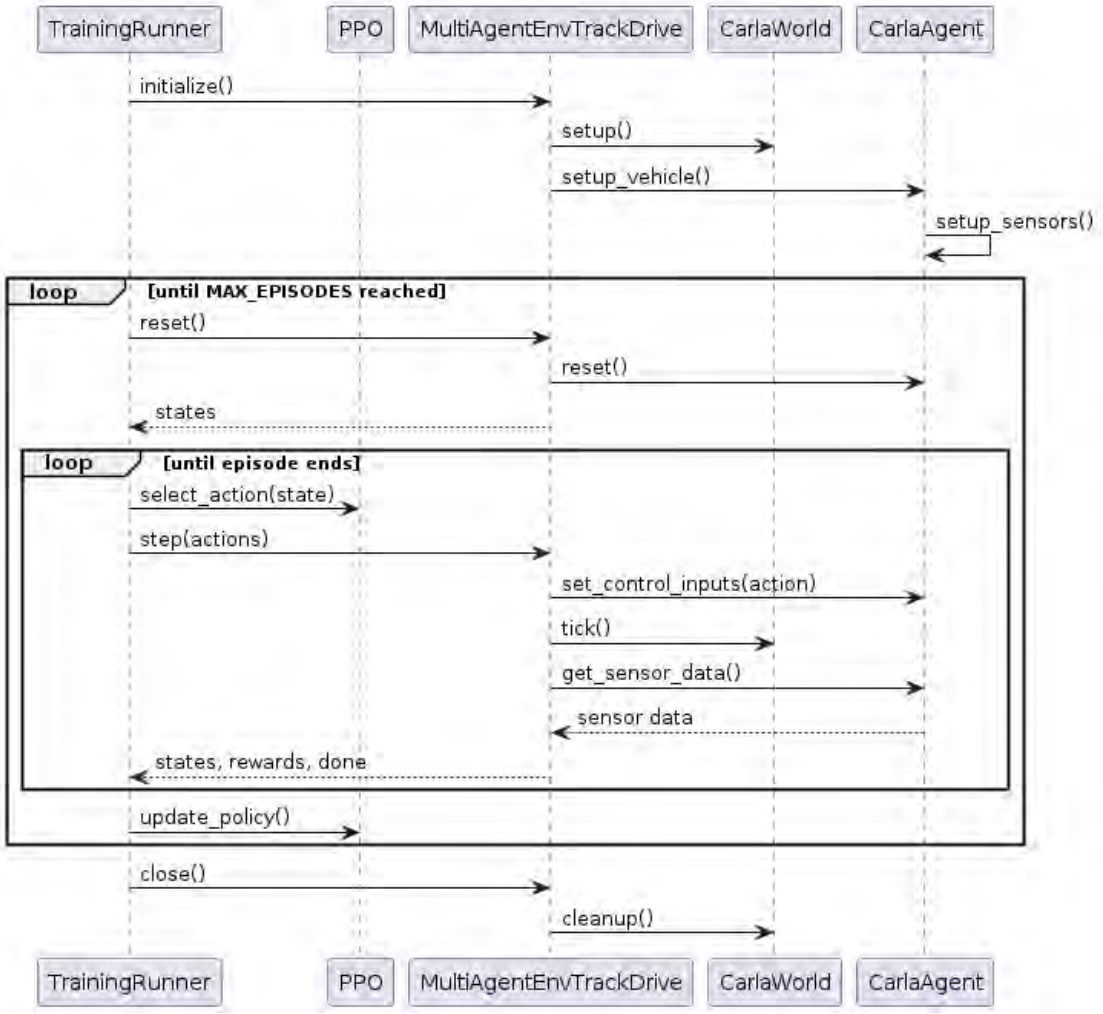


Figure 23.: A high-level overview of the training loop sequence

In the first part, the environment is initialized and all the vehicles are spawned as part of the `setup_vehicle()` method. The CARLA PythonAPI is utilized in the `setup()` method to configure the relevant settings for the simulation, such as enabling synchronous mode. Afterwards, the `CarlaAgent` objects find a suitable spawn location, which were previously defined for every map in the CARLA Editor. A ZUR racecar is spawned in the sim world and all the sensors are configured and added to the vehicle during this process.
In the second part, the actual training loop begins. This follows the classic RL training loop, where an action is selected and executed for every agent. By setting the selected actions as control inputs (during `set_control_inputs()`) and calling the `tick()` method on the world, the agent moves to a new state in the simulator. After executing a single tick and then calling `get_sensor_data()`, the new sensor data can be read from the vehicle again, which is used by the environment to create the new state, as defined previously. This loop is then repeated until the end of the episode is reached, either when time runs out or all vehicles have crashed.
After a pre-defined number of episodes have been completed, the buffered trajectories are used to update the policy, using the `update()` method of the PPO instance. Finally,

either through manual cancellation or if the maximum number of episodes defined have been reached, all vehicles get destroyed, the environment is closed and the connection to CARLA is disposed.

5. Results

In this section, the experimental setup is presented. The training and evaluation results are analyzed. The raw results data for each discipline is available on GitHub [38]. The averages are calculated using the arithmetic mean.

5.1. Experimental Setup

To train the agent for each discipline, the created tracks detailed in section 4.1 were used. Each map contains multiple tracks, facilitating the training of a number of agents simultaneously. The exact training setup parameters can be viewed in the appendix, table 7. The hardware used is detailed in table 2. The compute time to train the agents is an approximation, as the computer was also used for other work during the day, leading to significant variations in training times.

5.1.1. Training Metrics

To monitor and validate the training progress of the agent, specific metrics must be defined. Traditional metrics like loss are less straightforward to interpret for the actor and critic networks in PPO due to their roles in policy and value estimation, respectively. The following training metrics have been established:

- **Average Reward per Episode**: Measures the average total reward obtained by the agent per episode.
- **Average Speed per Episode (m/s)**: Represents the average speed maintained by the vehicle throughout each episode.
- **Average Survived Steps per Episode**: Counts the average number of steps the vehicle survives without crashing.

The scalar `Action Standard Deviation` is the exploration coefficient as detailed in section 4.3.4. This value decreases linearly over time. The fluctuations visible in the

Table 2.: Hardware and software used for training and validation

Hardware	**CPU**	**GPU**	**RAM**	
	Ryzen 7 5700x	RTX 3070 Ti 8 GB GDDR6X	96 GB DDR4 3600 MHz	
Software	**CARLA**	**Python**	**Pytorch**	**Cuda**
	0.9.15	3.8.0	2.3.0 + cu121	12.1.0

graph are due to shortened episodes where all agents have crashed, as the scalar is decreased every n steps.
It is important to note that the graphs presented are smoothed, as the scalars fluctuate over time. The lighter colors represent a smoothed average over the minimum and maximum values within 40 bins to display the varying values.

5.1.2. Validation Metrics

To measure the performance of the agent, each fully trained agent was evaluated on one or more previously unseen tracks. This is done for 100 runs, where each run begins on the start of the track. The following metrics were defined:

- **Average Completion Rate (%)**: Indicates the percentage of the track that the agent successfully completed during each run.
- **Average Speed (m/s)**: Represents the mean speed maintained by the agent throughout each run.
- **Crash Rate (%)**: Calculates the percentage of runs in which the agent crashed.
- **Speed Fluctuations**: A graph depicting the variation in average speed during each run, presented as a 2D map of the track.
- **Crash Locations**: A graphical representation of where crashes occurred during the runs, plotted on a map of the track.

It is important to note that the final evaluation is based on an `Action Standard Distribution` of 0.001, ensuring that the model's outputs are nearly deterministic.

5.2. Acceleration

5.2.1. Training Results

Figure 24 illustrates the agent's learning progression over time. Initially, the agent persists a significant amount of steps without crashing, primarily because it stays stationary, not having learned to drive forward. As the training progresses, there is a slow but steady improvement in performance, with the reward and speed metrics converging around the 800-episode mark, where they begin to plateau. This indicates that the agent has reached an optimum. Additionally, the graphs show dips in performance at around 420 and 620 episodes, from which the agent recovers quickly.
To evaluate the trained agent, the agent at episode 900 was selected. To train the acceleration agent from scratch, about 1 day of compute power was needed.

