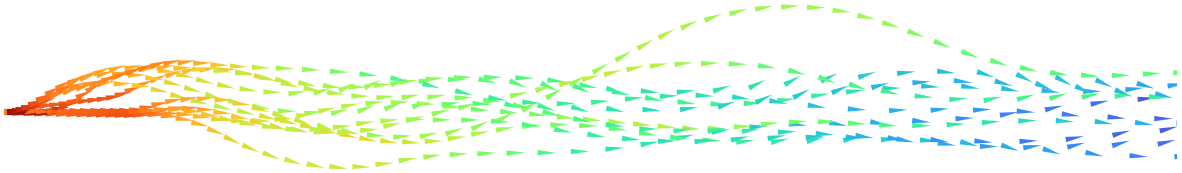
HOCHSCHULE
FÜR ANGEWANDTE
WISSENSCHAFTEN
MÜNCHEN

IPG
AUTOMOTIVE

Bachelor's Thesis

Soft Actor-Critic as Race Driver in CarMaker™



Author: Andreas Häge

Supervisors: Prof. Dr. Christian Möller
Dr. Stefan Engels

Start: 01.10.2020

Finish: 07.04.2021

Abstract

This bachelor's thesis shows that using Deep Reinforcement Learning, a virtual race driver can be trained within the virtual vehicle environment of CarMaker. The Soft Actor-Critic algorithm is used for this purpose. With this algorithm, unknown situations are preferably searched for and chosen by maximizing entropy. It uses TensorFlow's open-source implementation and API in Python. The CarMaker simulation environment is extended by an online communication interface and the simulation sequence is modified for faster training. For the respective training, the action space and state space are varied. This influences properties like the driver's learning speed and behavior. Also, experiments with the impact of the reward function are performed. The characteristics of the driver are also evaluated on unknown routes. The evaluation of the trained driver shows that the driving skills are comparable with the IPG RaceDriver, especially in corners. Problems arise on fast sections of the track, which are driven too slowly and using unrealistically varying control inputs. It is also possible for the driver to successfully master unknown circuits if they bear a certain resemblance.

Kurzfassung

Anhand dieser Bachelorarbeit wird gezeigt, dass unter Anwendung von Deep Reinforcement Learning ein Rennfahrer innerhalb der virtuellen Fahrzeugumgebung von CarMaker trainiert werden kann. Dazu wird der Soft Actor-Critic Algorithmus verwendet. Bei diesem Algorithmus werden dabei unbekannte Situationen mittels Maximierung der Entropie bevorzugt gesucht und gewählt. Es wird dabei die Open-Source Implementation und API von TensorFlow in Python genutzt. Die CarMaker Simulationsumgebung wird hierfür um eine online Kommunikationsschnittstelle erweitert und der Simulationsablauf für ein schnelleres Training modifiziert. Für das jeweilige Training wird dabei mit variierendem Beobachtungs- und Aktionsräumen für den Lerner gearbeitet. Somit werden Anpassungen hinsichtlich der Lerngeschwindigkeit und das Verhalten des Fahrers beeinflusst. Zudem wird mit dem Einfluss der Funktion für die Belohnung experimentiert. Die Eigenschaften des Fahrers werden dabei auch auf unbekannten Strecken evaluiert. Die Evaluation des final trainierten Fahrers zeigt, dass insbesondere in Kurven die fahrerischen Fähigkeiten mit dem IPG RaceDriver vergleichbar sind. Probleme ergeben sich auf schnellen Streckenabschnitten, welche zu langsam gefahren werden und hinsichtlich der größtenteils unrealistisch variierenden Steuerungsinputs. Dem Fahrer ist es auch möglich, unbekannte Rundkurse erfolgreich zu meistern, solange diese eine gewisse Ähnlichkeit aufweisen.

Content Overview

Figure Overview

Table Overview

Symbol Overview

Symbol – Physical Unit – Description incl. indices and abbreviations

Latin Symbols

s_t	-	State vector of the environment at timestep t
a_t	-	Scalar or vector of the actions in timestep t
r_t	-	Reward at timestep t
$\mathcal{A}$	-	Set of actions a_t
$\mathcal{S}$	-	Set of states s_t
p	-	Probability distribution for each action a_t at state s_t to transition to state s_{t+1}
G_t	-	Discounted cumulative reward or return
v_π	-	State-value for policy π
v_*	-	Highest possible state-value for the optimal policy
q_π	-	Action-value for policy π
q_*	-	Highest possible action-value for the optimal policy
v_k	-	State-value for state with index k
q_k	-	Action-value for action with index k
w_j	-	Input weight for a unit in a neural network with index j
I	-	Information content
H	-	Entropy
s	m	Longitudinal route coordinate
t	m	Lateral route coordinate
v	$\frac{m}{s}$	Velocity of the vehicle
A_G	-	Bounded set of actions for gas and brake control

A_S	-	Bounded set of actions for steering controls
$a_{S,t}$	-	Action at timestep t for steering controls
$a_{G,t}$	-	Action at timestep t for gas and brake controls
BPP	-	Brake pedal position
APP	-	Accelerator pedal position
c_x	-	Constant with index x

Greek Symbols

γ	-	Discount factor
π	-	Policy
α	-	Temperature or weight for the entropy bonus of the reward

Index of Abbreviations

AI ..... *Artificial Intelligence*

ANN ..... *Artificial Neural Network*

API ..... *Application Programming Interface*

DDPG ..... *Deep Deterministic Policy Gradient*

DP ..... *Dynamic Programming*

FIFO ..... *First-In First-Out*

IPC ..... I*nter-Process-Communication Socket*

MC ..... *Monte Carlo*

MDP ..... Markov Decision Process

RL ..... *Reinforcement Learning*

SAC ..... *Soft Actor-Critic*

SARSA ..... *State-Action-Reward-State-Action*

SWA ..... *Steering Wheel Angle*

TD ..... *Temporal Difference*

1 Introduction

Reports, that artificial intelligence (AI) solving games or other demanding tasks is becoming more and more superior compared to humans, have repeatedly reached the press in recent years [1–3]. Well-known examples of this are Google's AlphaGo, where the AI is at the top of the global ranking in the game Go with over 10^{171} possibilities [4], or OpenAI's AI, which beat the leading team in the competitive video game Dota 2 by training over accumulated 10,000 years of playing time against itself [5]. Even in more realistic racing simulation games like Gran Turismo Sport, Deep Reinforcement Learning Algorithms already achieve superhuman performance based only on nearly one million kilometers driven for training [6].

In view of the paper by Fuchs et al. [6], the question arises whether AI can also achieve very good performance in simulation applications that are specialized in vehicle dynamics like CarMaker. This could bring the realization of a real-life application of AI on real vehicles on racetracks one step closer and could lead the further possibilities to tackle other related optimization problems.

In contrast to the IPG RaceDriver which searches for best lap times by adjusting the vehicle velocity to its wheel slip and manually selected exponents, the AI learns the optimal vehicle handling by observing the vehicles behavior and environment.

To achieve this goal, a promising Reinforcement Learning (RL) algorithm is used, as done by Fuchs et al. [6]. This algorithm learns the control commands by observing user defined environmental values of the track and the car. By means of an adjustable reward, different goals can be defined, such as achieving best lap times or minimizing energy consumption on a route that must be covered within a certain time. The latter scenario is of interest to the author, since own experience in efficiency competitions has shown that the optimization of such a driving strategy can become highly complex.

With the help of the conducted experiments the expenditure and plausibility of the above-described problem definition is examined.

The methodology of this approach is explained in the following section of this chapter.

1.1 Research Design

An experimental research design is chosen to examine this type of problem. The individual steps are illustrated in Figure 1.1.

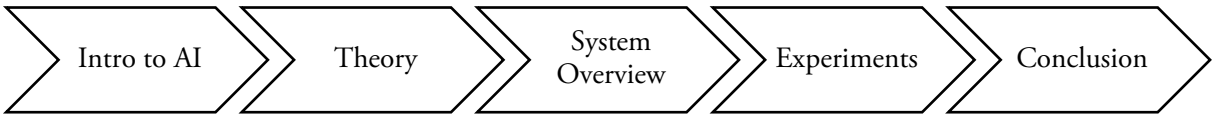


Figure 1.1: Research design steps

After the general introduction about AI, a knowledge base is built. The base is needed for the formulation of the problem at the program level to merge the components. Furthermore, the knowledge acquired in the theory is incorporated into the experimental design. The design of the experiments is then further iterated by means of the results until knowledge about the possible performance of the algorithm is available.

2 Preface to Machine Learning and Reinforcement Learning

In this chapter, terms like AI and its subsets Machine Learning, Deep Learning and Reinforcement Learning are introduced.

2.1 Brief Introduction to Machine Learning and Deep Learning

The concept of artificial intelligence can be tracked back over a hundred years before the first computer was built [7]. Since then, AI has become a broad field with many applications and ongoing research topics. Today, the learning approach of AI can be described formally as gathering knowledge from experience and using this knowledge to define simple concepts. Using this method, a human is not necessary to specify the exact knowledge needed [8]. A simple concept can be classification algorithms like logistic regression, linear regression, or any function, that generates a modification of the input. Combining multiple simple concepts to a more complex concept can be called AI deep learning or more with focus on the architecture "deep neural networks" [8]. The term "deep" refers herby to the depth or more specifically the number of stacked concepts or neurons connected in a row. Each row also consists of multiple neurons stacked together as illustrated in Figure 2.1. Using such a deep neural network, AI systems can solve problems that can be formally hard to describe for humans. Many well-known examples can be found in the fields of computer vision or speech recognition.

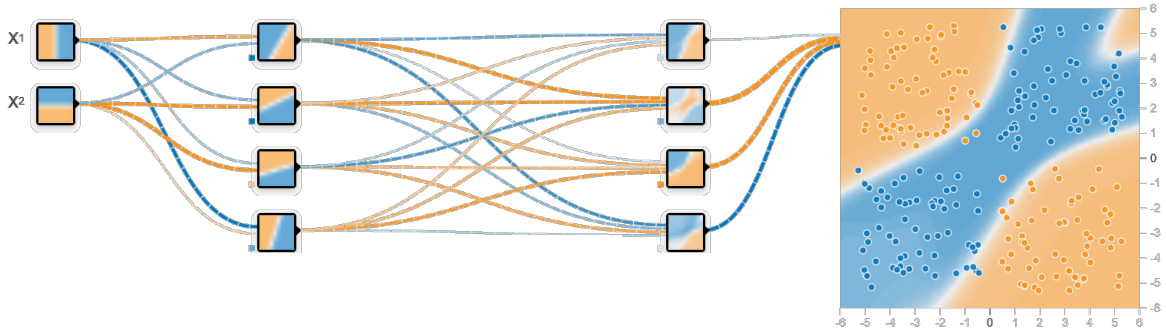


Figure 2.1: Screenshot of TensorFlow's playground [9].

These networks can be trained in many ways. Widely spread is the concept of supervised learning, where the networks parameters are trained to a set of labeled data with to goal to label new unseen data by itself. The capability to acquire self-knowledge from raw data is known as machine learning [8]. Further information about machine learning using AI deep learning algorithms can be found in Goodfellow et al. [8]. General information about neural networks is shown in chapter 3.3.

2.2 Brief Introduction to Reinforcement Learning

Reinforcement Learning belongs to the field of machine learning. The aim of RL is to train an algorithm for a specific environment to make a sequence of decisions until a terminal state is reached. The decisions or formally called actions are selected in a way to get the best possible solution. For this purpose, the programmer defines rewards and penalties. These are given for every decision is made by the algorithm. Thereby, the rewards and penalties can be modified to get different solutions. The learning algorithm, later called agent, usually seeks the highest possible sum of rewards, while considering finding new solutions by exploring stochastically other decisions.

The model of the environment has not to be known to the agent for some RL algorithms. Therefore, unlike supervised deep learning algorithms, the RL agent can be trained without knowing its environment. In other words, RL is learning by interacting with an unknown environment with the goal to maximize the reward.

Combining both methods is called Deep Reinforcement Learning, as can be seen in Figure 2.2. The illustration shows, that deep RL is just a combination of deep neural networks and classic RL. Figure 2.2 also clearly shows that with a more complex problem more elaborate algorithms are involved to solve the problem.

