UNIVERSITÄT
LEIPZIG

Institute of Computer Science
Faculty of Mathematics and Computer Science
Database Department

End-To-End Reinforcement Learning Training of a Convolutional Neural Network to Achieve an Autonomous Driving Agent on Tracks in Simulation

Master's Thesis

submitted by:
Georg Schneeberger

matriculation number:
3707914

Supervisor:
Prof. Dr. Erhard Rahm
Dr. Thomas Burghardt
M. Sc. Martin Lorenz

Abstract

This master thesis contributes to the domain of autonomous driving and Reinforcement Learning (RL). Agents are developed to solve a simplified autonomous driving task in simulation. The agent use Convolutional Neural Network (CNN) policies that takes raw camera inputs. The policies are trained in an end-to-end manner using RL. The driving task is modelled after physical experiments conducted at the Scads.AI research facility.

This thesis builds upon previous work at the Scads.AI by [1], and develops an agent that has to solve the same autonomous driving task. The agent from previous work was not able to reliably complete all tracks. Its performance suffered further under changing light conditions, motivating the research goals of this thesis.

An RL agent is developed to traverse simulated tracks of varying difficulty and light settings. The agent's policy consists of a CNN that is trained end-to-end using RL. The inputs to the Neural Network (NN) are camera images from the perspective of the agent. The thesis investigates different training approaches to develop a powerful agent, such as the reward functions and data collection parameters. The most promising policy is used to answer the three research questions.

As a result of the experimentation, a policy is developed that is able to navigate the tracks successfully. The policy has success rates of above 95% for all investigated tracks and light setting configurations. Further experiments show that the policy can be transfered to a physical robot in theory. The thesis shows that it is possible to train a CNN agent to solve the investigated autonomous driving task using end-to-end RL.

Georg Schneeberger
3707914

Contents

Acronyms

List of Figures

List of Tables

1. Introduction

Recent advancements in Artificial Intelligence (AI) technology have made it possible to develop automated solutions for a wide range of tasks that were previously thought to be too complex and unfit for machines to solve. Most notable in recent few years is the introduction of diffusion image models and large language models. These technologies were well received and moved AI tools into public discourse. All over the world people have recognized the potential of AI technologies and are now using them in their daily life and at work.

AI has already been of great importance in academia and industry for a long time. AI has been proved useful in many different fields, such as image recognition, natural language processing, and robotics. This encourages researchers and industry to further develop and use AI in their work. A promising domain for the application of AI is autonomous driving.

The development of autonomous vehicles promises to greatly reduce the number of traffic accidents and transportation cost [2]. The development of autonomous driving could have further downstream effects on our society and industry, such as improved logistic and transportation systems. As a result, researchers and private enterprises from all over the globe are making progress towards fully autonomous driving agents. Many companies started to integrate adaptive cruise control and lane centering assistance in their products [3]. Due to the recent developments in AI and the very high complexity of the task of autonomous driving, AI often plays a big role in these systems [4].

Predictions for the future of autonomous driving have been very optimistic and although huge progress has been made, the task of fully autonomous driving is still far from being solved [5]. This thesis aims at contributing to the research in this field by applying Reinforcement Learning (RL) to autonomous driving agents in a simulated environment. This work builds upon the work of Schaller [1] and will use the same task and evaluation metrics. This thesis focusses on improving the agent's resilience to changing light conditions by training a Convolutional Neural Network (CNN) end-to-end using RL.

2. Research Goals

The goal of this thesis is to contribute in the domain of autonomous driving by investigating the use of RL to train an autonomous driving agent that is resilient to changes in light conditions. The agent is evaluated on simulated tracks that consist of a series of goals indicated by two blocks, a tracks is successfully completed if the agents drives through all goals in order 2.1. This thesis builds upon previous work at the ScaDS.AI Schaller [1] and uses the same tracks and task specifications. The agent from previous work was not able to reliably complete tracks under changing light conditions, motivating the research goals.

The agent and the policy are designed to be resilient to light changes. To achieve this, the policy is trained using RL in a simulated environment with changing light conditions. The policy consists of a CNN, the inputs to the CNN are camera images from the perspective of the agent 2.1. Preprocessing steps are applied to the camera images to improve the performance of the policy under changing light conditions. The policy is trained end-to-end using an RL algorithm. This allows the policy to learn the relevant features from the camera images itself. Different training approaches and preprocessing steps to develop a powerful agent are investigated.

2.1. Question 1 - Is it possible to train an autonomous driving agent consisting of a CNN policy with end-to-end reinforcement learning to reliably solve the tracks of all difficulty levels?

The previous work by Schaller [1] showed that it is possible to train an agent using RL to solve the evaluation tracks, however the trained agents were not successful in reliably traversing the tracks of higher difficulty levels. The agents developed in previous work utilized an extensive preprocessing pipeline to extract the relevant information from the camera images for processing by the policy. The evaulation by Flach [6] showed that the preprocessing pipeline's performance depends heavily on the light settings of the environment.

This thesis will use a CNN network policy that is trained end-to-end using RL. It has already been shown to be possible to train an agent to solve the autnomous driving task using RL. However the agents developed in previous work used different preprocessing steps and policies compared to the agents developed here.

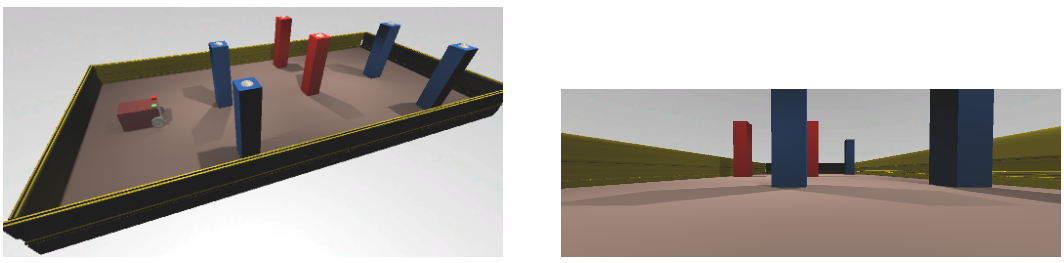


Figure 2.1.: Example image of a track and the agent's camera

Due to these differences in implementation and as a prerequisite for question 2 and 3, it is first important to investigate if it is possible to train a CNN policy to reliably solve the tracks of all difficulty levels. This raises question 1: Is it possible to train an autonomous driving agent consisting of a CNN policy with end-to-end reinforcement learning to reliably solve the tracks of all difficulty levels?

The question will be answered by developing agents based on related work. The training process of the agents will use common practices from RL adapted to the specific task. The developed agent's *success_rate* and *collision_rate* will be evaluated on all difficulty settings to answer the question.

2.2. Question 2 - Is it possible to use an end-to-end trained CNN policy to make the agent robust to changing light conditions?

Question 1 investigates if it is possible to develop a successful CNN-based agent solving the tracks of all difficulty levels. Question 2 then investigates if it is possible to make the CNN-based agents robust to changing light conditions.

The agents developed for question 1 will be used as a basis for the agents developed in this question. The agents will be augmented with preprocessing steps that improve the performance of the agents under changing light conditions. These preprocessing steps will be applied to the camera images before they are processed by the policy. The agent's policy will be trained in a simulated environment with changing light conditions to help the agent generalize and learn.

Similarly the *success_rate* and *collision_rate* will be primarily used to evaluate and compare the agent's performance. The performance difference between different light conditions will be used to answer the question. If the performance for all light conditions is comparable to the performance of agents in question 1, the agents can be considered robust to changing light conditions.

2.3. Question 3 - Is it possible to use a NN policy that can be executed in real-time on a physical robot?

One goal of the ongoing research at the ScaDS.AI is to build physical robots for demonstration and research purposes Flach [6]. The robots are based on the NVIDIA JetBot platform [7]. They are equiped with a camera, wheels and a small on-board computer. A future goal is to transfer trained policies onto these robots and execute them in real-life. However the limited processing power of these robots might not be sufficient for more complex agents that utilize Neural Networks (NNs). This raises question 3 - Is it possible to use a NN policy that can be executed in real-time on a physical robot?

The question will be answered by investigating the processing power required to run the preprocessing steps and NNs used in the agents. This will be evaluated empirically by creating recordings

of the agents in simulations. The recordings are then replayed on the physical hardware. The evaluation checks if the physical hardware is able to reproduce the preprocessing steps and policy from the replays in real-time.

3. Related Work

The development of self-driving cars represents one of the most intricate and ambitious challenges in the field of engineering, robotics and Artificial Intelligence (AI). The complexity and variability of real-world driving environments make this task extremely difficult. The environments include unpredictable actors, such as other drivers, pedestrians, and animals, as well as changing weather and road conditions. The environments include uncounted edge cases that are difficult to anticipate and code for explicitly. Consequently, many researchers and institutions include advanced Machine Learning (ML) techniques in their approaches towards solving autonomous driving.

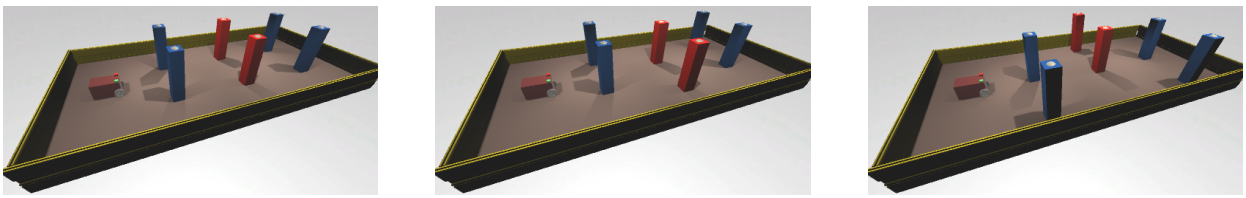
The development of autonomous driving systems started with driver assistance systems in the 1970s. These systems focus on assisting the driver in specific tasks, such as lane keeping, adaptive cruise control, and parking. This reduces the problem complexity. However modern self-driving systems aim to achieve full autonomy. For example the Tesla autopilot is capable of driving in many environments without human intervention.

This thesis will use Reinforcement Learning in the training of an autonomous driving agent. The agent is equiped with a single camera sensor. The agent's behaviour is controlled by a Convolutional Neural Network (CNN) policy that processes the visual input. The policy has to learn a specific self driving task with reduced complexity. The self-driving task is identical to previous research at the ScaDS.AI by Schaller [1]. However the agent's design and training is very different. The task consists of a simplified environment. The environment consists of an enclosed arena with different tracks that the agent has to complete. The tracks consist of a series of goals that the agent has to drive through. The tracks are grouped in three difficulty levels 3.1. The environment is simulated with three light settings, bright, standard and dark 3.2. The trained agent has to learn to navigate at all difficulty levels and light settings.

