DHBW
Duale Hochschule
Baden-Württemberg
Ravensburg
Campus Friedrichshafen

GLOBAL FORMULA RACING
GFR
DHBW RAVENSBURG / OREGON STATE UNIVERSITY

# Development of a Reinforcement Learning Algorithm for Planning and Control of a Formula Student Race Car

**STUDIENARBEIT**

in the course of studies

Electrical Engineering/Communication Engineering

at the Dualen Hochschule Baden-Württemberg Ravensburg

Campus Friedrichshafen

written by

Graf Phillip

June 2023

| | |
|---|---|
| Process time | 10.10.2022 - 17.07.2022 |
| Student ID | 3295951 |
| Project Advisor | Bjarne Johannsen |

# Declaration

appropriate to § 1.1.13 of attachment 1 to §§ 3, 4 and 5 of the „Studien- und Prüfungsordnung für die Bachelorstudiengänge im Studienbereich Technik der Dualen Hochschule Baden-Württemberg“ of the 29th of September 2017 in the version of the 25th of July 2018.

I hereby certify that I have written my thesis with the topic:

**Development of a reinforcement learning algorithm for planning and control of a Formula Student Race Car**

independently and have not used any other sources and aids than those indicated. I also confirm that the digital version corresponds to the printed document.

FN, 13.01.2023

_____

Phillip Graf

## Abstract

The aim of this work is to train a policy network that controls a vehicle over the track, by using the Proximal Policy Optimization algorithm. The outputs of the trained network are the acceleration value and the steering angle of the vehicle. In the final stage of this work, its inputs consist of the current velocity and the coordinates of the next twenty track cones left and right. The prospect of achieving better driving performance than with model-based approaches and at the same time reducing the number of autonomous system components holds significant promise.

The initial situation is an established simulator and a self-implemented Proximal Policy Optimization algorithm from the last project contributors. The result of this work is a policy that can drive a wide variety of track segments. This ability proved the functionality of the developed system and opens the way for the development of a performant policy in the next project phase.

The existing system was incrementally analyzed and adapted to the goal of a driving policy. Key contributions are namely the substitution of the self-implemented algorithm with the RLlib module, the change of distance ray inputs to cone coordinate inputs for the policy, and a semi-automated solution to find good reward parameters.

The successful work to develop an overall driving policy also revealed more opportunities for improvement for the next project phases. Namely, the generation of more tracks to train the policy on and the further fine-tuning of the reward function.

# Kurzfassung

Das Ziel dieser Arbeit ist es, ein Policy-Netzwerk zu trainieren, welches ein Fahrzeug über die Strecke steuert, indem ein Proximal Policy Optimization Algorithmus verwendet wird. Die Ausgaben des trainierten Netzewerkes sind der Beschleunigungswert und der Lenkwinkel des Fahrzeugs. In der letzten Phase dieser Arbeit bestehen seine Eingaben aus der aktuellen Geschwindigkeit und den Koordinaten der nächsten zwanzig Strecken Kegel links und rechts. Die Aussicht ist vielversprechend, bessere Fahrleistungen als mit modellbasierten Ansätzen zu erzielen und gleichzeitig die Anzahl der autonomen Systemkomponenten zu reduzieren.

Die Ausgangssituation ist ein entwickelter Simulator und ein selbst implementierter Proximal Policy Optimization Algorithmus der letzten Projektmitarbeiter. Das Ergebnis dieser Arbeit ist eine Policy, die eine Vielzahl von Streckenabschnitten fahren kann. Dieses Ergebnis hat die Funktionalität des entwickelten Systems bewiesen und ebnet den Weg für die Entwicklung einer leistungsfähigen Policy in der nächsten Projektphase.

Das bestehende System wurde inkrementell analysiert und an das Ziel einer fahrenden Policy angepasst. Die wichtigsten Beiträge sind das Austauschen des selbst implementierten Algorithmus durch das RLlib-Modul, die Umstellung der Eingaben von Entfernungsstrahlen auf Kegelkoordinaten für die Policy Eingaben und eine halbautomatische Lösung zum Finden guter Belohnungsparameter.

Die erfolgreiche Arbeit an der Entwicklung einer generell fahrenden Policy zeigte auch weitere Verbesserungsmöglichkeiten für die nächsten Projektphasen auf. Nämlich die Erstellung weiterer Strecken, auf denen die Strategie trainiert werden kann, und die weitere Feinabstimmung der Belohnungsfunktion.

# Table of Content

# List of abbreviations

| | |
|---|---|
| DQN | Deep Q-Network |
| FSAE | Formula Society of Automotive Engineers |
| GFR | Global Formula Racing |
| JSON | Java Script Object Notation |
| MDP | Markov Decision Process |
| OSU | Oregon State University |
| PPO | Proximal Policy Optimization |
| RL | Reinforcement Learning |
| TRPO | Trust Region Policy Optimization |

# 1. Project Description

## 1.1 Introduction

Participating in races is ultimately choosing a compromise between quick lap times and accident risk in favor of the lap times. In normal road traffic, you want to get from place a to b as safely as possible, which is why you accept major sacrifices in transport times. After all, you are still significantly faster than if you were walking.

In races, however, the aim is to test the hardware of the vehicle and the software of the autonomous system or the skill of the driver for the best performance. That is why move the vehicle so fast that the increased probability of a crash barely allows you to reasonably safely reach the finish line. That is why the vehicle is moved so fast that the increased probability of a crash barely allows it to reach the finish line reasonably safely. The reinforcement learning agent is an effort to push this explained compromise further towards better lap times on an autonomously controlled vehicle while having lowered crash probabilities at the same time. This is possible by the ability of the agent to determine more optimal trajectories and follow them with higher speed through more precision in controlling the car.

Thus this project aims to use a reinforcement learning algorithm to control the driverless car. To control the car, a neural network is used which is trained within a simulator. It takes the current velocity and seven over 180-degree distributed rays to the track outline as input and outputs the acceleration and steering angle for achieving a good as possible lap time. The input for this policy network is also changed as a major improvement in the course of this project phase.

The approach would combine three currently used system elements: the boundary planner, trajectory planner, and dynamic controls. The benefits are that the trained policy is very likely to observe better trajectories than conventional algorithms, resulting

in better lap times. With a successful implementation, we would also be the first team in the Formula Society of Automotive Engineers (FSAE) to use reinforcement learning for planning and control.

The project started last year with an implementation of a proximal policy optimization approach and an according simulator. The goal this year is to customize the training system and rewards so that a policy can be trained to drive many different track segments, proving the functionality of the system. The creation of sufficient documentation for the system is also made up for.

## 1.2 Rules and Constraint Analysis

From the Formula Student Rules 2023 file I could identify relevant rules for this project from the two chapters General Technical Requirements and Dynamic Events (Formula Student Germany GmbH). For an overview, they are listed here and discussed below.

General Technical Requirements

T 14.2 Data logger

T 14.3 Remote Emergency System

T 14.11 Autonomous Missions

Dynamic Events

D 2.3.2 Vehicles must not be driven in reverse.

D 3.1.1 The following track conditions are recognized: Dry, Damp, Wet

D 3.1.2 The operating conditions are decided by the officials and may change at any time.

From rule T 14.11 the following missions should be considered:

D 6 Autocross Event

D 8 Trackdrive Event

The first two rules of the General Technical Requirements are relevant for the integration of the agent into a real car and do not play a major role in the development. For example, how is the data prescribed in T 14.2 fed to the data logger? The integration question also arises for the two significant roles of the Remote Emergency System described in T 14.3. It not only turns the car off remotely during an emergency but also starts a preselected mission.

Rule T 14.11 lists the missions, in which an autonomous system has to participate, as its name suggests. They are Acceleration, Skidpad, Autocross, Trackdrive, EBS Test, Inspection, and Manual Driving. The reinforcement learning agent is currently developed for the Autocross and Trackdrive missions.

Rule D 6.1.3 states that the same track is used for both events, trackdrive and autocross. At the autocross event, one lap is driven over a track, and the trackdrive event is about ten continuous laps on the track. Besides training a fast-driving agent, there are two further challenges conceivable. The vehicle was placed with little distance to a starting line for both events. Here it shall be examined carefully that the agent does not recognize the starting line as the finish line and thus stops right after the first bit of acceleration. For the trackdrive event, it is furthermore stated in D 8.3.5 there is no lap signal and the autonomous vehicle has to count laps itself.

The rules of the Dynamic Event section give further aspects to think about. D 2.3.2 states that the vehicle must not be moved in reverse. Hence the implementation of the agent should not even allow the agent to perform a backward acceleration.

Furthermore, races are carried out in dry and wet conditions. Therefore two different tire types are allowed for use. Here should be thought about how to incorporate this into the training of the algorithm. Probably the change is so significant that two different trained agents are needed for a suitable solution. Or may an additional input flag is introduced, that signals the policy if the track is dry or wet. During training, only the friction values of the tires would then have to be adjusted as well as the moisture flag provided to the policy. In the vehicle, however, the computer vision system must then also recognize the weather situation or the flag is manually set to dry or wet before the start.

One essential run time requirement is the frequency of the policy runs. This means how many times a second should the values from lidar and camera be processed by the agent and the control parameters updated accordingly. The team behind the Cassie bipedal robot of Oregon State University (OSU) targets 20 to 50 Hertz (Warila). For this project 30 Hertz as a minimum seems also reasonable. This can be illustrated by imaging a car that drives a long corner at 100 km/h. Here it moves with a speed of around 27,8 meters per second. So there would be at least one update per meter moved, which should be enough to control the car with a good performance.

An important point to consider is the integration of the project into the working environment, once it is proven to work fine. For this matter, the use of pipelines from the Python code to the C++ environment prevents the efforts of rewriting the Zenith project in C++.

# 2. Fundamentals

## 2.1 Reinforcement Learning

Most machine learning methods can be split into the categories of supervised and unsupervised. In supervised machine learning, large data sets with labeled data are used. Here, the label means that the output expected by the network is already known for a data field. During training, this data is introduced into the network. Then, depending on how much the output of the network deviates from the label of the data field, the parameters of the network are adjusted. So that when the same input is entered again, the value of the label is with greater probability output. In this way, the more extensive and more diverse the available labeled data, the better the neural network can be trained. Unsupervised machine learning on the other hand is mostly about finding structures in unlabeled datasets. Reinforcement learning (RL) is a type of machine learning too but does not really fit into these classes. RL is based on unlabeled input data, but its objective is to maximize the rewards signal and not to find hidden structures in datasets (Sutton and Barto 3).

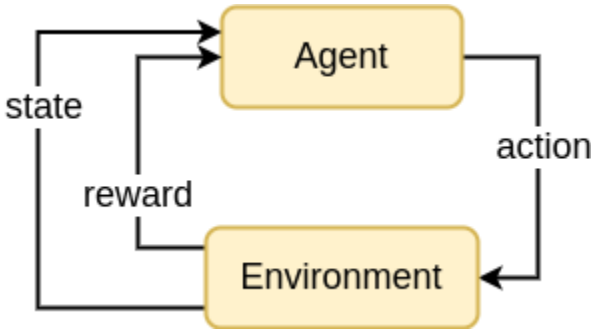


Figure 2.1 RL overview

In Figure 2.1 can be observed the main parts of a RL system (Sutton and Barto):

- The agent which makes decisions
- The environment the agent interacts with
- The policy used by the agent to decide on an action (a neural network)
- The reward given back to the agent after taking an action

As figure 2.1 illustrates, reinforcement learning is learning through interaction and behaving as to reach a certain result. The interaction between the *agent* and *environment* occurs in sequences of discrete time steps. The interaction is defined by *actions*, decided by the *agent* based on the current state. The state is provided by the *environment* and gets updated, with every action taken. The *policy* used by the agent to make decisions is a stochastic rule that maps states to actions. Maximizing the achievable cumulative reward is the objective of the agent. The policy defines the agent's behavior (Sutton and Barto 80, 81).