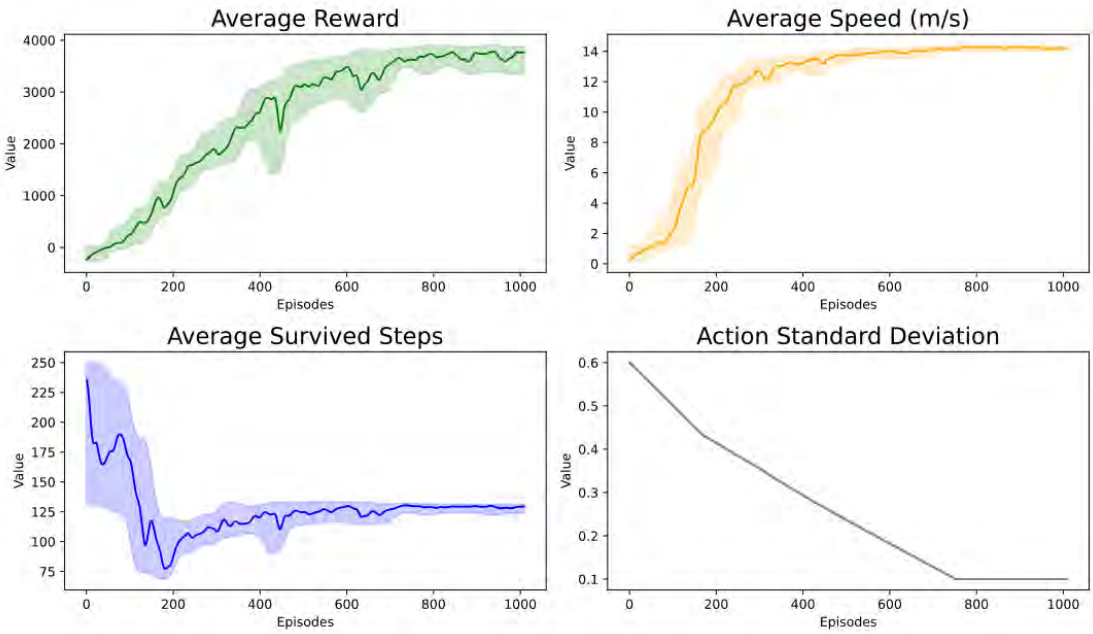


Figure 24.: Training results of acceleration

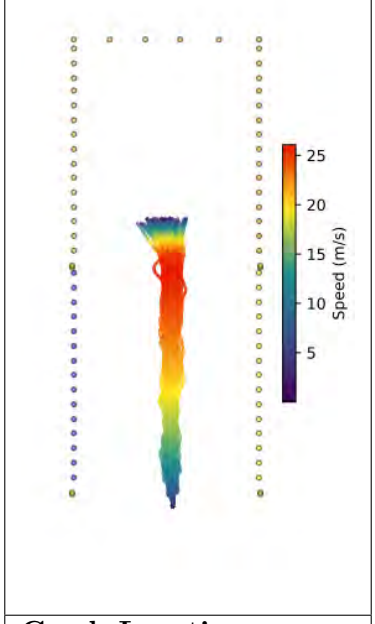
5.2.2. Validation Results

In table 3 the validation track is evaluated. As there were no crashes, the crash location map is omitted. It's important to note that the speed fluctuation graph was scaled horizontally to enhance the readability of the paths. The agent starts at the bottom of the track and makes its way to the center, where it crosses the finish line and brakes.

Table 3.: Evaluation results for acceleration discipline

Metric	Result
Average Completion Rate (%)	100.00
Average Speed (m/s)	14.53
Crash Rate (%)	0.00

Speed Fluctuations



Crash Locations

No crashes

5.3. Skidpad

5.3.1. Training Results

The charts in figure 25 display data of the training of the skidpad agent. The average reward increases steadily until the end of training. The speed seems to be steadily increasing until the end of training. Notable is the low minimum values of average reward, occasionally dropping below -1000, suggesting episodes of particularly poor performance likely due to frequent crashes which cause the episode to terminate early for the respective agent.
For the evaluation phase, the model at episode 9900 was selected. For training, approximately 10 days of compute power was needed.

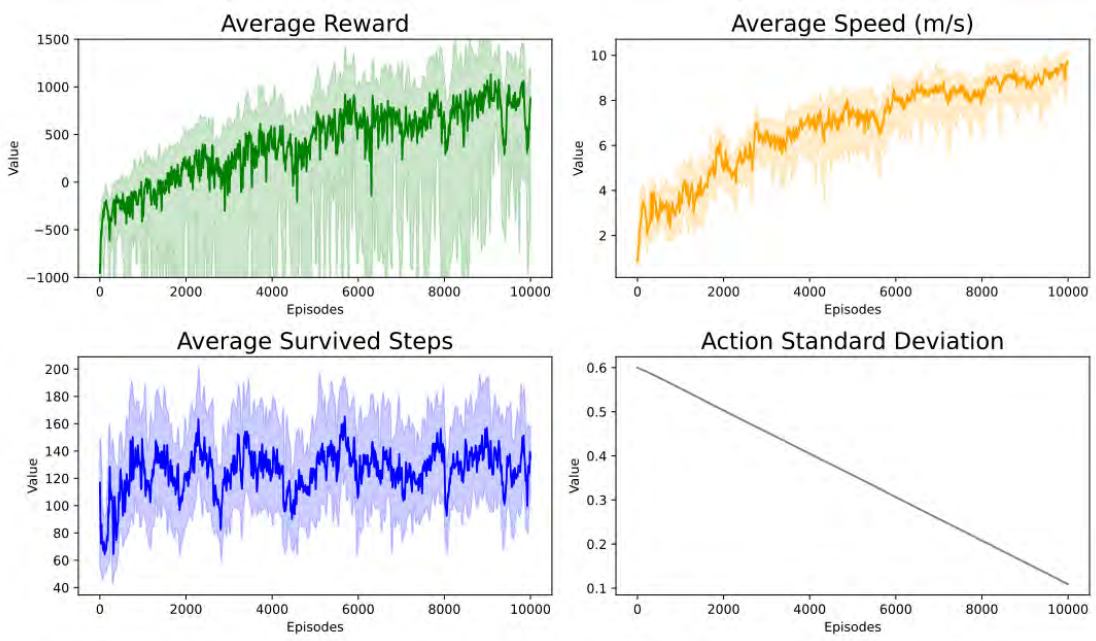


Figure 25.: Training results of skidpad

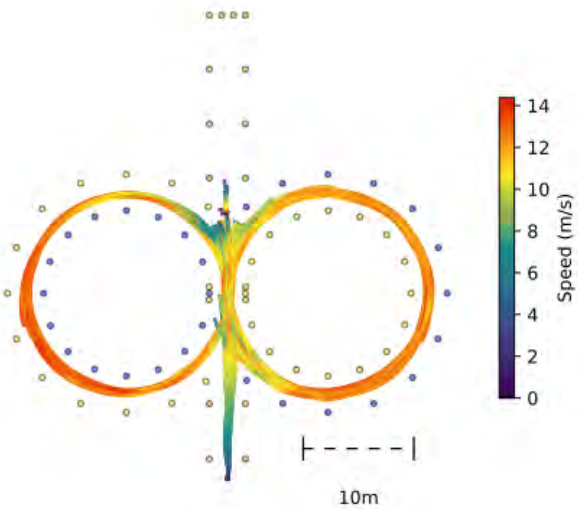
5.3.2. Validation Results

The figures in table 4 display the evaluation results. Most of the agents did not manage to complete the course. The average speed is about 37 km/h, reaching up to 50 km/h in the corners. The agent slows down in the center of the map where it must choose a direction. The crash locations are diversely spread all around the map, with the exception of the center. Crash locations are predominantly clustered around and after the central junction, a critical area where the agent must decide on direction changes.