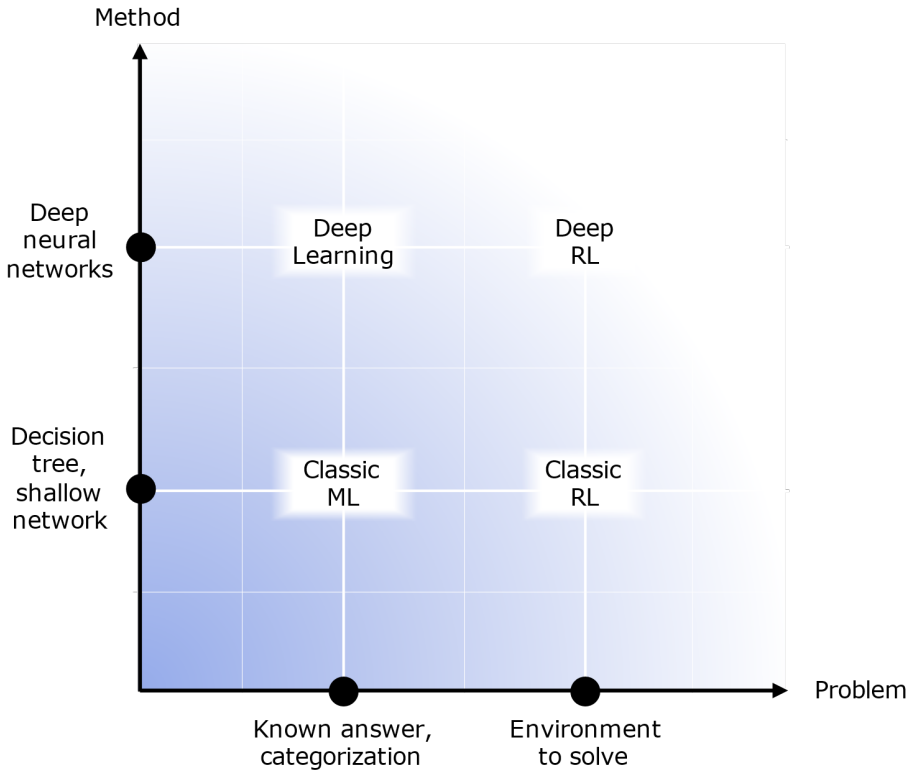


Figure 2.2: Complexity of problem and method based on an illustration of deepsense.ai [10]

3 Theory

As a fundamental basis for Reinforcement Learning, the *Markov Decision Process (MDP)* is explained first. Then common RL methods are presented. These will be used to solve the MDP. After the basics, the Soft Actor-Critic (SAC) algorithm is introduced.

3.1 Markov Decision Process

Commonly, the problem which is tasked to RL algorithm can be described as an MDP. Therefore, the approached problem of this thesis will be also described as MDP later.

A MDP is a mathematically idealized form of a reinforcement learning problem [11]. Key elements like returns, value functions and Bellman equations will be described here. These elements are necessary to understand the RL problem and the later used SAC.

The MDP can be divided into a learner, called the agent, and the environment, illustrated in Figure 3.1. For every discrete time step t the agent and the environment interact with each other.

The MDP model consists of following sets:

- states of the environment $s_t \in \mathcal{S}$
- for each state, a finite set of actions $\mathcal{A}(S_t)$ with action a_t
- rewards for every possible state transition $r_t \in \mathcal{R}(s_t, a_t)$
- a probability distribution p for each action a_t at state s_t to transition to state s_{t+1}

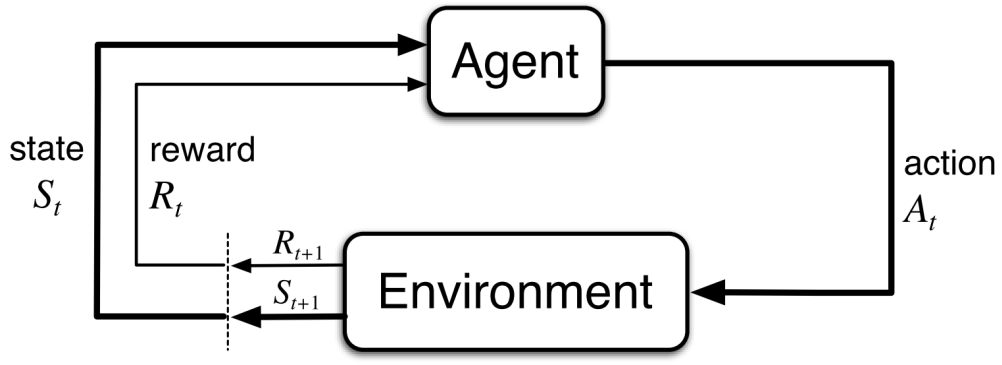


Figure 3.1: Interaction of agent and environment in a MDP [11]

The agent uses a learned policy to navigate through the environment. The policy π contains the probability distribution p to choose an action a_t for a particular state s_t.

The next state depends only on the previous state and action if the system satisfies the Markov property. This is the case, if the current state includes all information about all aspects of the past agent-environment interactions that have an impact on the future state. The Markov Property is important because the next action is to be assumed a function only of the current state. The later used algorithm also uses only the current step to estimate the best action.

The goal of the agent is to maximize the reward over time. The discounted cumulative reward received over time t is defined as return

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{3.1}$$

where $\gamma \in [0,1]$ is the discount factor. Depending on γ the agent is for small values near-sighted and for a discount factor approaching 1 far-sighted, which means future rewards are taken more strongly into account. With the discount factor smaller than 1 a convergence for an infinite horizon k against a finite number is ensured. Using a finite return, it is possible to compare multiple policies. A finite return is also important for a continuous environment with a possible infinite time horizon.

The probability of an action in a state being selected depends on an initially defined policy, which can be deterministic or stochastic. The initialization of the values can be chosen randomly. The policy can be depicted as the agent's brain and generally tries to maximize the return. To predict the expected return, if policy π is followed from each state in $\mathcal{S}$, following value function is defined:

$$v_\pi := \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{3.2}$$

The state-value $v_\pi(s)$ is the expected reward. If the policy π is better or equal to every other policy, the optimal policy is found, which generates the highest possible return. Regarding all states it is defined as

$$v_*(s) := \max_\pi v_\pi(s) \tag{3.3}$$

Like the value functions, an equation for every state-action pair can be defined, which is called the *action-value function* for policy π:

$$q_\pi(s,a) \coloneqq \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | S_t = s, A_t = a\right] \quad 3.4$$

An optimal action-value function will be defined like the value function:

$$q_*(s,a) \coloneqq \max_\pi q_\pi(s,a) \quad 3.5$$

Action-values represent the possible reward if action a is selected.

Using the equations of this chapter, the optimal policy which has the highest return, can be found. The application of these Bellman equations is explained in the next chapter.

3.2 Solution Methods for MDPs

To solve finite MDPs approaches like Dynamic Programming, Monte Carlo Methods, Temporal Difference Learning, and n-step Bootstrapping are common.

These methods are all similar in some ways. A basic understanding of these rather simple algorithms should lead to a more comprehensible understanding of RL and the later used state-of-the-art algorithm.

As a simplification it is assumed that all state values, action values and rewards are stored in a table. Using approximation methods, it is then shown how the calculation can be made more efficient for large state spaces without saving each value of each state. The shown algorithms are then compared.

3.2.1 Dynamic Programming

Dynamic Programming (DP) is a mathematical method to solve optimization problems. To get the optimal policy π_* of an MDP, the values v_{k+1} are evaluated iteratively using Eq. 3.2 until a defined residual is reached. The policy is then improved to be greedy to maximize the return. This process is then repeated until π_* is found. Maximizing the return means that the highest state-action pair according to Eq. 3.4 is chosen as the new policy $\pi'(s)$. π' is the optimal policy π_* if for every state $s \in \mathcal{S}$

$$v_{\pi'}(s) \geq v_{\pi}(s). \tag{3.6}$$

This procedure is called Policy Iteration. This method only works if the environments dynamics are completely known and the MDP is finite.

If only one step is used to compute the values v_{k+1} before doing Policy Improvement, the policy can be updated more often. Policy updates every step can have multiple advantages like a faster convergence, especially where steps are limited, resulting in less computational time.

A well-known example for solving a MDP with DP where the environment is a small grid world is shown in Figure 2. The initially unknown state-values are set to zero. The agent reaches a terminal state once a grey box is hit. The policy is initialized as a random policy where every action to move in a direction have equal probabilities. Every box, that is a nonterminal state, has a reward of -1. This encourages the agent to reach the target with as few steps as possible. For each step k one step of Policy Iteration is done. Since the environment is small, rewards are not discounted, meaning γ equals 1. The state-values are calculated using Eq. 3.2.

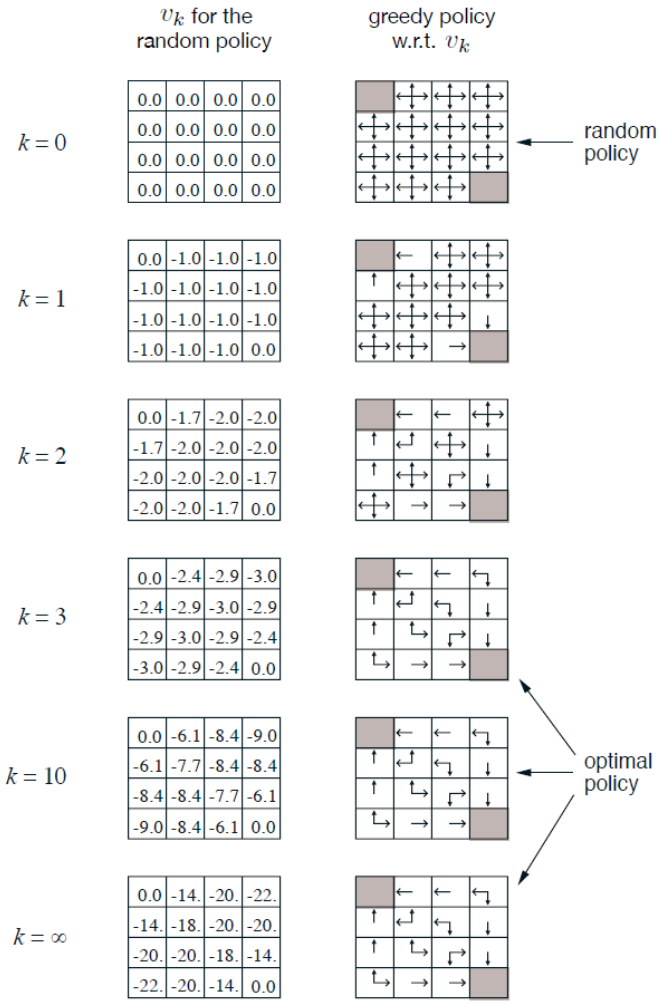


Figure 2.3.2: Grid world example showing the convergence of the state-values on the left and the policy on the right. [11]

3.2.2 Monte Carlo Methods

Put in a very simplistic way, Monte Carlo (MC) methods solve the MDP by trial and error. The actions are thus chosen stochastically. This is in contradiction to DP because it is a deterministic method.

To find the optimum, the First-visit Monte Carlo method randomly visits different states. During an episode[1], the initially unknown values $v_k(s)$ of each state approach the true expected value $v_*(s)$. This is done by averaging over all rewards each time the state is visited. This method can also be applied if the environment has a model which is unknown. In such a case, it is appropriate to search the optimal state-action pair $q_*(s, a)$ for each state.

The policy $\pi(s)$ is adjusted after each episode to be greedy. This can only be done, if all states were visited, otherwise the return is not known. To prevent that states are not explored enough to gain precise information about the return, a minimal selection probability is defined for the nongreedy actions. Giving the nongreedy actions a likelihood to be chosen randomly is one possible way to mitigate the exploitation vs. exploration problem.

3.2.3 Temporal-Difference Learning

The Temporal-Difference method (TD) combines ideas of DP and Monte Carlo methods. While Monte Carlo as described above only updates the state-values after each episode, in the one step TD method, called TD(0), an update is calculated after each state transition. Similar to MC a model of the environment is not needed.

To find the optimal policy all state-values $v_0(s)$ are initialized arbitrarily. An action is then chosen according to the initialized policy. For each step of the episode an updated state-value is then calculated with

$$v_k(s) \coloneqq v_k(s) + \alpha\,[r + \gamma\, v_k(s') - v_k(s)] \tag{3.7}$$

where α is a step size parameter. Notice that for the TD error in the square brackets the next state must be visited. Only then the old value can be updated.

[1] An episode is a sequence of actions from the initial state to some terminal state.

The step size parameter is of great interest because properties like the rate of convergence can be adjusted if the step size parameter is chosen accordingly.

Nearly identical update rules can be applied for the action-values $q_k(s,a)$, which is used in the SARSA algorithm. If this is slightly modified, an important RL-algorithm called Q-Learning is obtained.

State of the art algorithms like Soft Actor-Critic build on Q-Learning. For this reason, a short overview of SARSA and Q-Learning is given here:

a) State-Action-Reward-State-Action (SARSA) – On Policy Control

Unlike TD Learning, SARSA uses state-action pair values instead of state values. As already mentioned, the updated rule for this method looks nearly identical to Eq. 3.7 [11]:

$$q_k(s,a) \coloneqq q_k(s,a) + \alpha\,[r + \gamma\, q_k(s',a') - q_k(s,a)] \tag{3.8}$$

As indicated in the name and seen in Eq. 3.8, SARSA uses the current and next action-values and reward for the update rule. Since the policy must be followed for the required data, this is classified as an on-policy control algorithm.

b) Q-Learning – Off Policy Control

A relatively simple approach to approximate the optimal action-value function q_* is that the greedy policy is used to update the action values even though another policy is followed. This leads to

$$q_k(s,a) \coloneqq q_k(s,a) + \alpha\,\left[r + \gamma \max_{\mathrm{a}} q_k(s',a') - q_k(s,a)\right] \tag{3.9}$$

as the update rule for Q-Learning (with a deterministic policy). It is proven, that Eq. 3.9 converges to an optimal solution in a finite MDP, even if the greedy policy is not followed. The sole requirement is that all state-action pairs are kept updated [11].