A measure of the cumulative reward obtained by an agent over a succession of time steps is provided with the *return* function $G_t$. It is calculated from the obtained rewards $R_t$ after a series of interactions with an environment. Usually, the rewards are discounted by a defined factor $\gamma$, the farther they are in the future, as can be seen in the formula below. Thus the discount factor controls the relative importance of immediate rewards versus future rewards (Sutton and Barto 59, 60). The practical value of this is to account for the value of time. As an example of the current problem, rewarding finishing a track segment gives a higher return the faster the end is reached. More time steps in between mean a bigger discount defined by the factor.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+1+k} \qquad (2.1)$$

The mathematical framework used in reinforcement learning for the decision making problem is Markov Decision Processes (MDP). It describes the structure of the agent and environment as described above, interacting through states and actions based on a

policy. To make use of MDP as a framework, the environment has to fulfill the Markov Property. It states that the current state of the agent “depends only on its immediate previous state (or the previous timestep)” (Jagtap).

During the training, the actor is selecting actions from the probability distribution provided by the policy. Since the policy ought to be optimized during training, it is not reasonable to already follow the distribution exactly. In order to pursue the objective of maximizing the cumulative rewards, the actor should choose actions that are the most likely to yield the highest reward. This decision-helping information, which action leads to the highest return (cumulative reward), is estimated by the so-called action-value function (Sutton and Barto 70).

To summarize the previous elements: in reinforcement learning the policy, the agent and value function are interconnected components that work together to learn an optimal policy for the agent. The policy defines how the agent should behave in a given environment, by mapping states to actions. The agent interacts with the environment, and the policy is updated based on the feedback received from the environment. The value function estimates the expected future rewards that the agent can obtain by following the current policy. This is used to guide the learning process and to evaluate the quality of the policy. The policy and value functions are learned simultaneously, and they depend on each other. The value function helps to improve the policy by providing feedback on the quality of the actions taken, while the policy helps to improve the value function by exploring the state-action space more effectively.

Some projects that highlight the effectiveness of reinforcement learning are the board-playing agents AlphaZero and AlphaGo trained by DeepMind in chess and Go. The latter even defeated the world champion in 2016 (“AlphaGo”). Further fields of appliances are robotics, autonomous vehicles, healthcare, advertising, and energy management. The two-legged walking and running robot Cassie is one more example of the robotics field.

## 2.2 Proximal Policy Optimization

Proximal Policy Optimization (PPO) is a reinforcement learning algorithm that uses an actor-critic approach and combines it with trust region optimization techniques. It has been used in many successful applications such as OpenAI's DOTA 2 AI and DeepMind's AlphaGo Zero. The main objective of PPO is to find a policy that maximizes expected return by following a stochastic policy that lies within some distance from an initial or previously learned policy, called the Proximity Threshold. This constraint helps ensure that any changes made to the policy are small enough so that they do not significantly affect performance negatively (Van Heeswijk).

The PPO algorithm is selected over others like Deep Q-Network (DQN) or Trust Region Policy Optimization (TRPO) for multiple reasons. Mainly PPO is able to work with continuous action spaces (Warila). DQN lacks this characteristic and can only operate in discrete action spaces. The continuous action space means that float numbers can be expected as the action outputs of the policy. A discrete action space would give full integer numbers as action output, which is not appropriate for the control of a vehicle throttle and steering. Also, PPO is more stable during training than DQN. TRPO and PPO share "data efficiency and reliable performance" (Schulmann et al. 1). Here higher data efficiency means that fewer data samples are required from the environment for the optimization. But TRPO involves more complex mathematical calculations to compute an optimal policy update (Schulmann et al. 1). This makes it more challenging to implement than PPO and requires more computation time.

This chapter attempts to explain the mathematics and code implementation of the PPO reinforcement learning algorithm used behind the "train" command in Zenith. For this purpose, a complete algorithm diagram is provided in Figure 3.3 (Yu, PPO Part 1). This is followed by an explanation of every step and for some parts how they are implemented.

**Algorithm 1** PPO-Clip

1: Input: initial policy parameters $\theta_0$, initial value function parameters $\phi_0$
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function $V_{\phi_k}$.
6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left( \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \quad g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

typically via stochastic gradient ascent with Adam.
7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left( V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.
8: **end for**

Figure 3.3 PPO RL algorithm diagram by Eric Yang Yu (Yu, PPO Part 1)

The first step in the diagram is creating two feed-forward neural networks of arbitrary structure that represent the policy and value function. At this step could also pre-trained neural network weights with the same structure be loaded. It is common for PPO to be implemented using an actor-critic architecture because this approach has been shown to be effective for many reinforcement learning tasks. In the actor-critic architecture, there are two neural networks (as initialized above): an actor network that represents the policy and selects actions based on the current state, and a critic network that estimates the value function of the state. PPO updates the policy by computing a surrogate objective function that encourages the policy to stay close to the old policy while improving its performance based on the critic's estimation of the value function (Sutton and Barto 257, 258).

The second step represents the main training loop. The repetitions of this loop are known as iterations. It is usually stopped after a predefined number of iterations. Another abort condition could be that the average achieved return is not increasing significantly enough over a specific number of iterations. Therefore it would not be worth to train further.

Step three is the sampling of data from the environment with the policy network and its current weights. $\pi(\theta)$ usually denotes a policy network with the weights $\theta$. The obtained samples are stored in batches. A single sample contains three elements state, chosen action, and achieved reward.

With a specified amount of data sampled, the returns of the single trajectories are calculated in step four of the diagram. A trajectory is a collection of samples from one simulator run, from a start state to a terminal state. Returns are the cumulative reward received for all the subsequent states and actions taken after the current state. In the diagram are mentioned rewards-to-go. This is also a cumulative reward, but as the name suggests without taking the current reward into account, only looking at the steps after that. Both return and rewards-to-go can be used to process the advantages and optimize the value function. The here used return is calculated for every sample state in the collected batch.

As can be seen in Formula 2.1, there is a parameter $\gamma$ introduced to give values that are farther in the future a lower rating. The parameter gamma is a value between 0 and 1 that weighs the importance of instant rewards versus future rewards. It determines how much an agent values rewards that are further in the future. Values closer to 0 place more emphasis on immediate rewards, and values closer to 1 place more emphasis on future rewards. By incorporating the discount factor gamma into the return calculation, PPO is able to balance the trade-off between exploring new actions and exploiting known good actions.

Code Block 2.1 presents an example of how the returns could be computed. The passed batch object in it is a list of trajectories. Each trajectory is in turn a list of samples from a single track run. A sample consists of the elements state, action, and reward. Only the rewards are needed for the calculation of the return. The computed return is attached to each list. Formula 2.1 shows that it is efficient to start at the end of each trajectory and work backward since the return of each sample can be reused for the cumulative reward of its processor (Yu, PPO part 2).

```python
def compute_return(self, batch):
    # Algorithm Step 4
    # Batch is a list of trajectories
    for trajectory in batch:
        discounted_reward = 0
        # Sample structure = [state, action, reward]
        for sample in reversed(trajectory):
            reward = sample[2]
            discounted_reward = reward + discounted_reward * self.gamma
            sample.append(discounted_reward)
    return batch
```

Code Block 2.1 Calculating Returns

In step five the advantage is estimated to get a reference of how much better a new policy is compared to the policy before. This is the difference between the actual return of the state and the expected return. The actual return was calculated with the discount factor in step four. The expected return was a guess by the value function. If the

advantage is positive which indicates the value function was too optimistic with its estimation. Vice versa a negative advantage indicates the value function was too negative about the cumulative return for a state. This way the advantage provides a quality estimate of the actions decided by the policy. It helps to adjust the policy towards a state-action mapping that leads to higher rewards (Yu, PPO part 3).

// may include the standard GAE advantage method

The formula in step six returns the instruction of how to update the weights θ of the actor-network (the policy π(θ)). Most of the elements in it are already computed. The implementation for this and the next step is shortened only to details, to keep the amount of text within limits. Beginning with the ratio of policies $\pi_\theta(s_t, a_t)$ and $\pi_{\theta k}(s_t, a_t)$, both return a scalar probability value of taking an action a in state s, not a probability distribution when only provided with a state. The upper policy is the current one, and the lower policy is the one before the last update. If the ratio is greater than one, the current policy is more likely to take action a in the state s than the old policy. A value smaller than one signals that the current policy is less likely to take the action a in the state s than the last policy. Multiplied with the advancement value $A^{\pi\theta k}(s, a)$ of the last policy this gives a measurement in quantity and direction if the development regarding the two policies was positive or negative regarding the objective to maximize the cumulative reward. Regarding the use of the current advantage values or the ones of the last policy, both approaches seem to be plausible. Though in the original PPO paper of 2017, $A^{\pi\theta}(s, a)$ is used in the surrogate objective (Schulmann et al. 3).

The second term in the minimize function is g(). It too takes the advantage values and a parameter epsilon as input. Internally it also multiplies the advantage with the same ratio of policies, but if the value is smaller than 1 - ε or greater than 1 + ε it is clipped. A better visualization for this is the following representation of the same formula in step six (Schulmann et al. 3).

$$L^{CLIP}(\theta) = E_t\left[min(r_t(\theta)\,\hat{A}_t,\ clip(r_t(\theta),\ 1-\varepsilon,\ 1+\varepsilon)\,\hat{A}_t)\right] \qquad (2.2)$$

In Formula 2.2 above the $r_t(\theta)$ term is the ratio between the policies. $E_t[]$ represents the expectation of an empirical average after a number of iterations between sampling and optimization. The representation within $E_t[]$ makes it more apparent that if the policy ratio multiplied by the advantage is further away from one, it is clipped to the range defined by epsilon. The min() term finally selects the smaller of both objectives, “so the final objective is a lower bound (i.e., a pessimistic bound) on the unclipped objective” (Schulmann et al. 3). This way only too large steps that improve the objective are avoided. Still, they are included if they worsen the objective.

The sigma on the right sums up the computed $L^{CLIP}$ values for all samples in the batch. Or put differently, for all timesteps T in the batch. Then with the multiplication by 1/T, the $L^{CLIP}$ average of all the samples is retrieved, which is optimized in the end. The left sigma is not exactly mentioned but I suppose it shall indicate that the same batch of samples can be used for multiple optimizations of the policy, so-called epochs.

To finally minimize the loss and optimize the policy, the Adam optimizer of Pytorch is used. To get Adam to minimize the loss instead of maximizing it, as is indicated by the arg max in the formula, the loss is provided as negative. A maximized negative loss is ultimately minimized (Yu, PPO part 3). I suppose Adam is used instead of direct Stochastic Gradient Descent since it provides an automatic learning rate adaption based on historical gradient data.

In the final step seven of each reinforcement learning iteration, the critic network weights are optimized. For this purpose, the mean squared error of the predicted values with the current values is formed. The mean squared error (MSE) is used to assess the quality of an estimator.

$$MSE = \frac{\sum_{i=1}^{n}(Y_i - \widehat{Y}_i)^2}{n} \qquad (2.4)$$

In the mean squared error Formula 2.4, $Y_i$ is the observed value, $\hat{Y}_i$ is the predicted value and n is the number of observations. It returns “the average squared distance between the observed and predicted values” (Frost). The order of the two values in the counter does not matter due to the quatration. Due to the quatration, also only positive distances are returned and larger errors are weighted more heavily (Frost). So the mean squared error is calculated over all samples in the batch and then used for the update of the critic network weights with a gradient descent algorithm. With this last step finished the training loop is beginning all over again.

The main innovation in PPO is the surrogate objective function explained in step six. This lead to comparatively very stable training that does not quickly fall into local maxima and gets stuck there. There are many different implementation details like the advantage estimation that can have a significant impact on the result. Nonetheless, with the explanation of this chapter, a good foundation for further work is created.

## 2.3 Ray RLlib

In the course of the project, the decision was made to test a provided framework for the reinforcement learning algorithm implementation. Speaking about one of the current industry standards RLlib from the Ray ecosystem. It provides 25 reinforcement learning algorithms, PPO included. Without further effort, a multi-agent training mode is possible, which makes use of multiple processor cores. Further points that clearly support it (“RLlib”):

- Safer algorithm implementation
- Simplifies architecture by multitudes
- Automated hyperparameter search (with Ray Tune)
- Documentation is fine - many examples available

The terminology in RLlib is as follows: algorithms like PPO or TRPO are used to solve problem environments. The policy of an algorithm selects actions. And sample batches

(trajectories) are obtained by rollouts of the environment. Basically, the RLlib module provides a complete black box for the agent in Figure 2.1.