Research relating to Reinforcement Learning (RL) algorithms, CNNs and self-driving will be reviewed. The review will highlight the key ideas and approaches that will be used in this thesis to build an autonomous driving agent resistant towards light changes.

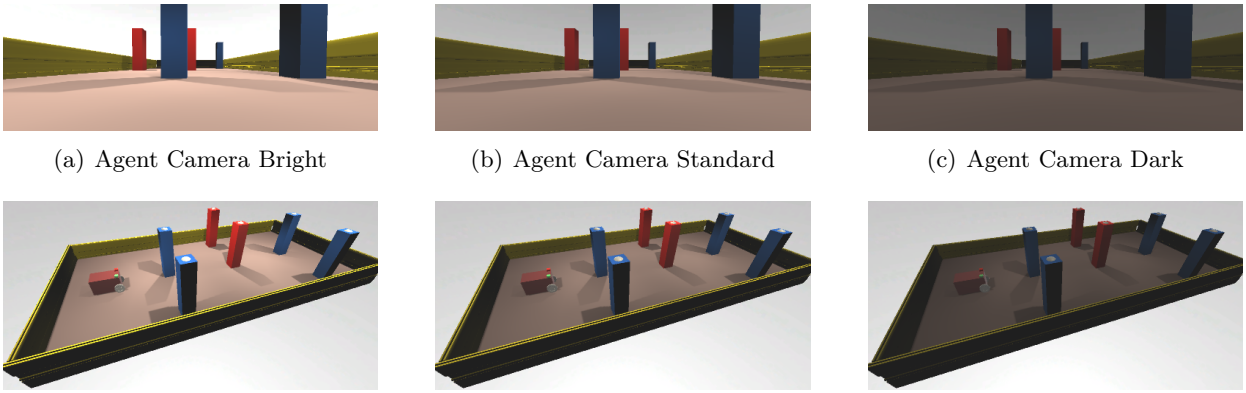
3.1. Self-Driving

Neural Networks (NNs) have been used in the domain of self driving for a long time. Pomerleau [8] developed one for the first self driving systems with NNs in 1989. The system was developed for



(a) Easy (b) Medium (c) Hard

Figure 3.1.: Example tracks for each difficulty level



(a) Agent Camera Bright (b) Agent Camera Standard (c) Agent Camera Dark

(d) Arena Bright (e) Arena Standard (f) Arena Dark

Figure 3.2.: Light settings

road following. The system used a camera image and a laser range finder as input. Compared to current network architectures, they used a very small fully connected NN. The network was trained using synthetic data. The data consisted of generated road images and steering commands. The network was trained using backpropagation to reproduce the steering commands. This approach did not use RL. The system was able to follow roads in real-life under certain conditions. Pomerleau [8] highlighted the advantages of NNs for self-driving. The training data, not the programmer, determines the salient image features. This has proven to be true with the recent success of CNNs in the domain of self-driving Kiran et al. [4].

There has been a lot of progress in the domain of self-driving in recent years. Sophisticated self-driving algorithms consist of many hardware and software components to achieve satisfying performance. Hardware components include various imaging approches such as Radar, Lidar and cameras [4]. Termometers and other hardware components are also used, for example to determine weather conditions. Software components might include separate object detection, occupancy and planning. For example Tesla's self-driving builds these components on top of a shared backbone that uses CNNs [9].

Self-driving in a real world environment is a very complex task, especially when including other traffic participants. Modern self driving agents consist of multiple complex components that interact with each other [4]. Multiple interacting components are shown in figure 3.3. The Scene Understanding components build a model of the current surroundings. The Decision making & Planning components use this model to decide and execute the next actions. The components are implemented using different algorithms and NNs. This can require seperate data collection, training and evaluation processes for the components. An example is the path prediction by Tesla Tesla [10] in 2019. This component is trained using Imitation Learning (IL) on a large dataset of human driving behaviour.

An autonomous driving system with similarly complex components is not feasible for this thesis. The simplified nature of the task and previous work by Schaller [1] suggests this is not required.

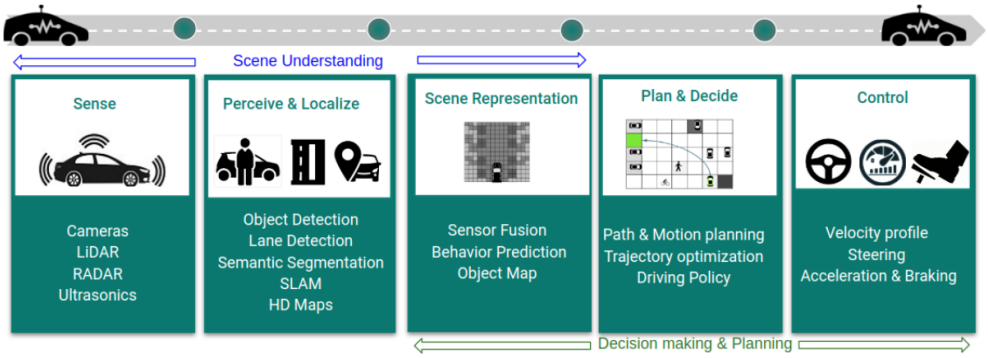


Figure 3.3.: Components in a modern autonomous driving system's pipeline. Image from Kiran et al. [4]

3.1.1. Previous Work at ScaDS.AI

This thesis builds directly upon the work of König [11], Flach [6] and Schaller [1]. König [11] built a self-driving agent that was trained to avoid collisions in a simulated arena. The agent used a hand-crafted preprocessing pipeline to extract features from visual input. The features represent the obstacles that the agent has to avoid. The agent's behaviour was controlled by a policy that consisted of a NN. This network used the extracted features as inputs. It was trained using an evolutionary approach in simulation .

Flach [6] investigated the feasibility of transferring this agent to the real world. The research showcased many challenges. The challenges are caused by differences between the simulated and real-life environments, the Simulation-To-Reality gap. The most notable problem was the object recognition part. The preprocessing pipeline had difficulties recognizing the objects in the real world. This results in further problems for the agent, since the agent's policy is based on the extracted features.

Schaller [1] investigated a different task than the two previous papers. An agent was trained to traverse a track by driving through a sequence of goals. The same tracks are used in this paper. The agent used a preprocessing pipeline to extract object features similar to König [11]. The NN policy was trained using the Proximal Policy Optimization (PPO) RL algorithm developed by Schulman et al. [12]. The agent could traverse the tracks, its performance decreased for tracks of higher difficulty. Further evaluation of the agent under different light settings showed that the agent is not robust against changing light conditions. The preprocessing pipeline was not able to extract the necessary features reliably under different light settings. This resulted in a performance collapse.

The instability of the hand-crafted preprocessing pipeline and promising results by CNNs from other RL researchers in the domain of self-driving Barla [13] motivate the choice of CNNs as the feature extraction method in this thesis.

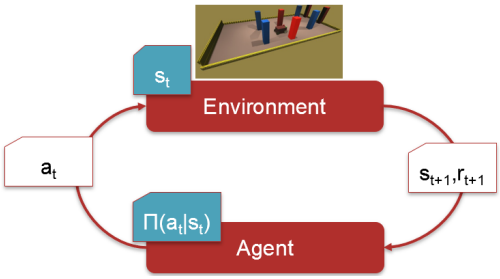


Figure 3.4.: RL Training Cycle: The agent selects action a_t based on policy $\pi(a_t|s_t)$ at state s_t and receives the next state s_{t+1} and rewards r_{t+1} from the environment. The states, actions and observed rewards are used to update the policy.

3.2. Reinforcement Learning

3.2.1. Introduction to RL

RL algorithms have been around for a long time, but only recently have they been able to achieve superhuman performance in games and control tasks [14]. RL algorithms formalize the problem as consisting of an environment and a policy π. The environment consists of a state space, an action space and a reward function that takes state-action pairs as input. Reward functions assign positive rewards to actions that are deemed to be desirable by the environment designers, for example scoring a goal in a football match. Reward function can also assign negative rewards to undesirable actions, for example collisions in a driving simulation.

The goal of RL algorithms is to build an agent that interacts with its environment and maximizes the cumulative reward over time. The policy selects actions given an observation. It controls the agents behaviour [15]. It is trained by the RL algorithm to learn the desired behaviour. The trained policy can then be used to solve problems in the environment.

The agent interacts with the environment during the training phase of RL algorithms. The policy takes some representation of the environment's state as input and selects actions to execute in the environment. The selected actions are executed in the environment by the agent which results in a new state. The reward function assigns rewards to these state transitions. The observed rewards are then used to update the policy 3.4.

3.2.2. Classification of RL Algorithms

RL algorithms are classified into two major groups. RL algorithms that use a model of the environment are called model-based algorithms, algorithms without such models are called model-free algorithms. Algorithms from both groups have been successfully used in a wide range of applications, model-based algorithms are often much more complex but have been shown to be successful at many tasks that require planning [16]. Model-free approaches are often simpler and more flexible. They have shown great success in various control tasks by Mnih et al. [14].

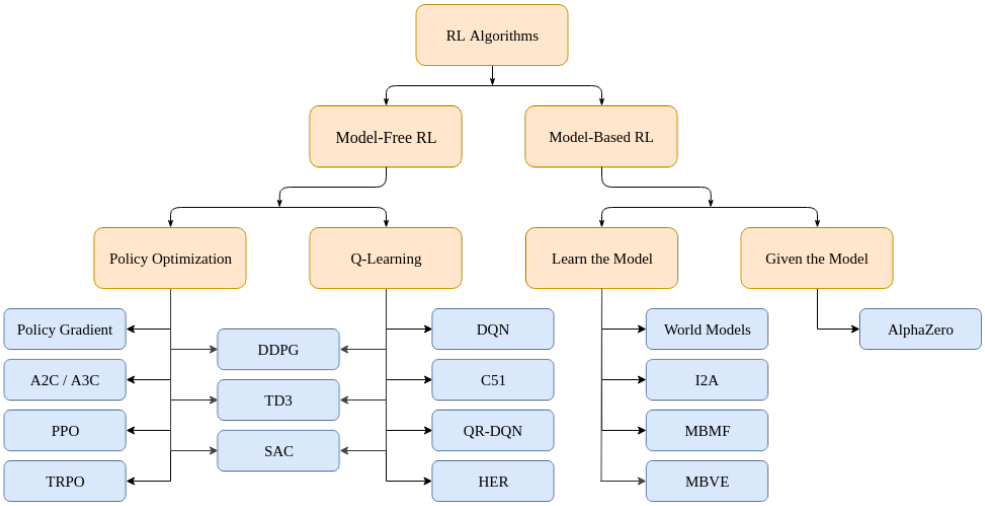


Figure 3.5.: Taxonomy of RL algorithms from OpenAI's Spinning Up course [17]

3.2.2.1. Model-Free RL Algorithms

Model-free RL algorithms take actions in an environment without an internal representation of the environment. Model-free algorithms learn from direct interactions with the environment.