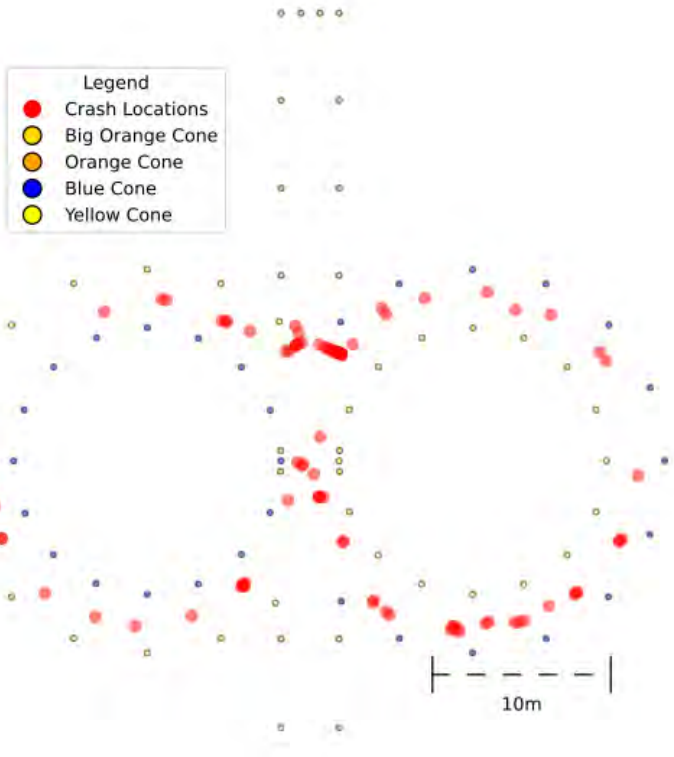
Table 4.: Evaluation results for skidpad siscipline

Metric	Result
Average Completion Rate (%)	47.65
Average Speed (m/s)	10.27
Crash Rate (%)	96.00

Speed Fluctuations



Crash Locations



5.4. Trackdrive

5.4.1. Training Results

Figure 26 illustrates the agent's rapid initial learning phase, with all metrics increasing steadily up to approximately episode 6000. After this point, a decline is observed. As the `Action Standard Distribution` decreases, indicating reduced exploration, the training shows signs of a decline in performance.
For the evaluation phase, the model from episode 6000 was selected because the scalar averages indicate the best performance.
Training the agent from scratch for 8000 episodes took approximately 7 days.

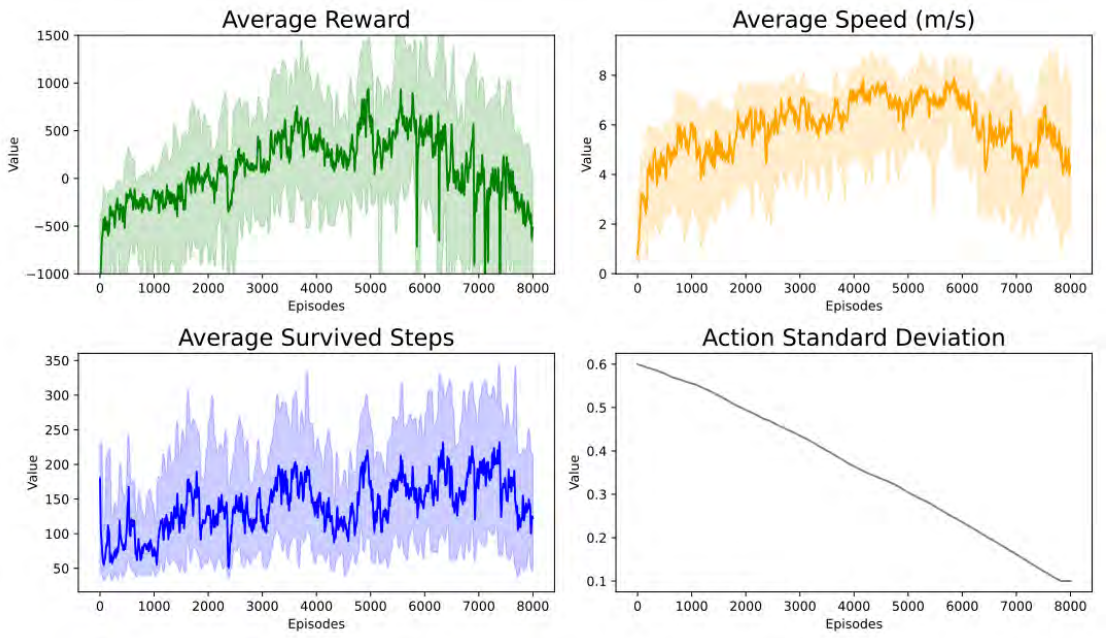


Figure 26.: Training results of trackdrive

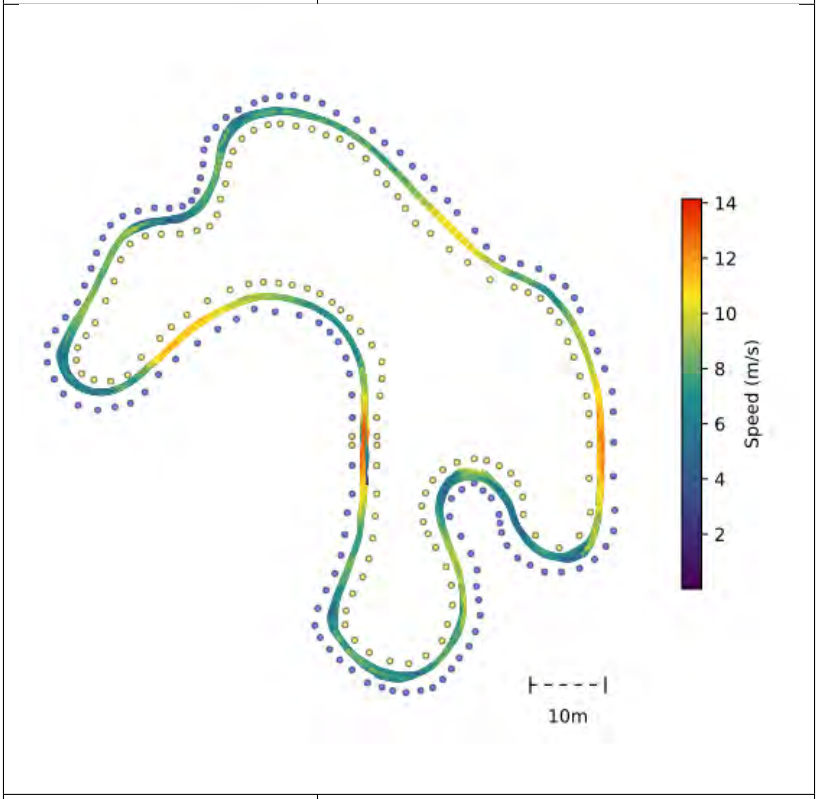
5.4.2. Validation Results

The figures in table 5 display the evaluation metrics. These metrics are only for a single lap of the track, as this provides a clear and focused assessment of the agent's performance under consistent conditions. The speed fluctuations show that the agent was able to stay within the cones, as well as drive with adequate speed. The yellow and red sections show that the agent drove quickly on straight sections, with speed reductions during cornering.
Interesting to note is that the crash locations are clustered primarily in or after turns.