The concept of a general black box for any reinforcement learning agent is possible because environments can broadly be defined as black boxes with standard interfaces too. Environments consist of defined action and observation spaces. The observations represent the complete state of the agent. Environments are expected to return a reward signal after every action taken (“RLlib Key Concepts”). The actions and observations are defined by the so-called public gymnasium space attributes of the environment. These are set in the initialization of the environment object. Gymnasium is a widely used standard Application Programming Interface module for reinforcement learning. It also provides a base class for environments.

During training and evaluation of agent policies, the only contact points to the environment are the methods reset and step as can be seen in Figure 2.3. Reset is used to bring the environment into the start position of a trajectory. The step method takes an action, performs the according changes in the environment, and returns the updated observation, a reward value for the step taken, and a boolean done signal. If the done signal is true, a reset is applied to start a new trajectory (“RLlib Environments”).

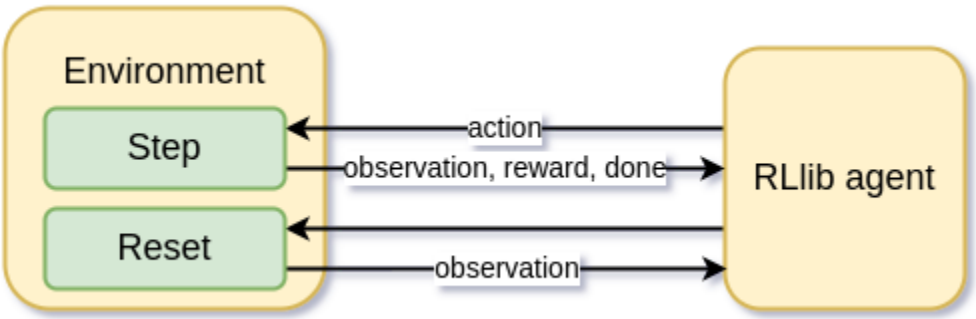


Figure 2.3 RLlib agent environment interaction

Starting as a small experiment, the RLlib module turned out to be of great importance for the success of the project. Primary with its large simplifying effect on the architecture and its correctly implemented reinforcement learning algorithms.

# 3. Work on Zenith

## 3.1 Investigation of the Takeover State

At the beginning of the project I meet with the two American team members who are also working on Zenith. They showed me the setup steps for Zenith that they got from a meeting with our predecessor. After a few days of figuring out how to create virtual environments that make use of the correct Python version and solving dependency issues, the setup was done. Running an initialization bash script without knowing what it does, created a number of random track segments and compiled the Global Formula Racing (GFR) simulator into an executable binary file. The next step was to understand how the commands work. A format description was provided in the latest project report, but a simple example could have saved an hour. This way I figured out how Zenith is used to train policies and how they can be evaluated. The processes were working, but the training itself was not successful.

Zenith's goal is to train policies through a PPO algorithm. The required environment is created by interfacing the GFR C++ simulator binary through the CDLL module in Python. For the implementation of the PPO algorithm, no source was documented. The sampling of data in the environment was already done in a Ray worker, which enabled distributed processes over multiple processor cores. The accumulating data of rewards and episode length was logged into a Tensorboard SummaryWriter object. The resulting file could later be read to see the collected data points in diagrams. This visualization allows a better understanding of the training. The feature that data points were logged and how they could be retrieved, was not documented with a single word.

The information described above took the team two months to discover. A few more explanatory words might have reduced this effort to a fraction.

## 3.2 Server Training

After the first month of evaluating different reward function approaches without success I still had the notion, that it is mostly about the amount of iterations to train a working policy. “After all the agent is self-training in the simulator” was the approach. At this time I have heard that we could use a server from the GFR team at Oregon State University. So I spend two weeks getting the required gateway access and started the Zenith setup on the server. Finally, I started a training with 8000 iterations, which took four hours. Then I secure copied the resulting training folder to my computer for the graphical evaluation of the policy, which is not possible on a Linux Secure Shell. Then I saw the sobering result, that the sixfold number of iterations still did not produce an even remotely functioning policy.

The newly created folder for training contained pairs of stored critic and policy network parameters denoted with a number in the file name. The number indicated the iteration in which they were stored. On average there were a lot of stored files for iterations up to 30, sporadically for up to 100 iterations, and very seldom for up to 1000 iterations. They seemed to be stored arbitrarily. I understood the numbers by checking the storing mechanism in the source code: the network weights for both policy and critic were only stored, if the current iteration had a higher average return than all the iterations before. This comprehension was disillusioning since the highest number appearing there was a maximum of 200 for the training of 8000 iterations mentioned above. Meaning the highest achieved reward was received in iteration 200 and all the 7800 iterations afterward were wasted.

Of course, a policy that can not increase the reward remains bad. At this point, I started to print the average reward per episode into the terminal output lines too. This allowed me to check during the training how the reward was increasing, not just at the end in the tensorboard logfile diagrams. With this new indicator, I again experimented with different reward function concepts and hyperparameter combinations. But none of them could

get the reward even slightly increasing for more than 150 iterations of training. Additionally as explained in Chapter 2.2, the actual goal of reinforcement learning is to increase the reward with every iteration taken. In this Zenith implementation, however, on average, only every 25th iteration was slightly better and thus stored. The visualization during the evaluation was likewise bad, with an agent that could barely drive some timesteps before crashing into the outline of the track or getting stuck in driving circles.

## 3.3 Training on Straight Track

To furthermore exclude that the environment is too complex for the agent to be directly trained in, I simplified the task by creating a straight track segment and trained only on it. The process of creating the track segment manually is described in the following paragraphs.

The JavaScript Object Notation (JSON) track files are created at the Zenith setup when executing the run.sh bash file in the project folder. It calls a method of the TrackFactory that creates ten random tracks and slices them into segments. These segments are stored as multiple “pieces” in ten different JSON files. The structure of these files is shown in Code Block 3.1. The complete track is saved in the same file too, but only its cones, no position and yaw to start from. The “pieces” element contains a list of track segment dictionaries. This detail that pieces are stored within an extra list is important for the later implementation to load both, full tracks or segments for evaluation.

```JavaScript
{
    "fullTrack": {
        "outerCones": [...],
        "innerCones": [...]
    },
    "pieces": [{
        "outerCones": [..., [2, 1]],
        "innerCones": [..., [6, 1]],
        "startingPosition": [4, 2],
        "startingYaw": -1.57
        }]
}
```

Code Block 3.1 Track File Structure

The “outerCones” and “innerCones” consist of small sublists, one for every cone. Every sublist contains two numbers, that represent the axis position X and Y of a cone in the mapping coordinate system. The order of the cones in the lists is reversed. So the last sublist defines the cones at the start of the track. The starting position of the agent should be well inside of the track outline to avoid getting the agent into a position where it cannot even act but still gets the punishment for crashing. Figure 3.1 illustrates that the agent is positioned in a radius of three length units away from the starting point. The yaw value defines the angle of the radius and therefore of the car. The yaw is defined

counterclockwise starting from the right side of the agent, as can be seen in Figure 3.1. A default yaw of plus one Pi is added in the reset function, probably as a necessary correction term for the output of the track generation. This default yaw addition is marked with a red arrow in the figure. To start the car with a yaw in the direction of the track, half a Pi needs to be defined as the starting yaw in the track file in Code Block 3.1. This kind of setting makes the manual definition of start positions a bit more complex. The small red arrow within the four green points of the car is visualizing the amount of acceleration and yaw the agent is going to apply for the current time step. Figure 3.1 shows only a zoomed perspective of the straight track, which is actually 60 length units long.

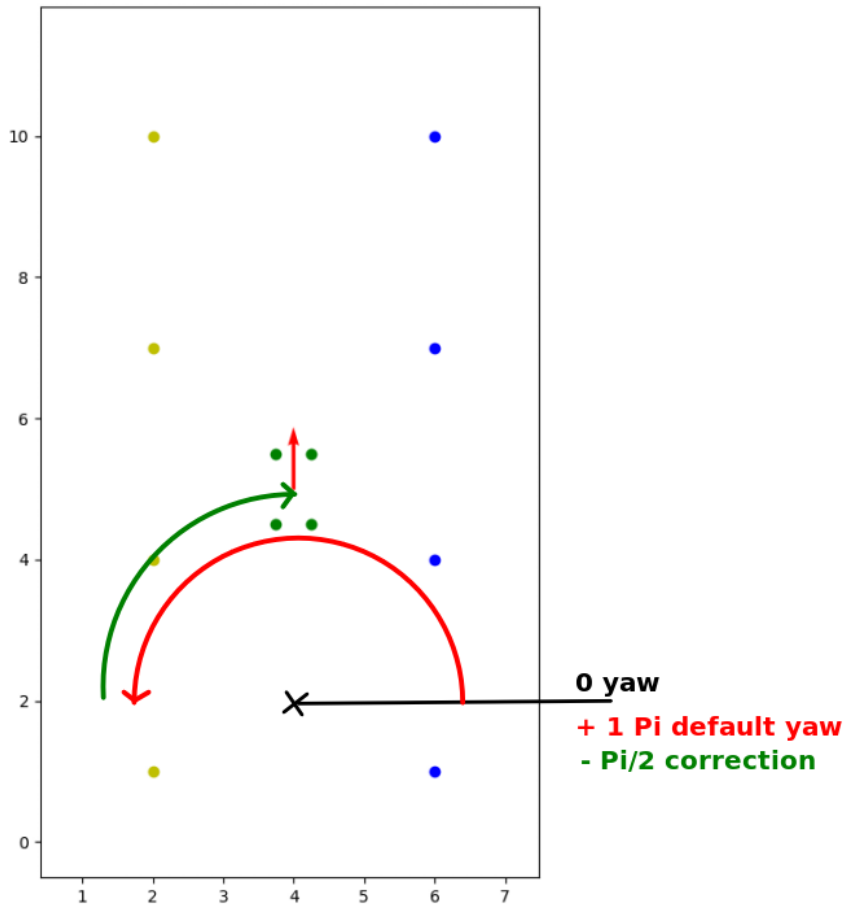


Figure 3.1 Starting Position Definition

To use only the straight track for training, it is defined in its own track file. This file is also saved in its own folder. Zenith only takes the folder path in which the track files are stored and makes random use of all tracks provided in the files of this folder.

With this preparation for a simplified environment done, I started again to search for a combination of a reward function and hyperparameter settings that would result in a constantly growing average cumulative reward over the iterations. After two weeks there were no significant improvements. Rewards would only grow sporadic and marginal up to 200 iterations. Once, in a not replicable training, the agent was able to reach nearly half of the straight track before crashing randomly into the outline.

At the point of drawing conclusions, I had the reward function modified as follows. To guide the agent towards driving just straight forward, it was punished for more distance to the middle of the track. The achieved reward was reduced linear towards zero if the agent is nearly at the track outline. This was easily implemented by checking if the X position of the car is near the value four as in Figure 3.1. Higher velocities were rewarded exponentially. Punishment was given for crashing into the outline or when the agent turned around and drove back toward the start. Thus basically it could only drive in a straight line towards the end of a track, which should be easy to learn if trained only on the same track. Still, the result was mostly a policy that would let the car circle around its starting position with a small radius and low velocity.

After many hours were spent experimenting only on the straight track without the slightest success, it dawned on me to think about the correctness of the implementation. A hint in the project documentation of Zenith led me to the PPO implementation of Eric Yu (Yu, PPO part 1). By comparison of the steps, it seemed the implementation of Yu was used as a foundation for Zenith, but with certain changes. The purpose or source of these changes was not documented.

To get an overview, I draw the diagram in Figure 3.2 that summarizes all PPO algorithm steps as they were implemented in Zenith. It gives a good insight into the present complexity and the resulting high probability of errors.

