HAW HAMBURG

MASTER THESIS
Daniel Leonid Riege

Real-World Reinforcement Learning for Bridging Sim-to-Real Gap in Miniature Autonomy

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Daniel Leonid Riege

Real-World Reinforcement Learning for Bridging Sim-to-Real Gap in Miniature Autonomy

Master thesis submitted for examination in Master´s degree
in the study course *Master of Science Informatik*
at the Department Computer Science
at the Faculty of Engineering and Computer Science
at University of Applied Science Hamburg

Supervisor: Prof. Dr. Stephan Pareigis
Supervisor: Prof. Dr. Tim Tiedemann

Submitted on: May 16th 2024

Daniel Leonid Riege

Thema der Arbeit

Real-World Reinforcement Learning zur Überbrückung des Sim-to-Real Gap in der Miniaturautonomie

Stichworte

Reinforcement Learning, Digitaler Zwilling, Autonomes Fahren

Kurzzusammenfassung

In dieser Arbeit wird ein Reinforcement-Learning-System im Maßstab 1:87 vorgestellt. Dies umfasst einen digitalen Zwilling mit einer vollständigen Simulation sowie ein tatsächliches, im Maßstab 1:87 skaliertes Auto, das mit einer Kamera, einem Servo und einem Motor ausgestattet ist. Verschiedene Experimente wurden durchgeführt, um die Fähigkeiten der gesamten Gym-Umgebung und einer Reinforcement-Learning-Policy zu testen, die versucht, im realen Umfeld autonom zu fahren, indem Erfahrungen aus der Simulation genutzt werden. Ziel ist es, den Sim-to-Real-Gap zu überbrücken, indem das Training in der realen Welt fortgeführt wird. Die Ergebnisse zeigen, dass die Reinforcement-Learning-Policy das Auto in der Simulation steuern kann, die Anwendung in der realen Welt jedoch noch weiterer Forschung bedarf. Durch die Verwendung eines Encoder-Actor-Setups konnte jedoch der Sim-to-Real-Gap für einen supervised gelernten Actor überbrückt werden. ...

Daniel Leonid Riege

Title of Thesis

Real-World Reinforcement Learning for Bridging Sim-to-Real Gap in Miniature Autonomy

Keywords

Reinforcement Learning, Digital Twin, Autonomous Driving

Abstract

A 1:87 real-world reinforcement learning system is presented in the scope of this thesis. This includes a digital twin with a full simulation and a real 1:87 scaled car, equipped with a camera, servo and motor. Different experiments were conducted to test the capabilities of the whole gym environment and a reinforcement learning policy, trying to drive autonomously in the real-world by using experience from the simulation. Ultimetaly to bridge the sim-to-real gap by extending the training into the real-world. Results show that while the reinforcement learning policy is able to drive the car in the simulation, the performance in the real-world needs further research. Using an encoder-actor setup, the sim-to-real gap could however be bridged for a supervised learned actor. ...

Contents

List of Figures

List of Tables

1 Introduction

Miniature Autonomy is a research project at the University of Applied Sciences Hamburg (HAW Hamburg) which develops and analyzes different methods and algorithms in the field of autonomous driving. The miniature scale (1:87) allows for a controlled environment where different scenarios can be tested and evaluated without the risks and costs associated with real-world experiments [31, 30]. The cooperation with the Miniatur Wunderland Hamburg allows to work with a large and detailed environment additionaly to the one at HAW Hamburg. Especially the Knuffingen section allows for testing in different scenarios such as complex intersections, urban environment, rural mountain roads as well as highways with construction sites. The project is part of the research group "Autosys", which stands for autonomos systems, at the Department of Computer Science.

1.1 Motivation

Before deploying autonomous driving systems in to the real world, they need to be tested and evaluated in a simulated environment. The sim-to-real gap describes the difference in performance between a model trained in a simulation and the same model deployed in the real world. This gap is caused by the differences in the environment, entropies in the system, and the sensors used to perceive the environment. Not all of these differences can be simulated accurately, which leads to a decrease in performance. To bridge this gap, the model needs to be trained in a way that it can generalize to unseen environments and adapt to the real-world conditions with its uncertainties.

Especially when training reinforcement learning models, the sim-to-real gap is a common problem [12]. Reinforcement learning is a machine learning paradigm where an agent learns to interact with an environment by taking actions to maximize a reward signal. The agent learns a policy that maps observations to actions by exploring the environment

and learning from the rewards it receives. In a simulated environment, the agent can learn a policy that performs well in the simulation but fails to generalize to the real world, which is often seen [2, 20].

Bridging this sim-to-real gap is the main motivation and problem statement of this thesis. The general idea is to train a reinforcement learning model in a simulated environment and transfer it to the real world, where it can continue the learning process in a real-world environment. This way the model does not have to train from beginning in the real world, but can use the knowledge it already gained in the simulation.

Using a 1:87 scaled environment has the advantage that an accurate tracking system can be installed easily and that the car can crash without causing any harm or damage. Especially during the training steps, with a lot of noise on the steering angle for exploration, this can occur quite often.

To have the model use the experience from the simulation in the real world, when continuing the learning process, the whole system should by design already have a reduced sim-to-real gap. Otherwise the problem could arise, that the simulation experience cannot be used in the real world at all and the model has to start with random actions again.

1.2 Objectives

To overcome these challenges and to bridge the sim-to-real gap, a digital twin of the environment is developed. This digital twin, designed as a farama gymnasium environment, also embodies a simulation of the miniature car used. The simulation is used to train a reinforcement learning model in the early stages that can be transferred to the real world. This model uses a sort of perception images as input and predicts raw steering angles for the car. This is done in an end-to-end like approach. A key aspect of the autonmous driving scenario is the handling of intersections by providing the model with a maneuver which direction to take when there are multiple options. Possible maneuvers are going straight, turning left or turning right. The choosing of the maneuver and therefore navigation from point A to point B is not scope of this thesis, but rather the execution of the maneuver. The car will also indicate the chosen maneuver by using blinkers.

To train in the real-world the simulation is used as a digital twin with the simulated car being replaced with the real counterpart. This counterpart, the tinycar, is also developed

in the scope of this project. It is a 1:87 scaled car that can be controlled by a computer and is equipped with a camera to perceive the environment. Using a steering servo and a motor, the car can be controlled to drive. LEDs such as blinkers are used to signal the maneuver the model has chosen. The digital twin is used to evaluate the actions based on a ground truth from the gym environment. The car is tracked by an overhead camera tracking system that provides the position and orientation of the real car in the environment. The tracking system is also developed in the scope of this project.After the termination of an episode, e.g. when the car leaves the track or collides with an obstacle, the car is automatically repositioned to a suitable starting position.

The objective for the trained reinforcement learning policy is ultimetaly to drive the real car autonomously on the track. The policy should be able to generalize to the real world and adapt to the real-world conditions. The policy should be able to drive the car on the track without colliding with obstacles and without leaving the track. The policy should be able to handle the intersections by executing the chosen maneuver.

1.3 Structure

This thesis first introduces the background of reinforcement learning and the different methods which could be used. Followed by this, design choices for the observation space, action space and reward shaping are discussed as well as the structure of the neural nets and the overall training process. The following implementation chapter describes the hardware setup of the tinycar, the network protocol used to communicate with the car, the software interface to control the car and the gymnasium environment with its simulation. The real-world environment, which is an option inside the developed gym environment, is described in detail, including the tracking system, the automatic repositioning of the car and the camera preprocessing. After that results of different experiments of the environment and with the trained models are presented. The thesis concludes with a discussion of the results and an outlook on future work.

2 Related Work

2.1 Steering Control with Lane Detection

Traditonal Approaches for controling a cars steering and throttle are based on controllers like PID controllers [11]. In order to feed the controllers with enough information like cross track error, heading error, the car needs to perceive the environment and extract the relevant information. For lateral control, this is often done by using a camera to detect the lane markings and plan a lane path accordingly [21]. The lane markings can be detected by using classical computer vision techniques to fit polylines onto the lane markings [4, 15]. These approaches however, are often limited to highway scenarios which have flatter curve radiuses. Other approaches to detect the lane markings are based on deep learning techniques [22, 36, 27]. This thesis will use such a lane segmentation model to detect lane markings, but will not use a traditional controller such as Pure Pursuit [18] or Stanley [14].

2.2 Imitation Learning End-to-End Systems

Another approach for lateral control is the use of end-to-end trained neural networks. These networks take an image as input and output the steering angle directly. ALVINN [24] in 1989 was one of the first approaches to use neural networks for steering control. However, it can only drive in very simple scenarios with few obstacles as the neural network layers were limited. Even though, it demonstrated the potential for an end-to-end neural network navigation a vehicle on the road. The approach was further developed by NVIDIA in 2016 with the PilotNet [7]. This approach was able to drive a car autonomously on the road and was the first to use a convolutional neural network (CNN) for steering control. Further developed by Comma.ai with the OpenPilot software [3], it shows that it can reliably apply to commerical use in real world traffic and compete

with other L2 autonomous driving systems [25]. The main advantage of this approach is that it does not require a lot of feature engineering and can be trained end-to-end. The disadvantage is that the model is a black box and it is hard to understand why the model makes certain decisions [23, 8].

Such a end-to-end approach with an CNN architecture heavily inspired by the PilotNet [7] will be used in this thesis for pre-training the encoder. The idea is heavily inspired by the work of Zou et al. [35], where they also train a CNN image decoder via supervised learning and use that to train an reinforcement learning actor with the feature vector as input.

End-to-End trained steering controller networks can go further than simply holding the car in the center of a lane. Research at MIT has shown that these networks are also capable of navigating intersections based on context information of the environment [5]. Their approach is capable of steering the car through an intersection and choosing the correct maneuver based on a rendered part of the map with navigation information embedded. OpenPilot now uses the same idea to include navigation features into their L2 autonomous driving system [1]. A similar approach is done by the authors of [10] where three different sub-networks are trained. Each responsible for a different maneuver (keep straight, turn left, turn right). The sub-network in control is chosen prior by another system. This thesis will incorporate the navigational idea in end-to-end networks. Instead of using seperate networks, each for a maneuver, a single network architecture is designed to predict the steering angle. Just as in [10], the decision on the maneuver is known prior by another system and is not part of this thesis. Using this approach, compared to [1, 5], has the advantage that the car could theoretically navigate intersection without the need of a map or position information.

2.3 Reinforcement Learning for Autonomous Driving

Based on the idea of an end-to-end trained steering controller, reinforcement learning can be used to train these networks instead of supervised learning [19, 29, 26, 34, 32, 33]. A problem often occuring in reinforcement learning is catastrophic forgetting. This is when a model forgets how to perform a task it has learned before, when learning a new task [9]. A common approach to mitigate this problem is to use a replay buffer. This will also be used in this thesis and since the input of the model is a feature vector instead of

raw pixels, the replay buffer can store much more data. Other approaches use machine learning techniques to sample data from replay buffers more efficient [17, 6].

Another common problem is the sim-to-real gap, where an agent performs well in a simulated environment but fails to generalize to the real world due to many entropies [12, 20]. In Allamaa et al. [2] they use a executable digital twin to update a nonlinear model predictive controller (NMPC) in the real world. The digital twin is trained in simulation and then used to update the NMPC in the real world. While the car is driving, the digital twin randomizes certain parameters and updates the weights of the NMPC. While this approach also uses a digital twin, as this thesis does, the approach is different. In this thesis the car actually faced various entropies in the real world and adapts to them. The digital twin is used to pre-train the model to achieve a better starting point in the real world.

3 Methodology

This chapter describes the methodology used to train the model. Especially the reinforcement learning algorithm used, the environment and simulation design, and the training process are described in detail.

3.1 Reinforcement Learning

In Reinforcement Learning (RL) an agent learns to interact with an environment by taking actions to maximize a reward signal. The agent learns a policy that maps observations to actions by exploring the environment and learning from the rewards it receives. The goal of the agent is to maximize the cumulative reward it receives over time. The agent interacts with the environment in discrete time steps, where at each time step t the agent receives an observation o_t, takes an action a_t and receives a reward r_t. The agent learns a policy π that maps observations to actions by maximizing the expected cumulative reward $R_t = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$, where γ is the discount factor.

This section alerady assumes that the agent has no model or knowledge of the underlying environment. This is called model-free reinforcement learning. Depending on the environment and the task, different RL algorithms can be used to train the agent. The most common RL algorithms are Q-Learning, Deep Q-Learning, Proximal Policy Optimization (PPO), Soft Actor-Critic (SAC) and Deep Deterministic Policy Gradient (DDPG). These can be categorized into value-based methods, policy-based methods and actor-critic methods.

Value-Based methods learn a value function that estimates the expected cumulative reward of taking an action in a given state. The agent then chooses the action with the highest value. Q-Learning is a value-based method that learns the Q-Value of taking an action in a given state. Deep Q-Learning uses a neural network to approximate the Q-Value function.

Policy-Based methods learn a policy that maps observations to actions directly. The policy is optimized to maximize the expected cumulative reward. Proximal Policy Optimization (PPO) is a policy-based method that uses a clipped surrogate objective to update the policy.

Actor-Critic methods combine value-based and policy-based methods. The actor learns a policy that maps observations to actions, while the critic learns a value function that estimates the expected cumulative reward. The actor is updated to maximize the expected cumulative reward, while the critic is updated to minimize the error between the estimated value and the actual reward. Soft Actor-Critic (SAC) is an actor-critic method that uses a maximum entropy objective to encourage exploration. Compared to SAC, Deep Deterministic Policy Gradient (DDPG) is another actor-critic method that utilizes deterministic policies, simplifying decision-making but potentially limiting exploration. While SAC promotes exploration through entropy regularization, DDPG relies on exploration strategies like noise injection. Another variant of DDPG is Twin Delayed Deep Deterministic Policy Gradient (TD3) [13], which will be used in this project and is described in the following section.

3.1.1 Twin Delayed Deep Deterministic Policy Gradient

For controlling the car, especially the steering angle, Twin Delayed Deep Deterministic Policy Gradient (TD3) [13] is used. TD3 is an off-policy actor-critic algorithm that learns a deterministic policy. It is an extension of Deep Deterministic Policy Gradient (DDPG) that uses a pair of critics to estimate the Q-Value of taking an action in a given state. The critics are used to reduce overestimation bias and stabilize training. The policy is updated to maximize the expected cumulative reward, while the critics are updated to minimize the error between the estimated value and the actual reward.

The target for the Critic network is calculated as follows:

$$y_t = r_t + \gamma \min_{i=1,2} Q_{\text{target},i}(s_{t+1}, a_{t+1}) \tag{3.1}$$

where y_t is the target value, r_t is the reward at time step t, γ is the discount factor, $Q_{\text{target},i}$ is the target Q-value of the i-th critic network, s_{t+1} is the next state and a_{t+1} is the next action. The target value is calculated as the reward plus the discounted minimum target Q-Value of the next state and action. The critics are updated to minimize the error between the estimated Q-Value Q_{critic} and the target Q-Value Q_{target}.

The target for the Actor network is calculated as follows:

$$J(\theta) = -Q_{\text{critic},1}(s_i, \pi(s_i)) \tag{3.2}$$

where $J(\theta)$ is the objective function, $Q_{\text{critic},1}$ is the Q-Value of the first critic network, s_i is the state and $\pi(s_i)$ is the action chosen by the Actor network. The actor is updated to maximize the Q-Value of the first critic network.

TD3 uses a target policy smoothing and target Q-value clipping to improve stability and performance. The target policy smoothing adds noise to the target policy to prevent the policy from collapsing to a single action. The target Q-value clipping clips the target Q-Value to reduce overestimation bias. TD3 also uses delayed policy updates to improve stability and performance. The policy is updated less frequently than the critics, which reduces the variance of the policy updates. Usually after every second step. This makes the training more stable and efficient.

3.1.2 Exploration Noise

To encourage exploration, noise is added to the action chosen by the policy. The noise used is an Ornstein-Uhlenbeck process, which is a stochastic process that generates temporally correlated noise. The noise is added to the action to encourage exploration and prevent the policy from collapsing to a single action. The noise is calculated as follows:

$$x_t = x_{t-1} \cdot \theta \cdot (\mu - x_{t-1}) + \sigma \cdot \text{random} \tag{3.3}$$

where x_t is the noise at time step t, x_{t-1} is the noise at the previous time step, θ is the decay rate, μ is the mean, σ is the standard deviation and random is a random number drawn from a normal distribution. The noise is added to the action chosen by the policy to encourage exploration.