A disadvantage in real application compared to SARSA is that due to the policy that explores randomly, the optimal function can only be achieved by ramping the exploration down. SARSA takes its own behavior policy into account because the behavior and the acting of the policy are the same [11].

Algorithms like Soft Q-Learning handle these “Exploration vs. Exploitation” problem learning maximum entropy policies [12]. The term “entropy” expresses, that not only the optimal policy is

learned, but the entire range of highest return policies [12]. This fact must be kept in mind for later when the SAC Algorithm is applied. Reasons are explained in Chapter 3.5.

3.2.4 Summary

Different solution methods for the MDP were presented. In the following the properties of the algorithms are compared to back up the final choice of the algorithm, the Soft Actor-Critic.

As already deducible in the previous chapters, the algorithms differ mainly in the included states for the choice of the next action. The differences can be divided into depth (length) and width of the update. The depth (length) represents the span of states or actions being considered for the update of the value or action. The width is used to give a representation of the other possible state transitions or actions considered for the update rule. Using these key figures, the algorithms can be easily distinguished as illustrated in Figure 3.3.

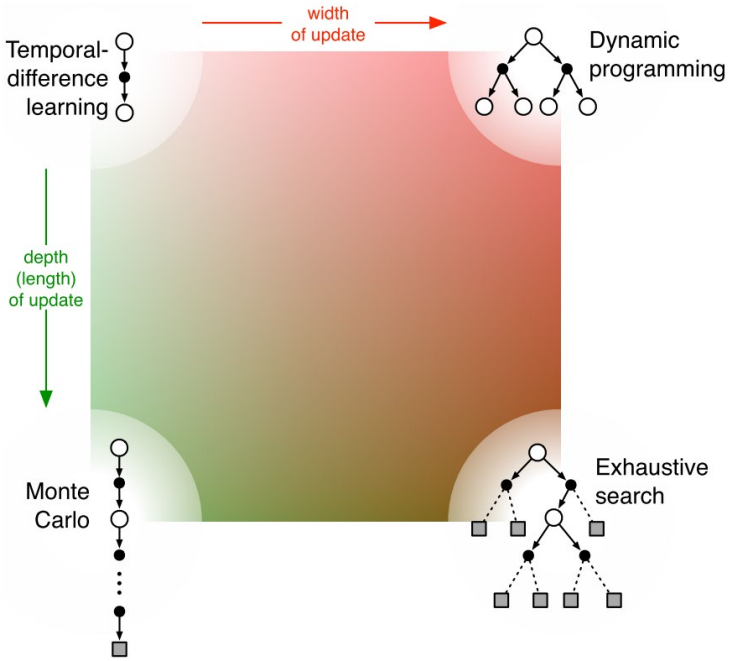


Figure 3.3: Comparing update depth and width between the introduced algorithms [11]

As to be seen, sample methods like MC and TD are on the left. Depending on the chosen depth TD methods can also have a deeper depth of update. These TD methods are then called TD(λ), where λ is the amount of time steps used for the next update. In MC, all steps of an episode are taken into consideration.

A further distinction can be made for on- and off-policy methods. While TD methods can be both, depending on the update rule (SARSA, Q-Learning), the only additional off-policy method can be conducted by some MC methods.

Since off-policy methods like Q-Learning do not rely on an exact model, that needs to be calculated for every time step, they would be favorable for the later discussed problem, driving a car in CarMaker. While this advantage lowers computation time, further challenges need to be faced to get the correct and fast predictions for action and state values.

3.3 Approximating Value Functions with Artificial Neural Networks

Previously, tables were used to store values for each state and action. For environments with big action and state spaces, the required memory, the access time and writing operations could increase the need of computation power in a way, making the calculation of a solution impossible.

A solution to avoid huge tables is by replacing these with functions. Different methods for function approximation can be used. These functions must be differentiable since gradient descent is used to converge against the true values of the states. Linear combinations of features and Neural Networks are possible approximation methods. The combination of Neural Networks and solution methods like Q-Learning is widely used in recent papers that deal with the subject of RL. For this reason, the focus in this thesis will be limited to Neural Networks.

The goal of this chapter is to get a general idea of the approximation of the value function $v_t(s)$ and the action function $q(s_t, a)$ using Artificial Neural Networks (ANN).

For better overview, an example of an ANN is shown in the following illustration:

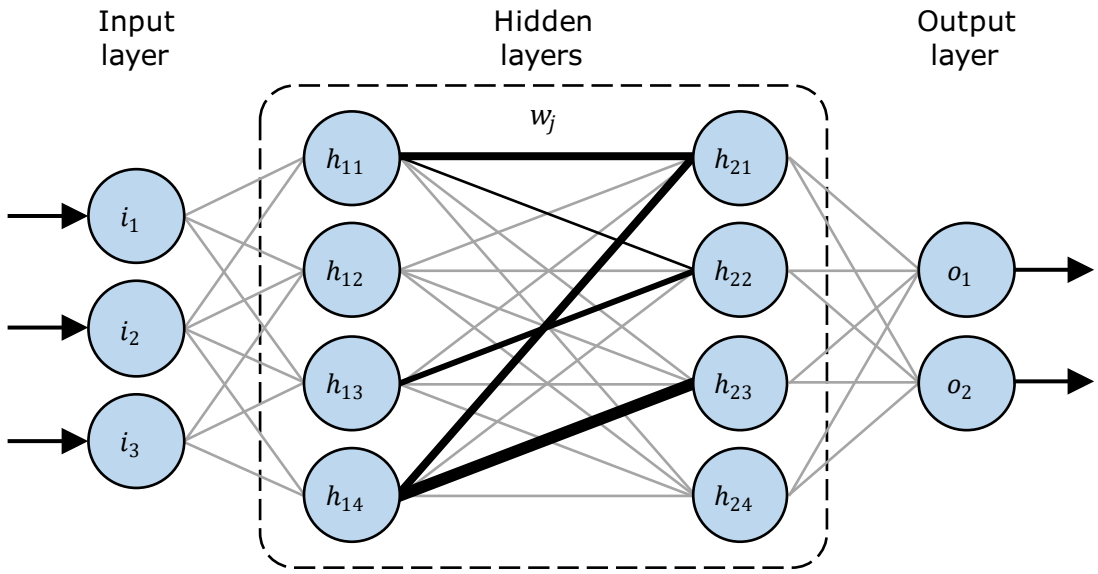


Figure 3.4: Fully connected feedforward artificial neural network

As can be seen, the ANN consists of an input layer, followed by two hidden layers and an output layer. Each layer contains an arbitrary number of neurons. Similarly, the number of hidden layers can be chosen freely.

Every neuron has one or more inputs. Using the inputs, a weighted sum is formed. Each input is individually weighted using w_j. The weight of the inputs is qualitatively indicted by the line thickness in Figure 3.4. Using a so-called activation function, the output is calculated including the

weighted sum. As illustrated, every neuron of each layer is connected to each neuron of the subsequent layer. As a result, the layers are called fully connected layers. The activation function is in most cases non-linear (e.g., rectified linear unit) [8].

This arrangement of multiple connected activation functions with correctly adjusted weights is used to estimate action and state values [11,13]. The weights are typically learned by training the neural network. Simplified, this is done by calculating the error between the output and real value. The TD error in Eq. 3.7 serves also as possible update metric to learn the value function [11]. Based on the error, the weights are then updated after each step and may converge to the true value. A common method for this procedure is called Backpropagation. More about this method and deep learning with ANNs can be read in the book “Deep Learning” by Goodfellow et al. [8].

3.4 Introducing Soft Actor-Critic

The Soft Actor-Critic (SAC) is a reinforcement learning algorithm that is based on actor-critic methods [14]. As with Soft Q-Learning, "soft" refers to the application of entropy as an exploration vs. exploitation regulator for stochastic policies.

To get a general understanding about the framework of the SAC, knowledge of actor-critic methods and entropy is essential.

3.4.1 Actor-Critic Methods

Actor-Critic methods combine two different methods. One method is the already known value-based policy update as used in Q-Learning and all other methods mentioned earlier. An optimal policy can also be found if the training is not done on state or action values but on the possible actions in the policy space itself. As an example, the first-visit Monte Carlo method can be modified in such a way, that the best policy is learned directly. An example is the so-called REINFORCE algorithm [11].

Both methods are given their own tasks, as can be derived from the name "Actor-Critic":

- Policy-based actor:
  ➔ Choses an action by returning a probability for each action in the action space.
- Value-based critic:
  ➔ Predicts the future return based on the actor's action.

A disadvantage of the REINFORCE algorithm is, that like first-visit MC, a gradient step can be done only after a whole episode. This leads to an inefficient training, because it is not bootstrapping [11,15]. Only the sum of rewards is known for the update. Single states are not distinguishable and therefore, the agent does not learn, which states lead to the return. This problem occurs often in policy spaces with high variance.

Looking back at Figure 3.3, it is identifiable for policy-based methods (bottom-left) that the policy updates are done farsighted resp. the decisions consider the whole depth. Since the policy gets optimized directly, the advantages are as follows:

- Even though learning is slower, the training process is more stable because convergence properties are better [11].
- The dimensionality of the policy is typically much lower than the state and action space. Therefore, less computational resources are needed.
- The learned policy can be stochastic, which is better for many environments.

The idea behind the actor-critic method is to compensate the disadvantages. These can be tackled by making it possible to do a gradient step after each time step. This is where the critic comes in hand.

Like in Temporal Difference Learning (see Chapter 3.2.3) an TD error is calculated based on the prediction of the critic and the actor's action. This is done by passing the current state values s_t to the actor and the critic as a tensor. The actor returns an action a_t, which is passed to the critic. The critic then computes the action value $q_t(s, a)$, that is used by the actor to update its policy (or to do a gradient step, if a neural network is used). The updated actor computes a new choice a_{t+1} for the state s_{t+1}. Based on these two actions a gradient $\Delta q(s, a)$ is calculated using the action values. The approximated gradient is then used to update the critic, while the actor is updated using $q_{t+1}(s, a)$. An overview of this process is shown in Figure 3.5.

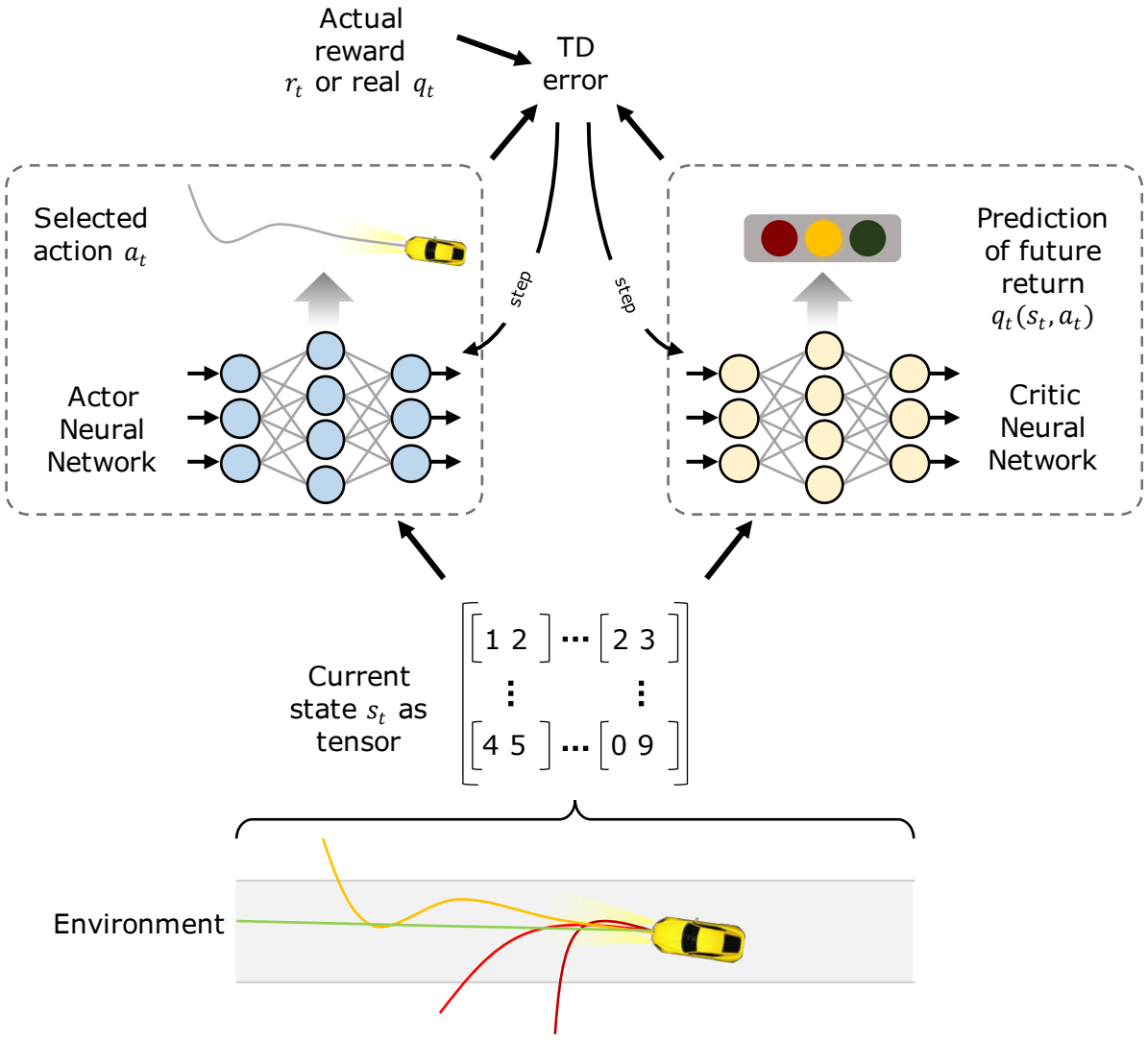


Figure 3.5: Actor-critic framework

3.4.2 Entropy

In information theory, entropy is the amount of uncertainty in a probability distribution [8]. In other words, entropy can be used as a measure for an agent to explore unknown states or state-action-pairs.