Value-based algorithms Model free algorithms can be further divided into two families. The first family are value-based approaches. These algorithms learn a function that assigns state-action pairs a value. This function is called the value function. This value represents the expected future reward Q.

The policy is not trained directly. The policy selects actions based on this value function instead. The state-action pair with the highest Q-value is selected for a given state.

The training process of the value function is done by updating the Q-values based on the observed rewards. The Q-learning algorithm is an early and common example of value-based algorithms Sutton and Barto [15]. The Q-learning algorithm updates the values based on the observed rewards and the maximum Q-value of the next state:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t))$$

α is the learning rate. It determines how quick values are changed. γ is the discount factor of future rewards.

The Q-learning algorithm can use a table to store the value function. The table contains one cell for each state-action pair and stores the Q-value. For many problems the use of these tables is not feasible due to the amount of state-action pairs [15]. Many extensions to the algorithms have been developed, for example deep q-learning. Deep q-learning was developed by Mnih et al. [14] and uses a deep NN to approximate the Q-values. The network learns to predict the Q-values for state-action pairs. NNs are general approximators. They can learn to generalize and return accurate value predictions even for previously unseen states.

Value-based RL algorithms have been used to great success for control tasks Sutton and Barto [15]. However they will not be used in this thesis, as they require discrete action spaces. The environment in this thesis consists of a continuous action space. An action space can be discretized for use by value-based algorithms. However this can lead to a loss of fidelity [4].

Policy-based algorithms The other family are policy-based algorithms. These algorithms optimize the policy directly instead of the values associated with states or state-action pairs. Instead of computing the learned probability of each action, the policy learns the statistics of the action distribution. Given a scalar action space. The action distribution can be represented as a gaussian probability Distribution:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

The policy can then be defined as the normal probability density over a real-valued scalar action. The mean and standard deviation of the distribution are given by parametric function approximators that depend on the state s. The mean is given by $\mu(s, \theta)$ and the standard deviation by $\sigma(s, \theta)$. The policy is then defined as:

$$\pi(a|s, \theta) = \frac{1}{\sigma(s, \theta)\sqrt{2\pi}} e^{-\frac{(a-\mu(s,\theta))^2}{2\sigma(s,\theta)^2}}$$

The parametric function approximators $\mu(s, \theta)$ and $\sigma(s, \theta)$ are trained by the RL algorithm. Any function approximator can be used, for example a NN with parameters μ Sutton and Barto [15].

The action distribution can be extended to multi-dimensional action spaces. This action distribution can be represented as a multivariate gaussian distribution. The function approximator then outputs the mean and covariance matrix of the distribution. This makes policy-based RL algorithms very flexible. They can be used for multi-dimensional continuous action spaces.

The policy can be updated using the gradient of the expected rewards with respect to the policy parameters. Algorithms that use this approach are called policy gradient algorithms.

Actor-Critic Algorithms Actor-Critic algorithms combine policy and value-based approaches. Actor-Critic approaches use a policy and a value function. The policy is used to select actions similar to policy-based approaches. The value function represents the expected future reward of a state. The policy is updated during training using this value estimate.

The PPO algorithm is a policy gradient Actor-Critic algorithm. It was developed by Schulman et al. [12] to improve the stability of policy-based algorithms. The PPO algorithm restricts the size of policy changes caused by parameter updates. This ensures the policy does not change drastically and improves the stability during training. PPO is currently one of the most popular algorithms in RL. It has proven to be very succesful in many continuous control tasks [12]. Schaller [1] used the PPO algorithm to train agents for the task that is investigated in this thesis.

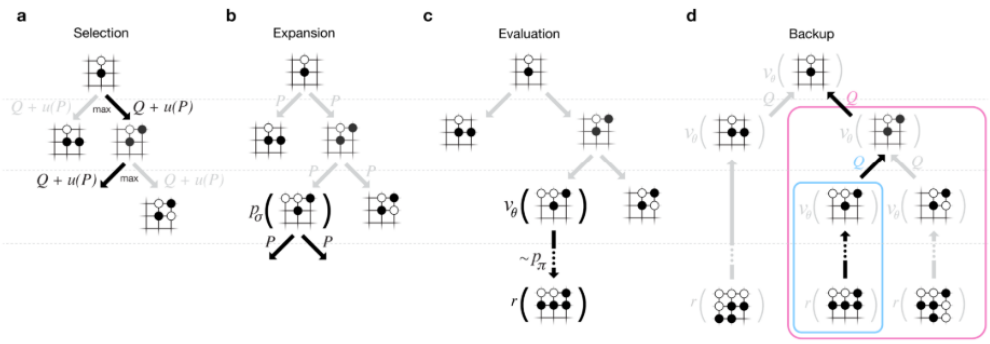


Figure 3.6.: AlphaGo Monte-Carlo tree search. Image from [18]

3.2.2.2. Model-Based RL Algorithms

Model-based RL algorithms use an internal model of the environment to predict future states and rewards. This internal model can be used to predict the outcome of actions, improving the overall performance of the algorithm. Model-based algorithms use the internal model to plan ahead. The internal model can be either provided by the programmer or learned by the RL algorithm.

Model-based approaches use the environment model to plan ahead and evaluate actions based on this planning. One example of a model-based algorithm is the Monte-Carlo tree search algorithm. The Monte-Carlo tree search algorithm builds a tree of possible actions and their outcomes. The internal model is used to simulate the actions and outcomes. The tree is then used to evaluate the possible actions and select the best one based on the simulated outcomes. In the context of RL the internal model represents the state transition function and the reward function.

AlphaGo by Silver et al. [18] is an example of a model-based algorithm with a provided environment model. AlphaGo was able to achieve breakthrough performace for the deterministic 2 player game of Go. AlphaGo uses a combination of a CNN policy and a Monte-Carlo tree search algorithm to evaluate the possible actions at a given state. The policy is used for the evaluation of states and the action selection in the search algorithm. The provided environment model simulates the states and actions along the tree paths 3.6.

The internal model can also be learned by the RL algorithm. Schrittwieser et al. [16] modified the algorithm of Silver et al. [18] to learn the internal model. The modified algorithm was applied to different kinds of problems, including single player continuous state and action spaces.

3.2.2.3. Comparison of Model-Free and Model-Based Algorithms

State of the art model-free and model-based algorithms have been used to solve all kind of RL tasks. Both approaches can be applied to the task of autonomous driving. The model-free PPO algorithm is used to train the agents in this thesis for a number of reasons. Model-free approaches are generally much simpler to implement since they do not require an internal model. Model-free algorithms also require less resources for the training and evaluation process.

3.2.3. CNNs for RL

CNNs are a NN architecture specifically developed for processing image data, they consist of a number of filters and a fully connected NN. The filters are applied to the image in a sliding window fashion. The filters detect patterns in the image such as edges and corners. Multiple successive applications of such filters enable the network to learn hierarchical information and recognize more complex structures. The fully connected NN analyses the results of the filters and makes the final prediction Sutton and Barto [15].

CNNs are often used in RL since RL problems often require an agent to process visual input. Compared to other feature extraction methods, CNNs can be trained end-to-end using the loss functions defined by the RL algorithms. CNNs can learn what features are important for the task at hand. Therefore a CNN will be used to process the camera images instead of a hand-crafted feature extraction method.

CNNs can take the raw camera/simulation images as input. Preprocessing steps are often applied to these images, e.g. grayscaling by Mnih et al. [14].

3.2.3.1. Preprocessing Steps for CNNs

There are many reasons for applying preprocessing steps to images for the use by CNNs. The images can be preprocessed to make the processing less computationally demanding, both in terms of processing speed and memory requirements. Preprocessing steps can also be applied with the goal of improving the performance of the NN during training and evaluation.

Preprocessing steps can change the image's height, width and channel dimensions. The datatype of an image can also be changed. The image's pixel values can also be changed.

The most common preprocessing step is grayscaling, the image's RGB color values are converted to a single grayscale value. This reduces the image's channel dimension from 3 to 1. Grayscaling reduces the computational requirements in terms of processing and memory. Furthermore it can improve the performance of the CNN. Grayscaling can also improve the network's generalization to different environments. The network does not have to learn to recognize objects based on their color. The grayscaling step removes color information from the image. This can be a disadvantage if the color information is important for the task at hand. Mnih et al. [14] used the grayscaling preprocessing step to make the processing less computationally demanding.

The image's width and height dimensions can be changed with different preprocessing steps. Images can be cropped to a smaller size. Cropping can also remove irrelevant parts of the image or change the aspect ratio of the image. Mnih et al. [14] used cropping to achieve square images for GPU processing. Images can also be resized without significantly changing the image's semantic content using upsampling or downsampling. Similar to grayscaling the reduction of the image size results in less memory and processing requirements.

Normalization is another common preprocessing step. The pixel values of the image are rescaled. The rescaling can be done by multiplication with a factor, for example from the range $[0, 255]$ to the

ranges $[0, 1]$ or $[-1, -1]$. Another common normalization approach standardizes the pixels to have a mean of 0 and a standard deviation of 1. Normalization approaches can improve the performance of the CNN during training, [19].

Data augmentation can be applied to images during the training phase of the CNN. This can help in improving the generalization of the CNN by making it robust to variations in the input data, [20].

Preprocessing steps that increase the contrast of images can increase the performance of CNN algorithms. The increased contrast can make the image features more distinguishable. This can increase the CNN performance. Vijayalakshmi et al. [21] review different approaches to increasing the image contrast, they show that contrast increasing steps can improve the performace of CNNs.

3.2.4. Extensions to RL

3.2.4.1. Reward Shaping

Reward shaping involves modifying the reward signal to guide the learning process more effectively. The term shaping comes from psychology and describes the idea of rewarding all behaviour that leads to the desired behaviour [15].

Reward signals are often sparse in RL tasks. Sparse reward signals assign non-zero rewards only in a few steps and can make learning difficult. This makes it difficult for the agent to learn which actions lead to future rewards.

Reward shaping allows for the creation of a shaped reward function that gives frequent rewards for actions that lead to the desired behaviour. The agent can learn desired behaviours faster and more efficiently.