3.2 Environment and Simulation Design

For the first training steps of the agent, a simulated environment is used. This simulation is a digital twin of the real environment used, so observation space, action space, reward signal and other information used, are the same and defined in the following.

3.2.1 Observation Space

The oberservation space for the agent is like in most end-to-end approaches a camera frame. But instead of using the raw camera frame, the frame is preprocessed by a lane segmentation model. This segmentation model takes the raw camera frame and detects all the lane lines and road edges and outputs a binary image for each class. So if the model detects road edges, dashed lines, solid lines as individual classes, three binary images are output. These binary segmentation images are the oberservation space for the agent. Figure 3.1 shows an example of the observation space but rendered as an RGB image, where each class gets a certain color. This format for the oberservation space is chosen, because it minimzes by design the sim-to-real gap. Simulating raw camera frames needs a lot of computational power and a hyper realistic simulation. By using a segmentation model, the simulation can be simplified since only lanelines have to be rendered. The segmentation model used for the real camera frames however, has to be very accurate so there are no differences between the simulated and real observation space. So the oberservation space is defined as:

$$s_t = \{0, 1\}^{C \times W \times H} \tag{3.4}$$

with C being the number of classes (lane line types), W being the width and H being the height of the camera frame.

However, the trained agent should not only be able to keep itself centered in its lane, but also be able to execute a certain maneuver whenever possible. These maneuvers are going straight, turning left or turning right, which are meant to be executed only at intersections. This one hot encoded maneuver vector (each index represents a maneuver) is also part of the agents input. But it is not provided as oberservation space by the environment. Instead is it given from the user or an external navigation system. This changes the policy to be defined as:

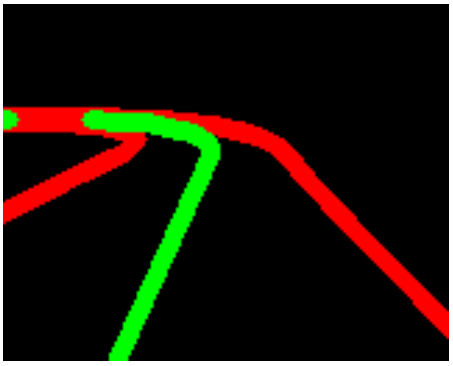


Figure 3.1: Rendered observation space of the simulated environment. The observation space itself would be in CxWxH format with C being the classes of the lane segmentation models output. In this example there are two classes visible. Road edges (here in red) and dashed lines (here in green). The image shown is the rendered RGB version of this observation space for better visualization.

$$\pi : (s_t, m_t) \rightarrow a_t \tag{3.5}$$

with s_t being the observation space given by the environment, m_t being the one hot encoded maneuver vector and a_t being the action chosen by the policy.

The general critic network defined in section 3.1.1, which is used to estimate the Q-Value of taking an action in a given state, would then be defined as:

$$Q_{\text{critic}} : (s_t, m_t, a_t) \rightarrow Q_t \tag{3.6}$$

with Q_t being the Q-Value of taking action a_t in state s_t with maneuver m_t.

3.2.2 Action Space

The action space for the agent is the steering angle of the car. The throttle is set to a constant value, so the agent only has to control the steering angle. This simplifies the action space and the training process. Since the observation space does not include any information necessary for longitudinal control, the agent would nonetheless use a constant throttle value. The steering angle is a continuous value between -1 and 1, where -1 is the maximum left steering angle and 1 is the maximum right steering angle. The actual

steering angle is calculated by multiplying the action value by the maximum steering angle of the car. For the simulated environment this has to be set to the same value as the real car, so the agent can learn to control the car in the simulation and transfer the learned policy to the real world. So the action space is defined as:

$$a_t \in [-1, 1] \tag{3.7}$$

3.2.3 Additional Information Calculated

For the reward signal and the evaluation process of the agent, additional information is calculated from the environment after every step. This information is the cross track error (CTE), the heading error and the current position, which will be used to render the driven trajectory. The CTE is the perpendicular distance between the car and the center of the lane. The center of the lane is the ground truth path which is manually labeled into the map. The heading error is the difference between the current heading of the car and the heading of the ground truth path. The current position is the position of the car in the environment based on the center of the rear axle. Both the CTE and position is given in meters and the heading error in radians.

3.2.4 Reward Signal

The reward signal used is soley based on the absolute value of the CTE, so direction does not matter. Based on previous experiments, a linear reward shaping gives the best results. Therefore the reward signal for this environment is defined as:

$$r_t = \max(-33 \times \text{abs}(CTE) + 1, -1) \tag{3.8}$$

This reward signal is used to encourage the agent to stay centered in its lane. The reward signal is clipped to the range of -1 to 1 to prevent the agent from receiving rewards that are too large or too small. The -33 is calculated by using the negative of the maximum reward 1 divided by the maximum CTE which should still give a positive reward, which is defined as 0.03 meters (for the 1:87 scaled environment).

3.2.5 Episode Termination

An episode is terminated if the car leaves the track, collides with an obstacle or the maximum number of steps is reached. The maximum number of steps is set to 1000. The car leaving the track is defined as the CTE being larger than 0.1 meters. The collision with an obstacle is defined as the car not moving for more than 10 steps, even though a velocity is set. For the real-world environment the eipsode reset will perform an automatic repositioning of the car to the nearest position within the lane. For the simulation, a random but defined spawn point will be used.

3.3 Model Architecture

As in [35], the system is trained in two steps. First, the encoder is pretrained using Imitation Learning (IL) and then the Actor and Critic network are trained using TD3. The encoder is pretrained to learn a feature representation of the observation space that can be used for the Actor and Critic network. For this, training data using a human driver and a stanley controller in the simulation is collected, by saving the observation space (lane segmentation image), chosen maneuver and action. The imitation learning network, as shown in Figure 3.2, is then trained to predict the action given the observation space and maneuver. This way the encoder indirectly learns a feature vector representation after the convolutional layers. This encoder from the IL network is then cut off and used for the Actor and Critic network and not updated during the TD3 learning process.

This has the advantage that the convolutional layers, which take the most processing time in neural networks, do not have to be backpropagated, speeding up the learning time. Especially when considering that training is also done in a real-world environment, where training should happen in real-time. Another benefit is that the encoder could potentially minimize the sim-to-real gap by design. The encoder compressed the highly dimensional observation space into a vector of 256 values. It will be evaluated, by first training the IL network completly in the simulation, to have an encoder which can compress the observation space from the simulation. And then training another model of the IL network using only real data, collected by a human driver, and using the weights of the fully connected layer, which come after the feature vector, of the prvious simulation-only model. This way the real data encoder should be able to learn a similar feature representation as the simulation-only encoder for similar oberservation spaces.

The outcome would be two encoders, one which will be used for the simulation and one for the real-world environment. The trained actors should then be able to generalize to the real world, since the feature representation is similar.

The Actor and Critic network are using a frame stack of the last 10 feature vectors as input. The actor part of the IL network only uses the current feature vector, simplifying this supervised learning step. Using a frame stack for the actor should have the benefit of giving the actor information about the past states and therefore the ability to predict smoother steering angles. Otherwise the steering behavior could potentially start to oscillate, which is a common problem in controllers that only use the current state. Similar to why the derivative of the error is used in PID controllers.

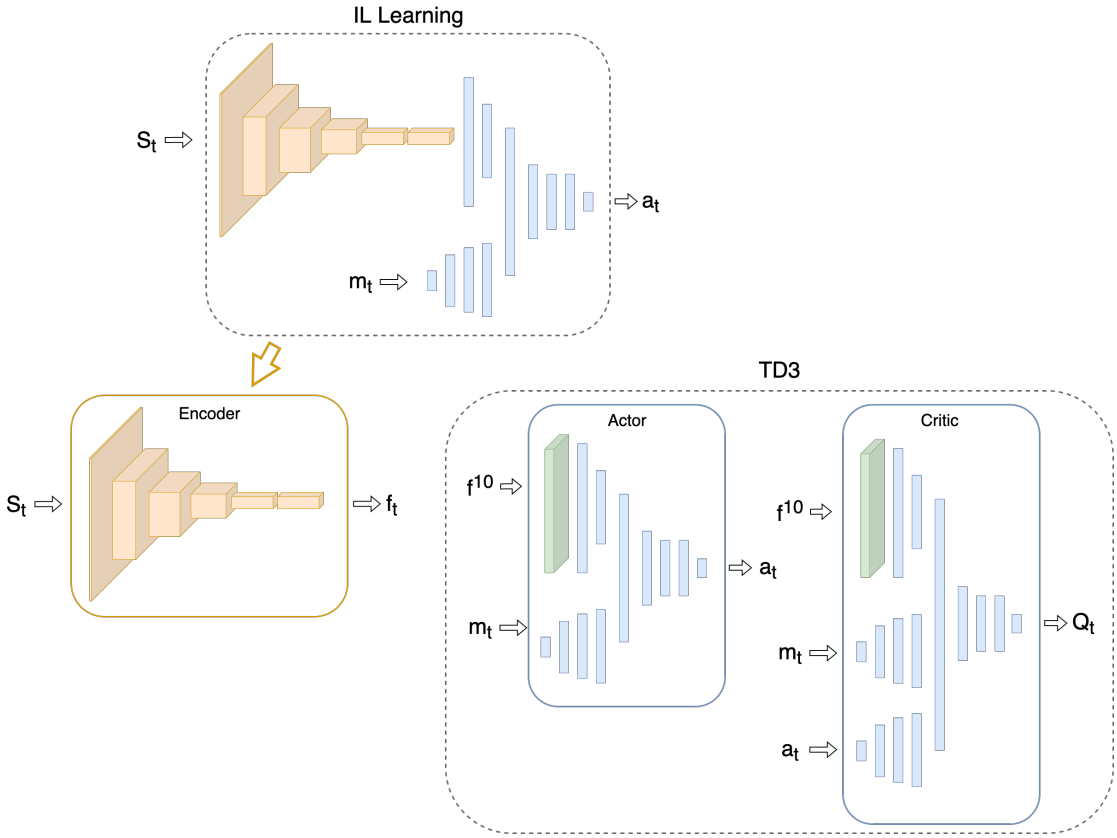


Figure 3.2: Overview of the model architectures used and the training process. The Imitation Learning (IL) network on the top is used to pretrain the encoder by using the observation space, maneuver and corresponding action, given by a human driver. The encoder is then cut out and used for the TD3 learning process on the bottom. Given the observation space, the encoder outputs a feature vector. The Actor and Critic network are using a frame stack of the last 10 feature vectors as input. These are then used to calculate the action and Q-Value. During TD3 learning, the encoder is frozen, hence only the Actor and Critic network are updated.

4 Implementation

In order to train a model in a simulation or in the real world, a simulation as well as a miniature car, which can be controlled by a computer, is needed. This chapter describes the implementation of the environemt as well as the technical setup of the tinycar.

The environment is written as a Farama Gymnasium Environment, which is a standard for reinforcement learning, previously known as OpenAI gyms. The gym environment which includes a map for the car, also includes a simulation of the car, which is used to train the reinforcement learning model in the first iterations. For that the car kinematics and camera images are simulated. When training in the real-world, the car and camera classes are replaced with control software for the tinycar. The control software is used to send commands to the car and receive camera data. The feedback on the position and orientation of the car is provided by a tracking system, which is also part of the real-world environment. The following sections describe these steps in detail.

4.1 Tinycar

The 1:87 scaled miniature car developed and used in this project, will be called tinycar. The tinycar consists of a steering servo, DC motor, LEDs for headlights, blinkers and taillights as well as a front facing camera. All is controlled by a custom designed PCB which utilizes an ESP32 microcontroller. The ESP32 is a low-cost, low-power microcontroller with integrated Wi-Fi capabilities. The ESP32 is used to control the car and to communicate with the computer running the control software. The ESP32 is powered by a LiPo battery which is charged via USB. When controlling the car during the real-world training, all the computation is done on the computer and the ESP32 only receives the commands and sends back the telemetry and camera data. The ESP32 is connected to the computer via Wi-Fi and the computer sends the commands to the car via a custom network protocol. Figure 4.1 shows the tinycar during a test drive. Even though network

Figure 4.1: The tinycar during a data collection test drive, remote controlled by a human, in the section of Knuffingen in the Miniatur Wunderland Hamburg. The car is equipped with a camera, LEDs for headlights, blinkers and taillights and is controlled by an ESP32 microcontroller. The cars body features a cyberpunk-inspired design, with only one headlight to break the symmetry.

latency is a big factor with this setup, the gain in computational power outweighs the latency [28].

4.1.1 Hardware Setup

The core concept of this vehicle centers on streaming camera data to an external, more powerful computer for processing, a difference to other approaches that conduct computations onboard [16]. Opting for a compact and efficient ESP32 over a Raspberry Pi Compute Module avoids increasing the cars size and energy demands. The ESP32 S3 model is chosen for its additional GPIOs and RAM, maintaining power consumption below 500 mA — even with active servo, motor, and LED lights — and enabling the use of a small 400 mAh LiPo battery. This battery allows for at least 48 minutes of operation. The custom designed PCB, designed to fit essential components like the DC Motor driver, LiPo battery charger, and camera interface, measures 22 x 45 mm. For enhanced usability, a USB-C port is incorporated at the vehicles rear, facilitating firmware updates, debugging, and battery charging via a TP4056 chip. The charging circuit includes a load-sharing mode as a safety feature to prevent overcharging of the LiPo battery, with two LEDs indicating the batterys status. When power is drawn through the USB-C port, the battery is disconnected from the circuit, ensuring accurate charge measurement for the TP4056 chip to terminate the charging process.

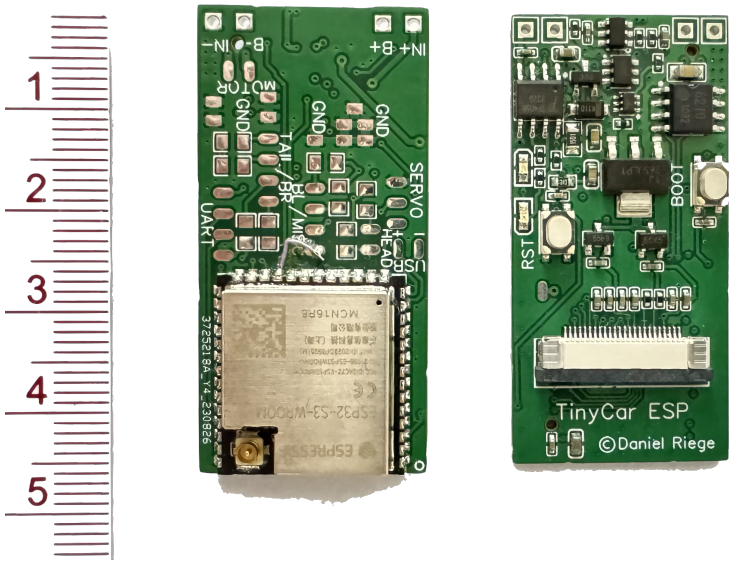


Figure 4.2: The custom designed PCB for the tinycar. The PCB features an ESP32 microcontroller, a motor driver, a servo connector, a camera connector, a LiPo battery charging and measuring circuit and a USB interface. The LEDs can directly be connected to the PCB, since the board also features resistors for the LEDs.

The motor driver, a BD6210F-E2 H-Bridge IC, offers up to 500 mA being drawn, sufficient for the vehicles 4.2 V DC motor. It supports bidirectional control and speed variation via PWM signals. Even though the trained model only drives forward, the later discuseed repositioning needs a reverse mode for the car. Despite components like the DC motor and servo performing best at 5V, the LiPo battery output varies between 4.2V when fully charged and 3.2V when nearly depleted. To circumvent the limitations of a 3.3V LDO voltage regulator, the motor and servo are directly connected to the battery, prioritizing power over consistent performance, as the regulated 3.3V would restrict maneuverability.

The dimensions of the car are kept within the range of a 1:87 scaled truck, with a wheelbase of 48.7 mm and a width of 32 mm. Figure 4.3 shows the blueprint of the car with the dimensions. The camera is angled at 75 degrees, making 70 % of the camera frame useful for lane detection. With an FOV 120 degrees, the car is able to look into corners, even though left turns might be harder to see, as discussed later.