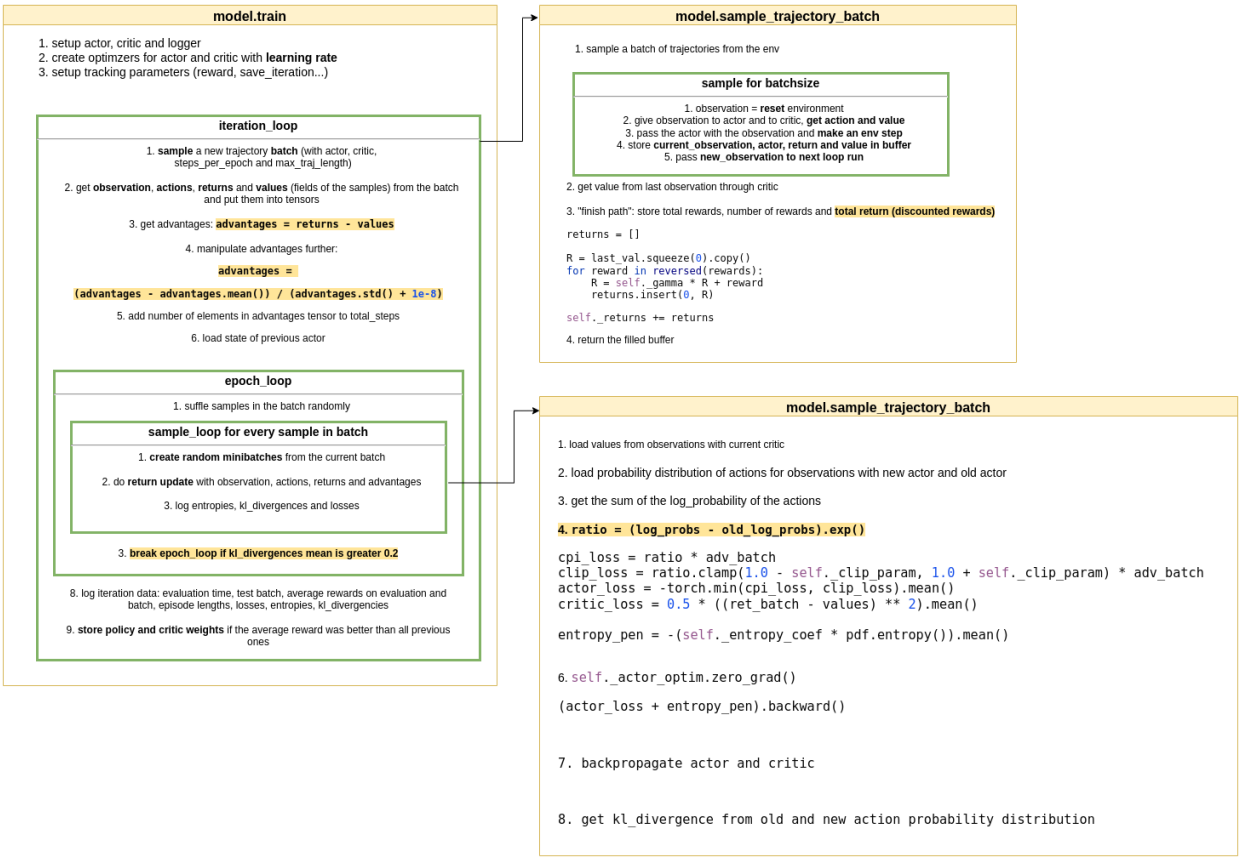


Figure 3.2 Previous Zenith PPO Implementation

## 3.4 Implementation of RLlib

At some point in searching for PPO implementation details and potential errors, an internet search engine gave me the proposal “rllib reinforcement learning framework” in the appearing dropdown menu. I read the line while pressing enter for my actual search topic. In a glimpse after reading the line, I realized that a framework for the algorithm implementation could solve the complexity and error issue observed in Chapter 3.3. Interestingly enough that we as a team have never read something about reinforcement

learning frameworks while searching over five months for different RL topics.

I used a week to check in an experimental implementation if Ray RLlib training is working on the straight track. The implementation was feasible with the number of available examples from different sources. Especially the incorporation of custom environments was fiddly. With the two functions for training and evaluation ready, I did a test training for 160 iterations on the straight track. The average cumulative reward was growing quickly per iteration. At 90 iterations the growth of the average reward stagnated. The visual evaluation confirmed the high reward values of the training by showing an agent that drove the car with a straight acceleration directly to the end of the track. The fact that I simply used only a simple quadratic reward for the velocity and small punishment for crashing made this result even more notable.

With this unexpected success, it was clear that the RLlib module is providing the new PPO algorithm implementation for Zenith. It is very likely to have a correct realization of PPO and it simplifies the architecture of Zenith greatly by reducing roughly 1500 lines of code to 170. To summarize, with this approach, more time can be spent optimizing the desired policy instead of troubleshooting the algorithm implementation.

Figure 3.3 illustrates how much of the algorithm complexity in Zenith is outsourced into the RLlib module. The upper diagram is representing the old implementation and the lower diagram is the new one. The box at the left of both is the run.py script of Zenith. The right box of both is the CarEnv class. The algorithm implementation is shown in detail in the centers. The environment is simplified in the diagrams to facilitate the comparison of the complexities. As can be seen, all the logic of the old implementation is now boxed in the PPO class of RLlib. To interface with this implementation, only three functions have to be accessed.

The next paragraphs describe the structure of Zenith as it currently stands. As in the previous version, the system is started by an execution of the run.py script. It parses the passed command line parameters and calls the train or eval function depending on the

user selection.

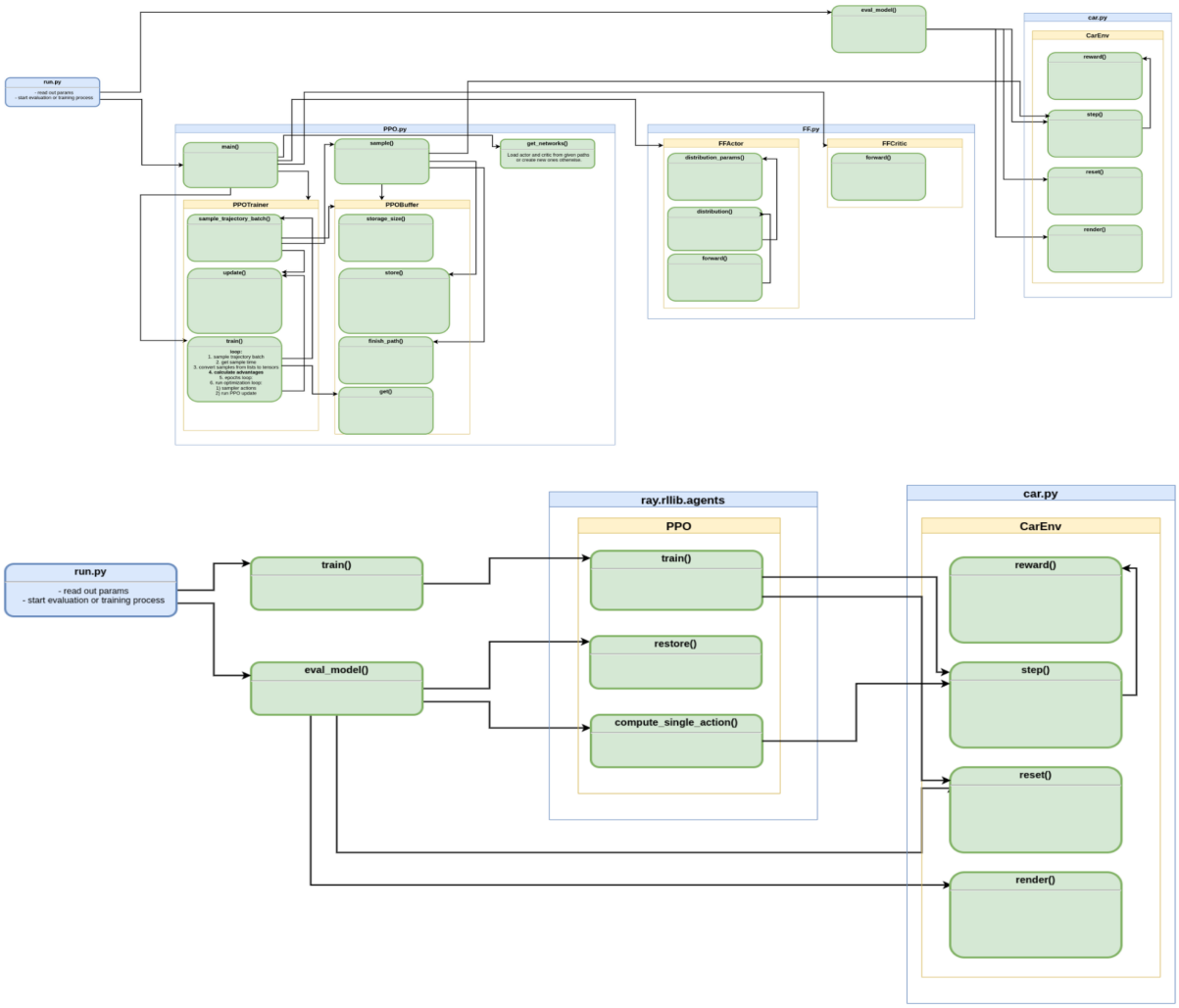


Figure 3.3 Simplified Zenith Architecture Comparison - old above, new below

How the functions of certain files make use of elements in other files is shown in Figure 3.4. External files are colored in yellow. The train and eval functions which are called in the run.py are stored in the train_model.py and the eval_model.py. The train function basically creates a PPOTrainer object from the RLlib module which is then provided with a registered environment and the training hyperparameters. Subsequently, the training loop is started which calls the train method of the trainer object and stores monitoring values like the average reward in a logger object. In order to use self-defined custom

environments in RLlib as in Zenith, it needs to be registered with a specific function. The process of finding the correct syntax for this registration was fiddly. As an example, the passed custom environment could not be written as CarEnv(args) but has to be imported as “env” to be accepted by the function. Hyperparameters and general settings are given to the trainer object in a default configuration dictionary, that is retrieved from the RLlib agent module and can be modified. This dictionary can also be printed into the console to see all available options.

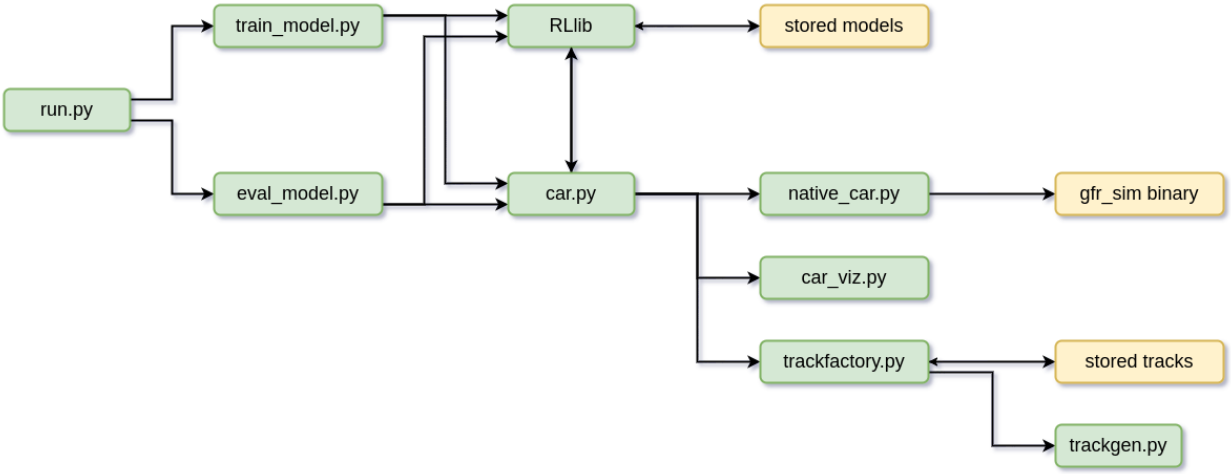


Figure 3.4 Current Zenith Architecture Overview

With the registered custom environment and the configuration dictionary in place, the PPOTrainer is created. If a file path to an already trained model is given, this model is loaded by the trainer. Otherwise, a new model is started. In the next step, the training loop starts. It runs for a specified number of iterations. In it is the train function of the PPOTrainer called, the model is saved in a checkpoint every five iterations and the results of the training are logged. The checkpoints are folders that contain files with information about the training itself and the current network weights. They are completely managed by the PPOTrainer. To continue the training of an already existing policy, only the checkpoint directory path needs to be referred to.

The environment CarEnv only required minimal changes in order to be interfaced by the

PPOTrainer. It is built on the gymnasium env class and already had gymnasium space attributes that define the action and observation formats. These formats form the specification for the input and output of the policy neural network. The observation space is a Numpy array with nine values. These are the two velocities v_x and v_y of the car, and seven distance rays to the outline of the track distributed over 180 degrees. The action space is a Numpy array too but with three values. It consists of the steering angle, acceleration, and speed limit parameters. The tracks and the C++ simulator are also prepared in the initialization of the CarEnv. The track segments are loaded from their JSON files in the TrackFactory class. The NativeCar class creates a C-compatible interface to the C++ binary gfr_sim.so with the CDLL module. This allows us to call the C++ functions as if they are Python functions.