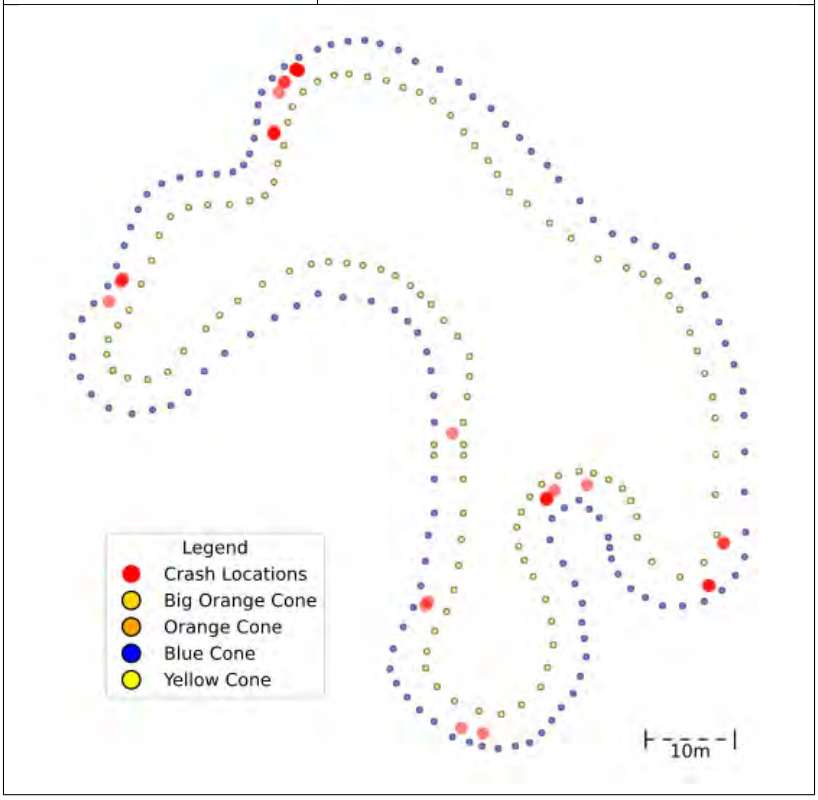
Table 5.: Evaluation results for trackdrive discipline

Metric	Result
Average Completion Rate (%)	72.97
Average Speed (m/s)	7.73
Crash Rate (%)	57.00

Speed Fluctuations



Crash Locations



5.5. YOLO Network

The following are the training results of the YOLOv8 nano network. The model was trained on the fsoco dataset [40] for 300 epochs, as suggested by the YOLO documentation [34]. Qualitative results of the network are available in appendix section A.2.2.

Figure shows loss and validation metric graphs over the course of training, generated by the YOLO training CLI. The displayed graphs are bounding box regression loss (box_loss), classification loss (cls_loss), dual focal loss (dfl_loss) for training and validation. The validation metric graphs show precision, recall and mean average precision (mAP). The graphs show consistent improvement in both training and validation losses. Around 100 epochs into training, improvements in class and bounding box losses begin to diminish.

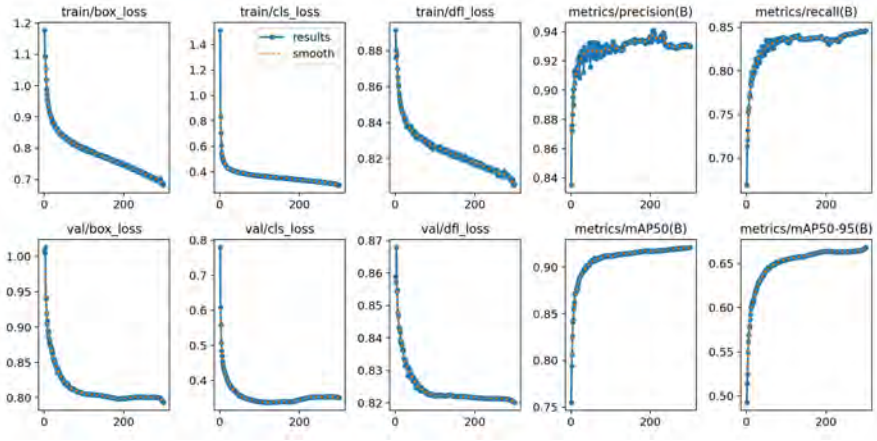


Figure 27.: Loss graphs of YOLOv8 training

The generated confusion matrix, displayed in figure 28, shows the detections on the x-axis and the true labels on the y-axis. For the different cone types it can be concluded that 12% of large orange cones, 14% of orange cones, 18% blue cones and 17% of yellow cones are falsely classified as background. The confusion matrix is generated at an IoU threshold of 0.7. Pairs of prediction and target boxes that do not meet this threshold are considered a misclassification.

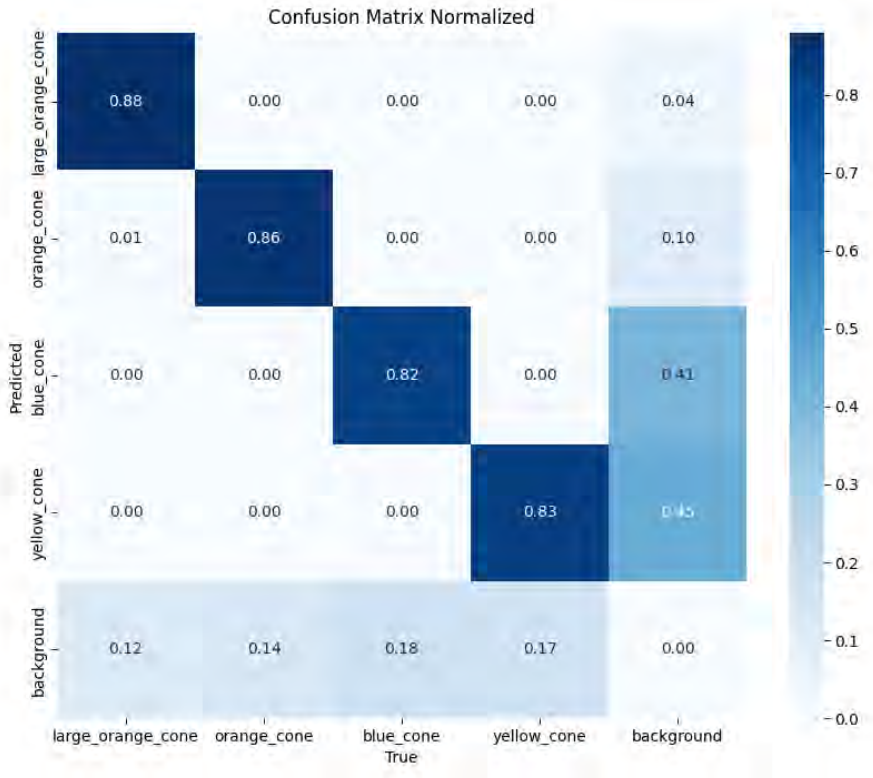


Figure 28.: Normalized confusion matrix of YOLOv8 training

6. Discussion and outlook

In this chapter, the results of the experiments are discussed and the insights are presented. The performance and behaviour of the agents is explored across all disciplines, highlighting key findings and areas for improvement. To conclude, the findings in relation to the defined tasks in 1.4 are interpreted and the extent to which the objectives have been achieved is assessed.

6.1. Simulation

Overall, the usability and robustness of the CARLA simulator is satisfactory for the purposes of this work. Although the initial setup posed some challenges, the extensive documentation and extensibility of the program offset these issues.
The correctness of the parameters for the vehicle setup could not be verified or reliably tested during the time frame of this thesis. Additionally, the capability of the physics engine's simulation for the more difficult aspects of racing dynamics could not be validated, as the vehicle appears to drift more rapidly when compared to videos of real FS vehicles. To reliably test this, test cases would need to be implemented in real life and the results would need to be compared to the simulator, which was not feasible due to time constraints.

6.2. YOLO Object detection

While the results of the YOLO training show some instability in later epochs, the overall performance of the network is sufficient for the simulated environment. While the confusion matrix shows almost 20% missclassifications for some classes, this can be attributed to the relatively high IoU threshold used to generate the matrix. The qualitative results show robust performance for the object detection task.

6.3. Acceleration

The acceleration agent displayed near-optimal behaviour during training as well as during evaluation. The training graphs are representative of the robust performance of the vehicle, showing a gradual increase in speed and a decrease of the crash rate.
In the evaluation phase, the agent displayed a level of performance that is near the theoretical limit. Observing the actions during the acceleration phase confirms that suspicion, as the agent outputs values very close to full throttle. The agent drives in the middle of the track with little to no unnecessary steering inputs. Since the reward function disregards the lateral position after reaching the finish line, the agent tends to drift away from the center, visible in table 3.