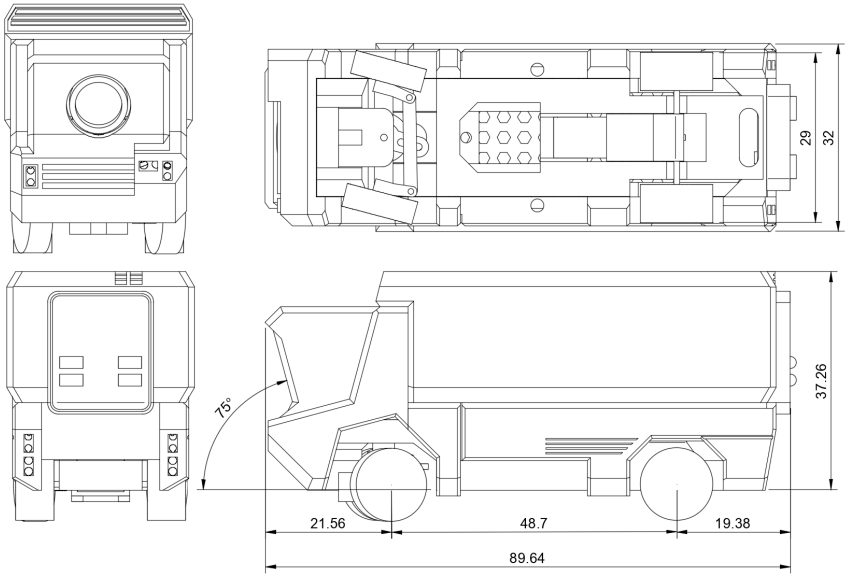


Figure 4.3: Full Overview of the tinycar with front perspective (top left), bottom view (top right), rear view (bottom left) and left side view (bottom right). All measurements are in mm. The front-mounted camera, positioned at a 75-degree angle, optimizes lane visibility and extends the lateral view with a 120 degree field of view. The car's dimensions are based on a 1:87 scaled truck, with a wheelbase of 48.7 mm and a width of 32 mm.

4.1.2 Network Protocol

As stated above, the car allows to be controlled via WiFi. This includes controlling the cars servo, motor and LED state as well as receiving camera frames and telemetry data, such as battery voltage or WiFi RSSI. In order to minimize the latency between capturing a camera frame and receiving it on the host computer, a custom UDP protocol is implemented. It consits of two different types.

The Tinycar Frame Protocol (TCFP) is developed for streaming camera images, inspired by the Real-Time Protocol (RTP) as outlined in RFC 3550. RTP, designed to accommodate various codecs and network clients, introduces significant overhead due to its versatility, which results in many of its bytes being unused in this application. A critical limitation of RTP for this context is its lack of packet numbering within a frame. Given that both TCFP and RTP utilize UDP for real-time communication, camera frames must be segmented into multiple packets to avoid exceeding the maximum datagram size, necessitating reassembly at the recipients end. RTPs structure includes a sequence number that increments with each packet, disregarding frame boundaries, and a fragmentation offset for reassembling frames, but it does not specify the total frame size or the exact number of packets required for complete frame reassembly. TCFP streamlines this process by eliminating unnecessary information, such as source identifiers, and by modifying the sequence number to increment per frame rather than per packet. It introduces a packet number within each frame sequence, starting from zero for the first packet, and includes the total number of packets needed for a frame. This allows the receiving end to easily identify missing packets, thereby facilitating frame integrity without the need for separate buffering and reordering processes. Consequently, TCFP reduces memory copy operations and latency by enabling direct packet placement into the buffer based on frame number and packet sequence, thereby simplifying the reordering process and enhancing efficiency.

The Tinycar Control Protocol (TCCP) is designed for controlling the car and receiving telemetry data. It is a simple protocol with a fixed size struct. The message contains the packet type and the actual data, such as the servo angle, motor speed, LED state, battery voltage or WiFi RSSI. The lack of a sequence number means that missing packets cannot be identified. However, as for TCFP, missing packets are not resent and the protocol is designed to be lossy, following the fire-and-forget principle. When starting the car, the host has to send a control message first, so that the car knows the IP address to send back telemetry or TCFP messages.

4.1.3 Software Interface

The host library to interact with the tinycar, is written in C++ to make the frame packet reassembly as efficient as possible. However, a python wrapper is provided for easier integration into a Python RL gymnasium environment. Each possible command, like setting a motor duty cycle or setting the blinker left, is a individual method call. Since the network protocol follows the fire-and-forget principle, no acknowledgment is sent back, making the command methods return as soon as the message leaves the transmit buffer.

4.2 Simulation

In order to enhance training time in the first steps of the reinforcement learning model, a simulation of the tinycar is used. The simulation features two main parts, the kinematics of the car and the camera resp. the lane segmentation simulation. Both of these components need a map to drive on and to render the camera image from. This projects uses two maps, both being a 1:1 replica of the environment at HAW Hamburg and a part of Knuffingen in the Miniatur Wunderland Hamburg. Figure 4.5 shows the map of Knuffingen. The maps are annotated by hand using a reference image, using a custom mapbuilder. The mapbuilder allows to draw the lane lines, as detected by the lane segmentation model, onto the reference image as well as a ground truth lane path which is the center of a lane. The mapbuilder then generates a map file which can be used by the simulation to render the camera image and to calculate the position and rotation of the car after a step. The map file is a json file which contains the nodes and edges for each lane line and the ground truth trajectory. The represented graph for the ground truth is handeled as a directional graph, important for the local path calculation.

4.2.1 Kinematics

The RL algorithm only controls the steering angle and the speed is kept constant at a speed of 0.5 m/s. Therefore, dynamics can be neglected and only the kinematics need to be simulated. The simulated vehicle is a bycicle model. This means that the vehicle has one steerable axle at the front. The range of movement in the longitudinal and lateral direction is restricted by this steering. So the steering angle can be used to determine

Figure 4.4: Example image of the mapbuilder tool. The tool is used to annotate the map of the environment. The mapbuilder generates a json file with graphs for each lane line and the ground truth trajectory. The lines are polylines, meaning nodes are connected by straight lines with no curvature. The reference image is only used for the tool and not part of the later generated map. The tool allows that paths can be split, needed for intersections on the ground truth trajectory.

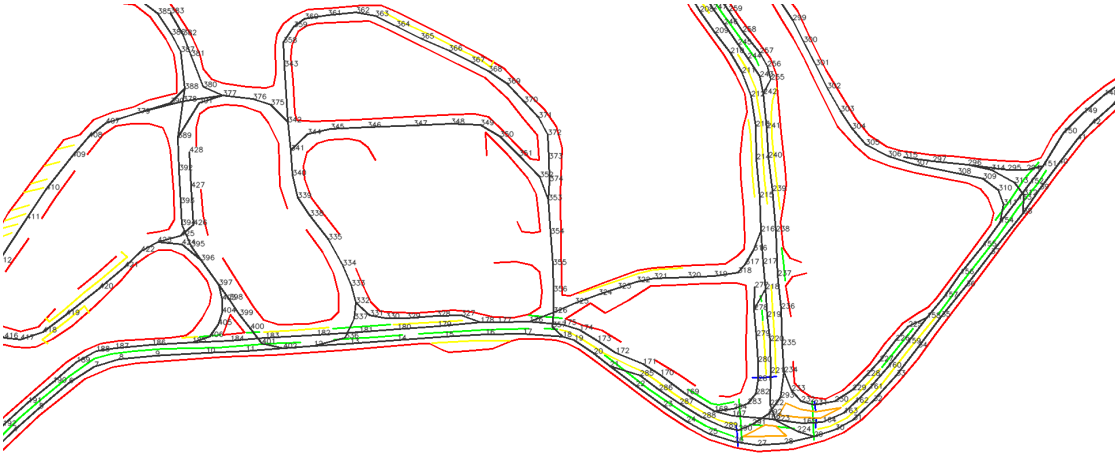


Figure 4.5: Part of Knuffingen in the Miniatur Wunderland Hamburg as a rendered map for the simulation and real-world environment. The map is annotated by hand using a reference image and a custom mapbuilder. The mapbuilder generates a json file with graphs for each lane line and the ground truth trajectory. The lane lines are not 100 % accurate, since they are drawn as polylines and therefore are not perfectly round. The numbers on the map are the node ids for the ground truth trajectory. These are needed to define certain spawn points when training the RL algorithm in the simulation.

the curve radius at the current timestep. Based on this, a linear transformation matrix can be defined to update the position and orientation of the car after a timestep. This transformation matrix conducts a translation to the center of the current curve, which is calculated by using the normalvector and curve radius, a rotation, given by the steering angle and current velocity, and a translation back to the outer curve. The transformation matrix is then applied to the current position of the car to get the new position and orientation.

4.2.2 Ground Truth Reference Path

After each simulation step (performing a position and orientation update for the car), the local path is calculated. This local path is a list of three edges of the hand annotated ground truth trajectory, based on the current position and heading of the car. The local path is needed to calculate the CTE and heading error for the reward function and episode termination. It is updated by checking the distance to the next edge of the ground truth trajectory. If the distance is smaller than the old current edge, the next

edge is set as the current edge. This is done for all edges directly connected to the current edge. Based on this new edge, the trajectory is followed two more times to form the local path. If the local path goes through an intersection, the given maneuver defines which edge to choose. The local path is then used to calculate the CTE and heading error.

4.2.3 Camera Rendering

As described before, the simulated camera image is not a representation of the real camera image but the lane segmentation, which is the preprocessing step for the real camera in the gymnasium environment. The used lane segmentation model predicts 5 classes of lane lines: Outer road edge, dashed lines, solid lanes, blocking area and wait line. The base information for the camera rendering comes from the map and current position/orientation of the car. The map defines all lane lines for the respective class. Figure 4.5 shows a rendered version of the map. The position and orientation of the car in addition to the mounting position of the camera, relative to the car frame, define the extrinsic camera matrix. The FOV and resolution of the real-camera define the intrinsic camera matrix.

Based on the CAD model of the tinycar, the extrinsic matrix is defined with:

$$\mathrm{E} = \begin{bmatrix} 0.0 & -1.0 & 0.0 & -0.005 \\ 0.374 & 0.0 & 0.927 & -0.037 \\ -0.927 & 0.0 & 0.374 & -0.015 \end{bmatrix} \tag{4.1}$$

The intrinsic matrix, based on the lens specifications and the chosen resolution of 160 x 128, is defined with:

$$\mathrm{I} = \begin{bmatrix} 95.34 & 0 & 80 \\ 0 & 76.27 & 64 \\ 0 & 0 & 1 \end{bmatrix} \tag{4.2}$$

In order to project the lane line points into the camera frame, the points first need to be transformed from world coordinate system to camera coordinate system. This is done by converting the points into homogeneous coordinates and applying the extrinsic matrix by a matrix multiplication. Now the z-axis of the points is the distance to the camera, forming the Z buffer. The Z buffer is used to sort the points by their distance to the camera. The points behind the camera are discarded. The remaining points are then projected into the camera frame by applying the intrinsic matrix. The points are then

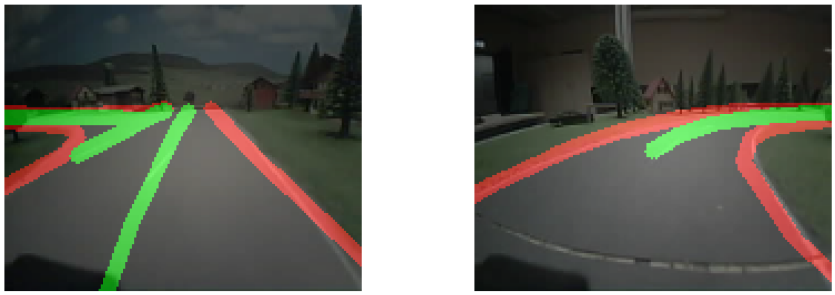


Figure 4.6: Two example images from the simulation's camera feed are overlayed onto the corresponding frames captured by the real camera of the tinycar. Given that the simulated map mirrors the real environment in a 1:1 scale, the car is positioned at two random locations, and its orientation and position is determined using the tracking system. Subsequently, the camera images are simulated for these positions. The simulated images are then overlayed with the actual camera feed to verify the accuracy of the simulation parameters. Small differences in the positioning of the lane lines can be seen.

drawn onto the camera image. For each lane line an individual image, ultimetaly forming a CxWxH tensor. The line thickeness in the drawing function is defined by the thickness of the lane lines of the segmentation model.

A special case which arises in this context, is the possibility that one end of the lane line is in front of the camera and the other is behind. This would result in a glitch, which does not project the lane line correctly. To avoid this, this case is detected and the intersection of the lane line edge with the camera plane is calculated. This results in a new edge with both ends in front of the camera.

Figure 3.1 shows an example of the camera rendering. Figure 4.6 shows an overlay of the rendered camera image at the same location as the real camera image.

4.3 Real-World Environment

The real-world environment is an option of the developed gymnasium environment. It uses the same map with the same local path calculation as the simulation but replaces the kinematics model and camera rendering with the real car and camera. The real-world environment consists of the tracking system, the automatic repositioning and the

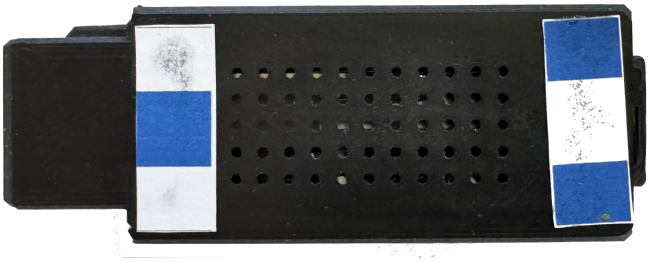


Figure 4.7: The two markers on the top of the tinycar, used to identify the car in the camera image. Only the white squares are used for the tracking. Since the front has two markers and the rear one, the orientation of the car can be calculated. The position of the car in the simulation is defined as the center of the rear axle, therefore its the center of the rear marker.

camera preprocessing. The camera preprocessing consits of only resizing the 5 lane segmentation images, after they are output from the lane segmentation model [27]. The tinycar provides with VGA resolution images, which are fed into this lane segmentation model. The model outputs 5 classes of lane lines, which are then resized to 160 x 128 and provided as oberservation space.

4.3.1 Tracking System

In the simulation the position and orientation of the car is calculated by the kinematics model based on the steering angle and velocity. In the real-world environment, the steering angle and velocity is sent to the car. In order to get a feedback on the position and orientation of the car after that step, a tracking system is needed. The tracking system consists of two cameras mounted over the track looking directly down, as seen in Figure 4.8. The cameras are calibrated so that the position and orientation of the car can be calculated by the position of the car in the camera image. This is done by undisorting the camera image and applying a correction factor, which maps the pixel position to the real-world position. The car is equipped with two different markers on the top, which are used to identify the car in the camera image. Figure 4.7 shows the markers on the car. The markers position is identified by filtering the image for a white color range and then applying a contour detection. All found contours are then filtered by their area and the distances between the contours, which has to be in a certain range. The position of the car is then calculated by the center of the rear marker. The orientation of the car is calculated by the angle between the front and rear marker.

Figure 4.8: Overview of the environment at HAW Hamburg with the two tracking cameras mounted over the track. Each camera captures one half of the track. The cameras are directly connected to a Raspberry Pi 4 via the Camera Serial Interface (CSI), which processes the camera image and calculates the position and orientation of the car. The Raspberry Pi 4 is connected to the host computer via Ethernet. Between the two cameras sits the WiFi access point used to communicate with the car underneath.

Each camera is directly connected to a Raspberry Pi 4, which is used to process the camera image and calculate the position and orientation of the car. The Raspberry Pi 4 is connected to the computer running the control software via Ethernet. The position and orientation are sent via a UDP message to the real-world environment software running on the host computer. Based on the ID, which identifies the two cameras and is also sent with the tracking data, the position is transformed into the world coordinate system. The transformation matrix used for that is determined by stitching the two camera images together manually. It is defined as:

$$M_2 = \begin{bmatrix} 1.01 & 0.0 & -21 \\ 0.0 & 1.01 & 410 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} \tag{4.3}$$

with M_2 being the transformation matrix for the second camera (left) relative to the first camera (right), as seen in Figure 4.8. The transformation matrix does a scaling and linear translation. Rotation is not needed since both cameras are mounted exactly in line to each other.

The cameras use a 640 x 480 pixel resolution with a frame rate of 30 fps. The resolution is sufficient to detect the car within 2.2 mm accuracy while keeping a low latency of 2 ms per frame. Using a higher resolution would increase the latency even though 2.2 mm accuracy is enough for stable control of the car, as seen in the results later.