Reward shaping can lead to unintended behaviour if the shaped reward function is not aligned with intended reward function. The agent can learn to exploit the shaped reward function and not learn the desired behaviour, Kiran et al. [4]. To avoid this problem, the shaped reward function can be changed over time to the intended reward function, Sutton and Barto [15].

3.2.4.2. Memory Mechanisms

RL algorithms are often employed in dynamic environments, such as self-driving. The agent has to remember past experiences to have an accurate representation of the current dynamic environment state. Various memory mechanisms can be used by the agent to make decisions.

Memory mechanisms typically store a fixed or variable amount of past observations. The stored observations are used together with the current observation to make decisions.

All NN architectures are able to process sequences of fixed length. The sequence of stored and current observations is concatenated to produce a fixed size input for the NN. This was used in the Deep Q-Network (DQN) algorithm by Mnih et al. [14]. This implementation of the memory

mechanism was also used by Schaller [1] to build an agent that can traverse the tracks in the arena.

Recurrent Neural Networks (RNNs) can be used to process sequences with varying length. The sequence of stored and current observations are fed into the RNN to produce a representation of the environment state that includes past observations. Long Short-Term Memorys (LSTMs) are a special version of RNNs. Wang and Chan [22] used an LSTM agent act in a dynamic ramp merging scenario.

3.2.4.3. Domain Adaptation

NNs are often trained in a simulated environment and then transferred to the real world. The NN often perform poorly in the real world due to differences between the environments. This difference is called the Simulation-To-Reality (S2R) gap. The S2R gap has been studied extensively by researchers in the field of computer vision [23] and RL [24]. The S2R gap can be reduced by using domain adaptation techniques.

The S2R gap can be reduced matching the simulated environment closely to reality. This can be done using high quality RGB renderings of the real-world environment for image processing agents. A more successful approach was the inclusion of additional depth sensors in the simulated environment.

Trained models can be fine-tuned to the real world environment. The model is trained in simulation. The trained model is fine-tuned with a dataset of real-world data. Ghadirzadeh et al. [23] used this approach to train a robotic agent in simulation using RL and then fine-tuned it in the real world.

Another domain adaptation approach tries to reduce the S2R gap by training the agent on a diverse set of environment variations. The agent is trained in a wide range of environments that are generated by randomizing the environment's parameters. For example the object textures and the agent motor power. The agent learns to generalize to the environment variations. The agent is transferred to the real world without fine-tuning. In the best case, the real-world appears to the model as just another variation. Tobin et al. [24] used this approach to train a robotic object focusing agent in simulation.

3.3. Imitation Learning for Self-Driving

As described before it is difficult to build self-driving agents for real world environments due to the environment complexity. The amount of complex edge cases make it very difficult to programmatically define the agent's behaviour. ML algorithms such as RL can be used to control the agent behaviour instead. IL is another approach to training an agent that interacts with its environment. IL requires a dataset that demonstrates the desired behaviour. The agent is trained on the dataset to mimic the expert behaviour.

In RL the programmer has to define a reward function that the agent uses to learn and improve its behaviour. In IL the agent learns exclusively from the dataset. As a result this dataset has to include a wide range of scenarios to produce a reliable agent that can handle edge cases.

There are two common approaches to training an agent with IL, Behavioural Cloning (BC) and Inverse Reinforcement Learning (IRL). BC is the simpler approach. The agent learns to mimic the expert behaviour directly. This is similar to supervised learning. IRL is more complex. The reward function is learned from the expert demonstrations first. The agent is then trained to maximize this reward function using RL.

Bojarski Bojarski et al. [25] used IL to train a self-driving lane following agent for real-world environments. This agent consists of a CNN that processes the camera images and predicts the steering angle directly. This NN is trained end-to-end to reproduce steering behaviour from recorded data. They demonstrate that IL is a viable approach for developing an autonomous driving steering agent without the need for multiple components such as object detection and path planning.

Tesla used IL to train the path prediction component of their self-driving systems. The fleet of Tesla vehicles allows them to collect a large amount of representative data in the real world. The reference dataset was generated from recordings of human Tesla drivers Tesla [10].

IL and RL can be combined to improve the training of the agent. The RL process generates samples via interaction between agent and environment. These samples and the expert behaviour dataset are used together to train the agent policy. Li and Okhrin [26] used this approach to train a car following agent.

3.4. Simulation for RL and Self-Driving

Simulations play a huge role in RL and the development of self-driving agents. Simulations provide a huge number of benefits over real world experiments. They are much cheaper and faster to run than real world experiments. Furthermore they can be run in parallel. In addition the programmers have direct and perfect control over the environment, as such programmers can for example change the simulation speed. This allows for fast experimentation and training of RL agents. Simulations also allow for the creation of scenarios that are not feasible in the real world. This is especially useful for RL agents that are trained to avoid collisions. Simulations also allow for the creation of ground truths such as perfect sensor data and object bounding boxes.

Interest in self-driving has also led to the development of dedicated simulators, such as the Carla by Dosovitskiy et al. [27] and the AirSim by Shah et al. [28] simulator. The Carla simulator provides researchers with useful features such as weather control and ground truths for object detection and segmentation.

Simulated environments often serve as baselines for RL algorithms, most famous are the atari games by Mnih et al. [14]. Towers et al. [29] developed the Python Gymnasium API for easy reuse and comparison of RL algorithms for different problems. The Gymnasium API defines an interface that can be used to model tasks as RL problems. A wide range of RL frameworks support the Gymnasium

API, for example Google's dopamine by Castro et al. [30] and OpenAI's Stable Baselines 3 (SB3) by Raffin et al. [31].

Advanced simulations like the Unity engine [32] and the physics simulator MuJoCo [33] can be integrated with the Gymnasium API. Some dedicated RL frameworks integrate directly with simulation engines. Schaller [1] used the ML-Agents framework Cohen et al. [34] to train the self-driving agent in Unity directly.

Unity will be used for the simulation in this thesis, the simulation will be integrated with the Python Gymnasium API and PPO algorithm. Compared to the ML-Agents framework, this allows for more flexibility and control over the simulation and training process.

4. Methods

This chapter explains how the environment, observation preprocessing, policy and training algorithm are implemented. The environment is implemented in the Unity game engine, this includes the arena, the jetbot agent and the reward function. The observation preprocessing, policy and RL training algorithm are implemented in Python.

The environment is describes first. The Python algorithm and Unity communicate via Remote Procedure Call (RPC). The arena, the jetbot agent and the developed reward function are then explained in detail.

The observation processing transforms the raw camera images from the Unity environment for processing by the policy. The observation processing steps apply image preprocessing and a memory mechanism to improve the performance of the policy.

The policy network's structure is then explained. This is followed by a section that ties the environment, observation processing and policy together. The interactions between these parts are shown visually in 4.19.

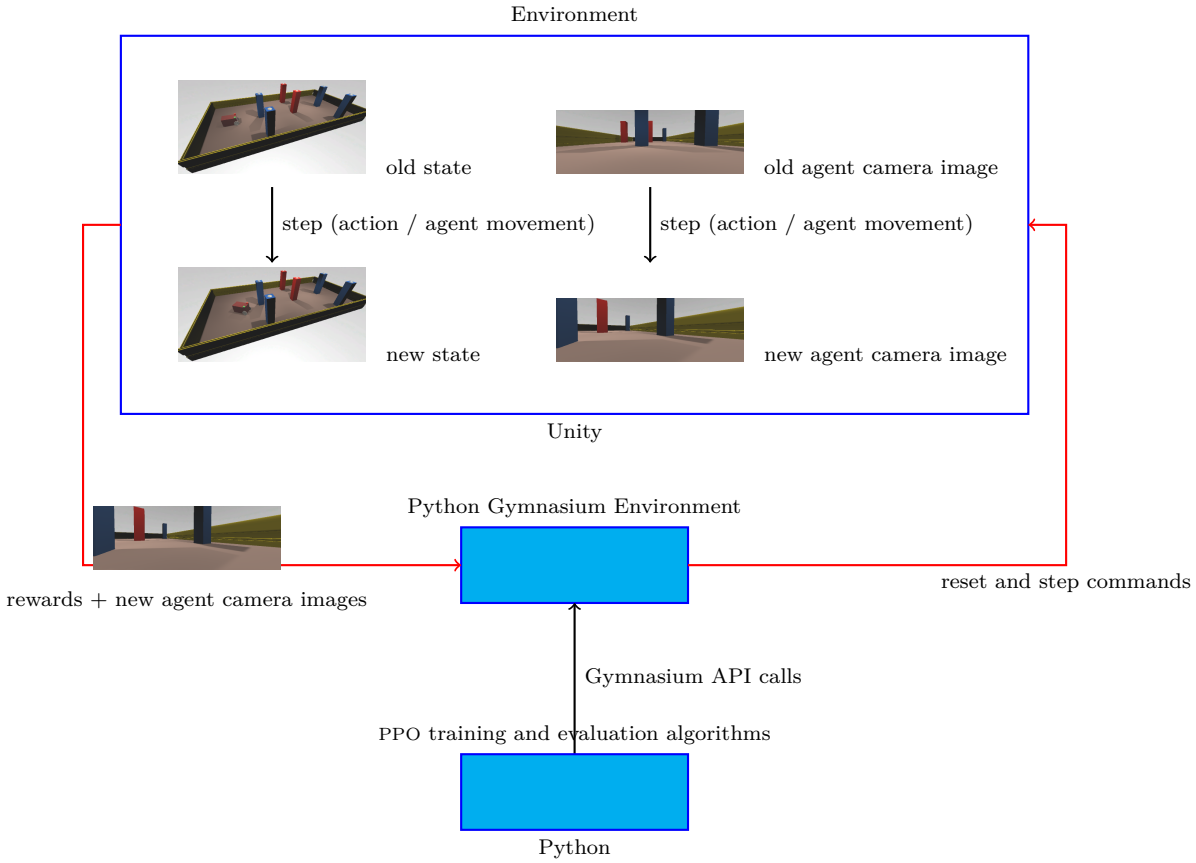
Finally the training algorithm is explained. The training algorithm is based on the PPO algorithm. The training algorithm is implemented in Python and uses the SB3 library [31].

4.1. Environment Description

4.1.1. Environment Simulation

The environment is simulated in the Unity game engine. The RL algorithm is implemented in Python and builds upon the SB3 library. This library is able to train a proximal policy algorithm on any environment that implement the Gymnasium API. The Unity environment is wrapped in a Python class that implements the Gymnasium API. This class is responsible for the communication between the RL algorithm and the Unity engine. The communication is implemented using RPC with the Peaceful Pie Perkins [35] library.