An example is illustrated in Figure 3.6. The chart shows two probability distributions. While the orange one has very low entropy, the blue one has maximum entropy. This is due to the equally distributed probabilities for each action. The blue case applies especially to unknown states. Actions with low probability have a high information content.

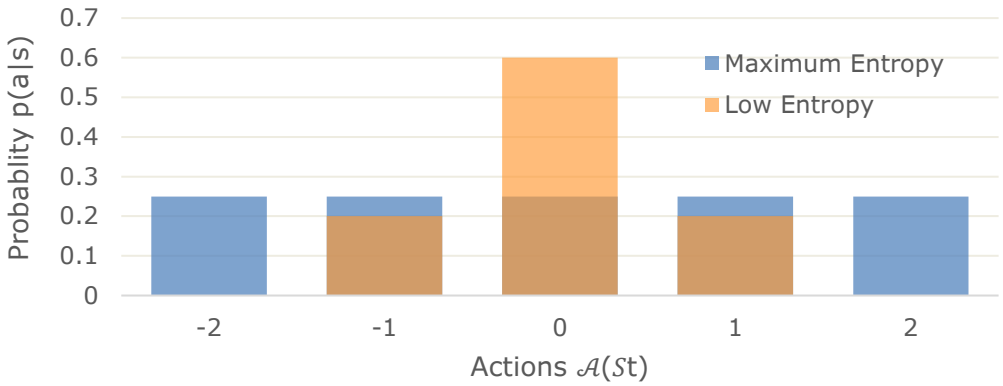


Figure 3.6: Two distributions of action probabilities in a state

The orange case can be state, that is already known very well. The action with the highest probability is likely to offer barely new information to the agent.

In RL, the agent normally seeks for the highest return, which means, that the entropy will lower over time. While the return converges to some value, it is possible that the agent found a local optimum. To counteract this problem, an entropy bonus can be added to the reward. This ensures a behavior, that explores unknown states.

The information of an action is measured in "nats" can be calculated using equation 3.11. The entropy is the expected value of all action's nats as seen in equation 3.12.

$$I(a) = -\log P(a) \tag{3.11}$$

$$H(a) = \mathbb{E}_{x\sim P}[I(a)] \tag{3.12}$$

The entropy can be then added to the reward. It is often scaled by a parameter called temperature.

Since the entropy is added to the reward, the temperature should be adjusted to the scale of the reward [11].

The SAC algorithm uses the entropy to put the agent in states, where entropy and reward are maximized [13]. Maximizing the entropy can lead to a better long-term reward and more diverse variations of learned policies.

3.4.3 Soft Actor-Critic

The Soft Actor-Critic is a model-free deep RL algorithm [13]. It is based on an off-policy actor-critic method that uses a maximum entropy RL framework. The actor tries to maximize both reward and entropy. Maximum entropy promotes random behavior and avoids collapsing to determinism.

For the problem discussed in this thesis, an updated implementation by Haarnoja et al. [14] is used. It contains an automatic gradient-based temperature tuning method. The temperature parameter adjusts the entropy. The goal of the tuning method is to scale the entropy correctly to obtain a good performance [14].

Several other important properties of the SAC are listed below:

- The ability to learn off-policy is used to train on earlier transitions, that are saved in a replay buffer. This is used to gain a higher sample efficiency.

- Earlier off-policy based deep RL algorithms like DDPG [16] are very hard to stabilize and are brittle to hyperparameter settings [14]. This is due to an interplay of an actor with a deterministic policy and a q-function network as a critic. The SAC combines a stochastic actor with one or two Q-functions as critics. This combination proved itself to be very stable in the SAC framework without the need of extensive tuning of hyperparameters [14].

By combining all properties of the SAC, a highly practical algorithm is a result.

The following pseudocode shows a simplified procedure of the algorithm. It is strongly based on the algorithm shown by Haarnoja et al. [14].

The first step is to initialize the critic and the actor net and a replay buffer, where all transitions will be stored:

Input: Initial parameters for Q-Networks and policy weights.

- Initialize target network weights.
- Initialize an empty replay buffer $\mathcal{D}$.

After that, a loop is used to do a specific number of iterations of a specific number of environment steps followed by some number of gradient steps:

---

for each iteration **do:**

  for each environment step **do:**
  - Sample action a_t from $\pi_\phi(a_t|s_t)$.
  - Sample transition from the environment s_{t+1}.
  - Store the transition in the replay buffer $\mathcal{D}$.

  end for.

  for each gradient step **do:**
  - Update the Q-function parameters using a stochastic gradient of a soft Q-function loss that contains samples of $\mathcal{D}$.
  - Update the policy weights using policy update rule of SAC that is multiplied by temperature α.
  - Update temperature α using approach of SAC.
  - Update target network weights.

  end for.

end for.

In case the networks converged to some value an output is generated containing the optimized policy and critic weights:

Output: Policy weights

The newly optimized policy can then be deployed.

The later used SAC implementation made by TensorFlow uses the same procedure [17].

4 System Overview

This chapter provides information about the software and the architecture design that is used to conduct the experiments. It starts with a high-level architecture overview and proceeds with additional implementations inside the components.

4.1 Architecture

Following figure gives an overview of the components in the architecture:

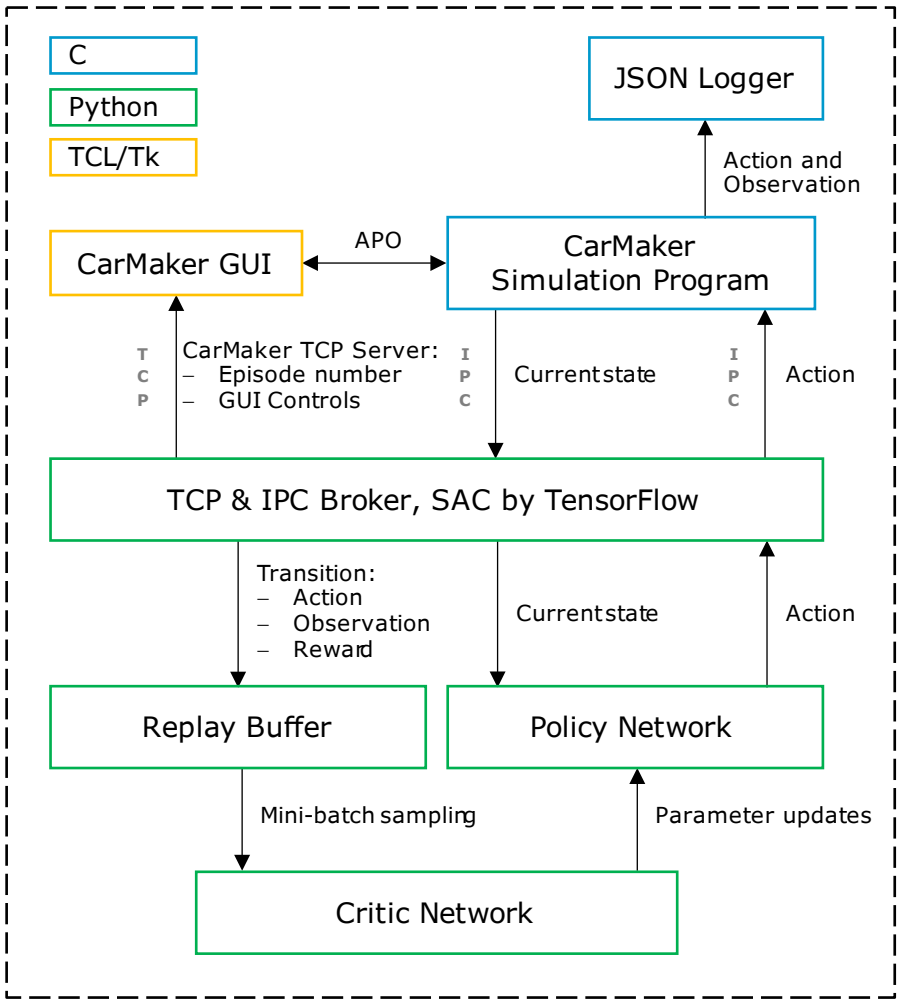


Figure 4.1: Architecture

High priority data exchange between CarMaker and TensorFlow is implemented by the so-called messaging library ZeroMQ. The library is available for Python and C and provides a high-level application programming interface (API) for safe data exchange between processes. An inter-process communication socket (IPC) is used. A synchronous connection ensures, that both processes communicate properly.

Every 125 cycles in CarMaker, the simulation program sends the current state to a python broker, while the latter sends an action, which is based on the last state. The broker translates the received string into an array. The reward function uses the new state and calculates the reward for the previous action. A first-in first-out (FIFO) replay buffer receives the transition and saves it.

In parallel, the policy network receives the state too and calculates the new action, that CarMaker receives at the next cycle. At the same time, the critic network receives a randomly sampled mini batch from the replay buffer. The batch is used to train the critic one step.

CarMaker saves the received action for 125 cycles and then waits for a new action.

A more detailed documentation of the training sequence or epoch can be found in the Jupyter Notebook file.

Multiple extensions are added to the simulation cycle of CarMaker. The knowledge of these modifications is important for the training process. Some of these extensions have a huge impact on the training process. However, they could also carry bugs with them. This possibility will be discussed in more detail during the experiments.

Table 4.1 on the following page shows an overview of all implemented features into the CarMaker simulation.

Table 4.1: Overview of implemented changes to the CarMaker simulation cycle

Name	Description	Background
Road border distance sensor	The sensor scans the road iteratively by moving a point along a line. The output is the distance in polar coordinates.	To calculate distance from the vehicle frame to the road border. The distance is implemented to the state space. The IPG Road sensor does not reliably sort the found road boarder points according to its distance. A simple algorithm is used because performance is sufficient.
Automatic teleportation	The multi border solver receives a new position and teleports the car to the new position. This is a modified implementation of the entry #771, that is found inside the IPG Tech Wiki.	To examine certain effects on training behavior and the policy. To avoid the termination of the CarMaker Simulation.
Direct Variable Access	Using CarMaker implemented functions.	To overwrite the IPG Driver. Also, the Slot Car Mode is activated if the vehicle leaves the track. This is done to avoid the termination of the CarMaker Simulation.
JSON Data Logger	Using a JSON library, that parses quantities to a JSON file. A new file is created for each episode.	No low-level access to data storage functions of CarMaker inside C program to create a file if an episode ended.
Simulation Speed Refactoring	CPU time is measured of the simulation cycle to calculate a factor. This factor overwrites the simulation speed chosen by CarMaker.	If the CarMaker simulation gets too fast, the simulation speed collapses. This results in an overall slower simulation speed.

4.2 CarMaker Simulation as Markov Decision Process in Python

The CarMaker simulation environment must be formulated as an MDP in python. The TensorFlow Agents module provides a python environment class, that includes the functions for an MDP. This class is used as a template. Following methods must be defined inside this class as shown in the following pseudocode:

```
Class inheriting python environment class:

      Initialization method:
       - Inputs: Discount factor, Number of the CarMaker instance.
       - Starts CarMaker GUI and Simulation.

       Observation/State spec method:
       - Returns dimension of the array.

      Action spec method:
       - Returns dimension, data type and limits for each value of the
         array.

      Reset method:
       - Sends CarMaker commands to put the vehicle in a normal state by
         one of the following methods:
           o Drive vehicle to a predefined path along the route using the
             IPG Driver.
           o Teleport car to start.
           o Teleport car to a random route section.
           o Change route and start a new simulation.

      Step method:
       - Inputs: Action
       - Receive new state.
       - Send an action based on the old state.
       - Calculate rewards based on new state.
       - Rule based episode ending and calling the reset method.
       - Returns reward, discount factor and end of an episode.

end class.
```

Features and properties of the state and action space are adapted iteratively for each experiment. Both are continuous and contain only floating-point numbers. Possible specs of the state spaces are illustrated in Figure 4.2.