4.3.2 Automatic Repositioning

The reset procedure in the simulated environment includes repositioning the car to a random spawn point. To prevent a manual repositioning in the real-world environment, an automatic repositioning procedure inside the reset method is implemented. The psuedocode for this procedure can be seen in Algorithm 1. The repositioning continuesly samples the nearest edge from the ground truth trajectory, based on the current position and orientation of the car. The end criteria for the repositioning is a distance of max 0.02 meters between the end node of the nearest edge and the current position and a heading error of max 0.34 radians. During an episode, every drive command is stored in a history list. This history stores the steering angle and velocity with a maximum of 30 entries. During the repositioning, if the history still contains entries, the last entry is popped and the car is driven in the opposite direction. If the history is empty, a local

path is calculated based on the nearest edge and the current position and orientation of the car. The CTE and heading error are calculated based on the local path. The Stanley control algorithm [14] is then used to calculate the steering angle. Based on that steering angle and a fixed velocity of 0.03 m/s, the car is driven until the end criteria are met. During this procedure the tinycar turns on its hazard blinkers for visual indication.

This procedure should ensure that if the car leaves the street and hits an obstacle, it can reposition itself by first always driving backwards until there is enough clearence to turn around and then driving forward to the nearest edge. It can happen that during the forward drive, the car gets stucks due to another obstacle, or that the history is too short to get enough clearence. This however, is evaluated in a later section.

Algorithm 1 Automatic Repositioning

Require: P and θ not None
  $P_e, \theta_e, E \leftarrow$ SampleNearestEdge(P, θ)
  while $\|PP_e\| > 0.02 \vee \text{abs}(\theta - \theta_e) > 0.34$ **do**
    if $\text{len}(H) > 0$ **then**
      $\alpha, v \leftarrow$ H.pop()
      drive$(\alpha, -v)$
    else
      findLocalPath(E)
      $\text{CTE}, \theta_{\text{error}} \leftarrow$ CalculateInfo(P, θ, E)
      $\alpha \leftarrow$ StanleyControl$(\text{CTE}, \theta_{\text{error}})$
      drive$(\alpha, 0.03)$
    end if
    $P_e, \theta_e, E \leftarrow$ SampleNearestEdge(P, θ)
  end while

5 Results

This chapter presents the results of the experiments conducted in the scope of this thesis. The experiments are divided into three parts. The first part is the presentation of the gymnasium environment with the tinycar performance especially the real-time capabilities and the tracking system. The second part is the presentation of the encoder. The last part is the presentation of the reinforcement learning trained actor in conjunction with the different encoders.

5.1 Environment

To make sure that there are no errors, which could affect the subsequent reinforcement learning, and the environment is suitable for the experiments, different tests are conducted. Besides the real-time capabilities of the tinycar together with the environment, the tracking system and the automatic repositioning with its stanley controller are presented.

5.1.1 Real-Time Aspects

The gymnasium environment requires certain real-time aspects to be met in order to do reinforcement learning in the real-world. Especially since the weights of the neural network are updated while the car still interacts with the environment. The only real-time operating system used in the whole system is the firmware of the tinycar, which uses FreeRTOS and therefore allows strict timing requirements. The tracking system, with its Raspberry Pis, uses Linux while the gymnasium environment is run on macOS. Both are consumer operating systems. So the timing diagrams will use worst case scenarios. Another big factor to measure is the latency through the network. Especially the time between capturing a frame on the tinycar and the receiving of that frame in the gym environment, which is transmitted over a wireless network.

Tinycar Frame Latency

Due to the lack of perfect clock synchronization between the host, running the gym environment, and the tinycar, a technique similar to a three-way handshake is employed for accurate time measurement. Following RFC 3550's guidance, both RTP and TCFP packets mark the start of a new frame with a timestamp, denoted as $t1$ in Figure 5.1. Upon receiving all packets of a frame, the host records its timestamp ($h1$). It then sends a Round-Trip Time (RTT) message via the TCCP protocol, to which, upon receipt, the tinycar logs the timestamp ($t2$) and replies with an RTT message including $t2$. When the host receives this, it sets $h2$. With these four timestamps the host can calculate the frame's latency.

Assuming the latency for an RTT messages is symmetric, the latency for a single RTT message (d_r) is calculated as:

$$d_r = \frac{dh}{2} = \frac{h2 - h1}{2} \tag{5.1}$$

Thus, the network frame latency is defined as:

$$d_f = dt - d_r = t2 - t1 - d_r \tag{5.2}$$

Table 5.1 shows the worst-case latencies for different camera resolutions. The jitter is the interarrival jitter as defined in RFC 3550. The compression and decompression times are the time it takes to compress and decompress the image using JPEG including the custom fragmentation and defragmentation process, as described in section 4.1.2. The compression is done on the ESP32, therefore taking longer compared to the decompression, done on the host computer. The network time is the frame latency as described above. The inference time is the time it takes to run the VGG lane segmentation model on the host, using pytorch with Metal backend. It is included to give the total latency to receive an observation for the gym environment.

The latency for a td3 learning step takes around 55.2 ms on the same Mac (worst case execution time). These are added to the tinycar latencies to give a total latency for a learning step of around 101 ms using the 320x160 resolution.

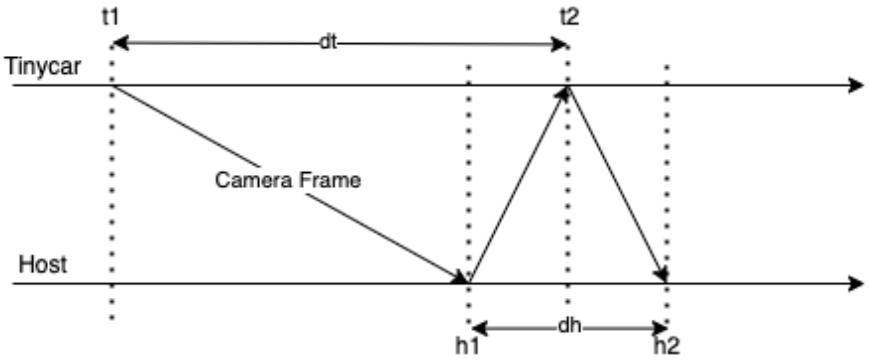


Figure 5.1: This illustration depicts the network latency measurement process for a camera frame transmitted from the tinycar to the Host. It shows the tinycar's timeline with t and $t2$, and the host's timeline with $h1$ and $h2$, representing key transmission and reception events. The durations dt and dh signify the camera frame transmission time and the RTT message round-trip time, respectively.

Resolution	Jitter [ms]	Compression [ms]	Network [ms]	Decompression [ms]	Inference [ms]	Total [ms]
1280 x 720	8.3	128.2	20.4	5.0	48.2	201.8
640 x 480	2.0	55.4	11.2	2.1	48.2	116.9
320 x 240	1.8	17.0	8.3	1.4	19.1	45.8
160 x 120	1.0	17.1	5.1	0.4	7.3	29.9

Table 5.1: Worst-case measurements over a period of 1 min. All measurements are in milliseconds. RSSI during measurements was around -35 dBm and the tinycar was 2 m in direct sight from WiFi Access Point. Inference was done using the VGG lane wegmentation model on a MacBook with M1 chip [27]. The inference time for resolutions of 1280 x 720 and 640 x 480 is identical, as both employ the same underlying model.

Resolution	Accuracy [mm]	Frame Processing [ms]	Network Latency [ms]	Total [ms]
1280 x 960	1.1	41.4	0.9	42.3
640 x 480	2.2	23.1	0.9	24.0
320 x 240	4.4	8.3	0.9	9.2

Table 5.2: Worst-case measurements over a period of 1 min. All time measurements are in milliseconds. The accuracy is the size of one pixel in millimeter, which depends on the resolution and distance between camera and ground. The Raspberry Pis for the tracking system and the host running the gym environment are connected via LAN cables. The tracking system uses OpenCV in C++ for processing the frames. The packets sent over the network are always 12 bytes, which includes the id for the camera, x and y position and the orientation.

Tracking System Latency

The tracking system latency is measured by the time it takes to process a frame from the overhead camera to the time it is available in the gym environment. Similar as above, the total latency consists of the time to process the frame using OpenCV and the network latency. Since the tracking data packet only uses 12 bytes, the network latency is measured by dividing the RTT time by 2. The camera frame consists of multiple packets, therefore the special measurement is required.

Table 5.2 shows the worst-case latencies for the tracking system for different resolutions of the overhead camera. The 12 bytes sent over the network includes the id for the camera, x and y position and the orientation. The frame processing time is the time it takes to process the frame using OpenCV in C++. The network latency is the time it takes to send the tracking data packet over the network. The total latency is the sum of the frame processing and network latency. The camera sends 30 frames per second. With the resolution of 1280 x 960, this means that the frame processing takes too long and only every second frame is processed. The accuracy, which is the size of one pixel in millimeter, is determined by using a 1 meter long ruler and counting the pixels for the meter in the image. This value depends on the resolution and the distance between the camera and the ground.

Location	False Positives [%]	Position Std Dev [mm]	Orientation Std Dev [deg]
1	2.6	0.250	0.721
2	1.9	0.536	1.180
3	1.7	0.778	0.901
4	3.4	0.538	1.182
5	30.2	0.689	1.231

Table 5.3: Susceptibility to errors for the tracking system. The tinycar is placed at five different locations and for each 1000 measurements are taken. The false positives are the number of measurements where the tracking system detects a marker which is not there. The mean and standard deviation are calculated for the position and orientation without the false positives. The position is in millimeter and the orientation in degrees.

5.1.2 Tracking System

For the tracking system the number of measurement errors and the standard deviation of measurements is determined. For that, the tinycar with its tracking markers is placed at five different locations. For each location 1000 measurements are taken. A ground truth for the measurements were not used for testing, as determining the exact position of the tinycar within 1 mm is not feasible. Measurements errors in the ground truth could potentially lead to wrong conclusions. Therefore only the deviation of the measurements is used and the number of measurements errors is counted. A measurement error would be a false positive, which means that the tracking system detects a marker which is not there. To determine the usability of the accuracy, a stanley controller using only the position data and virtual map with ground truth trajectory (as in Figure 4.5) is tested and evaluated in a later section.

The results are presented in Table 5.3 and show that for four locations the false positives and standard deviations are within a certain range. Location 5 however shows a significant increase in false positives. This problem can also be seen in the next section.

5.1.3 Stanley Controller

The stanley controller is mainly used in the automatic repositioning to bring the car back to the track after driving backwards up for a maximum of 30 steps. During the drive with the stanley controller, the steering angle is purely based on the position of the

Maneuver	CTE Avg [mm]	CTE Std Dev [mm]	H-Error Avg [deg]	H-Error Std Dev [deg]
Straight	14.645	24.592	11.193	13.409
Right	5.687	5.194	12.046	12.672
Left	11.877	18.576	20.178	28.430

Table 5.4: Benchmark for the stanley controller. The tinycar is driven with the stanley controller for 2000 steps for each maneuver in the real-world. The cross track error (CTE) and heading error (H-Error) are calculated for each step. The average and standard deviation are calculated for each maneuver. The CTE is in millimeter and the H-Error in degrees.

tracking system and its CTE and heading error to the ground truth trajectory. Table 5.4 shows a benchmark for the stanley controller, where the tinycar is driven for 2000 steps for each maneuver. During the drive the CTE and heading error are calculated for each step. Using the right maneuver, the tinycar has the lowest CTE and heading error. An artifical delay of 50 ms is added to also test, if the whole system delay is still within a feasible range. Figure 5.2 shows the positions of the tinycar rendered onto the map during this drive with the straight maneuver. It can be seen that near the two upper intersections the car struggles to keep the position within the lane. This also happens during the left turn maneuver. However, the right turn maneuver also passes the intersection, but from a different direction without any oscillations. Location 5 in Table 5.3 is in this area and shows the highest false positives, leading to false steering angles.

5.2 Encoder

The encoder is used to transform the observation from the gym environment into a format that can be used by the actor. More precise, it compresses information from a 160x128 image into a 256 dimensional vector. The encoder is trained via supervised learning, meaning that input and output are given in the training process. However, since the encoder learns an arbitrary representation of the input, the output of the encoder is not interpretable and thefore not known beforehand. But the steering angle for a given input can be easily collected by either driving the car manually or with the stanley controller. So the encoder and actor are trained simultaneously, as depicted in Figure 3.2. That way, the feature vector used by the actor includes information that distinct certain input images from others to predict the correct steering angle. Different experiments

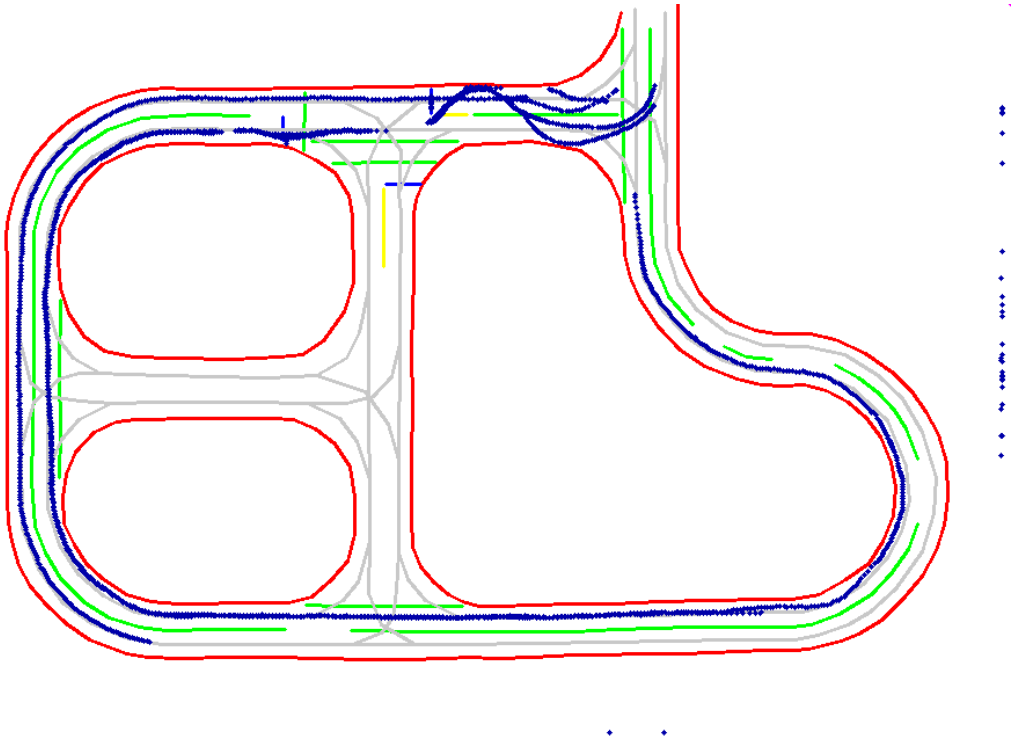


Figure 5.2: Positions of the tinycar rendered onto the map while steering with stanley controller in the real-world. The light gray lines show the ground truth trajectories and the blue line shows the actual driven path. The other colors visualize the lane lines. The figure shows that near the two upper intersections, the car struggles to keep the position within the lane. The blue dots on the right and lower side of the image show the false positives of the tracking system.

are conducted to train a suitable encoder. Especially one that generates similar feature vectors for simulation and real inputs, to minimize the sim-to-real gap in the beginning.

Three different training setups are tested for that. The first trains two encoders, one using simulation data, the other using real data. These encoders are not intended to be used for the final model, but to analyze the sim-to-real gap and as a baseline for the other two training setups. The second setup trains one encoder using simulation data and real data. So both datasets are mixed together and sampled randomly from during training. The third setup shares the weights of the actor to generate similar feature vectors for simulation and real inputs. With this setup, also two encoders are trained. But the one with simulation data is trained first and then the weights of the actor are copied to the second encoder. This second encoder is then trained with real data, but only the encoder part. The actor part is frozen. The idea is that the second encoder learns to generate feature vectors in such a way that the actor can use them to predict the correct steering angle.

The simulation training dataset is generated by using the stanley controller to drive on the map of the environment at HAW Hamburg and Knuffingen. Split between these two environments is 50-50, with each dataset consisting of 40,000 images. So the simulation data in total consits of 80,000 images. Using the stanley controller, an Ornstein-Uhlenbeck noise is added to the steering angle, so that the encoder also has images outside the current driving lane. This diversifies the dataset. The ground truth steering angle in the dataset however, is the clean stanley controller, which would then calculate a steering angle to get from outside the lane back into it. The spawn points are set so that each intersection is passed from each possible direction. Each spawn point is also driven three times, to include every possible maneuver. One episode in the data collection is 1000 steps, but only every second frame is saved.