The interaction is shown in 4.1.



red: RPC calls and returns

Figure 4.1.: Communication between Python and Unity

Parallel environment procesing The SB3 library supports the parallel processing of multiple environments. The config parameter n_envs specifies how many environments are simulated in parallel. The environments are simulated in the same Unity instance, see 4.2. The seperation between the Unity engine and the Python algorithm requires RPC calls for every environment step and reset. This can slow the training process. The overhead of sending calls to the Unity engine is reduced by bundling the calls for all environments in one RPC call. The *use_bundled_calls* config parameter enables the bundling of calls.

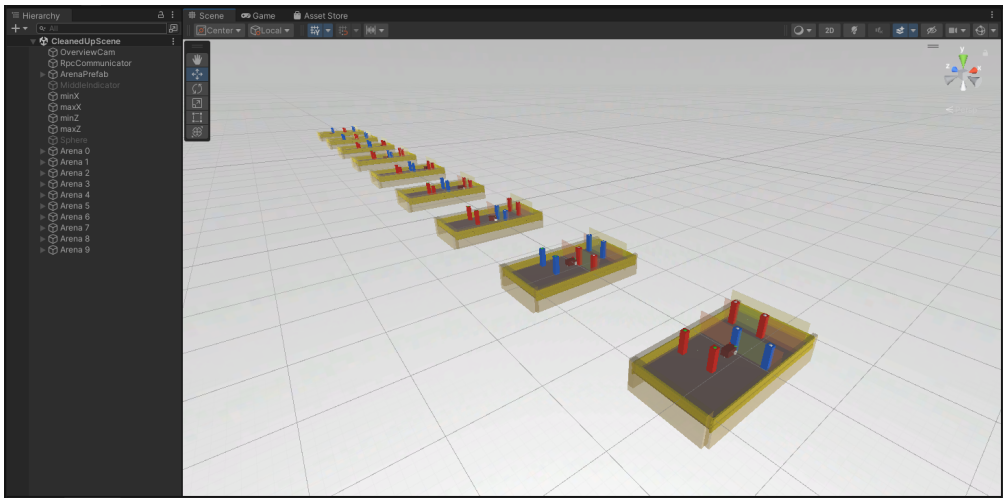


Figure 4.2.: Parallel simulation of multiple environments in one Unity instance

Environment steps The step function of the Unity environment is non-blocking. This means that Unity returns the call before the entire step transition is complete. The python algorithm can continue without waiting for the step to complete. This interaction is shown in 4.3.

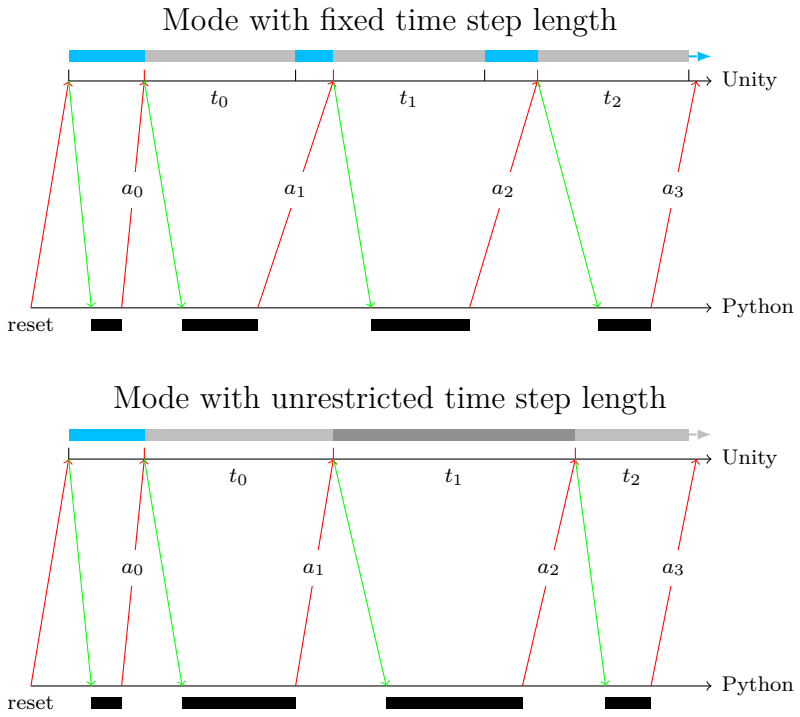
In conventional RL each step transition results in a new observation that is used to predict the next action. Since the Unity environment returns the call before the current step is finished, this is not possible. In this implementation the step call to Unity returns the observation from the start of the step instead. This observation does not capture the changes that have occurred in the environment during the step transition. The full changes are then visible to the agent when the next step call has been completed.

In summary, the returned observation is delayed by one step. The duration of the steps is dictated by the $fixedTimestepsLength$ environment parameter 4.1.4.3. Short step durations result in accurate observations, since their delays are small. Experiments show that the policy can learn to deal with this delay. The delay is negligeable due to the $fixedTimestepsLength$ of 0.3 seconds used in this thesis 4.4.

Alternatively there is the parameter use_fresh_obs. If this parameter is set to true, a new agent camera image is requested from Unity via another RPC call before predicting the next action. This can be useful if the policy is sensitive to the observation delay. However this increases the amount of RPC calls and slows down the training and evaluation process.

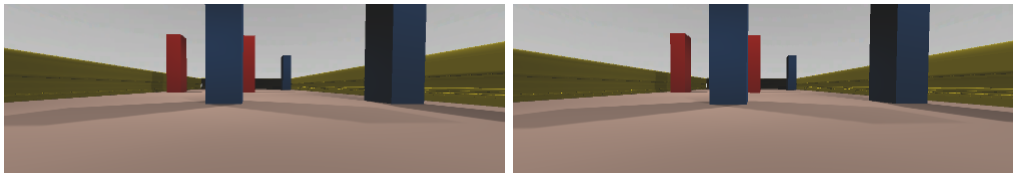
4.1.2. Arena Description

This section describes the simulated environment and agent in detail. The environment is a 3D simulation of a physical arena at the ScaDS.AI research facility. The simulated arena consists of a rectangular platform with enclosing walls. Simulated light sources illuminate the platform from above. The goal of our agent is to complete tracks in the arena by traversing the track's goals in order. Each goal consists of 2 cuboid pillars of the same colour. The pillars are coloured red or blue. The goals' colours alternate in the track. The distance between the pillars is fixed and the same for all goals. The positions of the goals depends on the episode's track. The tracks are grouped by the



grey: step simulation
blue: waiting for step command, no simulation
red: reset/step command
green: reset/step call return including agent camera image and rewards of all completed steps
black: processing of agent camera image and policy

Figure 4.3.: Python Unity Step call timeline



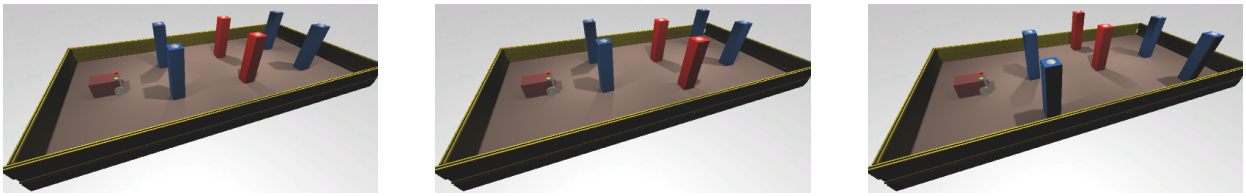
(a) Step n (b) Step n+1

Figure 4.4.: Agent Camera Images from two subsequent steps showcasing the negligeable changes

difficulty settings easy, medium and hard. An invisible finish line is positioned closely behind the last goal for each track.

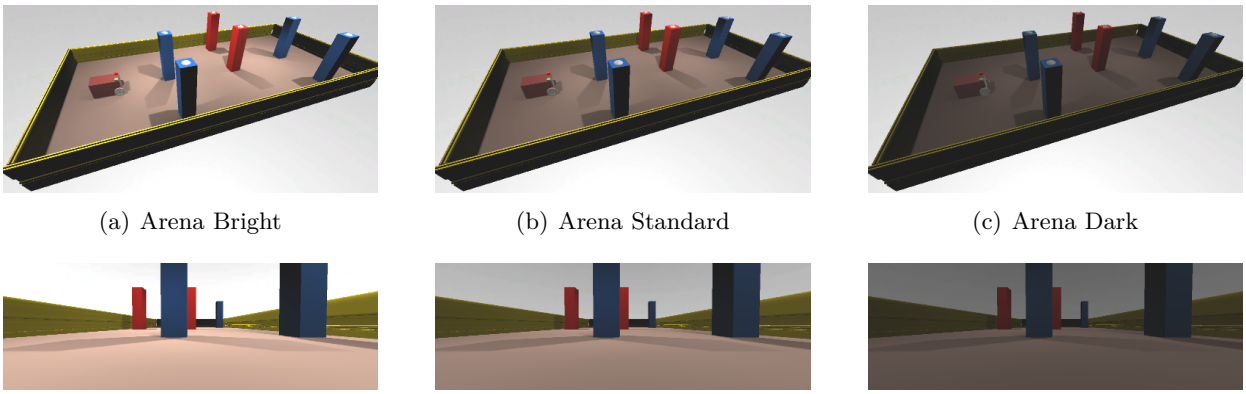
Track Description The tracks in the easy setting contain 3 goals that are positioned on the arena's center line with even distances between them. The medium setting contains 3 goals that are shifted on the center line towards the arena's walls. The hard setting contains 3 goals that are shifted on the center line towards the arena's walls, resulting in a zig-zag pattern. The zig-zag pattern is the most challenging for the agent to navigate, as it requires the agent to turn sharply to pass the goals. One track from each difficulty setting is shown in figure 4.5.

The tracks in each difficulty level are structurally very similar to each other. They differ in goal coloring and the orientation of the shift from the center line. In total there are 10 different tracks.



(a) Easy (b) Medium (c) Hard

Figure 4.5.: Example evaluation tracks for each difficulty setting.