6.4. Skidpad

The training for the skidpad agent was stable. Manual observation during the training phase revealed that the agent struggled to choose the correct course at the junction, even though there is an input parameter describing the required action as described in 4.4. This might be due to the high complexity at a single point of the race, where the network perceives near identical cone observations but needs to output conflicting actions depending on the lap counter input.
Compared to the other disciplines, the agent had the worst performance in the skidpad event. Only four out of 100 agents reached the finish line, which is a relatively low success rate. The reward for velocity is clipped at 10m/s, which the agent reaches nearly exactly as its average speed. As the agent attempts to reach this comparatively high speed to maximize reward, it begins to slip and attempts to start drifting. This is particularly visible in the video [7]. Due to this, adapting the reward function and retraining the agent would likely mitigate all of these issues. Additionally, because the skidpad event is timed only in the second lap on each side, a variable speed reward could be implemented in the future to mitigate the crashes in the junction.

6.5. Trackdrive

The primary distinction between trackdrive and the other disciplines is that the agent needs to adapt to a more varied range of possible states, as the track is not known beforehand. This represents significantly more complex challenge than merely navigating the same courses repeatedly. Nevertheless, the training and evaluation showed promising results. The training was stable to a certain point, visible in figure 26, after which the performance deteriorated. The decline in performance could be attributed to various factors, such as catastrophic forgetting, suboptimal hyperparameter settings, or an imbalance between exploration and exploitation.
Although the exact cause for the decline was not identified, the resulting agent demonstrates promising results. During the evaluation phase, as shown in table 5, the agent drives fast on straight sections and slows down in corners, similar to the behavior of a real car. Remarkably, it appears to follow a racing line, deviating from the center line on purpose despite the limited space between the cones. There was a significant crash rate of 57%, but the agents completed a large majority of the course before crashing. The crashes mostly occurred in locations where there are difficult racing conditions. Additionally, only visible in the complementary video [7], the agent starts to drift in corners, which leads to difficulty regaining control after the turn.

6.6. Conclusion

Based on the goals defined in section 1.4, the following section describes the completion status of each goal:

1. **Build and test a simulated environment to perform end-to-end reinforcement training**

   This goal was fully achieved. A customized simulation environment was successfully created, allowing for effective testing and iteration with reinforcement learning training.

2. **Create a working prototype agent that is conceptually applicable to the physical car**

   A functional prototype of the physical car was developed within the simulation, closely resembling the actual vehicle. The agent driving this simulated car demonstrated exceptional results. With further training and optimization, it is anticipated that the prototype will be capable of successfully navigating all races. However, we acknowledge that integration with the real car (sim2real) will be challenging and time-consuming, requiring significant effort.

3. **Evaluate and determine if end-to-end neural networks are a viable approach for the ZUR team**

   The viability of E2E neural networks was demonstrated, as they were shown to drive the car successfully in simulation. This approach is planned to be integrated into the ZUR environment as an alternative option to the classical pipeline. A challenge identified is the high turnover within the student team, which complicates the transfer of knowledge. Future students must be able to understand and retrain the models to ensure continued success in years to come.

6.7. Future Work

Considering the promising results, the following future tasks have been discussed and planned in collaboration with the ZUR team.

6.7.1. Short-term

Improvements in simulation speed

To train new models and evaluate new hyperparameters, architectures etc. an improvement in simulation speed is necessary. The bottleneck of the current training setup is the processing of the camera images to perform object detection on the cones, which leads to a training time of several days on the more complex trackdrive discipline. Now that the accuracy and reliability of the object detection could be verified, this task can be simplified during the training process, which allows for a method where the actual object detection network only needs to be used during evaluation and on the physical car, but not during training. The official CARLA documentation provides sample code that can be adapted so the cone positions can be projected on to the screen without the need for a rendered image [46]. During training, the positions of the cones on the screen can therefore be provided to the actor without a forward pass through the YOLOv8 network. The actor is trained on data that follows the same distribution as the outputs of the object detection network. After training, this module can be swapped out and replaced by the object detection to provide the cone positions instead.

Adjustable vehicle speed based on manual input parameter

Because the dynamics of racing change dependent on velocity, taking a model trained on highest possible speeds and just scaling the output values to a safer range for real life testing might not be a viable option. A variable speed parameter that is trained into the neural network will be implemented. This means that the reward function clips the

velocity at the specified value. The agent would learn to adjust its speed according to the input value, thereby training its ability to drive both slowly and quickly.

Reward engineering

The current implementation of the reward function for the skidpad discipline aims to balance a series of long and short term rewards. For example that the car is actively encouraged to stay at the center of the track and does not receive more reward by an increase in speed above a certain threshold. A more long-term reward signal would be the large negative reward for collision. The current balance of these more safety focused rewards have lead to, at times, risky behaviour, where the car drives at such high speeds that it intentionally starts drifting. To better understand the resulting behaviours, especially in combination with the discount factor hyperparameter, more variations of the reward functions will be tested.

ROS2 Implementation

The approach detailed in this thesis has so far only been tested in the CARLA simulator. As the 2024 Formula Student season is approaching, the newly built ZUR car will be tested further during the month of July. This includes the driverless components and therefore presents an interesting opportunity to also test the deep learning approach presented in this thesis in a real-world scenario. The current driverless systems use ROS2 [47] (Robot Operating System) python modules. Since the requirement of only using sensor data that is available on the physical car has been strictly adhered to, the current implementation can be quickly encapsulated as a ROS Node that receives sensor and image data and outputs vehicle controls. While its highly unlikely that the current model will work flawlessly for autonomous driving on the real car, it will allow the team to asses the behaviour and set new goals for the deep learning approach.

Lap counter

For the trackdrive and skidpad disciplines, an external track counter is needed. Without it, the agent would not know when to stop, or in the case of the skidpad event, it would be unable to determine the appropriate path. Due to the limitation that no external sensors are allowed during races, the counter would need to work with the same sensors as the driving algorithm. The basic idea is to train a smaller CNN specifically to classify whether the start is visible. Subsequently, a classic change point detection algorithm is applied to said network.

6.7.2. Mid-term

Larger Networks

After optimizing simulation speed, new network architectures can be tested more easily. The current size of the actor and critic network was increased gradually during the implementation of the approach. So far, an increase in trainable parameters appeared to also show a direct increase in performance. To optimize the driving behaviour further, larger and deeper networks for both the actor and the critic will be tested. It's crucial to find or estimate the boundaries of what the minimum number of trainable parameters

is and at which point an increase only results in a longer convergence time without any additional driving performance gains.

Partial Observability and Sequence Models

As detailed in chapter 3, the problem of autonomous racing can be considered a partially observable environment, making this a POMDP. Related projects on POMDPs in autonomous driving [48] have shown that performance can be improved by providing a history of observations to the agent, instead of only the latest observation. A simple version of this approach was implemented by concatenating the last five observations as explained in section 4.3.2. To not further extend the scope of the project with new architectures or topics, the decision was made to use a regular feed forward neural network to handle these sequential observations. This implementation, while already showing a performance improvement, does not use an architecture that is specifically intended to be used for sequential data, such as Recurrent Neural Networks (RNNs) [49], Long Short-Term Memory Networks (LSTMs) [50] or Transformers [51]. To improve the handling of sequential observations, the current implementation will be adjusted to use a sequence model that is capable of learning such temporal dependencies, which has proved to be beneficial [48]. Their performance in the autonomous racing task will then be compared to the current approach, ultimately deciding which type of model will be developed further.

Domain randomization

To expedite the transfer from simulation to real-world application (sim2real), a domain randomization approach will be explored. Given that the exact dynamics of the race car cannot be accurately modeled in software, certain aspects of the simulation, such as vehicle weight and wheel-slip, will be randomized within a range of plausible values. This approach allows the agent to experience a variety of conditions, some of which may closely resemble real-world scenarios by chance. Consequently, this variability is expected to train a more robust agent [52].

6.7.3. Long-term

Containerized Simulator and Learning

All training and evaluation runs presented in this thesis have been simulated using the personal devices of the authors. This was necessary, as the training tracks had to be edited and retested frequently using the UE4Editor that was built from source as explained in section 4.1. Now that a stable version of the setup for reinforcement learning is implemented, the simulated worlds and sources for running the training can be packaged into a containerized environment. Different architectures and approaches can then be tested without using resources on personal devices through the use of cloud resources provided by either ZHAW or the ZUR team.