The real training data is generated by manually driving the tinycar on the track at HAW Hamburg and Knuffingen. The total real dataset consits of 36,184 images, with 21,512 images from HAW Hamburg and 14,672 images from Knuffingen.

5.2.1 Feature Vector Comparison

To compare the feature vectors with each other, a sequence of 631 frames will be used. The frames were collected by driving the tinycar with the stanley controller over the track at HAW Hamburg. At each step, the frame of the real camera (which is post-processed

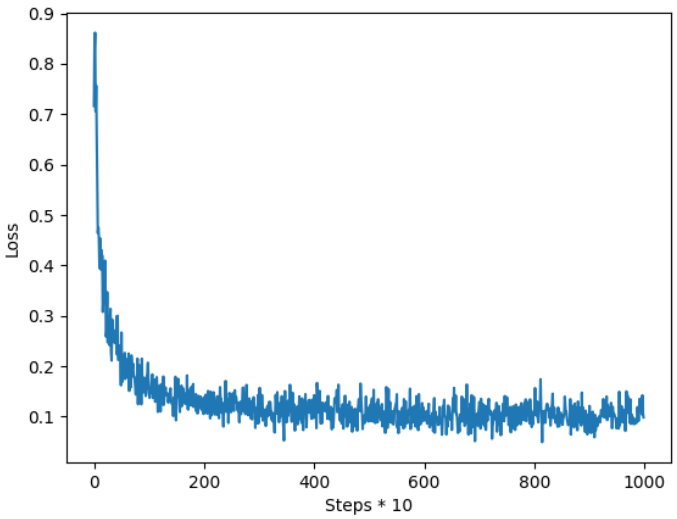


Figure 5.3: Loss curve during training process of the encoder with simulation dataset only. The loss function is mean squared error, which is averaged over the last 10 values and plotted every 10th step.

with the lane segmentation) is captured, parallel to a rendered frame in the simulation at the same location and orientation as in the real-world. Figure 4.6 shows an example where the raw camera image is overlayed with the rendered simulation image. So at each step in the sequence, two input images are saved. The feature vectors are then calculated with the respective encoder. The average, std deviation and maximum difference of these two vectors are used to determine the similarity of the feature vectors. These values are then averaged over the whole sequence.

The encoders are trained with a batch size of 32 and a learning rate of 1e-4. Adam is used as the optimizer. They are trained for 10,000 steps each. So 320,000 images are fed through the network and backpropagated, which is every frame 4 times for the simulation dataset. For the loss function the mean squared error is used. Figure 5.3 shows the loss curve for the encoder trained with simulation data only. Similar curves can be seen for all over training setups as well.

Using the two independently trained encoders, they produce a mean difference of the vectors of 1.143 for the 631 frames. The maximum difference is 10.084. This means that the maximum difference between two values in their feature vectors is 10.084. The standard deviation is 0.071. These values do not have a unit and the value of the feature

Setup	Mean	Max	Std Dev
Independent	1.143	10.084	0.071
Mixed	0.679	7.486	0.093
Shared Actor	0.636	6.035	0.056

Table 5.5: Difference between feature vectors of 2 different encoders respectively. Input images are of same location but one from the real-world environment and the other simulated. A low difference means a very similar feature vector for the same scene represented, either in simulation or real-world.

vector is not interpretable. However, they can be compared relative to each other with the other two setups. Table 5.5 shows the results for all three setups. These show that mixing training dataset or sharing the weights of the actor generates more similar feature vectors than independently training two encoders. Even though there is no big difference between the mixed and shared actor setup.

5.2.2 Supervised Training Driving Results

Since the encoder implicitly updated the actor when training the encoder, the similarity of feature vectors can also be tested by benchmarking the driving performance. In this experiment, the actor, which is initially trained using only simulation data, is used in conjunction with the encoder trained on real data using the shared actor setup. This setup gave the best results in the differences of the feature vectors (as seen in Table 5.5). This supervised learned encoder and actor are tested on the track at HAW Hamburg in the real-world, similar to the test in Section 5.1.3. The same benchmark is conducted with 5000 driven steps for each maneuver.

Table 5.6 shows the results of the benchmark. The CTE and H-Error are calculated for each step and then averaged over the whole maneuver. Similar to the stanley controller (Table 5.4), the right turn maneuver has the lowest average CTE, both in simulation and real-world. The results also show that the driving performance based on the CTE is similar to the stanley controller, indicating that the encoder does generate feature vectors based on the real input to predict a steering angle with the sim based actor, to keep the car in the lane. It can also be seen that the real-world driving performance is slighly better than the simulation driving performance.

Figure 5.5 shows an excerpt of the positions during the benchmark drive from Table 5.6 in the real-world. It visualizes how the right turn maneuver can handle the right and

Env	Maneuver	CTE Avg [mm]	CTE Std Dev [mm]	H-Error Avg [deg]	H-Error Std Dev [deg]
Real	Straight	13.572	19.033	14.683	19.493
	Right	9.622	8.613	14.014	13.286
	Left	14.289	18.601	16.327	18.718
Sim	Straight	13.512	14.314	13.952	13.724
	Right	11.690	6.879	15.006	11.514
	Left	17.297	19.477	20.207	19.257

Table 5.6: Benchmark for the supervised trained encoder and actor with the shared actor setup. The actor is purely trained on simulated data while the encoder for the sim env is also trained on simulation data and for the real env trained on real data only. For both environments the actor was trained purely with simulated data. For each maneuver the car is driven for 5000 steps. The CTE is in millimeter and the H-Error in degrees.

left turn without leaving the lane. The left and straight maneuver leave the lane during a right hand turn. This can also be seen at other right turns on the track. During this benchmark drive the car did not make the left turn with the straight maneuver, which is not intended nor controlled but a result of the random maneuver choosing while driving. On all the other left hand turns however, the car stayed in the lane like the right or left maneuver. In the simulation, the actor did not show this behavior. But during a left turn on t-intersections, the car takes the turn too wide resulting in higher CTE. The left turn on t-intersections is performed both for left and straight maneuver. Figure 5.4 shows an excerpt for the straight maneuver.

5.3 Actor

As described in Section 3.1.1, the actor is trained with td3. First in the simulation, so the actor can explore the basic behavior. Then in the real-world, where the actor can adapt to the real-world conditions and to bridge the sim-to-real gap. From the previous experiments, the two encoders which share the supervised learned actor achieved the best results therefore they will be used for training the TD3 actor.

The exploration noise is an Ornstein-Uhlenbeck process (as described in Section 3.1.2) with a θ of 0.15 and a σ of 0.4. The replay buffer size is set to 500,000 and after each step the critic is learned with a batch size of 256. The actor, as recommended for td3 [13], is updated every second step with the same batch size. The learning rate for the critic is

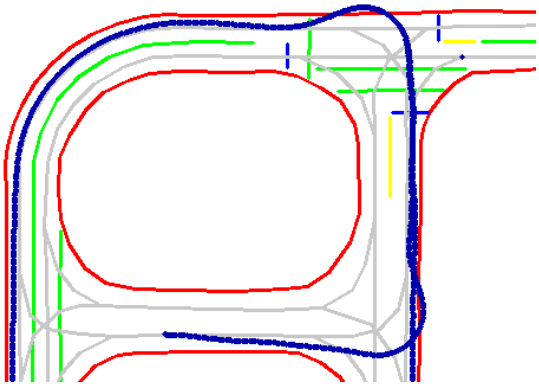
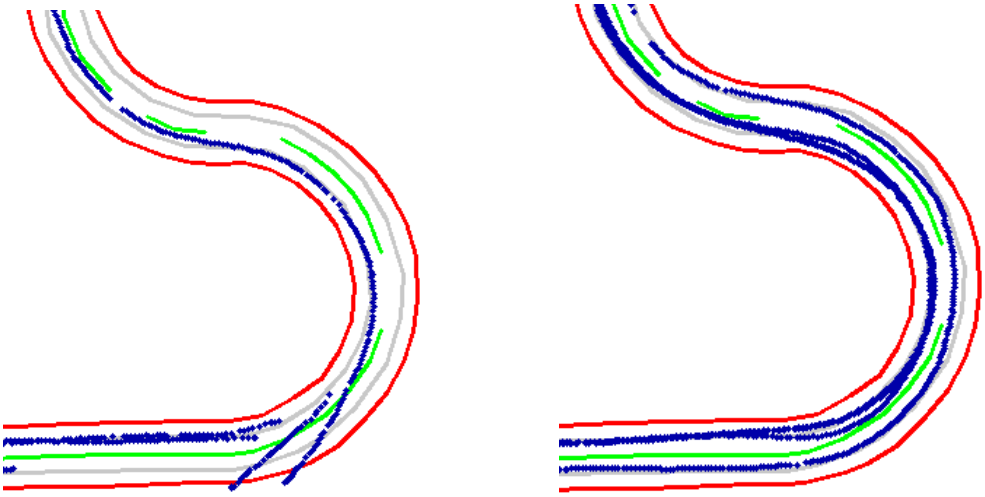


Figure 5.4: Supervised learning controller with simulation encoder steering the tinycar in the simulation. The figure shows an excerpt of the rendered positions in the map at two t-intersections with the straight maneuver. It can be seen that the car takes the left turns too wide resulting in overshooting and correction to middle of the lane after that. The light gray line are the ground truth trajectory. This executed maneuver would therefore result in a high cross-track-error.

2e-4 and for the actor 1e-4. The target update rate τ is set to 0.001. The discount factor γ is set to 0.99. For each episode a maximum step size of 1000 is set. For each episode a random maneuver and spawn point is chosen. The spawn points are entries to every intersection from every possible direction. In the simulation the actor is trained for 1000 episodes and in the real-world for another 400-500 episodes.

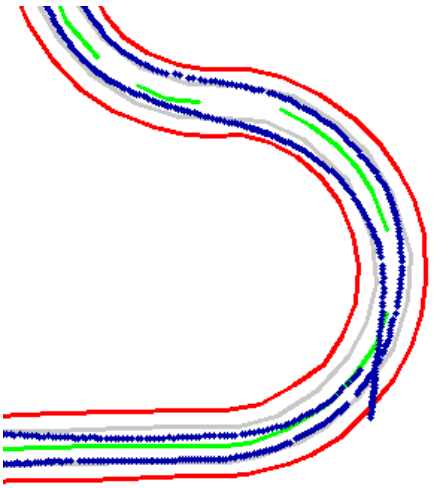
As described in Methodology, a linear reward shaping is used with a maxium reward of 1 and a minimum reward of -1 at an cross track error of 60 mm. If the cross track error is above 70 mm for at least 5 steps, the episode is terminated. The 5 steps threshold is used especially in the real-world environment in case of false postive tracking data. Added for the real-world environment is a crash termination. If the cars velocity is below 0.01 m/s for 5 steps, the episode is terminated. This is used to prevent the car from getting stuck when colliding with an obstacle.

To test the performance of the trained actor, different experiments are conducted. First, the actor is tested in the simulation to see if it can drive the car autonomously. Especially used for hyperparameter tuning and to see if the feature vector representation if sufficient for reinforcement learning an actor. Then the actor is tested in the real-world to see if it can adapt to the real-world conditions and bridge the sim-to-real gap. For that the car is driven on the track at HAW Hamburg. Knuffingen can only be used in the



(a) Straight

(b) Right



(c) Left

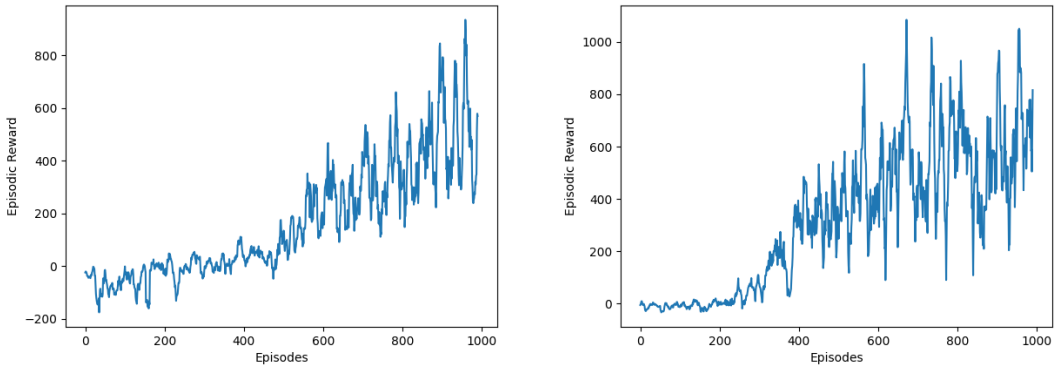
Figure 5.5: Positions of the tinycar rendered onto an excerpt of the map while steering with the supervised learned encoder and actor in the real-world. Encoder is trained with real data while actor is trained with simulation data. The light gray lines show the ground truth trajectories and the blue line shows the actual driven path. The other colors visualize the lane lines. The figure shows that the left and straight maneuver struggle to make the right turn while the right turn maneuver handles the right and left turn without leaving the lane.

Figure 5.6: Part of the map of the Knuffingen environment with a large intersection. From each direction the intersection has a distinct lane for turning or keeping straight. This lane splitting happens centimeters before the actual intersection. It can be seen at node number 277, 21 and 158.

simulation, since no tracking data is available for the real-world environment. For these experiments however, the car will be driven with the simulation only trained actor first. After continuing the training in the real-world, the car will be driven with the real-world trained actor. These two actors will be compared to check for differences. The last experiment will train the encoder again with a supervised learning process, but this time the weights of the td3 actor will be used. This is to test if the performance can be enhanced in case the td3 actor uses features from the simulation encoder which the real-world encoder cannot generate.

5.3.1 RL in Simulation

In the simulation the actor is trained for 1,000 episodes with a maximum of 1,000 steps per episode in the HAW environment and 2,000 steps in Knuffingen. The maximum step increase for Knuffingen is due to the larger map and bigger intersections with a lane splitting centimeters before the intersection, as seen in Figure 5.6. So that the actor has enough time to completly pass the intersection within one episode, each episode has a maximum of 2,000 steps.



(a) HAW Hamburg

(b) Knuffingen

Figure 5.7: Episodic reward during training in the simulation. The rewards are averaged over the last 10 episodes. The encoder used for the actor is trained with simulation data only. The maximum possible reward for the HAW Hamburg environment is 1000, since the highest reward at a CTE of 0 is 1 per step. For Knuffingen the maximum reward is 2000.

Figure 5.7 shows the rewards per episode during the training process. The rewards are averaged over the last 10 episodes. Both reward curves show that the actor is getting better by achieving higher rewards over time. They do not get the maximum reward of 1000 for the HAW environment or 2000 for Knuffingen. But that would imply that the CTE is always 0 for every step. The reward curve in the Knuffingen environment shows more fluctuations than in the HAW environment.

Table 5.7 shows the benchmark after driving the trained actors in the simulation environments. Same as in the training, the car is driven the double amount of steps in Knuffingen compared to in the HAW Hamburg environment, to match for the size of the intersections. As for the stanley controller and supervised learned actor, the right turn maneuver has the lowest CTE and H-Error. The results of the td3 learned actor in the HAW environment also have lower values compared to the supervised learned actor in the simulation. In the Knuffingen environment the results are in general worse than in the HAW one, for every maneuver.