(a) Arena Bright (b) Arena Standard (c) Arena Dark

(d) Agent Bright (e) Agent Standard (f) Agent Dark

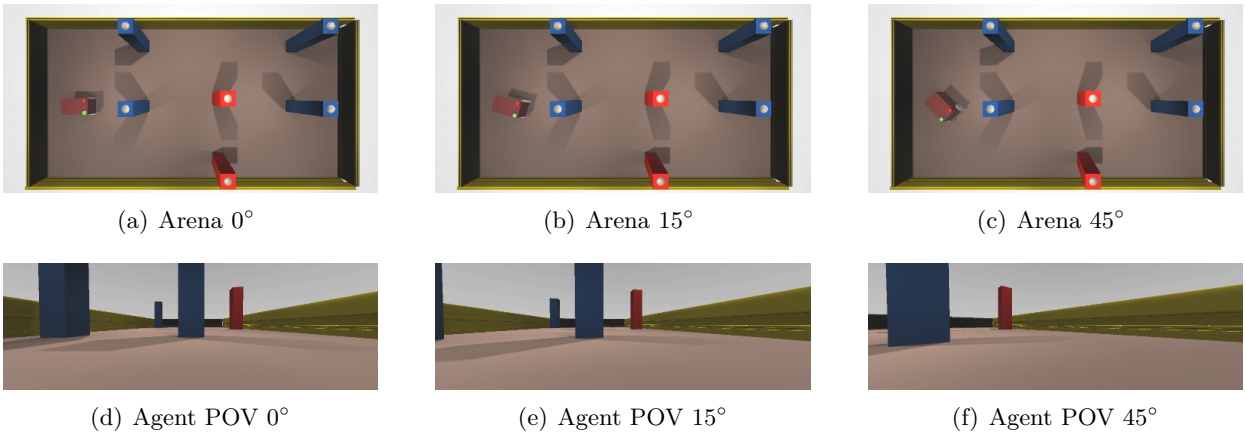
Figure 4.6.: Arena and agent camera at different light settings.

The full list is availible in the appendix A.1. The tracks are identical to the tracks used in previous work by Schaller [1].

Light Setting Description There are three light settings for the environment: bright, standard and dark. The different light settings are implemented by varying the light intensities of the arena's light sources and changing the horizon illumination of the agent's camera.

Initial Position of Agent The initial position of the agent at the start of an episode is fixed. The starting position is identical for all tracks. The inital orientation is defined by the environment parameter *spawnOrientation*. The parameter specifies the range of rotations around the z-axis.There are three options for the spawnOrientation parameter: Fixed, Random and VeryRandom. For the Fixed option the agent is spawned with an orientation of 0 degrees. In the Random option the agent is spawned with an orientation between -15 and 15 degrees. In the VeryRandom option the agent is spawned with a random orientation between -45 and 45 degrees.

During the training process an orientation from the range is sampled for each episode. In the evaluation process the agent is spawned with unique orientations from the range, see 5.2. The ranges for the spawnOrientation parameter influence the difficulty of completing the tracks. Depending on the spawn rotation and the selected track it might not be possible to see the entire first goal from the starting position. This is shown in 4.7.



(a) Arena 0° (b) Arena 15° (c) Arena 45°

(d) Agent POV 0° (e) Agent POV 15° (f) Agent POV 45°

Figure 4.7.: Example Spawn Orientations and agent camera views for a hard track

Arena Recording Video recordings of episodes can be generated by the unity environment. The episode recordings are created from three different perspectives. The first perspective is the agent's camera view. This video shows the images before preprocessing steps are applied. The second perspective is a top-down view of the arena. The third perspective is a side view of the arena.

Multiple episodes can be simulated in parallel. The video recording is generally not done for all episodes due to performance problems and the large amount of data generated. Example videos are shown in the appendix D.

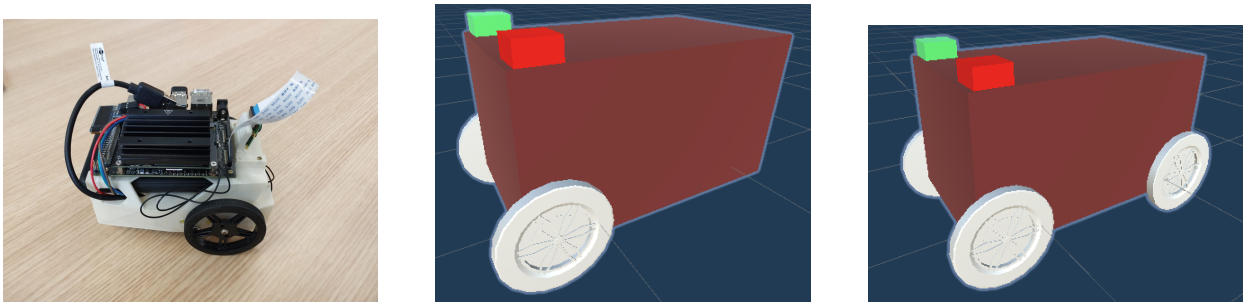
4.1.3. Agent Description

The agent is modeled after the NVIDIA JetBot, a small robot designed for educational purposes. The NVIDIA JetBot is equipped with a camera and a differential drive system. The agent's camera is mounted on the JetBot's front and captures the arena from the JetBot's perspective. The camera captures the arena in a 2D image format. In each step the agent receives an action to execute from the Python algorithm.

The actions consist of a $leftAcceleration$ and $rightAcceleration$ scalar value. The agent's wheels are controlled using these values for the entire duration of a step. The values are restricted to the range of $[-1, 1]$, see 4.16. The agent moves forward when the values are positive. When the right acceleration value is bigger the agent turns to the left and vice versa.

There are two versions of the JetBot agent in the simulation, the DifferentialJetBot and the FourWheelJetBot. The DifferentialJetBot has two driving wheels at the front and a ball supporting it at the back. The DifferentialJetBot applies by different torques to the two front wheels. The two acceleration values are multiplied by a constant factor and applied to the two wheels independently.

The FourWheelJetBot has 2 steering driving wheels at the front and two non-driving wheels in the back. The FourWheelJetBot steers by turning the front wheels in the desired direction. Equal torques are applied to both front wheels. The steering angle is computed from the difference in the two acceleration values. The torque is computed by multiplying the mean of the two acceleration values with a constant factor.



(a) Nvidia JetBot Photo (b) DifferentialJetBot (c) FourWheelJetBot

Figure 4.8.: Original Nvidia JetBot and simulated JetBot Designs

The FourWheelJetBot was used in the work by Schaller [1]. The DifferentialJetBot was developed to match the physical NVIDIA JetBot more closely. The two JetBot designs are shown in figure 4.8.

4.1.4. Episode Design

An episode represents one attempt of the agent at solving a track in the environment. Each episode consists of a series of steps starting from the initial position. Upon episode termination the end status is returned to python. This status is used to compute metrics such as the success_rate. The agent interacts with the environment in each step. The environment is simulated in Unity for the duration of the steps. This includes things like agent movement, collision detection and reward calculation.

4.1.4.1. Episode Initialization

The episode initialization prepares the arena. There are five important parameters that define how the episode is initialized. The initialization first generates the obstacles according to the *mapType* parameter. The *lightSetting* parameter defines the illumination for the entire episode. The *jetbotName* parameter dermines which jetbot design is used. The jetbot is then placed in the arena according to the *spawnOrientation* parameter.

The *videoFilename* parameter is used to define the filename for the video recording of the episode. No videos are recorded if this parameter is empty.

The call returns an initial observation to the python algorithm. This observation is used by the agent for the first step.

4.1.4.2. Steps

The agent selects actions to perform in each step. The actions consist of a pair of acceleration values 4.1.3. These values are applied to the agent's wheels for the entire duration of the step. The agent's movement is simulated in the Unity environment for the duration of the step. This results

in a new environment state. The Unity environment calculates the reward obtained during a step. It also registers agent collisions and episode timeouts.

4.1.4.3. Step duration

The duration of each step is defined by the environment settings. There are two destinct modes for the step durations. The first mode is the fixed timestep mode. In this mode the duration of each step is fixed and defined by the environment parameter $fixedTimestepsLength$. The second mode is the variable timestep mode. In this mode each step lasts until the environment receives the next action from the agent.

The fixed step duration mode was used for the training process after experimentation with both modes. The fixed step duration was set to 0.3 seconds. This allows for precise movements of the agent as the duration is quite short 6.1.

Fixed Duration The duration of each step is fixed in this mode. In each step the Unity environment receives an action from the policy. The environment simulation is started and the action is applied to the jetbot agent. The agent moves according to the action and interacts with the environment. The agent collects rewards. Collisions and timeouts are detected. The step and environment simulation is terminated when the fixed duration has passed. The duration is defined by $fixedTimestepsLength$ in seconds. The Unity environment then waits until the next step or environment reset. The unity environment does not accept new step commands when the current step is not terminated. An episode with fixed duration steps is shown in the upper part of figure 4.9. The figure shows how the Unity environment waits for new steps to start. It furthermore shows that the waiting time may not be consistent.

The fixed mode has many advantages. The fixed duration of the steps results in consistency of the step transitions. Given identical step durations and environment state, it is well defined how the agent will move in any step. The environment state after completing a step is well defined.

Furthermore the performance of the policy does not depend on the processing speed of the device. A policy in fixed mode can be transfered to other devices without changing the policy's behaviour. In case of slower policy computation, the unity environment pauses the simulation and waits for the next step command. In case of faster policy computation, the unity environment waits until the current fixed length step is completed before starting the next step.

Variable Duration The duration of each step is variable in this mode. The environment simulation is started when the first step is received. In each step the Unity environment receives an action from the policy. The step's action is applied to the jetbot agent. The agent moves according to the action and interacts with the environment. The agent collects rewards, collisions and timeouts are detected. The step is terminated when a new step is received from the policy. The new step is started instantly, there is no waiting time between steps. The duration of the steps is not fixed, it can change from step to step. It depends on the policy's computation and message transmission time. The environment simulation is not paused during the policy's computation time. An episode

with variable duration steps is shown in the lower part of figure 4.9. The figure shows how the Unity environment waits only for the first step. It furthermore shows that the step sizes may change from step to step.

The biggest advantage and disadvantage this mode is that the step duration depends on the policy's computation and message transmission time. If the policy is computed fast, the average duration of the steps will be shorter. Shorter steps allow for more precise movements. Shorter durations for steps can result in a more capable policy. A disadvantage is that the step duration can change due to a changed policy computation time. The policy computation time will be different on other devices or when there is additional load on the processing unit. Changes in step duration might break a trained policy, as the policy has learned to expect a certain environment change for the steps. This environment change might be different for steps with different durations. Example: The agent might decide to turn right, expecting to be able to stop the turning within 0.2 seconds. If the next step is computed too slow the agent might not be able to stop the turning in time. The agent might then crash into a wall.