Only two functions are used for further interaction between the PPOTrainer and the environment. One is to *reset* the environment, where the car is placed at the starting position of a random track segment that is contained in the track segment folder. Currently, the car is started always with velocity zero and an optimal yaw angle. An option to provide more randomness and thus create a broader training experience is to also define a random starting velocity, acceleration, and yaw within a certain range. The other function is provided with an action to execute a *step* in the environment. It returns the reward value for the step, the corresponding new observation, and a flag if the trajectory is ended or not back to the PPOTrainer object. If it was successful is signalized by the amount of reward. Both functions, step, and reset, use the NativeCar class as an interface to the actual C++ gfr_sim.so library.

The process is similar to evaluating a trained policy. At first, a CarEnv environment is registered to initialize a PPOTrainer as in the training process. Then the model is restored from a provided checkpoint directory. In the next steps, CarEnv is also registered as a gymnasium environment in order to create a CarEnv object. This stand-alone object is used to process the actions given by the policy in the PPOTrainer and provide it with an updated observation in order to load the next action. The process

of computing a single action to an observation without training on this, so to speak, just applying the policy, is called a rollout. In the real application, the policy is only used to perform rollouts. A rollout is basically performed by calling the compute_single_action method of the PPOTrainer with the loaded model. The second task of this environment is to render a track plot with the car, every time a new action has been processed. For the purpose of visualization exists an own class CarViz. It is used in CarEnv if a render flag at its initialization is set. In order to update the displayed plot, the render function of CarEnv is called.

This way the implementation of the evaluation is very similar to the one that is later used for the real-world application. A long series of rollouts is performed on the optimized policy. The difference is that the environment is not simulated by a CarEnv object, but provided by the car, which also carries the sensors and the computer where the policy network is run.

## 3.5 Hyperparameters

The training process for a policy in reinforcement learning algorithms is not only defined by the environment and the reward signal but further by hyperparameters for the training algorithm itself. Already mentioned are the clipping parameter and gamma. Further ones explicitly set are lambda, the training batch size, and the value function clip parameter.

The exact values for the values described are adapted from a report about experiments in the CarRacing-v0 environment from OpenAI (Giannoutsos et al. 30). Decisive for this were GIF files on the associated GitHub repository, which showed a well-driving agent. Though this is an environment with a different implementation, an experiment showed promising results. Therefore I used these values as a basis for all further tests.

The gamma value is for the weighing reduction of future rewards in the return calculation of the PPO algorithm, also called the discount factor. This is done because it gets more uncertain for certain situations the further they are in the future of a particular

step. The return is later used to estimate the advantage, which in turn is an important part of the surrogate loss function. Thus gamma defines the importance of future rewards in the decision process of the agent. The value is typically set between 0 and 1. Logically a value of 1 means that future rewards are equally regarded as instant rewards (Sutton and Barto 60). Gamma is set to 0.99 for further tests.

The clipping parameter is of central importance in PPO, since it defines how much of a maximal change of the policy network is allowed. Reducing the susceptibility for falling into local reward minimal and getting stuck there, this approach is the main innovation in PPO (PPO, Schulmann et al. 1). The clipping parameter is usually set to 0.2 which is also used for the tests.

A more sophisticated way to estimate the advantage is the General Advantage Estimation approach. This adds another layer of computation with a parameter lambda that controls the balance between bias and variance in the estimation of the advantage function. Ultimately lambda, like gamma, determines if the agent is focused on short or long-term rewards. It too is set to a value between 0 and 1. One is meaning that future rewards are considered the most, resulting in high bias and low variance (GAE, Schulman et al. 9). This parameter is set to 0.95 for the tests.

The number of collected samples that are used to update the policy is called the training batch size. The number of episodes defines how often a collected batch of samples is used for an update of the policy. A sample is a state(observation)-action pair. The choice of batch size is often a trade-off between the factors of generalization performance, convergence speed, and memory constraints. A larger collection of samples requires attention to have enough memory otherwise, the training process may become slow or even crash. Smaller batch sizes could lead to faster convergence of the training as it results in a higher update frequency. However, small batch sizes can result in a higher variance in the training, which makes it harder for the agent to learn the underlying patterns (Brownlee). Larger batch sizes on the other hand can lead to a

better generalization performance for the training of a policy since more samples are more representative of the environment. As an example, after training a policy in Zenith for 20 minutes, it takes an average of 130 steps to drive through a whole track segment from start to finish. On the other hand, is a full track consisting of around ten track segments, which results in 1300 steps for a full track run. To train on both elements, track segments, and full tracks, a training batch size of 5000 seems plausible.

The value function clip parameter is basically limiting the amount that the value function estimate can change over a single optimization step. The background is similar to the clipping parameter that is limiting the update of the policy network. Likewise, it prevents drastic large updates during the training, which could lead to instabilities. At some point during starting a training in Zenith one of the info messages from RLlib was to use 40.6 for the value function clipping parameter based on the average reward. I followed this instruction and due to the good results, I did not yet have the idea to try other values for this parameter.

All other training parameters configurable in RLlib are left at their default values. A task for the further development of this project would be to use the Ray Tune module to examine the best training parameters for Zenith.

## 3.6 Training on Full Tracks

With the experience of a policy that could perform quite well if trained only on one track segment, I wondered if this works too for training on complete tracks and not just segments. Zenith was at that time only designed for training and evaluation on segments. The JSON track files consist of “fullTrack” and “pieces” elements. Only the sub-elements of pieces, which are the actual track segments, are loaded by CarEnv objects when reset() is called. I figured that full tracks could be loaded with a little customization too. As the fullTrack and pieces elements both have the same data structure, left and right cone positions as well as a start position and start yaw. The episode length setting of RLlib is set to infinite, meaning that a trajectory run of the

agent during training is not aborted by the framework. This is an important detail, since the run on a full track requires a lot more steps than the training on small segments. On the other hand, using infinite episode length requires a reward function that encourages a quick ending of the trajectory run. Otherwise, the process would run infinitely, if the agent figures that waiting yields the highest rewards.

From the user side, only the specification for the track data path has changed. Instead of a single “--track-path” parameter for evaluation one has to provide a “--track-seg-path” or “--track-full-path” parameter with the directory path, in order to load the complete tracks or only the track segments. Full tracks and segments are currently not supposed to be used both at once for training or evaluation.

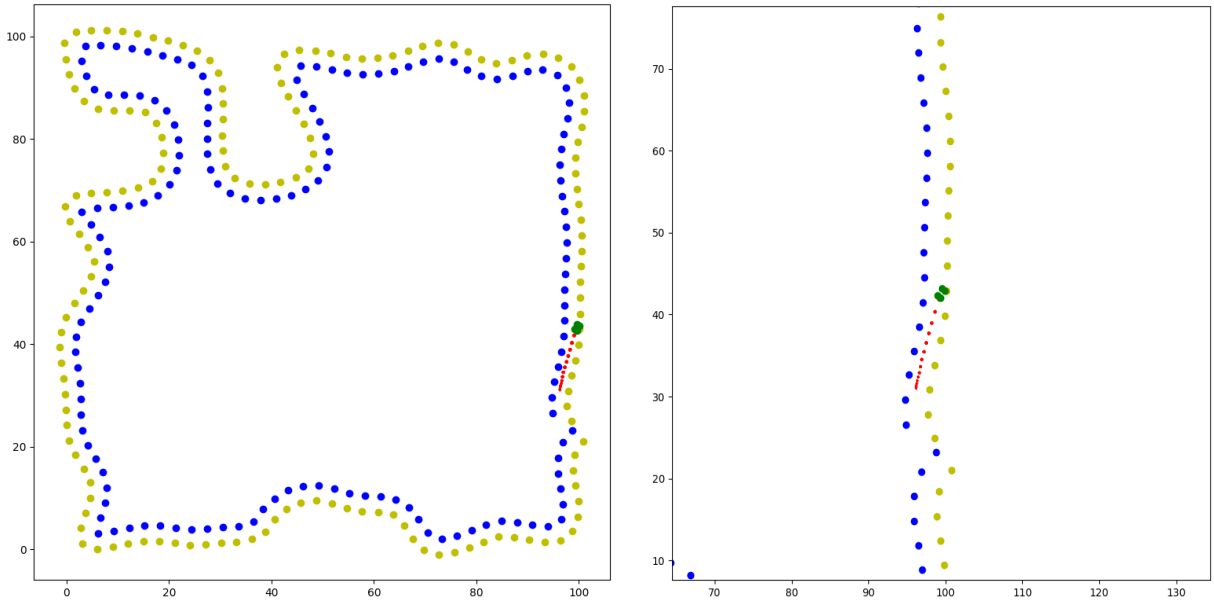


Figure 3.5 Full Track View and Full Track zoomed View

On testing the implementation the system worked as expected. But it turned out the starting position and yaw of the generated full tracks are not applicable. To avoid a bad starting position, the user of Zenith has to assure that a suitable starting position is set in the track data. This can be achieved by setting the Python input() function somewhere in the step method of the CarEnv class. That blocks the rendering of further

steps and the first position is displayed. Mostly the current full tracks have a clearly recognizable cone shift between end and start as can be observed in Figure 3.5 in both images. This can be used to determine a suitable starting point with the description of the starting point parameters in Chapter 3.3. The finish area is also recognized at the last four cones of full tracks (the last cones mean they are the first four in the JSON track file).

The first representation of a full track evaluation with a policy trained on segments looked like the left image in Figure 3.5. The overview is good for checking the track route but not to evaluate the trajectory, taken by the policy. I zoomed the plot of the track with every render in a fixed frame around the car. This way the plot window is following the car during the evaluation as can be seen in the right image of Figure 3.5.

With Zenith set up, I started a new training of 200 iterations and evaluated the policy. It was able to drive along a first straight and one curve but then crashed. A longer training did not change much about this result. Probably that is because the big finish reward is far from being reached and thus the agent falls into the behavior of crashing with a high velocity. The striking solution for this problem is to give rewards based on the track progress, not just on the velocity. As an indicator of the progress on the track, the index of the nearest cone in the cone list of one site could be used.

Another thing that became clear is that most generated full tracks have at least one curve that is too narrow or has an unrealistic placement of cones. A slightly critical example is the peaked curve in the middle of the left side of the full track shown in Figure 3.5 on the left. The quick solution for this problem is to check every generated track and sort out the unsuitable ones. The elaborate solution is to fix the track generator algorithm.

## 3.7 Implementation of Cone Input

During the presentation of previous results, my supervisor asked why the coordinates of the next ten cones were not used as input for the policy instead of the distance rays.

This approach would leverage the fact that the cameras mounted on the vehicle have a full view of the curve. The distance rays only allow a view into the curve, and depending on the rays' density, an estimation of how the visible part of the curve runs.

In order to provide the cones in a useful way, their coordinates have to be localized around the car. With the distance ray approach, the coordinates of the car and the cones are only relevant for the simulator. The policy only receives vehicle state data and the distance rays, which are independent of the position of the track and the car in the coordinate system. Localization means that the cones are transformed around the car, which is then at the center of the coordinate system. This way the view for the policy input is always the same, no matter where cones and the car are drawn in the coordinate system. A very advantageous aspect of this implementation is that the localized cone data is also output as localized from the optical vision system of the real vehicle (camera images with computer vision processing). This would eliminate some additional components used so far, such as the simultaneous localization and mapping algorithm and the solver to determine the driving lines.

From the Figures already shown in this document, I decided that the next ten cones on each side from the car onwards would be enough to estimate the ideal control parameters for the vehicle. Most curves are no longer than eight cones, and at high speeds on long straights, the distance over ten cones should also be enough to react to an incipient curve.

The cones are stored in two lists for the right and left sides of the track in every CarEnv object. They consist of sublists, that hold the x and y coordinates of a single cone. In the first step, these cone lists are reversed because they are generated by the track generator in such a way that the vehicle starts at the last cones in the lists. Then the absolute distance to all cones is calculated. From these new lists with the distances to all cones, the indices of the nearest cones left and right are selected. The indices of the nearest cones are also the starting point to select the next nine cones in the field of

view. These twenty cones are then localized around the car by an own function in the car.py file. The result can be compared to the original coordinates in the columns of Figure 3.6. The two plots at the bottom are the ten next cones, selected after the nearest cone. The plots above show respectively these ten next cones localized. The axis scaling of the plot windows distorts the track illustration partly noticeably. However, the two upper plots clearly show what is meant by localization. The two green dots symbolize the car. As can be seen, in the localized views, the car is at the coordinates origin and the field of view expands upwards.