When looking at the actual positions during a maneuver at an intersection, as visualized in Figure 5.8, it can be seen that the car starts oscillating. This happens not only at this intersection, but at almost every intersection. Also the trained actor in Knuffingen shows the same behavior, but not all the time. The right turn maneuver also cuts the

Env	Maneuver	CTE Avg [mm]	CTE Std Dev [mm]	H-Error Avg [deg]	H-Error Std Dev [deg]
HAW	Straight	12.736	8.412	13.633	11.188
	Right	8.930	7.549	14.377	11.063
	Left	16.533	20.158	17.788	16.199
Knuffingen	Straight	19.866	26.479	7.364	12.015
	Right	13.306	19.252	9.296	15.259
	Left	15.014	20.211	12.254	18.363

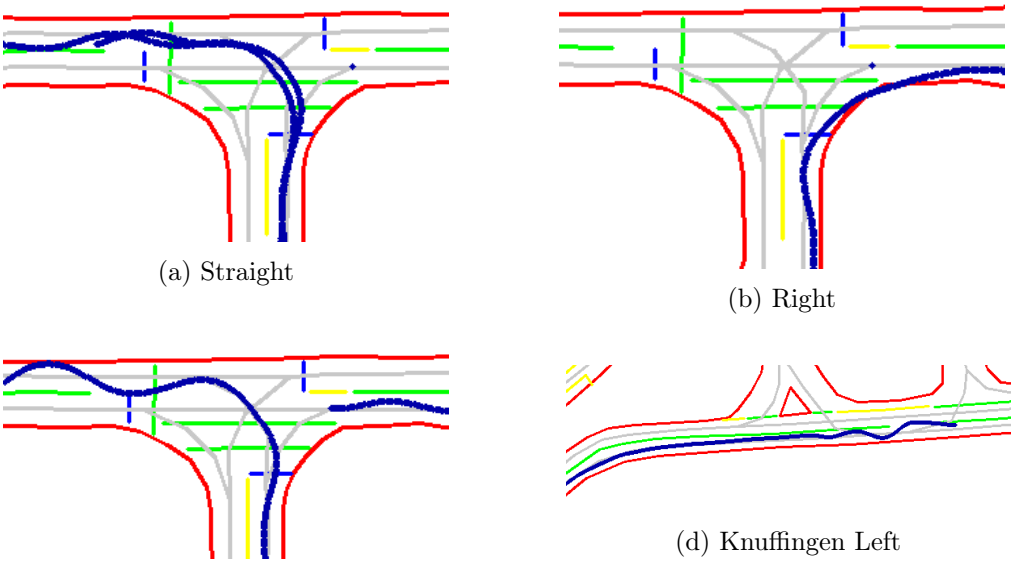
Table 5.7: Benchmark drive with the td3 actor in the simulation. The actor is trained for 1,000 episodes with a maximum of 1,000 steps per episode in the HAW environment and 2,000 steps in Knuffingen. For the test drive 5000 steps were driven in the HAW environment and 10000 in Knuffingen, to cover for the bigger map. The CTE is in millimeter and the H-Error in degrees.

corner, crossing the outer edge of the road. The highest oscillations can be seen during the left turn maneuver. This is also the maneuver with the highest standard deviation in the benchmark (Table 5.7).

In Knuffingen the actor does not tend to oscillate as much as in the HAW environment, but it does take the wrong lanes approaching an intersection. Figure 5.9 shows the positions during a maneuver in Knuffingen. The car always takes the right turn lane, regardless of the maneuver. All episodes are terminated early because the cross track error exceeds 70 mm at the last step. The right turn maneuver starts to cut the corner, which also exceeds the termination threshold of 70 mm. The left turn maneuver keeps the right lane for the longest time but then also turns into the right lane with a high heading error.

5.3.2 RL in Real-World

The Knuffingen environment in the Miniatur Wunderland Hamburg does not contain a tracking system for the cars which could be used for the benchmarking. Therefore the car is only tested in the HAW Hamburg environment, for the real-world tests. This also means that for the following tests, the td3 actor which was trained in the simulation using the HAW environment track will be used. This actor will be tested first in the real-world environment without any modifications besided swapping the encoder with the real-world trained encoder, from Section 5.2.2. The benchmarking is the same as in



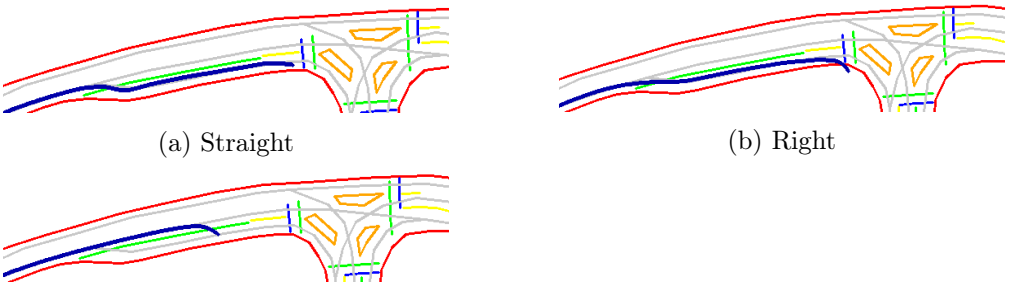
(a) Straight

(b) Right

(c) Left

(d) Knuffingen Left

Figure 5.8: Positions while driving in the simulation with td3 actor. The light gray lines show the ground truth trajectories and the blue line shows the actual driven path. The other colors visualize the lane lines. The figure shows that even though every maneuver is executed correctly, the car always starts oscillating. The right turn maneuver also cuts the corner, crossing the outer edge of the road. In Knuffingen the same behavior can be seen centimeters before the car is supposed to make a left turn.



(a) Straight

(b) Right

(c) Left

Figure 5.9: Positions while driving the td3 actor in the simulation in Knuffingen. The figure shows an intersection where the car always takes the right turn lane, regardless of the maneuver. The episodes are all terminated after the cross track error is above 70 mm.

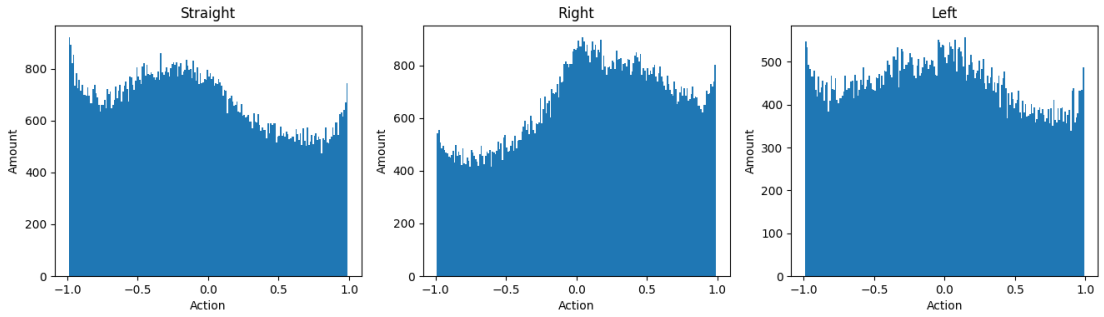


Figure 5.10: Histogram of the experience buffer used to train the actor in the real-world. The figure shows the number of steering angles in the buffer for each maneuver. The buffer comes from the td3 training in the simulation. It was cleaned afterwards, by skipping the first 100,000 episodes which mostly consists of steering angles either -1 or 1. Since the action space is continuous, the steering angles are quantized into 0.01 steps.

previous sections, where the car is driven for 5000 steps for each maneuver and the CTE and H-Error are calculated for each step.

After that the car is trained in the real-world with the same hyperparameters as for the simulation (Section 5.3.1). Only the number of episodes will be changed from 1000 to 300 episodes. Figure 5.11 shows the rewards per episode during the training process. The rewards are averaged over the last 10 episodes. In the beginning the rewards rapidly go down to values below 0. Over time however, they increase on average. But the rewards are still not near the maximum possible reward of 1000.

The experience buffer collected during the training process in the simulation is used to train the actor in the real-world. This should avoid a catastrophic forgetting otherwise experienced. The buffer is cleaned by skipping the first 100,000 episodes, which mostly consist of steering angles either -1 or 1. Figure 5.10 shows the histogram of the experience buffer.

During the 300 training episodes, the automatic repositioning successfully brings the car back to the track in 48 cases. This makes a success rate of 84 % for the repositioning.

Comparing the drive benchmark results from Table 5.8 with the simulation benchmark results (Table 5.7) shows that the real-world driving performance is worse than in the simulation. The CTE and H-Error are higher for every maneuver. The performance after training the actor in the real-world does not change for the straight maneuver, gets better for the left maneuver and worse for the right maneuver.

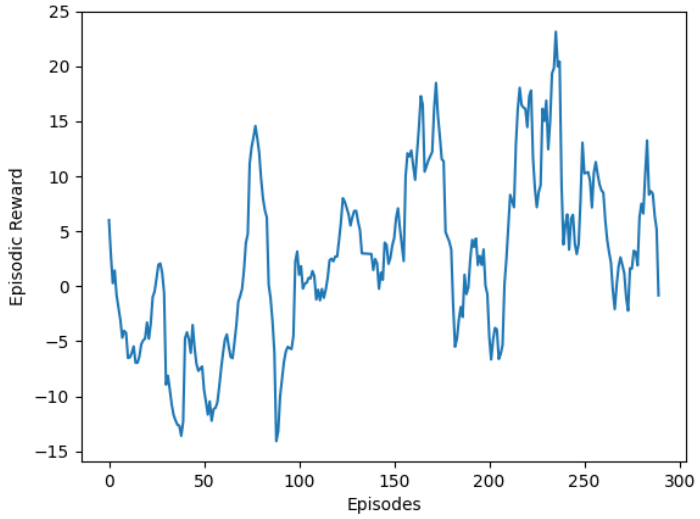
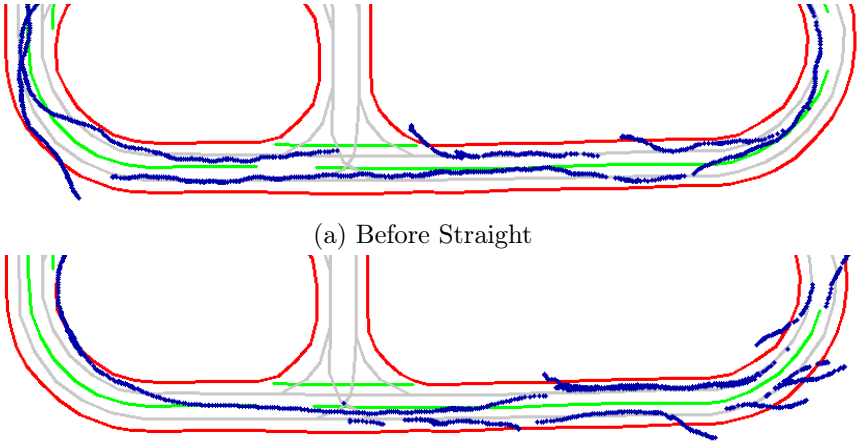


Figure 5.11: Episodic reward during training in the real-world at HAW Hamburg for 300 episodes. The actor is pretrained in the simulation for 1000 episodes. The encoder is different when pretrained, but both encoders are trained with the same supervised learned actor. The rewards are averaged over the last 10 episodes.

Phase	Maneuver	CTE Avg [mm]	CTE Std Dev [mm]	H-Error Avg [deg]	H-Error Std Dev [deg]
Before	Straight	29.646	22.848	19.543	18.513
	Right	29.395	22.410	22.863	22.450
	Left	38.137	28.070	33.879	24.004
After	Straight	29.267	22.360	22.554	22.055
	Right	49.846	29.333	31.331	20.243
	Left	26.979	24.065	22.534	21.993

Table 5.8: Benchmark drive with the td3 actor in the real world in the HAW environment. The before phase is the actor which is trained in the simulation using only the real encoder. The after phase is the actor which is trained for another 400 episodes in the real-world. It is based of the actor from before phase with the same encoder.



(a) Before Straight

(b) After Straight

Figure 5.12: Positions while driving in the the real-world in the HAW environment with two td3 actors. The before phase is the actor which is trained in the simulation using only the real encoder. The after phase is the actor which is trained for another 400 episodes in the real-world. The light gray lines show the ground truth trajectories and the blue line shows the actual driven path. The other colors visualize the lane lines.

This is also visible in the positions during the benchmark drive, as shown in Figure 5.12. The positions of the car show that in both phases the car struggles to keep itself straight and in the center of the lane. For both maneuvers the average CTE and standard deviation is the same in the before and after phase. Training in the real-world does not improve the driving performance using the real-world encoder and td3 learning algorithm.

To test if the oberservation space might be the problem for the td3 actor in the real-world, another actor is trained using raw camera data. So instead of using the lane segmentation, the 3 channel RGB image from the camera is fed into a seperate trained encoder and then into the actor. The encoder is trained with the same hyperparameters as the encoder trained with the lane segmentation. The same goes for the actor, which is trained for 400 episodes. Since the model does not use lane segmentation as input, the actor can not be pretrained in the simulation. Figure 5.13 shows the rewards per episode during the training process. The rewards are averaged over the last 10 episodes. Same as for the training with lane segmentation, the rewards increase over time on average but do not come close to the rewards in the simulation training. Since the episodic reward does not improve compared to the actor with lane segmentation, a drive benchmark is not conducted.

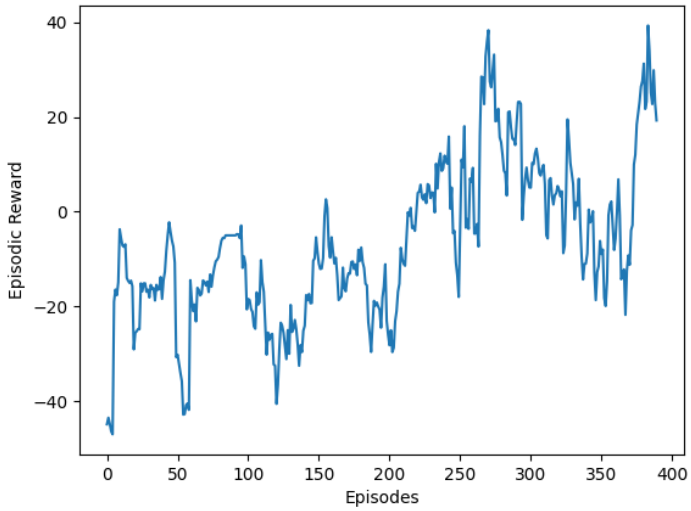


Figure 5.13: Episodic reward during training in the real-world with raw camera data. Input data is not the lane segmentation but the raw 3 channel RGB image from the camera. The actor is trained for 400 episodes without using a pretrained actor or existing replay buffer. The rewards are averaged over the last 10 episodes.

Setup	Mean	Max	Std Dev
Independent	1.143	10.084	0.071
Mixed	0.679	7.486	0.093
Shared Actor	0.636	6.035	0.056
Shared Actor RL Update	0.631	5.355	0.057

Table 5.9: Difference between feature vectors of 2 different encoders respectively. Input images are of same location but one from the real-world environment and the other simulated. A low difference means a very similar feature vector for the same scene represented, either in simulation or real-world. The values are the same as in Table 5.5 but with the encoder update setup included. The update is similar to the shared actor setup, but instead of the supervised trained actor with the td3 trained one.

5.3.3 Encoder Update with RL Actor

The results above show a significant decrease of average cross track error and heading error when using the pretrained actor from the simulation in the real world. Therefore, another test is conducted to see if the performance can be enhanced by training the encoder again with the td3 actor. The encoder is trained for 10,000 steps with the same hyperparameters as in Section 5.2. The encoder is trained with the same data as before, but this time the weights of the td3 actor from the simulation are used. The old encoder is then compared to the new updated encoder by using the same td3 actor from the simulation in the real-world.

Additionaly, the differences between the feature vectors from the simulation and real-world are compared, similar to Table 5.5. As references, the feature vector differences of this table are also included. The results are shown in Table 5.9 and are similar to the shared actor setup, but with a slightly lower max value. This behavior is expected though, since using a different actor for the encoder update should not change the overall feature vector differences too much. Therefore the driving performance with this encoder update should be tested to see if there is an improvement in the features generated.

Table 5.10 shows the results of the benchmark drive with the td3 actor in the real-world using the encoder update. The average CTE is worse for all maneuvers compared to using the not updated encoder. Even though, the loss curve during the encoder training process (Figure 5.14) shows a similar behavior as in the encoder training process with the supervised learned actor (Figure 5.3). The loss curve shows a significant decrease over time, expecting a better performance in the benchmark drive.

Phase	Maneuver	CTE Avg [mm]	CTE Std Dev [mm]	H-Error Avg [deg]	H-Error Std Dev [deg]
Before	Straight	29.646	22.848	19.543	18.513
	Right	29.395	22.410	22.863	22.450
	Left	38.137	28.070	33.879	24.004
After	Straight	31.536	26.273	24.394	18.444
	Right	34.583	23.967	20.088	17.428
	Left	41.079	25.405	20.990	18.676

Table 5.10: Benchmark drive with the td3 actor in the real world in the HAW environment. The before phase is the actor which is trained in the simulation using only the real encoder. The values are the same as in Table 5.8. The after phase is the same actor but with the encoder updated with this actor. The encoder is trained for 10,000 steps with the same hyperparameters as in Section 5.2.