Torque Vectoring

As the ZUR car is equipped with four independently addressable motors, using a single throttle value for all wheels is likely not optimal. Therefore, the agent could be enabled to output four distinct values, each corresponding to one wheel. This approach is expected

to facilitate automatic torque vectoring, enhancing the capabilities of RL without the need for a separate algorithm. However, this would require substantial effort, as it necessitates further modifications to the CARLA software to enable independent motor controls and adjustments to the current physics engine to accommodate these changes.

Bibliography

[1] S.-C. Lin, Y. Zhang, C.-H. Hsu, et al., "The architectural implications of autonomous driving: Constraints and acceleration", *SIGPLAN Not.*, vol. 53, no. 2, pp. 751–766, Mar. 2018, ISSN: 0362-1340. DOI: 10.1145/3296957.3173191. [Online]. Available: https://doi.org/10.1145/3296957.3173191.

[2] L. Liu, S. Lu, R. Zhong, et al., "Computing systems for autonomous driving: State of the art and challenges", *IEEE Internet of Things Journal*, vol. 8, no. 8, pp. 6469–6486, 2021. DOI: 10.1109/JIOT.2020.3043716.

[3] E. Yurtsever, J. Lambert, A. Carballo, and K. Takeda, "A survey of autonomous driving: Common practices and emerging technologies", *IEEE Access*, vol. 8, pp. 58 443–58 469, 2020. DOI: 10.1109/ACCESS.2020.2983149.

[4] A. Tampuu, T. Matiisen, M. Semikin, D. Fishman, and N. Muhammad, "A survey of end-to-end driving: Architectures and training methods", *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 4, pp. 1364–1384, 2022. DOI: 10.1109/TNNLS.2020.3043505.

[5] "Tesla release notes 25.03.2024". Accessed: 06.06.2024. (2024), [Online]. Available: https://www.notateslaapp.com/software-updates/version/2024.3.25/release-notes.

[6] R. McAllister, Y. Gal, A. Kendall, et al., "Concrete problems for autonomous vehicle safety: Advantages of bayesian deep learning", 2017. DOI: 10.17863/CAM.12760. [Online]. Available: https://www.repository.cam.ac.uk/handle/1810/266683.

[7] F. Ulrich and T. Wehrli. "End-to-end deep reinforcement learning for autonomous racing dynamics - a bachelor thesis", Youtube. (2024), [Online]. Available: https://youtu.be/fPDbPVxBh8M.

[8] E. D. Dickmanns, "Computer vision and highway automation", eng, *Vehicle system dynamics*, vol. 31, no. 5-6, pp. 325–343, 1999, ISSN: 0042-3114.

[9] C. Cadena, L. Carlone, H. Carrillo, et al., "Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age", *IEEE Transactions on Robotics*, vol. 32, no. 6, pp. 1309–1332, 2016. DOI: 10.1109/TRO.2016.2624754.

[10] L. Chen, P. Wu, K. Chitta, B. Jaeger, A. Geiger, and H. Li, "End-to-end autonomous driving: Challenges and frontiers", *arXiv preprint arXiv:2306.16927*, 2023.

[11] M. Mohanan and A. Salgoankar, "A survey of robotic motion planning in dynamic environments", *Robotics and Autonomous Systems*, vol. 100, pp. 171–185, 2018, ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2017.10.011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0921889017300313.

[12] D. A. Pomerleau, "Alvinn: An autonomous land vehicle in a neural network", in *Advances in Neural Information Processing Systems*, D. Touretzky, Ed., vol. 1, Morgan-Kaufmann, 1988. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/1988/file/812b4ba287f5ee0bc9d43bbf5bbe87fb-Paper.pdf.

[13] E. Perot, M. Jaritz, M. Toromanoff, and R. de Charette, "End-to-end driving in a realistic racing game with deep reinforcement learning", in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, Jul. 2017.

[14] M. Jaritz, R. de Charette, M. Toromanoff, E. Perot, and F. Nashashibi, "End-to-end race driving with deep reinforcement learning", in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, 2018, pp. 2070–2075. DOI: 10.1109/ICRA.2018.8460934.

[15] M. Bojarski, D. D. Testa, D. Dworakowski, et al., "End to end learning for self-driving cars.", *CoRR*, vol. abs/1604.07316, 2016. [Online]. Available: http://dblp.uni-trier.de/db/journals/corr/corr1604.html#BojarskiTDFFGJM16.

[16] M. Bojarski, C. Chen, J. Daw, et al., "The nvidia pilotnet experiments", *arXiv preprint arXiv:2010.08776*, 2020.

[17] J. Chen, B. Yuan, and M. Tomizuka, "Deep imitation learning for autonomous driving in generic urban scenarios with enhanced safety", in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019, pp. 2884–2890. DOI: 10.1109/IROS40897.2019.8968225.

[18] B. Price and C. Boutilier, "Accelerating reinforcement learning through implicit imitation", *Journal of Artificial Intelligence Research*, vol. 19, pp. 569–629, Dec. 2003, ISSN: 1076-9757. DOI: 10.1613/jair.898. [Online]. Available: http://dx.doi.org/10.1613/jair.898.

[19] A. Salvaji, H. Taylor, D. Valencia, T. Gee, and H. Williams, "Racing towards reinforcement learning based control of an autonomous formula sae car", 2023. arXiv: 2308.13088 [cs.RO].

[20] N. Hamilton, P. Musau, D. M. Lopez, and T. T. Johnson, "Zero-shot policy transfer in autonomous racing: Reinforcement learning vs imitation learning", in *2022 IEEE International Conference on Assured Autonomy (ICAA)*, 2022, pp. 11–20. DOI: 10.1109/ICAA52185.2022.00011.

[21] *Zurich uas racing website*, Accessed: 12.05.2024. [Online]. Available: https://zurichuasracing.ch/formula-student/.

[22] *Formula student rules 2024*, 1.1, Accessed: 12.05.2024, Formula Student Germany, 2024. [Online]. Available: https://www.formulastudent.de/fileadmin/user_upload/all/2024/rules/FS-Rules_2024_v1.1.pdf.

[23] "Nvidia jetson xavier technical specifications". Accessed: 01.06.2024. (2024), [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-series/.

[24] *Zed 2i product specifications*, Accessed: 20.05.2024. [Online]. Available: https://www.stereolabs.com/products/zed-2.

[25] C. M. Bishop and N. M. Nasrabadi, *Pattern recognition and machine learning*. Springer, 2006, vol. 4.

[26] D. Bertsekas, *Reinforcement learning and optimal control*. Athena Scientific, 2019, vol. 1.

[27] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[28] M. Hausknecht and P. Stone, "Deep recurrent q-learning for partially observable mdps", in *2015 aaai fall symposium series*, 2015.

[29] V. Mnih, A. P. Badia, M. Mirza, et al., "Asynchronous methods for deep reinforcement learning", in *International conference on machine learning*, PMLR, 2016, pp. 1928–1937.

[30] J. Peters and S. Schaal, "Natural actor-critic", *Neurocomputing*, vol. 71, no. 7, pp. 1180–1190, 2008, Progress in Modeling, Theory, and Application of Computational Intelligenc, ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2007.11.026. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231208000532.

[31] F. AlMahamid and K. Grolinger, "Reinforcement learning algorithms: An overview and classification", in *2021 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, IEEE, Sep. 2021. DOI: 10.1109/ccece53047.2021.9569056. [Online]. Available: http://dx.doi.org/10.1109/CCECE53047.2021.9569056.

[32] R. Girshick, "Fast r-cnn", in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Dec. 2015.

[33] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection", in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016.

[34] G. Jocher, A. Chaurasia, and J. Qiu, *Ultralytics yolov8*, version 8.0.0, 2023. [Online]. Available: https://github.com/ultralytics/ultralytics.

[35] 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.