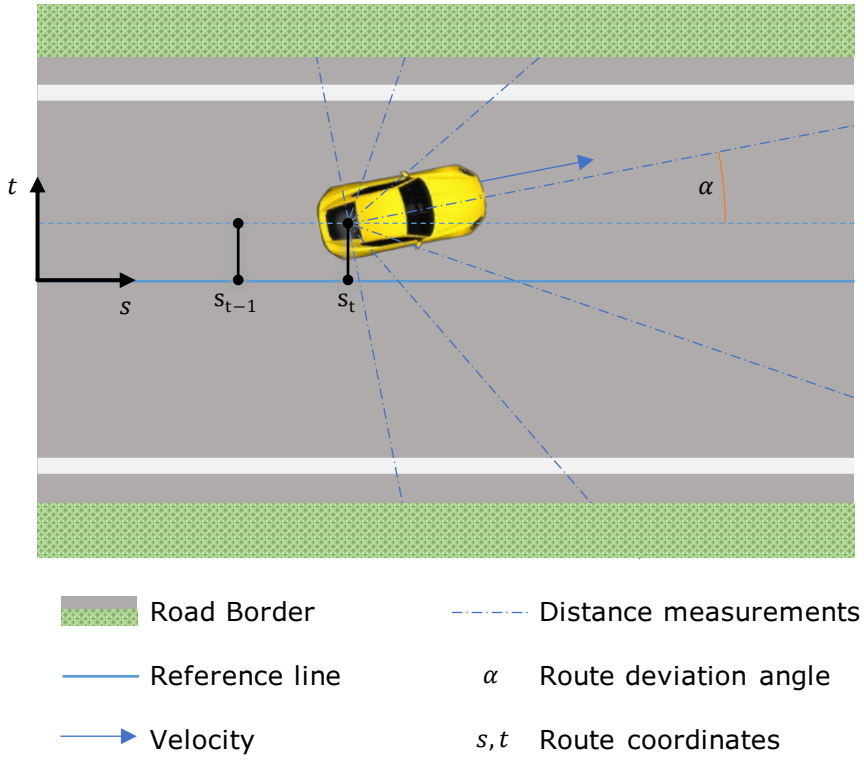


Figure 4.2: Visualized features of the state space

During the experiments two types of reward functions are examined by changing their weights. These reward functions use values of the state space to calculate the reward. For the maximization of the return, the episode length is also examined. An episode is always ended if the vehicle leaves the road. Extensions to the rules are also examined based on the outcome of the agent's behavior during the experiments.

The exact formulations of the state space, the action space and the reward functions can be found in Chapter 5 for each experiment.

The replay buffer is filled at the beginning with a predefined number of initial collect steps by using a random policy before the beginning of each epoch.

4.3 Hyperparameters

In a first step, all hyperparameter values are derived from best practices values found by Haarnoja et al. [13]. The replay buffer size and the mini-batch size are aligned to values used by Fuchs et al. [6].

The table below shows an overview of the hyperparameters. Almost all hyperparameters were varied during some experiments. They can be recognized by the given range of values instead of a single value. Parameters for the state space are not included in this table. State space feature parameters are explained in the chapters about the experiments themselves. Same applies for initial condition parameters, which apply for the beginning of each episode during an epoch.

Table 4.2: Overview of hyperparameters

Hyperparameter	Value
Mini-batch size	128 – 1024
Replay buffer capacity	100000 – 400000
Reward scale	1 – 15
Critic Learning Rate	4e-4 – 4e-6
Actor Learning Rate	4e-4 – 4e-6
Alpha Learning Rate	4e-2 – 4e-6
Discount Factor	0.9 – 0.998
Actor Net Size	2 Hidden Layers with 256 units each
Critic Net Size	2 Hidden Layers with 256 units each
Initial collect Steps	0 – 50000
Train steps per iteration	1 – 2
Number of steps for policy update	1 – 4
Number of parallel environments	1 – 4

5 Experiments

With the help of the experiments the capabilities of the SAC algorithm are evaluated. The experiments also check the robustness of the IPC interface and especially during the first trials, troubleshooting is done to ensure no additional noise is added to the data.

The newly gained knowledge during the experiments is then used to tune the learning environment by changing hyperparameters of the SAC itself and in CarMaker.

Difficulties occurred mainly in the duration of the experiments, which is due to the high computational effort of the training. For this reason, the repetition of long trainings with many episodes is dropped.

5.1 First trials

The first experiments were done on a relatively easy scenario. This includes a flat circuit with large curve radii as shown in Figure 5.1. The route also includes a section with a curvilinear transition inside the blue rectangle.

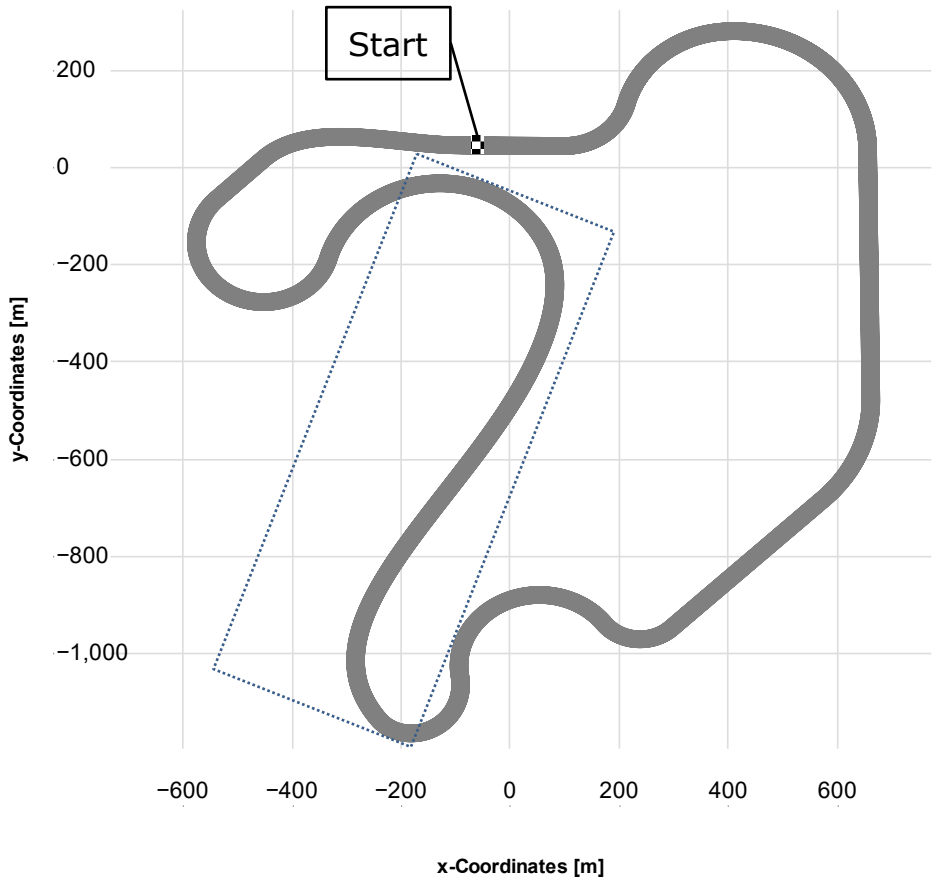


Figure 5.1: Track for first experiment

The goal of this test is to verify the feasibility of the interaction between CarMaker and the TensorFlow API. The Python script is also checked for functionality in the process. The results of the first practical test with the Soft Actor Critic algorithm in connection with CarMaker are used as a starting point for further experiments.

The hyperparameters of the TensorFlow example are used here [17].

5.1.1 Action & State Space

To achieve a good training behavior in the shortest possible computing time, the action and state space include as few variables as possible. This also has the advantage that in case of wrongly chosen environment or action variables, different variables can be tested.

The action space $\mathcal{A}$ contains two continuous bounded sets A_S and A_G. A_S represents the set of all steering commands and A_G the set of all possible gas and brake signals. The signal $a_{S,t} \in A_S$ is the steering wheel angle (SWA) in radians with the origin in neutral steering wheel position (driving a straight line). Accelerator pedal position (APP) and brake pedal position (BPP) are dependent on the sign of $a_{G,t}$. Within the python script the set $\mathcal{A}$ is defined by a bounded array spec, which is a data type in TensorFlow. Mathematically, this bounded array can be defined as follows:

$$\begin{aligned} &\mathcal{A} = [A_G, A_s], \mathcal{A} \in \mathbb{R}^2 \\ &A_G = \{x \mid (|x| \leq 1)\} \\ &A_S = \{x \mid (|x| \leq c_S)\} \ with \ c_S = 9.42\, rad \\ &a_{S,t} \in A_S, \qquad a_{G,t} \in A_G \\ &BPP = -a_{G,t} \cdot 1_{\{a_{G,t} \leq 0\}} \\ &APP = a_{G,t} \cdot 1_{\{a_{G,t} > 0\}} \end{aligned} \tag{5.1}$$

The observation or state space $\mathcal{S}$ can basically contain any parameters without constraints of the CarMaker environment. For these trials, only as few variables as possible are included. A bounded array spec is also applicated. Boundaries are min and max values of each variable. It should be noted that no road border distance sensor is used, because it was not available during this stage. Aside from the RL-driver implementation, a plain vanilla CarMaker was used.

The state vector $\mathcal{S} \in \mathbb{R}^n$ includes the following variables:

- Magnitude of vehicle velocity v
- t-Coordinate
- 5 Road sensors every 7 m in front of the vehicle:
    - all measuring deviation angles between vehicle direction and reference line
    - road width for first and second sensor
- Steering angle of last time step $a_{S,t-1}$

Figure 5.2 shows an illustration of most variables:

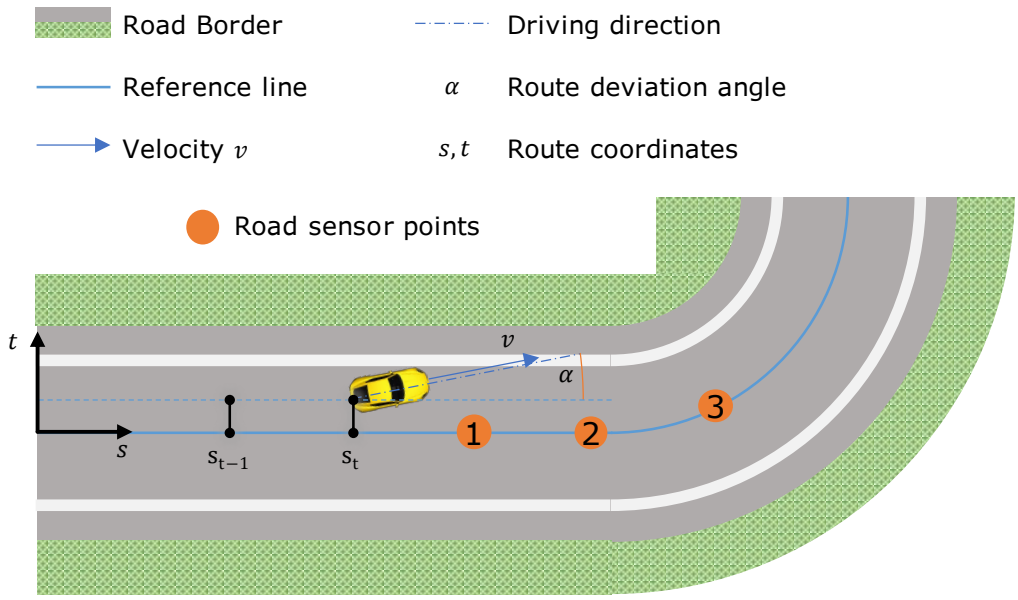


Figure 5.2: First trial state space

5.1.2 Reward Function

The goal of the reward function is to maximize the covered distance on the track. Therefore, the reward r_t depends on the velocity v of the vehicle. To ensure that the agent drives in the correct direction, v is weighted with the cosine of the route deviation angle α. The agent receives a penalty if the vehicle leaves the road. The road is considered left if the t-coordinate times 2 of the vehicle is greater than the track width. The exact formulation of the reward function is given in Eq. 5.2:

$$r_t = \begin{cases} v \cdot \cos(\alpha \cdot c_\alpha) \cdot c_v, & if\ 2 \cdot t < road\ width \\ v^2 \cdot c_p, & otherwise \end{cases} \qquad 5.2$$

The parameters c_v, c_α and c_p weight individual components of r_t. They are initially set to 1.

5.1.3 Initial Conditions and Reset Algorithm

In the CarMaker simulation environment, initial conditions are always the same. The initial vehicle velocity v_0 is set to 0. The start position is on a straight, all road sensors return the same. Therefore, all values of s_0 are 0.