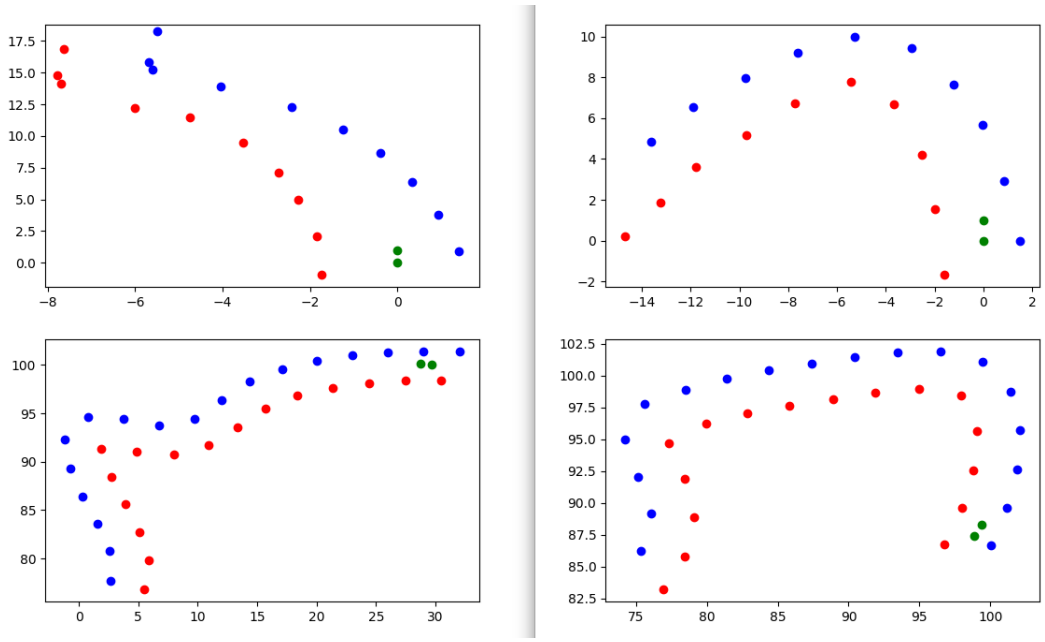

Figure 3.6 Localized Cone Coordinates Plot

The function used to plot the localized coordinates is kept in the car_viz.py file for debugging and validation purposes. A commented example call is also provided in the wrap_observation function of car.py, where the coordinates are actually localized. As learned from the fixed distance ray input, the validation of the correct processed input data is fundamental for all further steps. Training on incorrect observation data is pointless. To make sure that not only the localized coordinates of the first step are correct, I kept the visualization call in the wrap_observation function and set an

input(“Break!”) afterward. This way the evaluation is blocked on every step because the input() is waiting for an enter press on the keyboard. With the arbitrary blocker, I could check the cones' transformation for many steps on their correctness.

An important detail to keep the simulation realistic is the measurement error of the cones by the vision system, adding up on more distant cones. An example of the increase in perceptual error over distance is given in Figure 3.7. At 10 meters distance there is around 1 meter deviation from the real value. At 20 meters the deviation is already 5 meters deviation. The resulting deviation can be captured with a polynomial function of second grade. This is done with the curve_fit method of the Python scipy module. The fitting is done in an extra file cone_error_function_approx.py within the utils folder. The fitting function has two lists that hold the x and y-axis data from the diagram in Figure 3.7. The outputs of the function are the three parameter values for the second-grade polynomial that are printed into the console. It is assumed that the error function values do not have to be adopted very often. Thus these printed function parameters are copied manually into the car.py function get_coord_with_distance_error. This function takes the two coordinate values of one localized cone, calculates the absolute distance, applies the distance error according to the function, and recalculates the coordinates of the cone. So the function is called after the cones are localized and before they are passed as observation to the policy network.

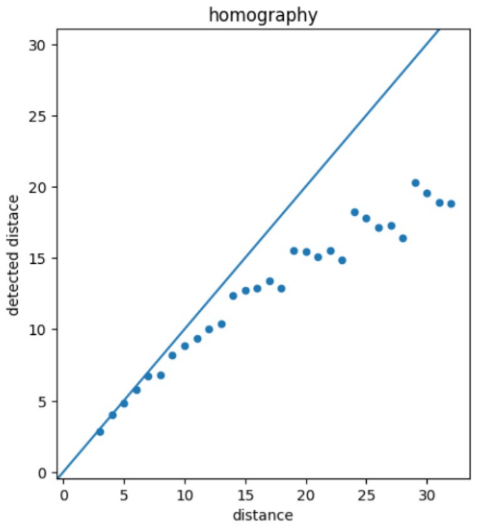


Figure 3.7 Cone Distance Detection compared to actual Distance

(Source: Depth Perception for Autonomous Racing, 24)

If viewed with attention, one can see in Figure 3.6 that the farthest away cones in the localized cone plots above have a significantly lower absolute distance to the car, than the same cones in the plots below. So the error function implementation is also validated visually.

# 4. Training based on Distance Rays

The experiments of this training section brought a lot of understanding of the exact operation and processes of Zenith. After the successful training of a policy on the straight track, I also evaluated the same policy on different curvy tracks without many results. Likewise, the training on individual curvy tracks did not yield any significant results. Although one observable characteristic of the accidents made sense with later obtained information. The agent was always able to learn the first straight part of track segments if they had any. But in the first turn, it was strangely always directly crashing. The idea was that it could not be that hard to drive curves if one is also able to learn to drive a straight line. Evaluating different reward function approaches to keep the agent in the middle of the track also had no effect on the direct crashing behavior. A short time after these events we had a code review with two team leads. The review brought a crucial implementation error in the environment to light.

At this time the policy observation consisted of the x and y velocity of the vehicle and seven distance rays to the track outline evenly distributed over 180 degrees. The concept was that these rays are always originating from the front of the car as shown in the top representation of Figure 4.2. Like the field of view that human car drivers have. In the observation preparation for the agent, the rays were calculated as expected over 180 degrees, only with the significant problem, that the car yaw was not used to rotate these ray angles. This caused the rays to solely represent a view towards the north, no matter in which direction the car was heading. Thereby in any situation, the agent was only looking upwards in the coordinate system, nevertheless, if he was in a curvy towards south or east. The problem is visualized in the bottom representation of Figure 4.2. That explains why the agent was perfectly good at learning to drive the north-oriented straight track but failed instantly in mastering any curve-like trajectory. The field of vision was prevented.

With the car yaw added to the calculation of the distance rays, the training quickly resulted in a policy that could drive along a curvy track. To verify the fixed ray calculation I printed the array of seven distance numbers into the terminal for every step taken and evaluated a policy that was well-trained on the curvy track. For most of the track, the car was driving a good trajectory. This implies that the most distant ray is somewhere in the middle of the field of view. The rays at 0 and 180 degrees are usually quite short since the track outline is directly right and left of the car. The printed distance ray arrays displayed exactly this pattern, which was sufficient proof.

When checking the velocity inputs of the policy I confused the x and y velocities that are handed over. I thought they are relative to the coordinate system. So to get the actual speed of the car, one had to calculate the absolute value of the two. As I learned too in the code review, the velocities are relative to the car. This means that velocity x is the forward velocity of the car and y is the sideward velocity if the car would drift for example. With this understanding, it really makes sense to pass both velocity values with a short description of the successors of the project.

With the distance rays in order, I started a new training on the slightly curvy track segment for 160 iterations, and behold, the policy worked fine. The trajectory taken is shown in Figure 4.1. There the agent already reached the finish area of the track. The red dots are plotted on every step of the agent. Thus their distance from each other indicates where the car was driving slowly and where it was faster. Curves were no longer an ominous obstacle. I evaluated the policy on other tracks too, and could definitely see that the policy also worked on tracks it was not trained on. Not to speak of good performance, but one could see how the agent was turning into the curves.

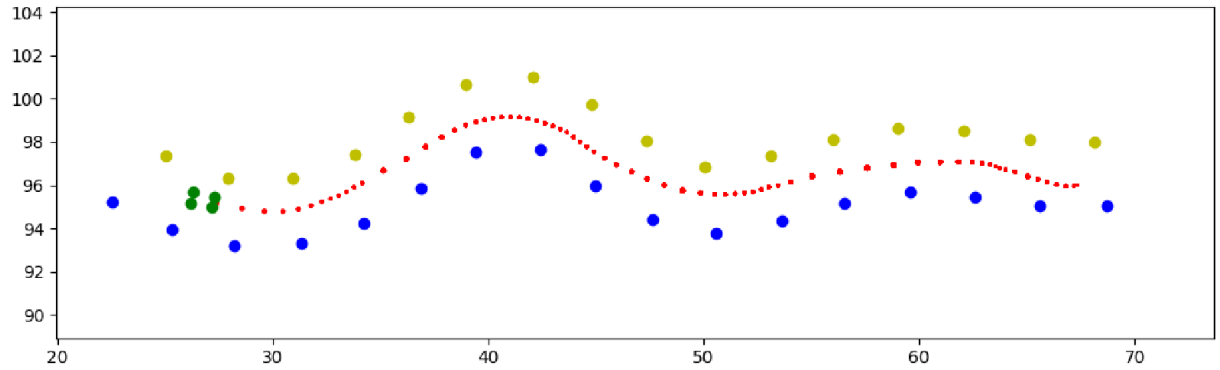


Figure 4.1 Evaluation of the first successful training on a curvy track

After this significant success, I decided to start training the policy on different track segments again to get a better performance in more diverse situations, not just on specific track segments. The first task for this kind of training was to create a vast selection of tracks that represent these diverse scenarios. The track generation makes use of a hull algorithm to create a specific number of random tracks. These tracks are sliced into segments that are used to train on. The starting position on these track segments is also determined by the track generation. Most of the produced segments are quite realistic. But some of them are not or the car is poorly placed outside of the trajectory. The unrealistic segments may be tough to learn and may even negatively influence the overall generalization of other scenarios. Furthermore, if the car is placed outside the track, it gets a punishment in the first step and could not even make one decision to learn from. This is certainly not beneficial for the overall training success too.

From this observation, it was clear that the track segments must be selected manually by hand. This is done by creating an empty JSON track file in its own folder. The file only contains one segment at a time. This new folder is then passed to Zenith for evaluation, where the specific track is used for visualization. The rendered visualization made it easy to rate the start position and the segment's realism. Fine segments are copied into another new track file that is used as a collection of valid track segments.

The folder with these selected track pieces is named “training_track_selection” and is also added to the GitHub repository of Zenith.

The next training run based on these selected tracks showed that a more sophisticated reward function is required to train on different track segments. For the single-track training still, only velocity reward and crash punishment worked fine. But on multiple tracks, this led to the behavior that the agent was driving some curves successfully. But in most situations, it just accelerated and crashed right into the outer curve boundary. So there was not enough incentive to try driving through the curves, which would lead to a much higher cumulative reward.

The agent being able to reach the track end sometimes, I figured that a big reward for finishing the track segment would be a sufficient incentive for avoiding the high-speed corner crashes. At this point, there was no implementation for detecting in the CarEnv class if the agent already reached the end of the segment. The finish detection is achieved by checking if the car is completely within the area of the last four cones. The information of the four cones that form the finish area is stored in the *state* object when the environment object is reset. From there it can be used in a new finished method, that works similarly to the done method. It makes use of the shapely function *contains* to check if all four corner points of the car are within the finish area. A boolean True is returned if this is the case. With this control option, the reward function is extended with a big bonus reward if the segment is finished. The current velocity is multiplied by 30 to incentivize a quick reach.

A second method to guide the agent more in the center of the train is achieved by rewarding greater distances to the track outline. The distance is determined by a likewise new method border_distance. It uses the shapely exterior.distance method of shapes to check the outline distance for every corner point of the car. The minimal distance is then returned. The value is depending on the track width and ranges between 0 and 1.6. In the reward function, the border distance is squared and then

used as a factor for the reward so far. The squaring has the advantageous effect to extra punishing small distances below one since an even lower factor is created by the squaring. Likewise, greater distances above one are extra rewarded with a steeper growing reward factor through the squaring.