[36] M. Towers, J. K. Terry, A. Kwiatkowski, et al., *Gymnasium*, Mar. 2023. DOI: 10.5281/zenodo.8127026. [Online]. Available: https://zenodo.org/record/8127025 (visited on 07/08/2023).

[37] *Gymnasium documentation*, May 2024. [Online]. Available: https://gymnasium.farama.org/api/vector.html.

[38] F. Ulrich and T. Wehrli. "Ba-e2e-seldr repository". (2024), [Online]. Available: https://github.com/joschi27/BA_E2E_SELDR.

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

[40] N. Vödisch, D. Dodel, and M. Schötz, "Fsoco: The formula student objects in context dataset", *SAE International Journal of Connected and Automated Vehicles*, vol. 5, no. 12-05-01-0003, 2022.

[41] B. Osiński, A. Jakubowski, P. Zięcina, et al., "Simulation-based reinforcement learning for real-world autonomous driving", in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, 2020, pp. 6411–6418. DOI: 10.1109/ICRA40945.2020.9196730.

[42] R. Chopra and S. S. Roy, "End-to-end reinforcement learning for self-driving car", in *Advanced Computing and Intelligent Engineering*, B. Pati, C. R. Panigrahi, R. Buyya, and K.-C. Li, Eds., Singapore: Springer Singapore, 2020, pp. 53–61, ISBN: 978-981-15-1081-6.

[43] U. Tewari, "Which Reinforcement learning-RL algorithm to use where, when and in what scenario?", *Medium*, Dec. 2021. [Online]. Available: https://medium.datadriveninvestor.com/which-reinforcement-learning-rl-algorithm-to-use-where-when-and-in-what-scenario-e3e7617fb0b1.

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

[45] N. Barhate, *Minimal pytorch implementation of proximal policy optimization*, https://github.com/nikhilbarhate99/PPO-PyTorch, 2021.

[46] *Bounding boxes - CARLA Simulator*, [Online; accessed 26. May 2024], Mar. 2024. [Online]. Available: https://carla.readthedocs.io/en/latest/tuto_G_bounding_boxes.

[47] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild", *Science Robotics*, vol. 7, no. 66, eabm6074, 2022. DOI: 10.1126/scirobotics.abm6074. [Online]. Available: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074.

[48] A. E. Sallab, M. Abdou, E. Perot, and S. Yogamani, "Deep reinforcement learning framework for autonomous driving", *Electronic Imaging*, vol. 29, no. 19, pp. 70–76, Jan. 2017, ISSN: 2470-1173. DOI: 10.2352/issn.2470-1173.2017.19.avm-023. [Online]. Available: http://dx.doi.org/10.2352/ISSN.2470-1173.2017.19.AVM-023.

[49] H. Salehinejad, S. Sankar, J. Barfett, E. Colak, and S. Valaee, *Recent advances in recurrent neural networks*, 2018. arXiv: 1801.01078 [cs.NE].

[50] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory", *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997, ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735.

[51] R. Girdhar, J. Carreira, C. Doersch, and A. Zisserman, "Video action transformer network", in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2019.

[52] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world", in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017, pp. 23–30. DOI: 10.1109/IROS.2017.8202133.

List of Figures

Acronyms

CNN Convolutional Neural Network. 6, 50

E2E end-to-end. 1, 2, 4–6, 29, 48, 49

F:SAE Formula SAE. 6

FS Formula Student. 1, 2, 7, 11, 12, 21, 27, 31, 47, 50

FSG Formula Student Germany. 7–10

IL Imitation Learning. 5, 6

MDP Markov Decision Process. 15

POMDP Partially Observable Markov Decision Process. 15, 51

PPO Proximal Policy Optimization. 18, 31, 35, 37, 60

RL Reinforcement Learning. , 2, 3, 6, 12–14, 16–18, 21, 25, 27, 35, 52, 57

TRPO Trust Region Policy Optimization. 18

ZUR Zurich UAS Racing. , 1, 2, 5, 7, 11, 24, 25, 31, 35, 49–51, 57, 62

A. Appendix

A.1. Project Management

A.1.1. Milestones

Weekly meetings with the supervisor were held during the project, where milestone progress, issues and other updates were discussed. Table 6 shows the defined milestones.

Table 6.: Project milestones and due dates

Milestone	Due Date	Description
#1 - Simulator setup	22.02.24	CARLA installed with UE4Editor, Source control including UE assets
#2 - Gym-Style envs	14.03.24	Implemented Gym-Style environments for the different maps in CARLA
#3 - Implementation validation	28.03.24	Validated correctness of PPO implementation by implementing a prototype that has access to privileged simulator information (position, rotation etc.)
#4 - Vision MVP	03.04.24	Implemented MVP that uses non-privileged sensor information (camera, accelerometer etc.) to navigate acceleration track
#5 - Final vision implementation	09.05.24	Implemented final prototype, near-final reward functions for all disciplines, hyperparameter tuning
#6 - Project report done	01.06.24	Near-final draft of thesis report done

A.1.2. Official Task Description (Complesis)

The highly regarded international engineering competition Formula Student (FS) challenges student teams to design, build, and race a Formula-style racing car. In addition to the Combustion Vehicle (CV) and Electric Vehicle (EV) events with human drivers, the Driverless Cup (DC) is a competition that challenges student teams to create an autonomous system that maneuvers the race car. The event provides a platform for students to improve and showcase their innovative ideas and technical skills in the field of autonomous driving. Launched in 2019, the Zurich University of Applied Sciences Racing (ZUR) team built a fully functional car in 2021 and successfully participated in the EV events in 2022 and 2023. ZUR's goal is to further boost its performance in 2024 with a new and improved car.
Developing autonomous racing vehicles is an exciting and challenging endeavor. The driverless team is organized into groups developing object perception, simultaneous localization and mapping (SLAM), trajectory planning, and vehicle control. Object Recog-

nition identifies the relative position and color of the cones that mark the racetrack. Using this information, a SLAM algorithm allows the vehicle to create a map of its environment while simultaneously determining its position within that map. Trajectory planning involves generating optimal paths in the map for the vehicle to follow. Vehicle control involves the implementation of algorithms that manage the vehicle's accelerator, brake, and steering systems to follow the path determined by the trajectory planning algorithm.
Currently, our autonomous racing car systems are deployed on an NVIDIA Jetson Xavier using ROS2 Foxy Fitzroy, ZED 2i camera, and software written in Python as well as C++. We use AWS for cloud services, all running in a Linux environment. Familiarity with these technologies is an advantage, but not a requirement. We are committed to fostering a learning environment where skills can be developed. Equally important is the team experience, we encourage frequent presence in our workshop to work directly with our members. It's not just about building a car, it's about building a team. Being there and achieving goals together is part of the fun and learning process.
The field of AI is transitioning from addressing specialized, deterministic problems to contending with broader, more complex problems. This shift from narrow AI to artificial general intelligence has been largely enabled by the advent of deep learning techniques. End-to-end deep learning represents a paradigm where a complex system is learned as a single unified function, mapping raw inputs to desired outputs without explicitly designed intermediate steps or representations. This approach, powered by deep neural networks capacity to encapsulate relationships and dependencies within their hidden layers, has found widespread adoption across various domains such as natural language processing, image recognition, and autonomous driving. In autonomous driving, end-to-end learning should be capable of directly comprehending the mapping from raw sensor data to vehicle control commands, circumventing the need for explicitly programmed intermediary steps, like object detection or path planning. In this Thesis, you can explore possibilities for such an approach together with the Centre for Artificial Intelligence and Zurich UAS Racing.