An episode is ended if the vehicle leaves the road or $|\alpha|$ is greater than 90 degrees. For this first test, the entire simulation is stopped, and the same test scenario is started.

5.1.4 Results

At episode 202 or 40000 steps the greedy policy of the agent maneuvers the vehicle along the road without touching the road border. The vehicle is driven at a velocity of about 60 km/h and receives hasty steering commands, which are impossible in real terms. The behavior of the RL-driver is shown in Figure 5.3 as an orange driving path line. The steering wheel angle is visualized as an offset line to the path in blue. Similar is shown for the IPG RaceDriver which is used here as a reference.

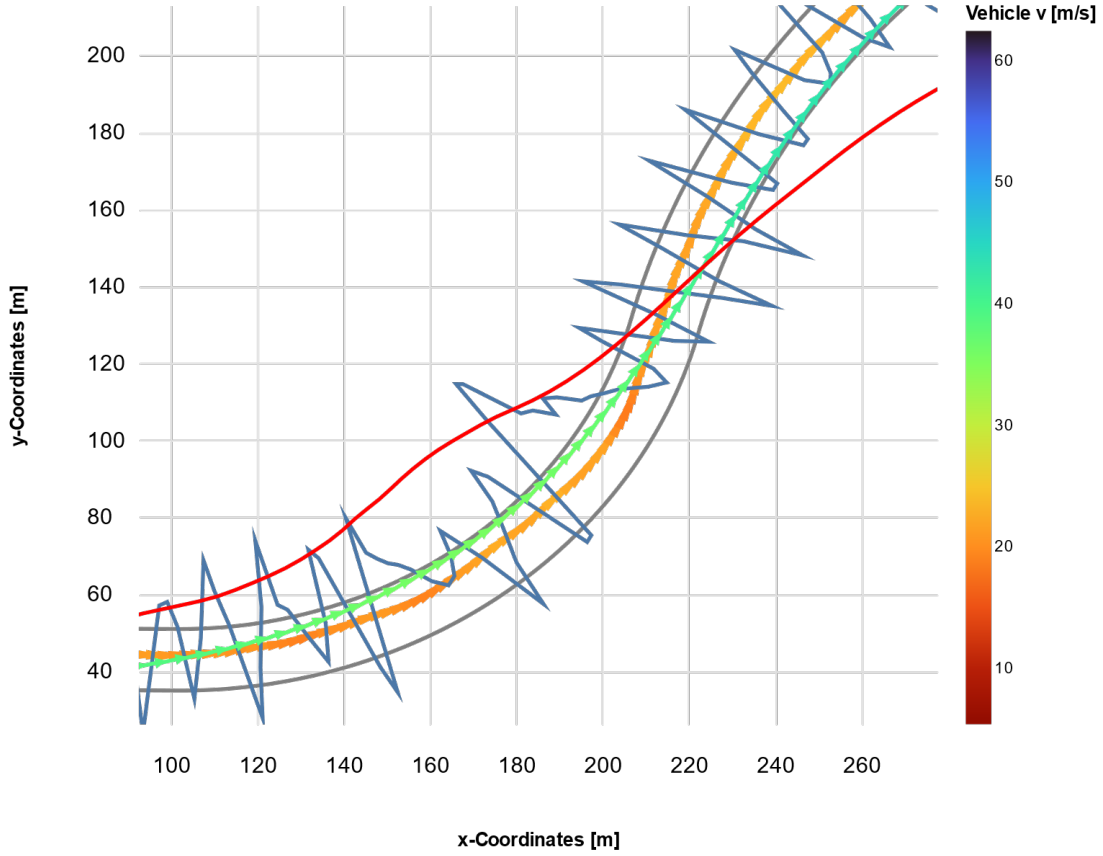


Figure 5.3: Driving path and steering wheel angle of the RL-Agent (orange and blue) and the IPG RaceDriver (lime and red)

An improvement over the steps of the steering behavior is recognizable as shown in the following histograms in Figure 5.4. The sum of each steering command is normalized to 1, to make comparison easier between the episodes. All episodes were evaluations of the greedy policy.

A video of the final evaluation is available at the following link: https://youtu.be/uAONUPHNisk

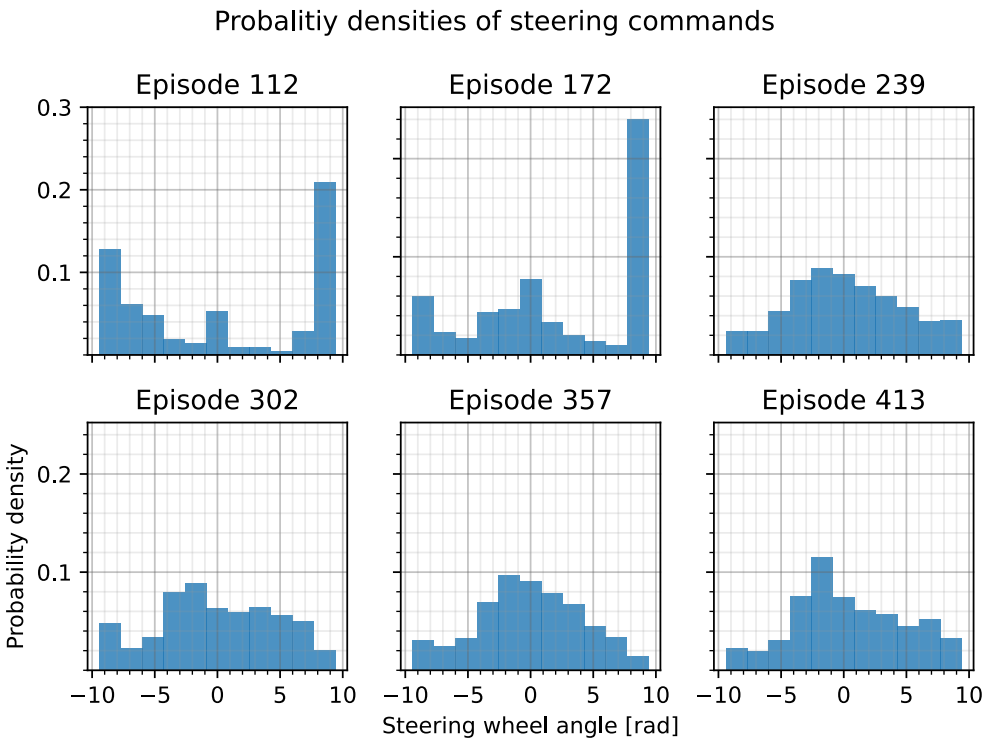


Figure 5.4: Probability densities of the steering commands during the first training

Last plots indicate that the agent makes more smooth steering commands during the training. But it should be kept in mind, that after 80,000 steps or 230 episodes, the steering behavior remained the same. The return behaves similarly and converges at the same rate as shown in Figure 5.5:

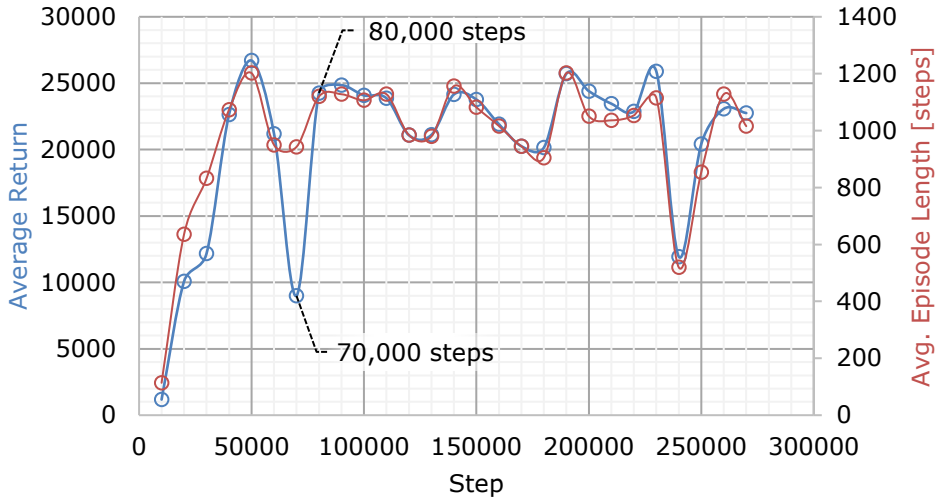


Figure 5.5: Average return in evaluation of the first trial

Figure 5.6 shows the average speed converging with similar behavior as the return. The red line represents the moving average. The blue line is a third-degree polynomial function fitted to the black data points. Both lines resemble each other regarding the limit.

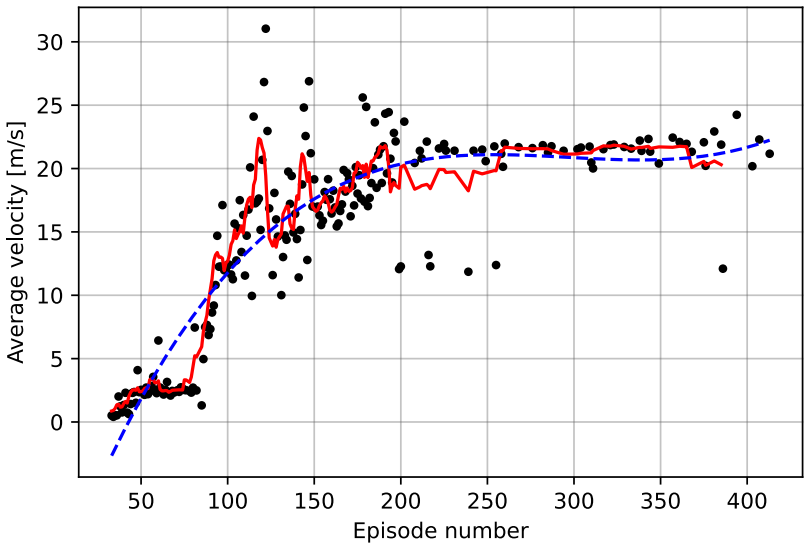


Figure 5.6: Average velocity over episodes during training in first trial

Also, the agent is not able to drive faster, because he loses traction due to the abrupt steering actions. The speed converges to 76 km/h.

Notable is, that the agent stays always in the same direction as the route. This is most likely due to the reward function which promotes this behavior.

Reasons for this behavior and the fast convergence of the return as shown in Figure 5.5 could be, that state space is too small or unknown variables. Some of these could be:

- Unkown side slip angles leads to unkown heading of the vehicle
- Unkown components of velocity in x- and y-direction may worsens driving ablities
- Road deviation angle not suitable to predict route

The agent learns to avoid the road border after just 70,000 steps by decelerating to standstill until the episode is ended as shown in Figure 5.7. This behavior also seems to slow the training down and may be avoided by adjustments to the reward functions in later experiments.

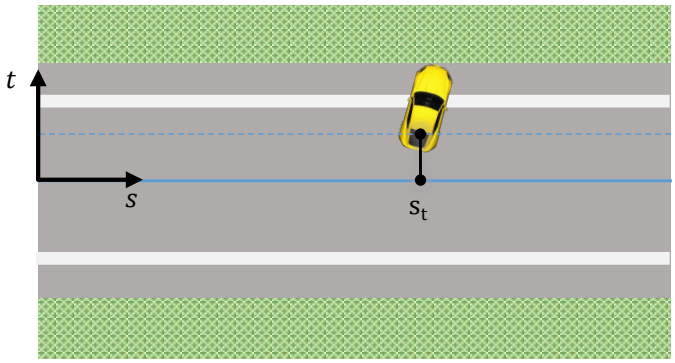


Figure 5.7: Dilemma for the agent: Receiving a penalty by crossing the road border or do nothing, which slows down the exploration of the environment.

Therefore, the next experiment tackles the steering problem by adding velocity in x and y to the state space, separating the y component from the reward. However, evaluation of this test with over 700,000 steps did not reveal any further improvements.

For this reason, the first area to be optimized is the performance of the training. Especially at the first steps where episodes are very short, the restart of the CarMaker simulation takes around 4-5 seconds. Therefore, more than 50% of the wall time for the first 100 episodes is restarting the CarMaker simulation. To solve this problem, later experiments avoid stopping the simulation by using later described mechanisms.

5.2 Optimization of Time Efficiency for Training

In this chapter, the focus is set on the highest possible utilization of the computing resources. In addition, the goal is also to further improve the training process itself and its design.

5.2.1 Reset Algorithm

First, the previous experiment shows that most of the time, loading a test scenario and restarting the virtual vehicle environment is time consuming. Especially during first episodes, loading time is much longer than training time as shown in Figure 5.8. This is especially true at the beginning of the training and whenever the episodes are of short duration.