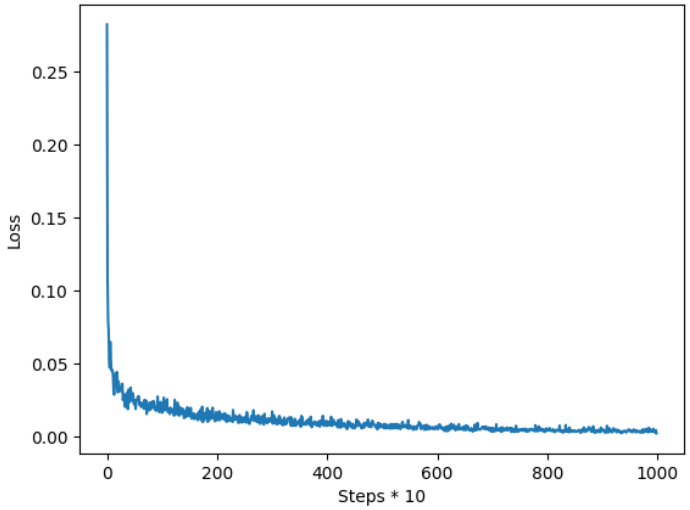


Figure 5.14: Loss curve during training process of the encoder with the td3 actor. The loss function is mean squared error, which is averaged over the last 10 values and plotted every 10th step. The training data used is the dataset consisting of only manually driven images in the real-world. The weights of the shared actor encoder used before are used as starting weights for this updated encoder.

6 Discussion

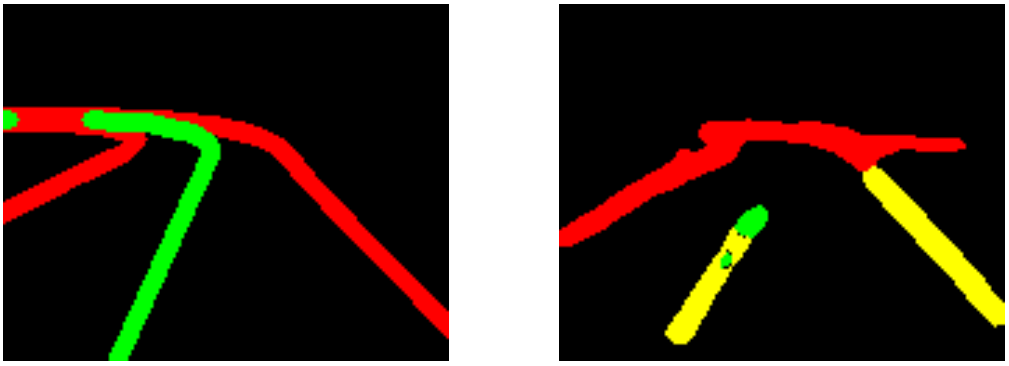
This chapter discusses the results of the experiments and the overall project. The main goal of this thesis was to train a reinforcement learning model in a simulated environment and transfer it to the real world. The model should be able to drive the car autonomously on the track and handle intersections by executing the chosen maneuver. Additionaly the model should be able to generalize to the real world and adapt to the real-world conditions.

This chapter uses the same main structure as the results chapter. It starts with a discussion of the environment. Especially how useful it is for the reinforcement learning task. The following section discusses the encoder with its improvements and the overall ability to minimize a sim-to-real gap. The actor is discussed in the next section with the results of the actual real-world reinforcement learning.

6.1 Environment

The gymnasium environment includes both the simulation of the car and the digital twin for the real-world. Especially the abilities of the real-world environment are interesting to see if there are big differences to the simulation and if the real-time aspects and tracking system work as intended.

Figure 4.6 shows that the simulated camera matches the real camera image quite well. The simulated image does not take lens distortion into account and the base info of lane lines in the map are a list of straight lines, which could never match the curvature of the road completly. However, the difference between the simulated lane segmentation and the real lane segmentation, from the neural network, is more significant. It can be seen in Figure 6.1 that the real lane segmentation is more noisy and has more false positives. This could be due to the neural network not being accurate enough and needing more training data than the 708 images it was trained on [27]. The given lane segmentation



(a) Simulated (b) Real

Figure 6.1: Example of the difference between the simulated lane segmentation and the real lane segmentation. The input for the real lane segmentation is at the exact same location and orientation as the simulated image. The real lane segmentation is done by a neural network.

model was already trained with additional 324 images taken with the new tinycar. Both in the HAW environemt as well as Knuffingen. This step seems necessary since the images taken for the old lane segmentation model were taken with a complete different camera and car. The still resulting difference between the simulated and real lane segmentations are tried to be minimzed by the encoder, which the actor uses as input. This will be discussed in the next section.

Table 5.1 and Table 5.2 show the latencies of the tinycar and the tracking system. The biggest latency and bottleneck of the system is the tinycar latency. As expected has the lowest camera resolution the lowest latency. With the encoder using this lowest resolution of 160x120 pixels one could argue that this resolution is sufficient to use. However, the difference between the 320x240 to the 160x120 resolution is not as big as between the other resolutions and since the lane segmentation model performs better IOU scores with a bigger resolution [27], the 320x240 resolution is used. With the latency for a training step, the whole latency for one step is 101 ms. This is quite high but with the speed of 0.03 m/s set, the car would travel 3 mm in one step. As later results show, where the car drives with an artifical delay using the supervised learned actor, this delay might bring some problems but is no breakpoint.

The latency for the tracking system does not matter as much as the latency for the tinycar, which is systemically relevant. The overhead camera resolution of 640x480 pixels is used, since it is the middle ground between high accuracy and reasonable latency. Even

though there is a 24 ms latency, it is only the half of the tinycar latency. Using the higher accuracy with 1.1 mm would result in a latency of 42.3 ms with which it would not be able to maintain the rate of 30 Hz. Therefore 640x480 pixels is used.

As Table 5.3 can show, the false positive rate of the tracking system lies between 1.7 and 3.4 percent. Together with the standard deviation for the position and orientation, the tracking systems accuracy is enough for the task. This is also demonstrated in the results where the car drives autonomously on the track steered by a stanley controller solely based on the tracking system. With the presenence of false positives however, an improvement could be made which finds false positives and predicts a location in that cases. The termination conditions, when training in the real-world, already take false positives into account by checking if the condition is met for a certain amount of time. This is done to prevent the car from stopping when a false positive occurs. The reward shaping however does not take false positives into account. This could be a possible improvement for the future.

The stanley controller tested shows that the accuracy of the tracking system is enough to autonomously drive the car. It also shows that the latency introduced by the tracking algorithm is not a problem. The average cross track errors are within reason, especially for the right maneuver. As Figure 5.2 shows, the higher CTEs for the left and straight maneuver most probably occur due to tracking issues in a certain area. Table 5.3 also showed that problem in the Location 5 row, which is in that direction. Figure 6.2 zoomes into the problematic intersection and reveals the most probable cause of the problem, which is the wait line. Since the wait lines are white and much thicker than normal dashed or solid lines, this could be the cause of the tracking system losing the car. The markers which are used by the tracking system are white as well. This behavior however only occurs during specific lightning conditions. The white tones of the street markings and markers on the car are not completly the same, which the tracking algorithm can distinct through filtering. But sometime when the lightning conditions are not optimal, the tracking system produces these false positives more often causing big runaways as seen in Figure 5.2. For later tests, this was fixed by changing the color filter thresholds to meet with the current lightning conditions. A more robust solution would be to use a different color for the markers on the car, which are completly different from all the colors seen in the environemt.

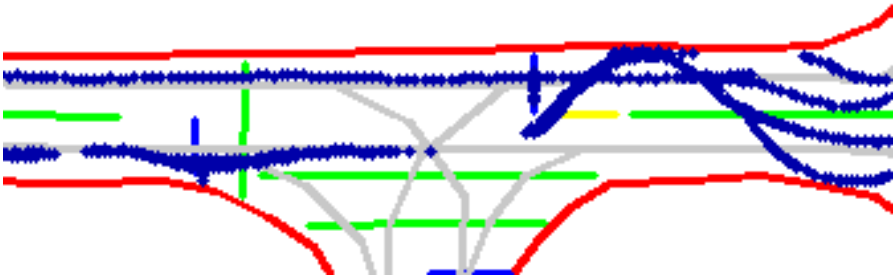


Figure 6.2: Excerpt from Figure 5.2 showing the problematic intersection zoomed in. Before the big runaway, the tracking system lost the car and produces false positives at the wait line above. Same can be seen more left at the wait line.

6.2 Encoder

As shown above, the real lane segmentation model is not as accurate as the simulated lane segmentation. The encoder, which takes these images and produces a small feature vectors, should minimize this problem by some factor. So that the feature vectors for the actor are more similar. This way the actor only has to handle a smaller difference between the simulated and real-world data, ultimetaly reducing the sim-to-real gap.

As Table 5.5 shows, through different techniques the encoders can be improved to produce more similar feature vectors. Even though the difference between training one encoder with both datasets or two encoders with a distinct dataset but share weights of the actor does not show a big difference. However, the standard deviation using the technique with the shared actor weights is smaller by 50 %. The reason why they do not produce more similar feature vectors might be due to the wrong training process. When training the two encoders with the shared actor weights, the loss function only tries to minimize the difference in the steering angles but not of the feature vectors. Because the real training data does not have an exact replica with simulated images, the loss function could not be used to minimize the difference in the feature vectors. This would only be possible if at one training step the network would get both a simulated and a real image at the exact same location and orientation. This can be done in the HAW environment where a tracking system is installed but not in Knuffingen which lacks such a system. This could however be a major improvement for the future. The steering angle should still be used in the loss function since otherwise the encoder could produce always the same feature vector or feature vectors which do not differ enough to drive a car autonomously.

The side product of training the encoders is an actor which can be used to drive the car. The actor trained with the shared actor weights technique is used to drive the car in the real-world, as seen in Table 5.6. The benchmark shows that the encoder trained with the real-world data is in fact able to drive the car with an actor trained with simulation data. This is expected though, since the encoder is just trained to create feature vectors for the actor, which generates steering angles similar to the correct ones from the real-world training data. The weight setup of the actor does not matter that much since the encoder would just produce feature vectors which could potentially overcome any issues. However, the results also show that in both environments the car struggles with the left and straight maneuver more than with the right maneuver. This is the same behavior seen with the stanley controller. With the stanley controller the problem was the false positives of the tracking system. The training data for the encoder however does not need the tracking system at all. The intersection, which creates problems for the stanley controller, does not create any for the supervised learned actor and encoder. As Figure 5.4 shows, the problem is an overshooting in the steering when approaching the t-intersections from a certain direction, where the car does have to turn left or right. Since the straight maneuver always has to turn left at these situation (it is a rule set in the simulation), the problem not only occurs for the left maneuver but also straight. This would explain why the left and straight maneuver show the same results, especially when approaching the intersection from that direction. For the overshooting no plausible explaination could be found, since the training data does not contain this behavior. The training data for the simulation is generated by the stanley controller, which does not show this behavior, as seen above.

Another problem the left and straight maneuver have is driving right curves, as seen in Figure 5.5. Only when choosing the right maneuver, the car makes that turn without leaving its lane. Even though the simulation training data is perfectly distributed with the same areas driven with every maneuver, the real-world training data is not. A lack of the problematic curve with left and straight maneuver compared to other curves, might be the cause of the problem. The simulation data is collected by driving with a stanley controller. For each spawn point the same amount of steps is driven with each maneuver. The real-world data is collected by manually driving the car. Every section of the track is included in this dataset with every maneuver. But it might be that the left and straight maneuver need more data for the problematic curve since driving right curves is not as common as driving left curves for these two maneuvers. This could be a possible explanation for the problem.

6.3 Actor

The actor is the main part of the reinforcement learning model. It takes the feature vectors from the encoder and produces steering angles. The actor is trained with the TD3 algorithm in the simulation and the real world.

Training the encoder in the simulation in the HAW environment and Knuffingen shows an improvement in the reward taken per episode over the time. The episodic rewards in the HAW environment shows an smoother curve than in Knuffingen, as seen in Figure 5.7. This could mean a better performance of the actor in the HAW environment which is also a much simpler track layout. Table 5.7 shows the results of the benchmarking drives of these two actors. It also shows that the actor in the HAW environment performs better than in Knuffingen. The reason for this could be the more complex track layout in Knuffingen with bigger intersections, as Figure 5.6 shows. The presented intersection has distinct lanes for keeping straight or turning. The intersections in the HAW environment are much simpler, smaller and do not have distinct lanes. They can be more compared to intersections found in suburban areas. The actor in Knuffingen has to make a decision which lane to take based on the given maneuver before the intersection is even in sight. This also means that during that time, 3 lanes have to be considered, from which one is the oncoming traffic. The actor has to make a decision based solely on the lane markings and the given maneuver, with no context information. This can be even challenging for a human driver. This can already indicate that just lane segmentation images might not be enough to handle more complex environments as Knuffingen. Unfortunately, this is hard to test further since accurate results in the real-world would need a tracking system.

Results from the simulation, as shown in Figure 5.8d, show that the actor has the same problems in Knuffingen as in the HAW environment. Both tend to start oscillating near a decision needed intersection. But while the actor in the HAW environment still executes the right maneuver, in Knuffingen it tends to execute the wrong one, as seen in Figure 5.9. On this intersection coming from this direction, the actors always chooses to take the very right lane, even though a left maneuver might be chosen. During the training the spawn points are set to intersections only, so that the actor learns these in particular. But still the training episodes might not be enough for a complex environemt like Knuffingen or the lane segmentation is simply the wrong approach. When looking further into the intersection in Figure 5.9 it can be seen that the straight lane does not have a dashed nor solid line to the oncomimg traffic lane. This could be a problem for the actor since it

maybe does not know it can drive in that area based on the lane segmentation only. This is one example making it hard to drive in Knuffingen with lane segmentation only.

For the training in the real-world the actor used the experience buffer from the simulation training as a starting point to prevent catastrophic forgetting. Even though the episodic rewards increase over time, the overall performance is not near the performance in the simulation. The used pretrained model does also underperform the simulation actor by a big factor indicating a too big sim-to-real gap in the first place. For training the model in real-world would probably need a good working pretrained model first. Otherwise using a pretrained model does not make sense if the model as so start from the beginning nonethless. Training the real-world actor for more than 2000 episodes might give better results, since a general lerarning curve is existent, but that is not feasible with the current setup due to time constraints and recharging the car.

The pretrained actor might struggle in the real-world because the differences in the feature vectors ars still too prominent. The encoder does not minimize the sim-to-real gap enough. Using the supervised trained actor in the real-world works well however. As said before, this is probably because the encoder just learns feature vectors which activate certain neurons to give the correct steering angle. Using the same technique was also tested with the pretrained actor, as presented in Section 5.3.3. But the new updated encoder even performs worse than the encoder used before. Even though the loss curve and results of the feature vector differneces look promising. Maybe the actor utilizes the feature vectors in a different way than the supervised learned actor. One could argue that the lane segmentation input is not enough for the actor to make a decision. But the lane segmentation input works for the actor in the simulation and for the supervised learned actor in the real-world. Using raw camera image for the actor in the real-world also shows no improvement and the same results. Because the raw image actor cannot use a pretrained model, the episodes it was trained might just be not enough. That however would show, that the actor in the real-world using the lane segmentation can not use the experience and knowledge of the simulation actor at all and also has to start from the beginning. In the end, the cause could either be that the td3 learning algorithm with the exploration noise used is not suitable for the problem in the real world and/or that the encoder/lane segmentation sim-to-real gap is too big to start with.

7 Conclusion

In the scope of this thesis a complete system to test and train machine learning and reinforcement learning models for autonomous driving in a miniature environment was developed. The system consists of a digital twin of the real 1:87 city environemt at HAW Hamburg. Using a simulation of the car and the environment, different tests and trainings can be conducted, including the training of a reinforcement learning policy. With a setup of overhead cameras above the real track the position and orientation of the real car is known after every step. This way the reinforcement learning can also be done in the real-world, transfering the models from the simulation. The tinycar, which was also developed for this thesis, is a 1:87 scaled car that can be controlled by a computer and is equipped with a camera to perceive the environment and lane markings. By streaming the camera data to the computer running the digital twin and reinforcement learning algorithms, the images can be used as an observation for these models.