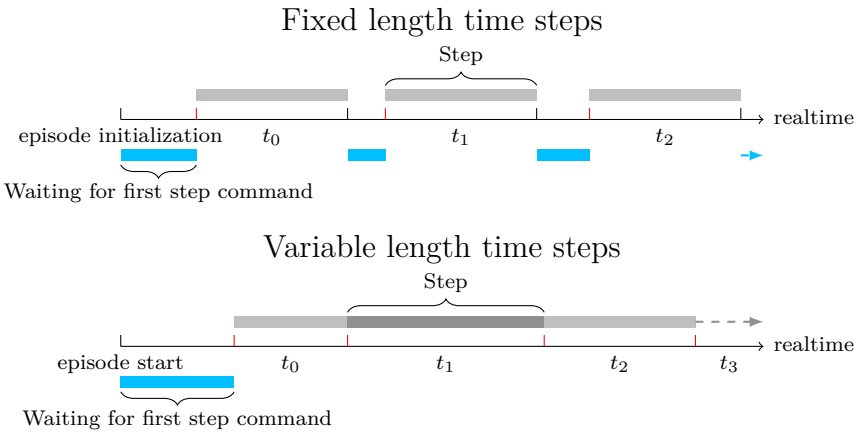


Figure 4.9.: Timeline of steps in Unity for fixed and variable duration

4.1.4.4. Episode Termination

The episodes are terminated based on the agent's interactions with the environment. Episodes are always terminated when the agent reaches the finish line or a timeout is reached. Episodes are also terminated when the agent collides with an object and the *collisonMode* parameter is set to *terminate*. The timeout is defined by a fixed timelimit of 30 seconds which is increased by further 30 seconds for each passed goal. The timelimit is necessary to terminate episodes where the agent does not reach the finish line. This can be due to the policy's learned behaviour or collisions.

The episode termination status is used by the python algorithms to evaluate the agent performance. The possible termination statuses are:

Success The episode is considered to have been terminated successfully if the agent passed all of the track's goals in order.

Timeout The timeout status is returned when the agent does not reach the finish line within the timelimit and not all goals have been passed successfully.

FinishWithoutAllGoals The FinishWithoutAllGoals status is returned when the agent reaches the finish line without passing through all goals.

Collision Collisions of the agent with the goal posts and the arena walls are detected by the environment. The environment parameter *collisionMode* defines how the agent's collisions are handled. The termination status Collision is returned when the agent collides with an object and the *collisionMode* parameter is set to *terminate*.

4.1.5. Reward Function

The reward function is a function that maps state-action pairs to a scalar reward. These state-action pairs are episode steps. The reward function assigns rewards to each step based on the agent's actions in the environment.

The RL algorithm trains the agent to maximise the cumulative reward over an episode. The reward function is a critical component of the RL process. It is important to design a reward function that is closely alligned with the goal. The agent could learn unintended behaviour if the reward function is not designed carefully.

Event Reward The goal of the training process is to achieve an agent that traverses the tracks without collisions and passed through the goals in order. The eventReward function describes this behaviour, see 4.10. The event reward function is evaluated in each step. A positive reward is awarded to the agent in case it passes a goal or reaches the finish line during the step. A penalty is applied in case the agent misses a goal, collides with an object or the timeout is reached. The magnitude of positive rewards increased was increased compared to the negative rewards. This means that positive events are more important, a single negative event does not cancel out positive rewards.

The maximum amount of reward obtainable from this function per episode is:

$$num\_goals * goal\_reward + track\_reward = 3 * 100 + 100 = 400$$

An agent that maximises the eventReward function will complete the track without collisions. In theory the event Reward should be enough to train an agent that traverses the tracks successfully without collisions.

$$EventReward(s_t, a_t) = \begin{cases} 100, \text{completed the track} \\ 100, \text{passed a goal} \\ -1, \text{missed a goal} \\ -1, \text{collision with wall or obstacle} \\ -1, \text{timeout} \\ 0, \text{otherwise} \end{cases}$$

Figure 4.10.: Event Reward function
s_t: state t a_t: action in state t

Dense Reward Functions The eventReward function provides the agent with a reward signal that is closely aligned with the desired behaviour. However the eventRewards are only awarded in very few steps. In the optimal episode there are only 4 steps where the reward is non-zero. Three steps where the agent passes a goal and one step where the agent reaches the finish line. The eventReward function is a sparse reward function. Sparse rewards can make it difficult for the agent to learn the desired behaviour. In the case of the eventReward, the agent might never learn to drive towards a goal since there are no rewards that encourage that behaviour. The agent would rely on random exploration alone to discover the positive reward associated with successfully driving through a goal. Additional reward functions were developed to help the agent learn to complete the tracks. The additional reward functions are dense reward functions. They assign non-zero rewards in most steps. This is comparable to encouraging route following behaviour with a trail of candy versus a big cake at the end of the route.

Dense rewards functions can exasterbate the issue of the agent learning unintended behaviour. Assigning rewards to sub-goal can lead to the agent behaving differently than as indended. The agent might not learn to achieve the overall goal, Sutton and Barto [15]. This is to be kept in mind when designing and testing the dense reward functions.

Velocity Reward The first dense reward function is the velocityReward. It was already developed in previous work Schaller [1] to encourage the agent to drive at full speed. The velocityReward function assigns a reward proportional to the agent's velocity. The reward from driving forward might also help at discovering other rewards, e.g. the positive reward for passing a goal awared by eventReward function.

Orientation Reward The second dense reward function is the orientationReward. The orientationReward function assigns a reward based on the agent's orientation. The reward is proportional to the cosine similarity between the agent's orientation and the direction towards the next goal. The orientationReward function encourages the agent to face the next goal. Together with other rewards this can help teach the agent to drive through the next goal.

Distance Reward The third and last dense reward function is the distanceReward function. The distanceReward function assigns a reward proportional to the difference in distance between the

$$
\begin{aligned}
R(s_t, a_t) &= c_1 \cdot DistanceReward(s_t, a_t) + c_2 \cdot OrientationReward(s_t, a_t) \\
&\quad + c_3 \cdot VelocityReward(s_t, a_t) + c_4 \cdot EventReward(s_t, a_t) \\
DistanceReward(s_t, a_t) &= \Delta distance(Agent, NextGoalPosition) \cdot \Delta T \\
OrientationReward(s_t, a_t) &= S_C(NextGoalPosition - AgentPosition, agentDirection) \cdot \Delta T \\
VelocityReward(s_t, a_t) &= v \cdot \Delta T \\
EventReward(s_t, a_t) &= \begin{cases} 100, \text{completed the track} \\ 1, \text{passed a goal} \\ -1, \text{missed a goal} \\ -1, \text{collision with wall or obstacle} \\ -1, \text{timeout} \\ 0, \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 4.11.: Complete reward function R with all its components
S_C: cosine similarity c_i: weights
s_t: state t a_t: action in state t

agent and the next goal. The distanceReward is positive if the agent gets closer to the next goal during a step transition. The distance to the next goal is measured from the midpoint between the two goal posts. The distanceReward function encourages higher driving velocity, as higher velocities result in bigger distance differences and thus rewards. The distanceReward function also encourages the agent to drive strait towards the next goal, as this results in the biggest distance differences.

In theory the agent should be able to learn to complete the tracks successfully from the distanceReward alone. Maximizing the distance reward would result in the agent driving through all goals in order.

The distance reward is opinionated. It guides the agent towards the midpoint of the next goal. This reward signal may result in a suboptimal path for the agent. The policy might learn to drive through the goal posts at the midpoint. This is not necessary for the desired behaviour. This is a tradeoff that was made when designing the distance reward function.

Composite Reward Function The four individual reward functions can be combined in many ways to give the agent dense and meaningful rewards. In previous work by Schaller [1], the eventReward was combined with the velocityReward to train the agent. In this work the individual reward functions are combined in the composite reward function 4.11. The composite reward function applies scalar weights to the individual reward functions and returns this weighted sum of rewards to the agent. It is crucial to select an appropriate combination of weights for the individual reward functions during training. The weights determine the importance of the individual reward functions and can influence the learned behaviour.

If the velocityReward is weighted too high the agent might learn to drive at full speed in circles without ever reaching the finish line. It is also to note that setting a specific reward function's weight to zero essentially removes that reward function.

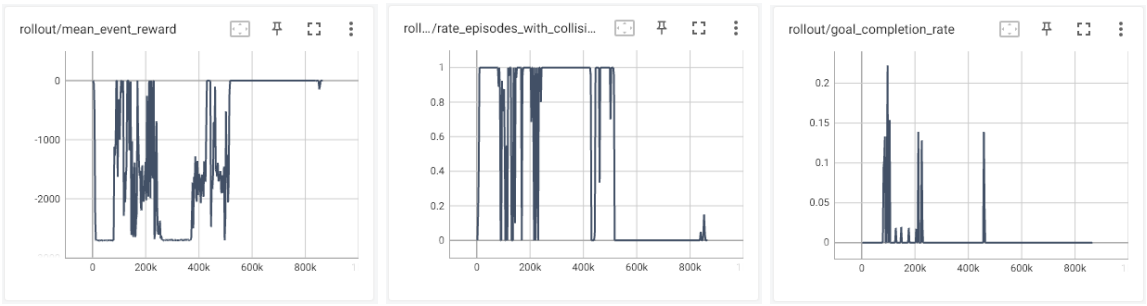


Figure 4.12.: Event Reward only training with *collisionMode unrestricted*

4.1.6. Collision Mode

The variable *collisionMode* describes how the environment handles collisions of the agent with the goal objects and walls. The *collisionMode* can take 5 different values described in table 4.13. In previous research the episodes were terminated upon collision and when missing a goal Schaller [1]. Furthermore there existed two different training regimes. The first regime trained the agent on the full maps, while the second regime trained the agent on a map with only one goal.

The *collisionMode* parameter was introduced to allow for more flexibility in the training process and fine control over the event rewards. The inital implementation was the unrestricted mode. Training with the unrestricted mode and event reward showed that the agent's policy started out with extreme negative rewards. The agent quickly learned to avoid these negative rewards by avoiding all collisions and movements. The policy got stuck in a local optimum 4.12. The policy did not learn to complete the track during the remainder of training. The unrestricted mode potentially triggers the collision's negative rewards multiple times per step. This results in a big skew of the eventReward function towards negative rewards.

The modes *oncePerTimestep*, *oncePerTimestep*, *terminate* and *ignore* were introduced to limit the negative rewards caused by collisions. Reducing the negative rewards could lead to a policy that does not avoid collisions and movement as much as. This could lead to more exploration of the environment and a succesful policy.

collisionMode	Behaviour upon Collision	Reasoning
unrestricted	Negative reward is given for each frame with collision. This can result in multiple penalties given during a single step.	Default behaviour
oncePerTimestep	Negative reward for collisions is given only once per step.	Limits the negative reward caused by a collision in a step.
oncePerEpisode	Negative reward for collision is only given once per episode per object.	Limits the negative reward caused by ongoing collisions.
terminate	Episode is terminated instantly.	No ongoing collisions. The early termination could speed up the policy training.
ignore	Collisions are ignored, no negative reward is given.	The agent might learn to avoid collisions based on other rewards.