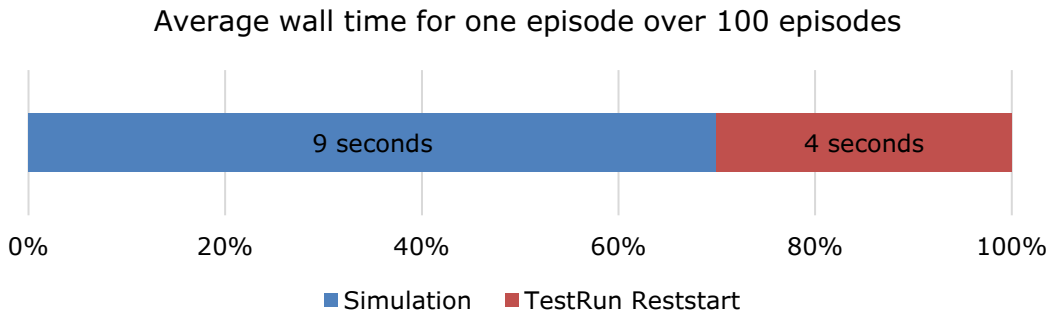


Figure 5.8: Average wall time for one episode during first trials

For this reason, various approaches have been tested to avoid restarting the simulation or to reduce the time as much as possible.

Multiple reset algorithms for the vehicle where tested:

- IPG Driver and Slot Car Mode[2]:

  This is programmed in a simplified way as follows, whereby target conditions like a maximum path distance were also added.

```
while Car is not on Road:
      while Target conditions not met:
            Activate Slot Car Mode and IPG Driver
```

[2] The Slot Car Mode prevents the car from leaving the track. Simplified, the mode creates an invisible wall at a lateral distance from the route line.

The dead time needed to drive the car back to the correct position can be avoided by using at leat two CarMaker instances.

- Teleporting the vehicle with multiple variations:
  - Teleporting always to start
  - Teleporting to random position along the path
  - With a initial velocity of 0.
  - With a random initial velocity between 0 and 100 km/h

The reset algorithm with the IPG Driver had to be discarded with the reason of unpredictable behavior leading to delays and interruptions of the training and possibly adding more noise to the loss functions. Especially if the driver model is switched to the IPG Driver, gas control commands did not transfer to torque to the wheels. This resulted in multiple delays with each up to 10 minutes during the training. Reasons for this behavior remain unknown.

In contrast to the first algorithm, the teleportation of the car eliminates nearly completely the simulation restart time. In addition, the initial conditions for the vehicle are static if the initial speed is set to 0. A new value for the initial speed is directly send to the multi body solver of CarMaker which resulted to slipping tires at the beginning of each episode. An initial speed other than 0 has therefore been discarded. It should be noted that for unknown reasons the car is not always correctly initialized after the teleport, which manifests itself in unrealistic behavior, such as turning the vehicle in dependence to the steering input on the spot.

Figure 5.9 shows the difference between a random and fixed start position. Both are very similar, but the small differences suggest that a randomly chosen start position will be beneficial later in the training process, as there is more variation in the sections of track driven. At a fixed start position, the agent must accomplish the first track section, a straight e.g., until the agent is able to observe different track sections which can be e.g., a left turn or an S-curve.

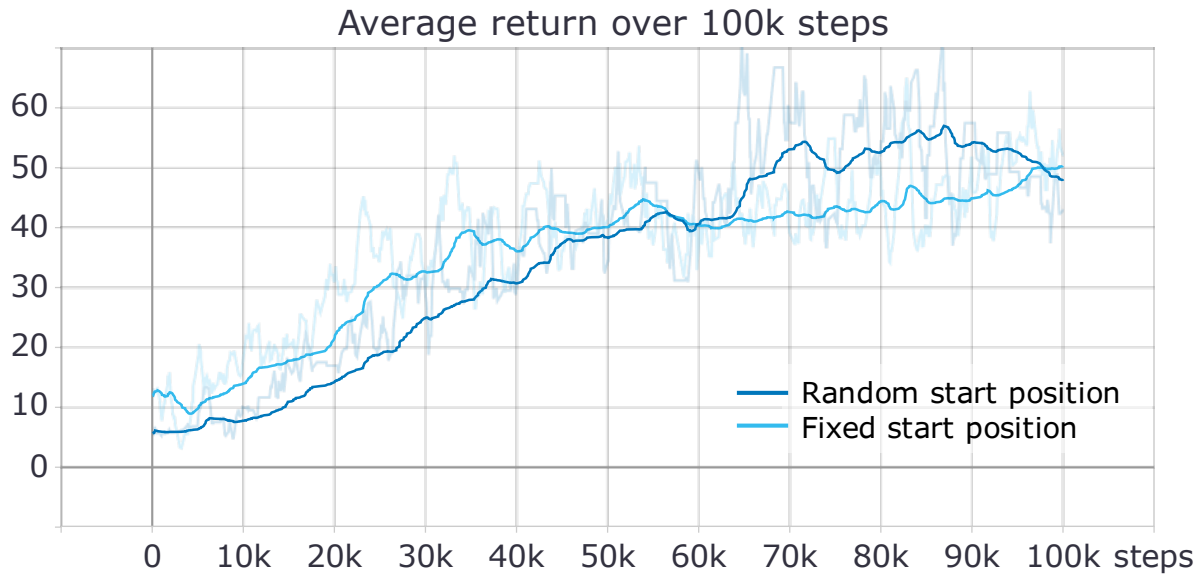


Figure 5.9: Average return with and without random start position

5.2.2 Performance

The general architecture proves to be robust and performant during the experiments. However, the performance collapses when the maximum real-time factor is selected. The real-time factor then fluctuates between 0.5 and 2. A manual setting of the real-time factor with a gradual increase before it collapses again yields a factor between 2 and 10 which depends on the hardware. Figure 5.10 shows the differences between the tested hardware configurations.

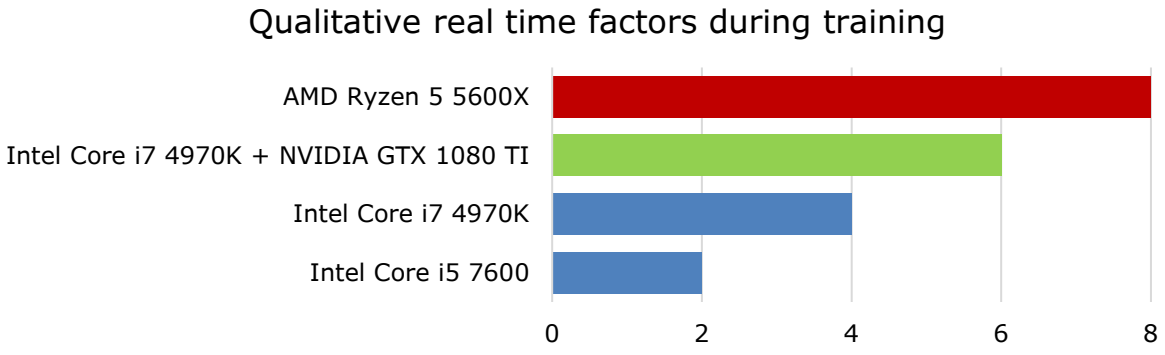


Figure 5.10: Qualitative comparison of real time factors during training for different hardware setups

The reasons for the differences in performance are not further investigated.

Multiple CarMaker instances in parallel were also tested. However, experiments with this setting showed an overall lower performance when computing on one computer. There may be advantages in a distributed setting, which is not part of this work.

5.2.3 State Space

The convergence behavior depends on the dimensionality of the state. To confirm this, two different setups of a training are compared. Apart from that, all hyperparameters remained the same.

Figure 5.11 shows the courses of the average return of 10 evaluation runs every 10,000 steps. The pale lines display the raw data. The progression of the time series is smoothed for a better comparison using TensorBoard. It is visible that a smaller feature size results in a faster convergence.

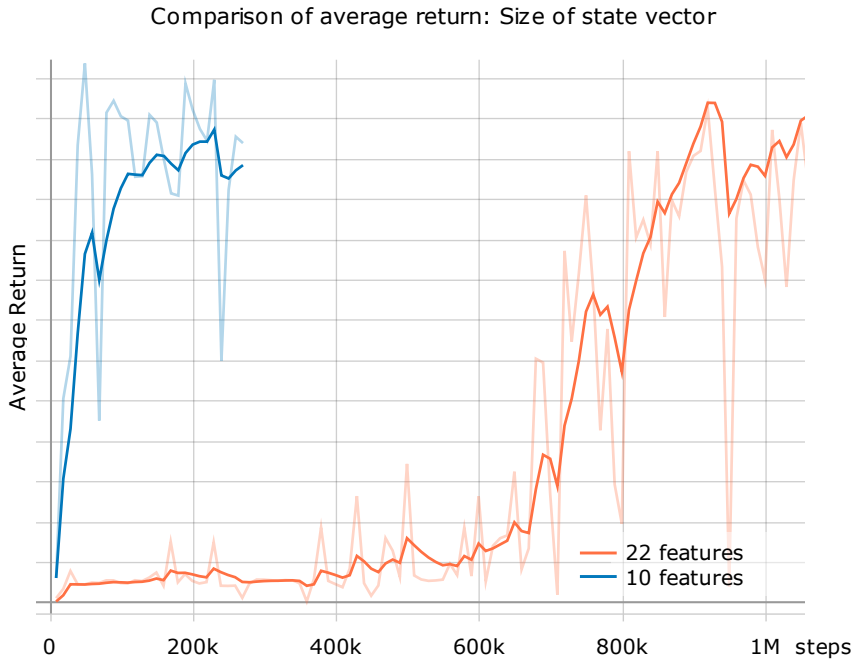


Figure 5.11: Influence of the number of features regarding the convergence behavior

The loss function for the actor shows the same convergence behavior. A significant change of the loss or the average return after convergence is not evident. In fact, the average return for the training with 22 features remained nearly the same after 10 million steps.

It should be noted that the feature size of the state space has a significant impact on the driving behavior. This topic is addressed in chapter 5.3.

5.3 Optimization of the Track Performance

The first trials show an unrealistic behavior regarding the change in steering angle (see Figure 5.3) and gas and brake inputs. Abrupt steering behavior has a negative impact on the vehicle's handling. For example, complete changes in steering direction take place with high frequency in curves and in straights. In the latter, this behavior leads to a high degree of driving dynamic instability at high speeds, which often results in a complete loss of control of the vehicle and thus leads to an early end of the episode. In addition to the steering behavior, it is also apparent that sections of road that can be driven quickly are driven too slowly. Adjustment levers for steering behavior and speed are examined in more detail in this chapter.

In addition, the reward function is redefined because the previous trials do not promote an ideal driving line.

5.3.1 Driving behavior

Figure 5.12 shows the vehicle's driving path on the left at episode number 242. It is shown as a thicker line with blue tones. A thinner blue line is plotted as an offset to the path. It qualitatively shows the steering wheel angle. A larger distance from the path is linked to a larger steering angle in the respective direction. The heatmap on the right shows the distribution of the steering commands during the training in the curve on the left. Data from 20 episodes are averaged into a bin here.

After about 240 episodes, it can be seen on the right that the steering angle is selected to be the same within the reading accuracy. With increasing training, the variance increases again. Possibly, conclusions can be drawn here about the entropy algorithm of the SAC, which increases the temperature factor when the training stagnates. However, the exact reasons remain unknown.

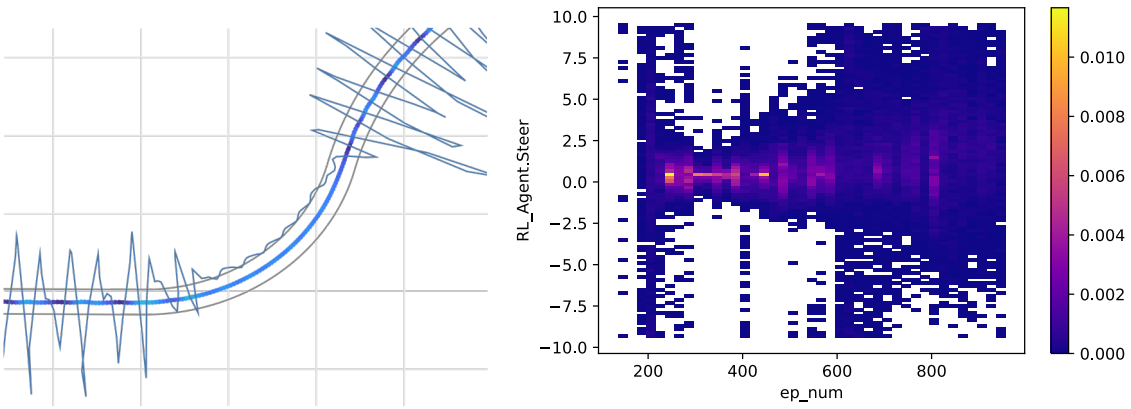


Figure 5.12: Steering behavior in the first curve. Shown on the left is the steering angle as offset to the driving line. On the right, the distribution of steering commands in this curve during training is shown.

To create a more realistic steering behavior, the following options are proposed and tested:

Table 5.1: Modifications to the training to obtain a more realistic steering behavior.