A.2. Additional Information

A.2.1. Training Parameter Definitions

NUM_EPISODES is the amount of episodes the simulation is run for. NUM_AGENTS is the amount of agents deployed. Notably, in the trackdrive discipline, only eight tracks were created as placing cones by hand was time consuming. MAX_EPISODE_LENGTH is the maximum amount of steps an episode can have before it is forced to end. Once UPDATE_FREQUENCY amount of steps are completed, the actor and critic networks are updated. GAMMA is the discount factor γ as explained in 3.3.2. LR_ACTOR and LR_CRITIC are the learning rates of the agent and critic networks respectively.
K_EPOCHS specifies the number of epochs for which the policy is updated in each PPO update cycle. EPS_CLIP is the epsilon value used to clip the probability ratios during the PPO update to ensure stability as explained in 3.3.9. ACTION_STD is the starting standard deviation for the action distribution as explained in 4.3.4. ACTION_STD_DECAY_RATE defines the rate at which the action standard deviation decays. MIN_ACTION_STD sets the minimum threshold for the action standard deviation. ACTION_STD_DECAY_FREQ specifies the frequency (in steps) at which the

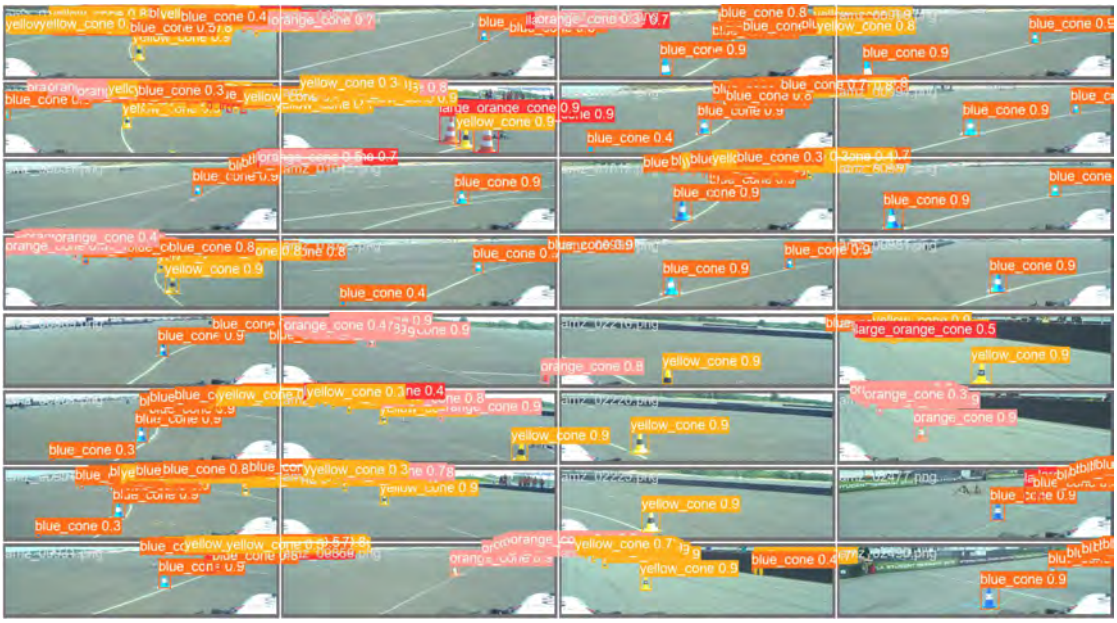
Table 7.: Parameter definitions for acceleration, skidpad, and trackdrive

Parameter	**Acceleration**	**Skidpad**	**Trackdrive**
NUM_EPISODES	1000	10000	8000
NUM_AGENTS	20	20	8
MAX_EPISODE_LENGTH	250	300	400
UPDATE_FREQUENCY	750	900	1200
GAMMA	0.90	0.90	0.90
LR_ACTOR	0.0005	0.0005	0.0005
LR_CRITIC	0.0005	0.0005	0.0005
K_EPOCHS	10	10	10
EPS_CLIP	0.2	0.2	0.2
ACTION_STD	0.6	0.6	0.6
ACTION_STD_DECAY_RATE	0.003	0.0002	0.00016
MIN_ACTION_STD	0.1	0.1	0.1
ACTION_STD_DECAY_FREQ	1000	1000	1000
SAVE_FREQUENCY	100	100	100
CAMERA_FOV	90	90	90
MAX_CONE_DISTANCE	20	20	20
NUM_DETECTIONS	10	10	10

action standard deviation is decayed. SAVE_FREQUENCY determines how often the model's state is saved during training.
CAMERA_FOV is the field of view of the camera in degrees. MAX_CONE_DISTANCE defines the maximum distance at which a cone can be detected by the sensors. This is used for the normalization of the distance. NUM_DETECTIONS indicates the maximum number of cone detections passed to the observation.

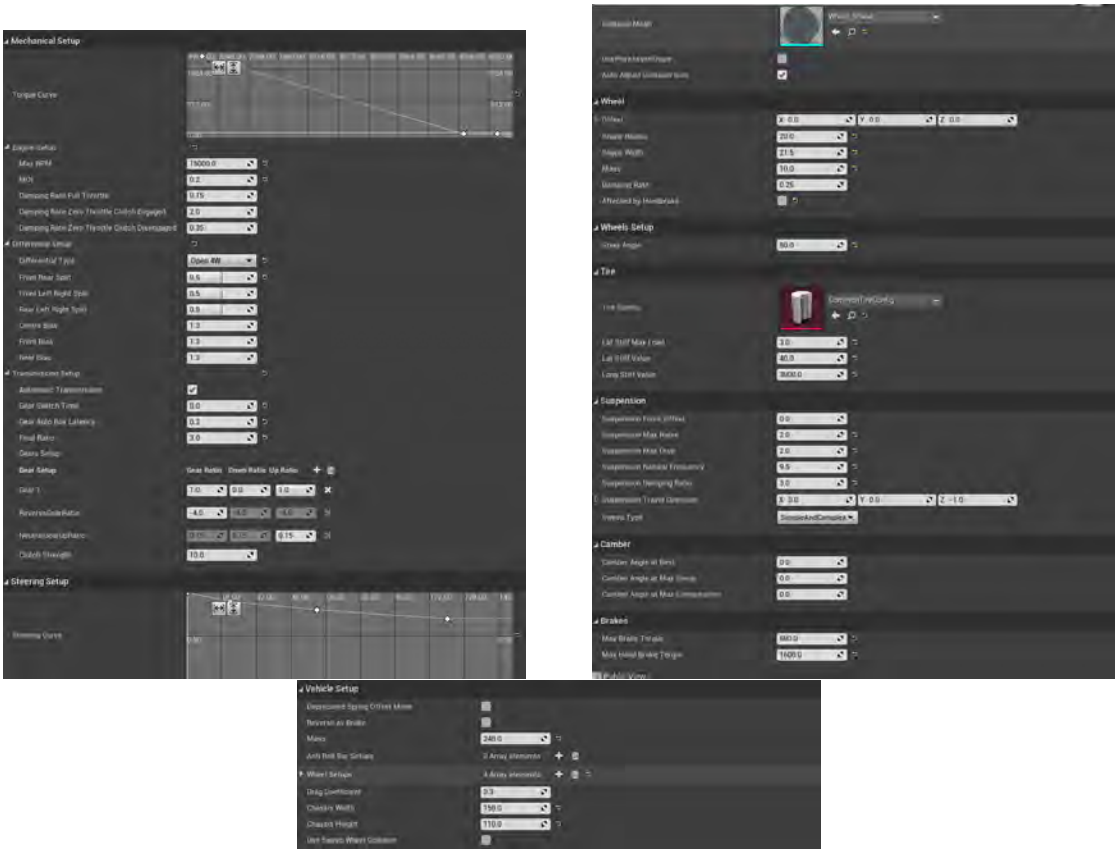
A.2.2. YOLO validation samples

The following are prediction outputs from the validation set during the YOLO training.



A.2.3. ZUR Vehicle setup in CARLA

In this section, the vehicle settings for the ZUR car blueprint in CARLA are documented:



A.2.4. Code

For this project, more than 10000 lines of code were created. The original repository is hosted on the internal ZHAW GitHub, which is not publicly accessible. Due to this, only a copy of the repository is hosted on [38].

A.2.5. Artificial Intelligence declaration

The authors of this work acknowledge that certain parts of the text and code have been generated or assisted by openAI's large language model, ChatGPT. Specifically, ChatGPT was used to help refine wording, provide rephrasing suggestions, and assist in the initial drafting of some sections as well as generating certain code passages, finding bugs and supporting coding structure.