The first results show, that the whole system is capable of training neural networks to autonomously drive the tinycar to even handle intersections. The latencies of the tinycar with the camera stream over a wireless network are low enough to be used for accurately controlling the car. The tracking system is able to track the car with a high accuracy and a low latency. During the reinforcement learning training process, the automatic repositioning, which uses the tracking systems data to steer the car onto the track using a stanley controller, is able to conduct the necessary episode reset automatically. In 84 % of the cases the car is able to reposition itself onto the track without any human intervention, drastically reducing the time needed for training.

Using encoders to reduce the dimensionality of input data for the actor reduces the training time in the real-world, which needs to work in real-time, and can minimze the sim-to-real gap early on. The supervised trained models, from which the encoders are ectracted, also showed promising results in controlling the car autonomously. With a given maneuver, which is set to either straight, right or left, the car is able to handle intersections and take the right exit. Its performance in the real-world, using an actor

trained in the simulation, is also similar to results driving the car with a stanley controller. The supervised actor is able to drive the car on the track without leaving it or colliding with obstacles.

The reinforcement learning models, which are trained in the simulation, are able to handle the same intersections in the simulation, even though problems like oscillations occur. In the complex environment of Knuffingen, the actor struggles to take the right lane when approaching an intersection. Taking these models to the real-world shows however, that they do not generalize well enough to stay within the lane. Even after training for another 400 episodes in the real-world, the models do not improve significantly.

Altogether, this thesis still presents a system usable to further test and investigate the challenges of reinforcement learning in the real-world and to bridge its sim-to-real gap. Using several experiments, it can be shown that the chosen observation and action space is sufficient for this task, while the chosen algorithms and model architectures might not.

7.1 Outlook

Further improvements can be made for the tracking system, which is only able to track the car with certain markers on top. Especially the white markers used, sometimes are mistaken for the white wait line markings. Using a different color and better filtering techniques can improve the false positives rate of the tracking system. Other approaches might be the usage of neural networks to detect and track the car. Especially to distinct different cars on the track, so that multiple cars can be used at the same time. That can open more doors to even simulate traffic scenarios and more complex interactions between the cars.

The gym environment can be extended to include more complex scenarios, such as traffic lights, car or other obstacles. This can be used to train the models to even drive within the full traffic simulation in Knuffingen at the Miniatur Wunderland Hamburg.

The sim-to-real gap in the perception can also be improved by maybe using a completly different approach. Instead of having a distinct model for the lane segmentation task and a model as an encoder, which encodes these lane segmentation images into a feature vector, an autoencoder setup could be used. The autoencoder can be trained in the same way as the lane segmentation model, however, the bottleneck of the autoencoder

can be trained to create the same features for real images as for simulated ones. This way the idea of simulating only lane segmentation images can still persist, which is a lot easier than to simulate real photorealistic images, but the features extracted from these images can be used in the real-world as well. This might reduce the sim-to-real gap in the perception part of the system and make the actor trained in the simulation be useful in the real-world as well.

Bibliography

[1] : *A Drive to Taco Bell*. Dezember 2022. – URL https://blog.comma.ai/taco-bell/. – Zugriffsdatum: 2024-04-30

[2] Allamaa, Jean P. ; Patrinos, Panagiotis ; Van der Auweraer, Herman ; Son, Tong D.: Sim2real for Autonomous Vehicle Control Using Executable Digital Twin. In: *IFAC-PapersOnLine* 55 (2022), Januar, Nr. 24, S. 385–391. – URL https://www.sciencedirect.com/science/article/pii/S2405896322023461. – Zugriffsdatum: 2024-02-05. – ISSN 2405-8963

[3] Alsubaei, Faisal S.: Reliability and Security Analysis of Artificial Intelligence-Based Self-Driving Technologies in Saudi Arabia: A Case Study of Openpilot. In: *Journal of Advanced Transportation* 2022 (2022), Juli, S. e2085225. – URL https://www.hindawi.com/journals/jat/2022/2085225/. – Zugriffsdatum: 2023-12-05. – ISSN 0197-6729

[4] Aly, Mohamed: Real Time Detection of Lane Markers in Urban Streets. In: *2008 IEEE Intelligent Vehicles Symposium* (2008), Juni, S. 7–12. – URL http://arxiv.org/abs/1411.7113. – Zugriffsdatum: 2021-12-14

[5] Amini, Alexander ; Rosman, Guy ; Karaman, Sertac ; Rus, Daniela: Variational End-to-End Navigation and Localization. In: *2019 International Conference on Robotics and Automation (ICRA)* (2019), Mai, S. 8958–8964. – URL http://arxiv.org/abs/1811.10119. – Zugriffsdatum: 2022-02-20

[6] Atkinson, Craig ; McCane, Brendan ; Szymanski, Lech ; Robins, Anthony: Pseudo-Rehearsal: Achieving Deep Reinforcement Learning without Catastrophic Forgetting. In: *Neurocomputing* 428 (2021), März, S. 291–307. – URL http://arxiv.org/abs/1812.02464. – Zugriffsdatum: 2022-03-01. – ISSN 09252312

[7] Bojarski, Mariusz ; Del Testa, Davide ; Dworakowski, Daniel ; Firner, Bernhard ; Flepp, Beat ; Goyal, Prasoon ; Jackel, Lawrence D. ; Monfort,

Mathew ; Muller, Urs ; Zhang, Jiakai ; Zhang, Xin ; Zhao, Jake ; Zieba, Karol: End to End Learning for Self-Driving Cars. In: *arXiv:1604.07316 [cs]* (2016), April. – URL http://arxiv.org/abs/1604.07316. – Zugriffsdatum: 2022-03-13

[8] Bojarski, Mariusz ; Yeres, Philip ; Choromanska, Anna ; Choromanski, Krzysztof ; Firner, Bernhard ; Jackel, Lawrence ; Muller, Urs: Explaining How a Deep Neural Network Trained with End-to-End Learning Steers a Car. In: *arXiv:1704.07911 [cs]* (2017), April. – URL http://arxiv.org/abs/1704.07911. – Zugriffsdatum: 2022-02-20

[9] Cahill, Andy: *Catastrophic Forgetting in Reinforcement-Learning Environments*, University of Otago, Thesis, 2011. – URL https://ourarchive.otago.ac.nz/handle/10523/1765. – Zugriffsdatum: 2022-03-01

[10] Cai, Peide ; Sun, Yuxiang ; Chen, Yuying ; Liu, Ming: Vision-Based Trajectory Planning via Imitation Learning for Autonomous Vehicles. In: *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, Oktober 2019, S. 2736–2742

[11] Chung, Wankyun ; Fu, Li-Chen ; Hsu, Su-Hau: Motion Control. In: Siciliano, Bruno (Hrsg.) ; Khatib, Oussama (Hrsg.): *Springer Handbook of Robotics*. Berlin, Heidelberg : Springer, 2008, S. 133–159. – URL https://doi.org/10.1007/978-3-540-30301-5_7. – Zugriffsdatum: 2024-04-30. – ISBN 978-3-540-30301-5

[12] Dulac-Arnold, Gabriel ; Levine, Nir ; Mankowitz, Daniel J. ; Li, Jerry ; Paduraru, Cosmin ; Gowal, Sven ; Hester, Todd: Challenges of Real-World Reinforcement Learning: Definitions, Benchmarks and Analysis. In: *Machine Learning* 110 (2021), September, Nr. 9, S. 2419–2468. – URL https://doi.org/10.1007/s10994-021-05961-4. – Zugriffsdatum: 2024-02-06. – ISSN 1573-0565

[13] Fujimoto, Scott ; van Hoof, Herke ; Meger, David: *Addressing Function Approximation Error in Actor-Critic Methods*. Oktober 2018. – URL http://arxiv.org/abs/1802.09477. – Zugriffsdatum: 2024-05-10

[14] Hoffmann, Gabriel M. ; Tomlin, Claire J. ; Montemerlo, Michael ; Thrun, Sebastian: Autonomous Automobile Trajectory Tracking for Off-Road Driving: Controller Design, Experimental Validation and Racing. In: *2007 American Control Conference*. New York, NY, USA : IEEE, Juli 2007, S. 2296–2301. – URL http://ieeexplore.ieee.org/document/4282788/. – Zugriffsdatum: 2024-04-30. – ISBN 978-1-4244-0988-4 978-1-4244-0989-1

[15] Jung, Soonhong ; Youn, Junsic ; Sull, Sanghoon: Efficient Lane Detection Based on Spatiotemporal Images. In: *IEEE Transactions on Intelligent Transportation Systems* 17 (2016), Januar, Nr. 1, S. 289–295. – ISSN 1558-0016

[16] Kasten, Markus: *Hardwareplattform für autonome Straßenfahrzeuge im Maßstab 1:87*, HAW Hamburg, Bachelorarbeit, September 2021. – URL https://autosys.informatik.haw-hamburg.de/publication/2021markuskasten/. – Zugriffsdatum: 2021-09-27

[17] Lo, Yat L. ; Ghiassian, Sina: Overcoming Catastrophic Interference in Online Reinforcement Learning with Dynamic Self-Organizing Maps. In: *arXiv:1910.13213 [cs]* (2019), Oktober. – URL http://arxiv.org/abs/1910.13213. – Zugriffsdatum: 2022-03-01

[18] Macenski, Steve ; Singh, Shrijit ; Martin, Francisco ; Gines, Jonatan: *Regulated Pure Pursuit for Robot Path Tracking*. Mai 2023. – URL http://arxiv.org/abs/2305.20026. – Zugriffsdatum: 2024-04-30

[19] Makantasis, Konstantinos ; Kontorinaki, Maria ; Nikolos, Ioannis: *A Deep Reinforcement Learning Driving Policy for Autonomous Road Vehicles*. Mai 2019. – URL http://arxiv.org/abs/1905.09046. – Zugriffsdatum: 2022-07-24

[20] Nagabandi, Anusha ; Clavera, Ignasi ; Liu, Simin ; Fearing, Ronald S. ; Abbeel, Pieter ; Levine, Sergey ; Finn, Chelsea: *Learning to Adapt in Dynamic, Real-World Environments Through Meta-Reinforcement Learning*. Februar 2019. – URL http://arxiv.org/abs/1803.11347. – Zugriffsdatum: 2024-02-06

[21] Netto, M. ; Blosseville, J.-M. ; Lusetti, B. ; Mammar, S.: A New Robust Control System with Optimized Use of the Lane Detection Data for Vehicle Full Lateral Control under Strong Curvatures. In: *2006 IEEE Intelligent Transportation Systems Conference*, URL https://ieeexplore.ieee.org/document/1707416. – Zugriffsdatum: 2024-04-30, September 2006, S. 1382–1387. – ISSN 2153-0017

[22] Pan, Xingang ; Shi, Jianping ; Luo, Ping ; Wang, Xiaogang ; Tang, Xiaoou: Spatial As Deep: Spatial CNN for Traffic Scene Understanding. (2017), Dezember

[23] Park, Mingyu ; Kim, Hyeonseok ; Park, Seongkeun: A Convolutional Neural Network-Based End-to-End Self-Driving Using LiDAR and Camera Fusion: Analysis Perspectives in a Real-World Environment. In: *Electronics* 10 (2021), Januar,

Nr. 21, S. 2608. – URL https://www.mdpi.com/2079-9292/10/21/2608. – Zugriffsdatum: 2022-02-20. – ISSN 2079-9292

[24] Pomerleau, Dean A.: ALVINN: An Autonomous Land Vehicle in a Neural Network. In: *Advances in Neural Information Processing Systems* Bd. 1, Morgan-Kaufmann, 1988. – URL https://proceedings.neurips.cc/paper/1988/hash/812b4ba287f5ee0bc9d43bbf5bbe87fb-Abstract.html. – Zugriffsdatum: 2022-03-13

[25] Reports, Consumer: Active Driving Assistance Systems: Test Results and Design Recommendations. URL https://data.consumerreports.org/wp-content/uploads/2020/11/consumer-reports-active-driving-assistance-systems-november-16-2020.pdf. – Zugriffsdatum: 2023-12-05. – Forschungsbericht

[26] Riedmiller, Martin ; Montemerlo, Mike ; Dahlkamp, Hendrik: Learning to Drive a Real Car in 20 Minutes. In: *2007 Frontiers in the Convergence of Bioscience and Information Technologies*, Oktober 2007, S. 645–650

[27] Riege, Daniel: *Segmentierung von Straßenmarkierungen durch maschinelles Lernen für die Miniaturautonomie*, HAW Hamburg, Bachelorarbeit, Dezember 2021. – URL https://autosys.informatik.haw-hamburg.de/publication/2021danielriege/. – Zugriffsdatum: 2022-03-02

[28] Riege, Daniel ; Pareigis, Stephan ; Tiedemann, Tim: Real-Time Aspects of Image Segmentation of Road Markings in Miniature Autonomy. In: Unger, Herwig (Hrsg.) ; Schaible, Marcel (Hrsg.): *Real-Time and Autonomous Systems 2022*. Cham : Springer Nature Switzerland, 2023 (Lecture Notes in Networks and Systems), S. 97–107. – ISBN 978-3-031-32700-1

[29] Sallab, Ahmad E. ; Abdou, Mohammed ; Perot, Etienne ; Yogamani, Senthil: Deep Reinforcement Learning Framework for Autonomous Driving. In: *Electronic Imaging* 29 (2017), Januar, Nr. 19, S. 70–76. – URL http://arxiv.org/abs/1704.02532. – Zugriffsdatum: 2022-07-24. – ISSN 2470-1173

[30] Tiedemann, Tim ; Fuhrmann, Jonas ; Paulsen, Sebastian ; Schnirpel, Thorben ; Schönherr, Nils ; Buth, Bettina ; Pareigis, Stephan: Miniature Autonomy as One Important Testing Means in the Development of Machine Learning Methods for Autonomous Driving: How ML-based Autonomous Driving Could Be Realized on a 1:87 Scale, Januar 2019, S. 483–488

[31] Tiedemann, Tim ; Schwalb, Luk ; Kasten, Markus ; Grotkasten, Robin ; Pareigis, Stephan: Miniature Autonomy as Means to Find New Approaches in Reliable Autonomous Driving AI Method Design. In: *Frontiers in Neurorobotics* 16 (2022). – URL https://www.frontiersin.org/articles/10.3389/fnbot.2022.846355. – Zugriffsdatum: 2022-07-24. – ISSN 1662-5218

[32] Youssef, Fenjiro ; Houda, Benbrahim: Comparative Study of End-to-end Deep Learning Methods for Self-driving Car. In: *International Journal of Intelligent Systems and Applications* 12, Nr. 5, S. 15. – URL https://www.mecs-press.org/ijisa/ijisa-v12-n5/v12n5-2.html. – Zugriffsdatum: 2024-02-05

[33] Youssef, Fenjiro ; Houda, Benbrahim: Deep Reinforcement Learning with External Control: Self-Driving Car Application. In: *Proceedings of the 4th International Conference on Smart City Applications*. New York, NY, USA : Association for Computing Machinery, Oktober 2019 (SCA '19), S. 1–7. – URL https://dl.acm.org/doi/10.1145/3368756.3369038. – Zugriffsdatum: 2024-02-05. – ISBN 978-1-4503-6289-4

[34] Zhang, Qi ; Du, Tao ; Tian, Changzheng: *Self-Driving Scale Car Trained by Deep Reinforcement Learning*. Dezember 2019. – URL http://arxiv.org/abs/1909.03467. – Zugriffsdatum: 2024-02-05

[35] Zou, Qijie ; Xiong, Kang ; Fang, Qiang ; Jiang, Bohan: Deep Imitation Reinforcement Learning for Self-Driving by Vision. In: *CAAI Transactions on Intelligence Technology* 6 (2021), Nr. 4, S. 493–503. – URL https://onlinelibrary.wiley.com/doi/abs/10.1049/cit2.12025. – Zugriffsdatum: 2024-03-03. – ISSN 2468-2322

[36] Zou, Qin ; Jiang, Hanwen ; Dai, Qiyu ; Yue, Yuanhao ; Chen, Long ; Wang, Qian: Robust Lane Detection from Continuous Driving Scenes Using Deep Neural Networks. In: *IEEE Transactions on Vehicular Technology* 69 (2020), Januar, Nr. 1, S. 41–54. – URL http://arxiv.org/abs/1903.02193. – Zugriffsdatum: 2021-04-21. – ISSN 0018-9545, 1939-9359

Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original