These improvements lead to a policy that finished multiple different track segments after only 140 training iterations. But two new issues could be observed from the behavior of the new policy. The border distance is fine to guide the agent through the track but it becomes a hindrance when it is about driving the most ideal curved lines possible. Furthermore curves after a long straight part are not recognized, which is why the agent mostly crashed at these points. More distance rays as input for the policy and training without the border distance reward/punishment are taken as countermeasures.

The idea is that the ideal trajectory can be estimated significantly better with more information about the track in front. To supply more information, more distance rays are introduced that are also concentrated around the middle of the field of view. With the declining length of the rays and the arrangement around the car, this concentration of the rays is not obvious in the middle representation of Figure 4.2. But it definitely clarifies that the additional rays greatly improve the direction estimation compared to the representation with seven rays above.

The training with more rays reduced the situations where the policy would accidentally crash into the outer curve border. In some corners, however, the behavior remained the same despite the improved vision. I figured the agent probably estimated more rewards by crashing with a high velocity than by reaching the finish area. To circumvent this, I implemented a step counter variable as an additional CarEnv attribute. This variable is incremented with every step taken in the environment and reset to zero if the environment is reset. Instead of rewarding the agent for a higher velocity when reaching the finish area, it is now higher rewarded if fewer steps are required to reach the finish area. After some experimenting with the reward scaling (amount and how quickly the

reward grows with fewer taken steps) the results even showed approximating ideal trajectories in some rare scenarios, as can be seen in Figure 4.3.

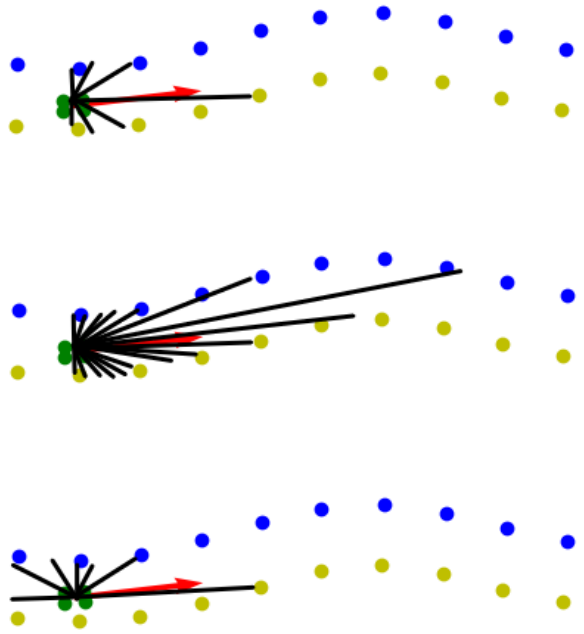

Figure 4.2 Distance Ray Visualization

Another change was to add some randomness to the starting values of acceleration, steering angle, and velocity of the car. This was done to train on more diverse scenarios with an unchanged amount of track segments. From the results, it is not clear if the added randomness improved the results.

But that was a very good example, on average the trajectories were rather mediocre and the agent still crashed frequently. Even on similar curves, it did not perform as well as in the one in Figure 4.3.

At this point, I adopted Zenith as described in Chapter 3.6 to train and evaluate on full tracks too. The full tracks are already created by the track generator function. They have the same data fields as the track segments. So from a data point, they are actually just longer track segments, but with a different label in the JSON track file. The training and evaluation was likewise not successful, but I got the idea to ultimately reward the

progress on the track, instead of just rewarding high vehicle velocities. This would end the balancing of the compromise between rewarding for high velocities and rewarding for the number of steps since the start. The first lead to high-speed curve crashes and the second could lead to a slow driving agent. Also if rewarded at the finish area for using fewer steps is not going to help if the track is long and the agent figures that a fast crash yields more reward. This made it clear that it was not the highest possible speeds that were to be rewarded, but the fast progress on the track. The implementation of progress detection is addressed in Chapter 5.

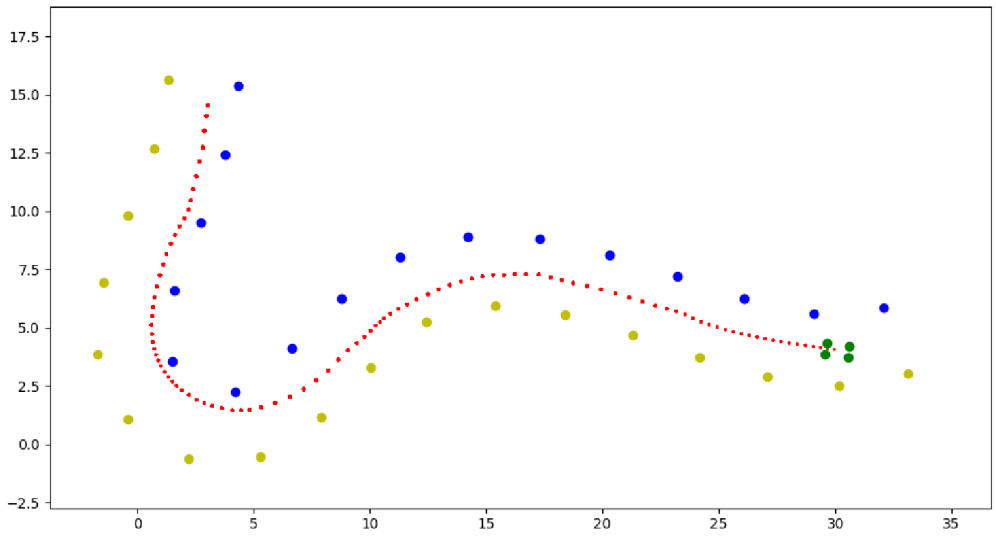


Figure 4.3 Approximating Ideal Trajectory

From the experiments described in this chapter it became clear that the distance ray observation is alright to get a basically driving agent, but it limits the generalization abilities of the policy towards high performance. The list of rays provides a good pointer in which direction to drive, but the information is insufficient to plan good trajectories through curves. The limitation of the rays can be imagined as driving as a human on a curvy pass on the side of a steep mountain. At the beginning of the curve, we may see to the center of the curve, but the mountain limits the view to the curve exit. The agent receives an even more imprecise view since the distance rays only provide a somewhat

discrete view of the observable curve entrance. This obstacle became very clear when the trained policy did not react to very steep curves in time and just crashed into them. The problem of the discrete view could be countered by introducing a lot more distance rays over the same “field of view”, but this would not solve the problem of insufficient trajectory planning ability.

# 5. Training based on Track Coordinates

The objective to plan better trajectories led to the alteration of policy inputs from distance rays to the cone positions directly. For that, necessary implementations like coordinate localization are described in Chapter 3.7. The vision system of the car detects the cones with a growing distance error, the farther they are away. This error is also incorporated into the cone manipulation for the agent observation. The cone manipulation is done individually for every observation of every step. The manipulation results can be seen in Figure 3.6 in the two upper plots.

After adopting the observation space definition in the CarEnv initialization, the training worked right away. Although the training results were a failure. The reward function for the distance ray training did not work fine for the new cone coordinate inputs. At this point velocity, crashes, and steps to the finish area were included. The agent drove very fast into the outer border of the first curves. That indicated too high velocity rewards. And indeed the results of the next training with lower velocity rewards brought a better cornering ability. By experimenting further it became obvious that the balancing between rewarding velocity to get the agent driving but not crashing, and rewarding of lower steps to the finish area but avoiding a very careful agent to reach the area is even harder with the new coordinate inputs.

To solve this conflict, which became also apparent during the training on full tracks as described in Chapter 3.6, a method was implemented to measure the actual track progress, instead of only rewarding velocity. This is done by using the orthogonal intersection between the car position and the track outline as visualized with the black lines in Figure 5.1. Once the intersection point is known, the distance along the track outline from the starting point can be determined. In the first attempt, this was implemented manually. But the results were incorrect. The final implementation with the constantly used shapely module simplified the code a lot and led to reliable results. The

average progress distance of both progresses from the left and right sides of the track is used, to avoid distorting progress indications when driving through curves.

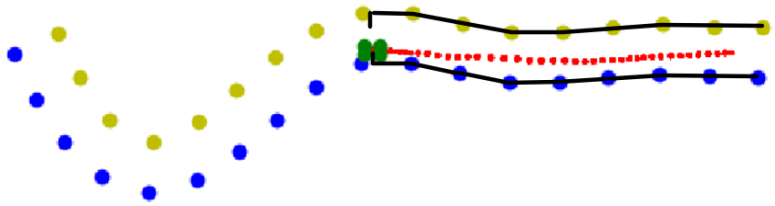

Figure 5.1 Car Progress Determination

The substitution of the velocity with rewarding progress made the training significantly more stable, but the results were still far from good. Although there could already be observed one advantage of the training with the cone coordinate input. Formerly not detected small steep curves were now clearly approached. Still, they were driven through only in rare cases, mostly it came in the curve to an accident. However, there was now clear evidence that the curves were recognized by the policy network.

By experimenting further manually with the reward function scaling, the results got a bit better. Small curves were handled well, but long straights and big curves were driven in strange trajectories if at all. At the straights the agent would slow down, only to accelerate again shortly afterward, like a human driver that is not able to process the situation. The fact that the agent could drive small curves quite fine, suggested that the agent had difficulties handling the cone distance error, which gets significant after 12 meters of distance. A quick training with the same halfway working reward function, each with and without the cone distance shifts did not show a noticeable difference.

The reward scaling could also be a cause. Formerly a lot of time was used to find a good combination of the reward components. To solve this tedious problem a new script is written. It starts many training sessions of hundred iterations with a random scaling of the reward function components.

The individual components are:

- Reward for *progress* on the track
- Rewared for *reaching* the *finish area*
- Punishment for *crashing*

Next to the short script, a small modification of Zenith was required to also pass the reward components per command line for the training. Since there is no exact understanding of the general reward scaling, a broad range of randomly inserted values for the components is given in the first run of the script. The exact ranges can be seen in the next code block.

```python
progress_reward = str(random.uniform(1.2, 2))

finish_reward = str(random.randint(50, 550))

crash_punishment = str(random.uniform(5, 20))
```

The training is executed and the random reward parameter values are written into a text file that is saved into the checkpoint folder. Finally, all the training sessions are visually evaluated, and if a fine working policy is found, the reward component values are retrieved from the text file. This script is run on a server for multiple hours or days. A hundred iterations are enough to see if a reward scaling configuration is worth using it further or not.

From Ray there is also a package called “Tune”, that automatically determines the best combination of a given hyperparameter range. It can quickly be implemented with RLlib environments. It even includes reward scaling as an evaluative parameter. But since it only offers the feature of tuning the general reward scaling, it is not suitable for to

determine a good combination of the individual reward components. Nevertheless, Tune would be very good for finding even better hyperparameters later, when a good reward function has already been found.

The first randomized reward parameter training was successful. On the sixteen resulting policies was one where the agent did not hit the wall directly with high accelerations, but followed large and small curves really well. With these working parameter values I started the script again arbitrarily for 24 iterations. This time I narrowed the range of the randomly initialized parameters down to the ones of the successful training as in the code block below.

```python
progress_reward = str(random.uniform(1.6, 1.8))
finish_reward = str(random.randint(60, 120))
crash_punishment = str(random.uniform(12, 20))
```

One of these trainings led to an again slightly better reward component combination of:

--prog-rew 1.76 --finish-rew 89 --crash-punish 17.48

With this syntax they are passed to Zenith through the command line. These reward component values led to a policy that drove both small and large curves quite well and also drove straight fairly safely. Not in quite good ideal lines and partly again with the slight braking, but relatively quickly and without crashing.

With this best (but not perfect) trained policy so far, I got the idea again to continue the training on the single full track that is stored in the training_track_selection folder, next to the many segments. This was the only complete track with somewhat realistic curves and no abrupt changes that I could find in fifteen generated tracks.

After some started trainings and active observation of the training monitoring parameters, I wondered why the reward was not growing further, when the policy showed definitely more potential in the evaluations. I realized that the ratio between the progress reward and the crash punishment is right to get the agent driving through the track, but far from good lap times. Then I noticed that the finish reward of 89 was large enough to incentivize higher track finishing on the short track segments but was a lot too small for large tracks with around 1000 steps to finish, instead of 120. As can be seen in Code Block 5.1 the finish reward is discounted by the number of steps times a factor of 0.02.