Proposed Modification	Implementation	Reason	Result
Replacing the angle of the steering wheel in the action vector	▪ Angle velocity ▪ Angle acceleration ▪ Torque on steering wheel	Adaptation of the maximum values to a human driver possible	Very high variance of input values persists. When presetting the steering speeds with small selection interval, difficulties arise during curves, but the steering behavior is more realistic. If high steering speeds are possible, the steering behavior remains unrealistic. Best results are achieved by defining the acceleration. The training time is longer for achieving a good performance. Steering by torque results in severe understeer.
Modifications to the reward function	▪ Penalty for fast angle velocity ▪ Penalty for side slip angle of the vehicle	Direct the agent's behavior toward giving as few control commands to the steering wheel as possible.	No improvements discernible.
New features for the state space	▪ Side slip angle ▪ Yaw rate of the vehicle ▪ Lateral and longitudinal acceleration	Effects on the driving behavior due to abrupt steering maneuvers are inexplicable for the agent due to missing features.	Improved handling due to no loss of control of the vehicle due to skidding.

To reward a better driving line, the Reward function is defined as shown in Eq. 5.3. This function rewards the choice of the smallest possible radius of curves, since a higher reward is achieved with less distance.

$$r_t = \begin{cases} \Delta sRoad \cdot (1 + Steer.AngVel)^{-1}, & if\ on\ road \\ v^2, & otherwise \end{cases} \qquad 5.3$$

Combined with the new features, the steering wheel control using angular acceleration and the new reward function, the agent achieves an overall better driving result. Results are shown in the next chapter.

5.3.2 Results

The new setup achieves the desired results of smoother steering behavior and more dynamic driven corners. Thanks to the new features, the agent achieves decent velocities and lap times, which are comparable to the IPG RaceDriver with some restrictions. The steering behavior shows still smaller oscillation, which is also clearly visible in the raw control data. The selected angular acceleration shows a very high variance, just as in the first tests. Figure 5.13 shows the new driving line and the steering angle as a blue offset line in the first curve of the track.

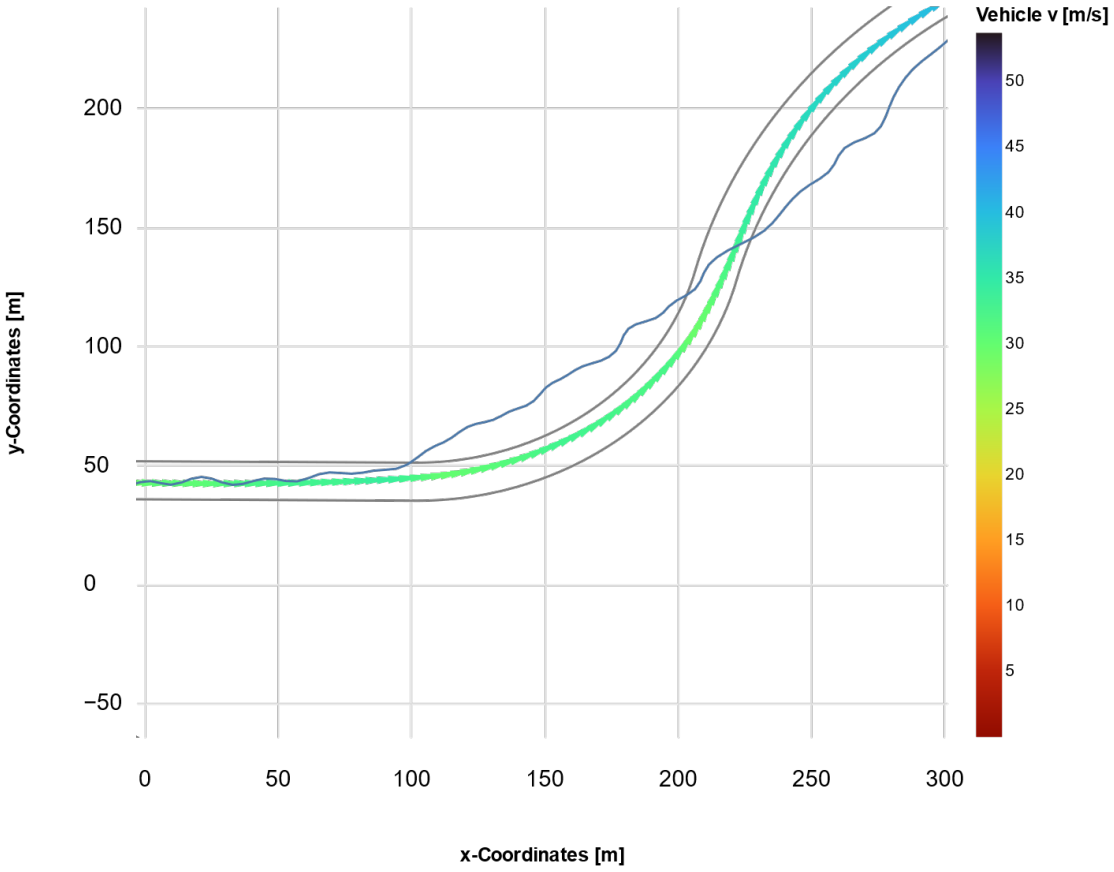


Figure 5.13: Driving path and steering wheel angle with the final setup.

The path shown above required 8 million steps of training. Actor loss and the return, however, already converge from one million steps. Similar results could already be reproduced at this point.

A video of the agent's driving performance is available here: https://youtu.be/T9s_7QHB_yc

In the video and in Figure 5.14, it is noticeable that the agent does not extend the maximum speed in the straight lines. Reasons may be that high speeds cannot be explored sufficiently during training, since in early training the termination of the episode by leaving the lane is particularly frequent here. Possibly the agent lacks the necessary sensors for this, which look far enough ahead.

The velocity profile of the first lap is shown in Figure 5.14. To make the straight lines recognizable, the amount of curvature is shown as a red dotted line. With the final policy, the agent needs approximately 140 seconds for the first lap, which is 16 seconds slower than the IPG RaceDriver with manual adaptions.

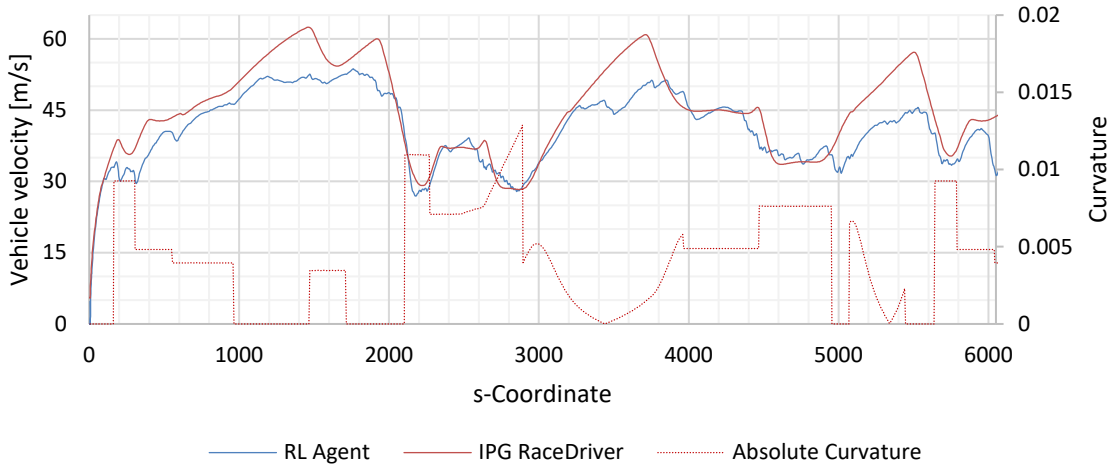


Figure 5.14: Velocity profiles of the first lap of the RL-Agent and the IPG RaceDriver

6 Conclusion

This research aimed to identify the applicability of a reinforcement learning algorithm in general. Based on a first implementation of the software architecture and the SAC, a state-of-the-art RL algorithm, it can be concluded that RL is suitable to tackle certain optimization problems in an online simulation in CarMaker. The results indicate that not only the problem investigated here of driving a car around a racetrack can be optimized, but also other problems. To what extent the level of difficulty, i.e., the number of features in the action and state space, can be increased should be the topic of further investigations.

Especially in view of this work, the behavior of the agent needs a more detailed examination. For all experiments, command inputs to the vehicle control turned out to be noisy. This also applies for the loss of the critic, which contained a very high level of noise during all training sessions.

7 Outlook

Since the time span between results and start of the experiments was one of the major bottlenecks of this work, a higher computing performance would be very beneficial for further examinations. In particular, the performance gain for training the neural network should be investigated on a GPU, rather than on a CPU as in this work. In this context, it is also advantageous to run CarMaker instances in parallel on multiple computers, while the training is done on a single GPU. Also, the software architecture could also offer further performance. As an example, fixing the automatic maximum real time factor while training offers an estimated performance gain of at least 20%. The scenario should also be simplified by providing the agent with driving assistance systems such as an anti-lock braking system or traction control.

Based on a higher computational power, the following can then be investigated:

- Hyperparameter optimazation e.g. discount factor, update step size, higher batch size etc.
- Training of a generalizing agent on various vehicles and routes
- Other rewards e.g. driving efficient, following a defined path

Independent of the computing power, the control behavior and lap times of the agent can also be further optimized by adjusting the state and action space:

- Adding important features to the state space
- Identify unnecessery features of the state space
- Changing the TD(0) error to a TD(n) error by editing the SAC agent of TF-Agents [6]

As a further outlook on what is potentially possible and specifically as motivation for follow-up work, the following key points are provided:

- Creation of a fully generalizing agent that can handle all types of vehicles on all race tracks
- Integration of the SAC algorithm into the CM GUI with selectable UAQs for action and state space, as well as a own reward function, in order to be able to carry out optimizations as an end user.

8 References

[1] Kaufmann, E., Loquercio, A., Ranftl, R., Müller, M., Koltun, V., Scaramuzza, D.: 'Deep Drone Acrobatics', *Robotics*

[2] 'Computer bluffen nicht: Künstliche Intelligenz', https://www.faz.net/aktuell/wissen/physik-mehr/forschung-spielende-computerprogramme-14440369.html?printPagedArticle=true#pageIndex_2, accessed November, 2020

[3] 'Maschine schlägt Mensch - im Curling: Künstliche Intelligenz', https://www.spiegel.de/wissenschaft/curling-roboter-schlaegt-athleten-wenn-automaten-die-besseren-sportler-werden-a-00000000-0002-0001-0000-000173324649, accessed November, 2020

[4] Silver, D., Huang, A., Maddison, C.J.*, et al.*: 'Mastering the game of Go with deep neural networks and tree search', *Nature*, 2016, **529**, (7587), pp. 484–489

[5] Berner, C., Brockman, G., Chan, B.*, et al.*: 'Dota 2 with Large Scale Deep Reinforcement Learning' (13.12.2019)

[6] Fuchs, F., Song, Y., Kaufmann, E., Scaramuzza, D., Duerr, P.: 'Super-Human Performance in Gran Turismo Sport Using Deep Reinforcement Learning' (18.08.2020)

[7] Carlucci Aiello, L.: 'The multifaceted impact of Ada Lovelace in the digital age', *Artificial Intelligence*, 2016, **235**, pp. 58–62

[8] Bengio, Y., Goodfellow, I., Courville, A.: 'Deep learning' (MIT Press, Massachusetts, 2017)

[9] 'Playground' (TensorFlow)

[10] 'What is reinforcement learning? The complete guide', https://deepsense.ai/what-is-reinforcement-learning-the-complete-guide/, accessed February, 2021

[11] Sutton, R.S., Barto, A.: 'Reinforcement learning: An introduction' (The MIT Press, Cambridge, MA, London, 2018)

[12] Haarnoja, T., Tang, H., Abbeel, P., Levine, S.: 'Reinforcement Learning with Deep Energy-Based Policies' (27.02.2017)

[13] Haarnoja, T., Zhou, A., Abbeel, P., Levine, S.: 'Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor' (04.01.2018)

[14]Haarnoja, T., Zhou, A., Hartikainen, K.*, et al.*: 'Soft Actor-Critic Algorithms and Applications' (13.12.2018)

[15]'An intro to Advantage Actor Critic methods: let's play Sonic the Hedgehog!', https://www.freecodecamp.org/news/an-intro-to-advantage-actor-critic-methods-lets-play-sonic-the-hedgehog-86d6240171d/, accessed January, 2021

[16]Lillicrap, T.P., Hunt, J.J., Pritzel, A.*, et al.*: 'Continuous control with deep reinforcement learning' (10.09.2015)

[17]'SAC minitaur with the Actor-Learner API', https://www.tensorflow.org/agents/tutorials/7_SAC_minitaur_tutorial

Appendix

1. GitHub Repository:
   https://github.com/Andreas-aha/CM_RL_Driver

2. Jupyter Notebook:
   https://github.com/Andreas-aha/CM_RL_Driver/blob/main/RL/CarMaker_RL.ipynb

3. Raw project directory with last experiment incl. all data storage files:
   https://nx11340.your-storageshare.de/s/iasYKnz5PNiTJ9o
   Password: cQaZ32Pj