Python
```
def reward(self):
    car = self.state["car"]
    reward = 0

    progress = self.get_track_progress()
    progress_delta = progress - self._last_progress
     reward += self._rew["progress_rew"] * progress_delta - 0.3
    self._last_progress = progress

    if self.finished():
        print("finished track segment")
         return self._rew["finish_rew"] - 0.02 * self._steps_since_start
```

```
    if self.done():
        return -self._rew["crash_punish"]
    return reward
```

Code Block 5.1 Final Reward Function of Zenith

Formerly this factor was 0.1, which means that earlier finishing only reduces a punishment instead of increasing a reward. Not sure if this even made a difference, I lowered the factor to 0.02, which led indeed to a good result. The policy finishes the track in circa every fifth evaluation. It shows better performance on the track segments than before the training on the full track. Most curves are driven quite well as can be seen in the trajectory markings in Figure 5.2. Straights still cause an uncertain drive style but seldom lead to crashes. A learned characteristic that needs improvement is the strong short braking in certain situations. In the evaluation, this manifests itself as the red speed arrow disappearing briefly and then reappearing immediately. This could be achieved by lowering the low progress punishment in the reward function in Code Block 5.1 from value -0.3 lower to -0.4 or -0.5. A higher punishment for low progress could circumvent these breaks since they are more expensive from the reward perspective. These short breaks could also be interpreted as impulses to set out for a drift, which the agent does in some curves. But in most of the occurrences, it does not provide a benefit. Furthermore, Figure 5.2 shows that the agent learned to drive good cornering lines, but has not yet learned to deflect before the tight corners. The well-trained policies are committed to the Zenith GitHub repository by adding them to the *policies* folder.

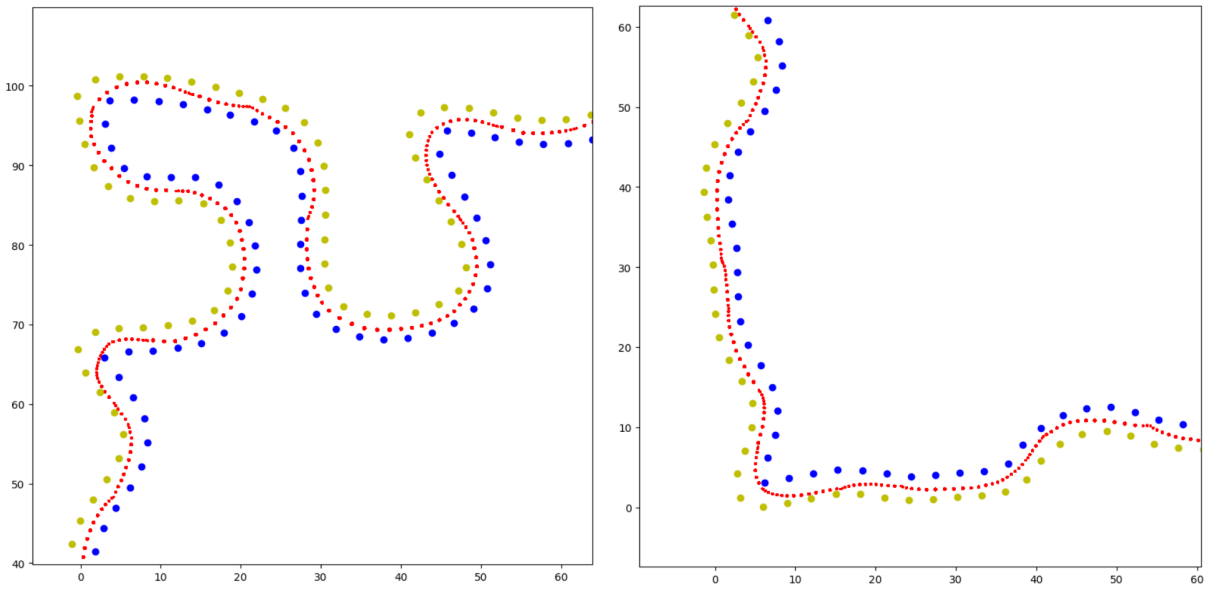


Figure 5.2 Trajectory of the best Policy Training

In total the policy is trained in 348 iterations. 151 iterations for the training started by the random reward scaling script and 197 iterations in the training on the full track. These two trainings took effectively under two hours. In addition to the effective 348 training iterations for the good policy, it should be remembered that the random reward component script also required 16 times 101 iterations for the first run and 24 times 151 for the subsequent second run. By roughly assuming realistic 20 minutes of processing time for 100 iterations, the two runs of the random reward scaling script took 17.5 hours.

What can be observed in the right image of Figure 5.2, is the really narrow generated track. By zooming it could be checked that the space between the left and right cones is actually three length units. The Formula Student rules demand a minimum track width of three meters for the track drive event (Formula Student Germany GmbH, 128). Formerly it was assumed that one length unit is corresponding to one meter. During the consultation with my supervisor, the question came up about how big the car is actually represented in the simulator. Currently, it is described as a rectangle of the size 0.5 by 1 length units. However, the actual car has a size of 1.35 by 2.86 meters. When I just input this size, the car was clearly too big for many track segments. This indicates that

one length unit does not represent one meter, which means that the size of the track and the car in the simulator should be carefully investigated in the future.

To see the result of a policy trained on multiple different full tracks, that are not too narrow, a new track generation is started with a TRACK_WIDTH value of four instead of three. This led to very angular, not usable results. Another indication is to understand and improve the track generator in the future.

Finally, to summarize this chapter, the input change from distance rays to the cone coordinates together with a new approach to find reward component scalings proved to be very successful. From now on, direct progress on the track is rewarded instead of the speed of the vehicle, which has led to a much better learning behavior of driving curves. It also turned out that the track and car size should be checked for further improvements. Additionally, the generation and checking of further full tracks are required. If these conditions are met one could start again to train a better policy. In case the now-found reward function does not work anymore due to the possibly changed car-to-track size ratio, the procedure with the random scaling script can be performed again. An important insight is the great role of reward scaling combined with using as few reward components as possible. I strongly recommend not expanding the basic form of the current reward function in any way. Instead, the scaling of the individual reward components should be further tuned. Every component added, for example, a penalty for braking, further complicates finding a good scaling of all components. From the experience of this chapter, the potential of better performance through finding a better reward scaling should not be aggravated by introducing more complexity to the reward components.

# 6. Conclusion and Outlook

To reach the goal of a well-driving policy, students created a simulator and a self-written PPO algorithm implementation in the last year. We took the project over with the goal to find a reward function that leads to a well-driving policy. During the first few weeks of working into the project, we realized that it was hardly documented. At first, we followed the notion that the training of a successful policy is mostly about the number of iterations we could train it. Thus we set the project Zenith up on a server and trained for many thousand iterations, only to realize that merely the first hundred iterations were actually useful. We assumed that our reward function was not formulated in a way, that would lead to a constant growth of the achieved reward during the training. After simplifying the environment to a single straight track and a reward function that only allowed driving forward, the agent would still drive circles around its starting position.

With this failure, I started to doubt the correctness of the PPO implementation. From this point, I continued to work on the project alone. While cross-checking the implementation with online sources (it was not documented) I learned about the module RLlib, a framework to simplify reinforcement learning systems. The first training on the straight training following the RLlib implementation was a success. With RLlib for the algorithm implementation and a corrected distance ray input, the policy was able to learn to drive along the track segments. Further training experiments showed a rather limited generalization behavior of the policy. Driving long curves was no problem, but steep small curves after long straights almost certainly lead to a crash.

At this point my supervisor had the idea to shift from the mere distance rays directly to the cone coordinates as input for the policy. Together with the usage of a progress indicator in the reward function and the approach to use a script that starts many trainings with different reward component scalings to find a good scaling combination, the training produced a well-driving policy. Still away from a really good performing

policy, this result proved the expected potential of the RL approach for the autonomous system and that it definitely makes sense to continue developing it.

It also turned out that the ratio between car size and track dimensions needs investigation. If the car is changed to its realistic size, it does not fit onto the tracks that seem to have the correct dimensions. The clarification of the size ratio may lead to the need to adjust the track generation algorithms as well. Once these issues are resolved, further fine-tuning of the reward function can proceed, and a performant competitive policy is very likely achievable.

Finally, this project phase of the “RL based planning/control” ends with a working policy, a simplified and evolved Zenith system, provided documentation, and suggestions on how to proceed further.

# 7. Bibliography

"AlphaGo." *DeepMind*, https://www.deepmind.com/research/highlighted-research/alphago. Accessed 17 April 2023.

Brownlee, Jason. "How to Control the Stability of Training Neural Networks With the Batch Size." *Machine Learning Mastery*, 21 January 2019, https://machinelearningmastery.com/how-to-control-the-speed-and-stability-of-training-neural-networks-with-gradient-descent-batch-size/. Accessed 8 May 2023.

Formula Student Germany GmbH. "Formula Student Rules 2023." *formulastudent*, 2023, https://www.formulastudent.de/fileadmin/user_upload/all/2023/rules/FS-Rules_2023_v1.0.pdf. Accessed 12 January 2023.

Frost, Jim. "Mean Squared Error (MSE)." *Statistics by Jim*, https://statisticsbyjim.com/regression/mean-squared-error-mse/. Accessed 23 April 2023.

Giannoutsos, Andreas, et al. *A deep insight in Q-Learning and Policy Gradient Algorithms while playing CarRacing-v0*. NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS, 2021. *GitHub*, https://github.com/spyros-briakos/Car-Racing-v0-GymAI/blob/main/Report.pdf. Accessed 6 5 2023.

Jagtap, Rohan. “Understanding the Markov Decision Process (MDP).” *builtin*, 21 November 2022, https://builtin.com/machine-learning/markov-decision-process. Accessed 15 April 2023.

“RLlib.” *Ray.io*, https://www.ray.io/rllib. Accessed 24 April 2023.

“RLlib Environments.” *Ray documentation*, https://docs.ray.io/en/latest/rllib/rllib-env.html. Accessed 24 April 2023.

“RLlib Key Concepts.” *Ray documentation*, https://docs.ray.io/en/latest/rllib/core-concepts.html. Accessed 24 April 2023.

Schulman, John, et al. *HIGH -DIMENSIONAL CONTINUOUS CONTROL USING GENERALIZED ADVANTAGE ESTIMATION*. University of California, Berkeley, 2016.

Schulmann, John, et al. *Proximal Policy Optimization Algorithms*. OpenAI, 2017. *arxiv*, https://arxiv.org/pdf/1707.06347. Accessed 17 April 2023.

Sutton, R., and A. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 2015.

Van Heeswijk, W. “Proximal Policy Optimization (PPO) Explained.” *towardsdatascience*, 29 November 2022, https://towardsdatascience.com/proximal-policy-optimization-ppo-explained-abed1952457b. Accessed 10 January 2023.

Verma, P., and S. Diamantidis. “What is Reinforcement Learning?” *synopsis*, 27 April 2021, https://www.synopsys.com/ai/what-is-reinforcement-learning.htm.

Accessed 10 January 2023.

Warila, John. “Reinforcement Learning Planning/Control.” *global-formula-racing*, 2022, https://wiki.global-formula-racing.com/doc/intro-john-warila-UrrYT7sipe. Accessed 10 January 2023.

Yu, Eric Yang. “Coding PPO from Scratch with PyTorch (Part 1/4).” *Medium*, 17 September 2020, https://medium.com/analytics-vidhya/coding-ppo-from-scratch-with-pytorch-part-1-4-613dfc1b14c8. Accessed 18 April 2023.

Yu, Eric Yang. “Coding PPO From Scratch With PyTorch (Part 2/4).” *Medium*, 17 September 2020, https://medium.com/@eyyu/coding-ppo-from-scratch-with-pytorch-part-2-4-f9d8b8aa938a. Accessed 20 April 2023.

Yu, Eric Yang. “Coding PPO from Scratch with PyTorch (Part 3/4).” *Medium*, 17 September 2020, https://medium.com/analytics-vidhya/coding-ppo-from-scratch-with-pytorch-part-3-4-82081ea58146. Accessed 21 April 2023.