Figure 4.13.: Collision Modes

4.2. Observation Processing

The policy controls the agent's actions in the environment. The policy is optimized using the PPO RL algorithm. The policy takes a representation of the environment state as input and produces an action as output. This representation is called the observation.

The environment state is represented as an image from the agent's camera. This section explains how the observation is processed before it is given to the policy. The steps aim to enhance the policy's performance and generalization capabilities.

The agent's camera image has three channels, red, green, and blue. Its dimensions are defined by the config parameters $agentImageWidth$ and $agentImageHeight$. These parameters define the agent's field of view. The policy receives an image from the agent camera as input. First the image is preprocessed. This changes the image content and dimensions. The preprocessed image is then combined with the memory mechanism to produce the inputs to the policy.

4.2.1. Preprocessing

The images from the agent camera are preprocessed before they are fed into the NN. The preprocessing steps are applied to reduce the size of the NN's input space. They are also applied to reduce the impact of different light settings on the policy's performance. Three preprocessing steps have been implemented, downsampling, grayscale conversion, and histogram equalization. The preprocessing steps can be enabled and configured in the environment parameter *image_preprocessing*. Each preprocessing step transforms an image into a new image. This can change the image dimensions and pixel domains. The steps are applied on the previous step's output. The steps are applied in order of downsampling, grayscale conversion, and histogram equalization. The first step takes the image from the agent camera as input. The NN uses the last step's output for inference.

The preprocessing steps are applied to all images from the agent camera. The table 4.1 shows how the preprocessing steps make images of different light settings more similar. The images in the table were generated using the configurations employed int the training runs.

Benefits of reducing the input space size

The reduction of input space size brings a variety of benefits. Some benefits are execution driven, like the reduction of the NN's inference time. The NN can make decisions quicker if it has to process fewer inputs. The reduction of the input space size also reduces the amount of data that has to be stored in memory. This is especially important during the training phase since many observations have to be stored in the rollout buffer.

The reduction of the input space size also reduces the number of parameters the NN has to learn. This can lead to a more robust policy. Networks with smaller input spaces and less parameters are less prone to overfitting. The policy can generalize better to unseen situations.

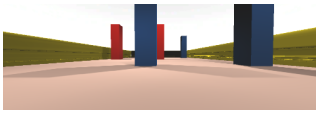
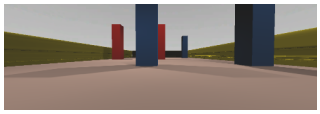
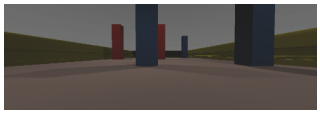
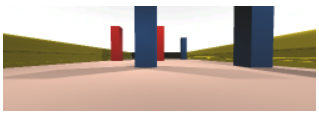
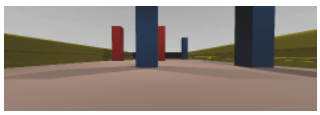
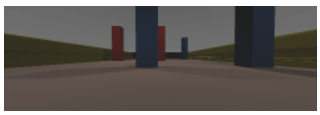
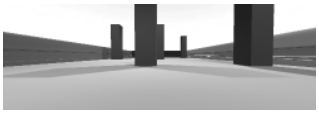
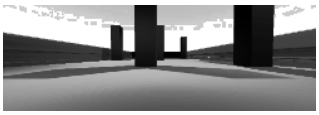
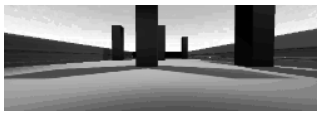
light Setting	Bright	Standard	Dark
Agent Camera Image from Unity			
Downsampled			
Grayscale			
Equalized			
Final Input Image			

Table 4.1.: Preprocessing steps applied to images from the agent camera at the different light settings. The steps are applied in order from top to bottom.

Benefits of reducing the impact of different light settings

The policy has to be able to complete the different tracks for three different light settings. The policy is not provided with information about what light setting is active for the current episode. This means that the policy has to learn to produce correct actions for images of all three light settings. If images of the light settings look very different, the policy has to learn to the light settings as well as what action to take. The preprocessing steps are applied to make images of different light settings more similar. The NN is relieved of learning the different light settings. This can make learning easier and faster. The preprocessing steps can also make the policy generalize better to unseen light settings. The policy can be trained on a single light setting and achieve good results for the other light setttings 6.4.

4.2.1.1. Downsampling

The downsampling step changes the resolution and dimensions of the image. The step is configured with the *downsampling_factor* parameter. The downsampling factor is a positive integer and determines how much the image is downscaled. A *downsampling_factor* of one results in no downsampling. The width and height dimensions of the input image are divided by *downsampling_factor* to produce the new image dimensions. The downsampling process divides the input image in a grid. The grid cells are *downsampling_factor* pixels wide and high. The pixel values of the new image are calculated by averaging the pixel values of the grid cells. The new image has a lower resolution than the input image. Downsampling preserves the image content while reducing the size of the input space substantially.

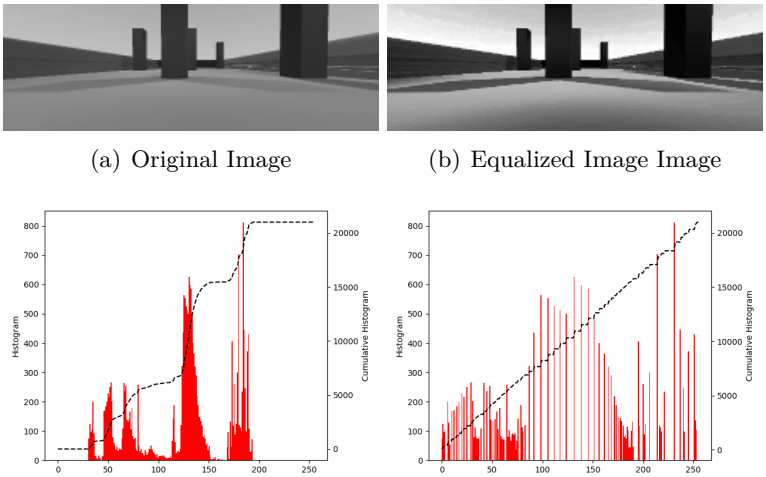
4.2.1.2. Grayscale

The grayscale step converts the image to grayscale. The step can be enabled or disabled with the *grayscale* parameter. The agent's camera image has three channels, red, green, and blue. The grayscale step changes the amount of channels and returns an image with a single grayscale channel. The grayscale image has the same width and height dimensions as the input image. The grayscaling reduces the input space size by a third. The pixel values of the grayscale image are calculated as the weighted sum of intensities of the red, green, and blue channels. The weights are determined by the human eye's perception of the different colors. The grayscale value is computed as $Y = 0.2125 * R + 0.7154 * G + 0.0721 * B$.

The conversion to grayscale removes color information from the image. The alternating goal colors red and blue are no longer distinguishible. The advantage of grayscale images is that the policy's NN does not have to learn the three dimensional colorspace of RGB images.

4.2.1.3. Histogram Equalization

The histogram equalization step increases the contrast of the image. The step can be enabled or disabled with the *equalize_histogram* parameter. The histogram equalization step requires



(a) Original Image (b) Equalized Image Image

(c) Original Image Histogram (d) Equalized Image Histogram

Figure 4.14.: Histogram equalization of a grayscale image from standard light setting

a grayscale image as input. The step changes the pixel values of the image. The step does not change the input space size. The histogram equalization step changes the pixel values of the image. The histogram of the pixel intensities are made more uniform. First the cumulative histogram of the pixel intensities is calculated. The pixel values are then mapped to new values by using the cumulative histogram. The pixel values are distributed more evenly over the pixel value domain 4.14.

The histogram equalization step can make images of different light settings more similar. The step also increases the contrast in the images. This can make it easier for the NN to detect the objects.

4.2.2. Memory Mechanism

Memory mechanisms are a common approach to enhancing a RL agent. Memory mechanisms can be implemented in many different ways. Our memory mechanism uses frame stacking, similar to Schaller [1] and Mnih et al. [14]. The memory is configured with the $frame_stacking$ parameter. The memory is a simple first-in-first-out buffer that stores the last $frame_stacking$ images. The memory is updated by inserting the new image and removing the oldest image. The inserted images are the outputs of the preprocessing steps. The images from the memory buffer are concatenated along the channel axis to produce the NN input. This NN input is essentially an image with many channels. The memory buffer is reset at the start of every episode. The reset fills the entire buffer with zeros. The zeros are used as a placeholder. The first-in-first-out mechanism removes the zeros over the first $frame\_stacking$ timesteps.

The memory mechanism enables the policy to use information from previous time steps. The policy can use this information to detect objects that are not visible in the current image. Depending on the agent's position and orientation, the next goal might only be partially visible or not visible at all. The past images can help the policy reason about the next goal's position and how to continue.

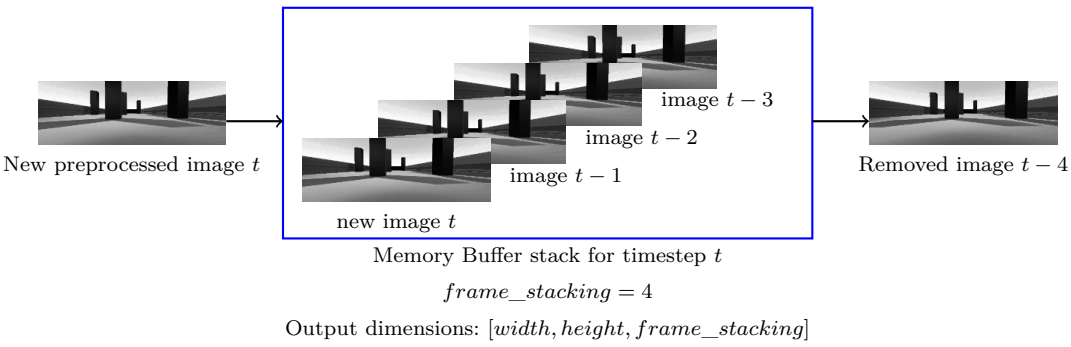


Figure 4.15.: Memory Mechanism

4.3. Policy

The policy dictates the agent's behaviour. The policy is responsible for selecting actions from observations. The policy is trained using the PPO algorithm. The algorithm has proven to be very reliable Schulman et al. [12]. The algorithm can be used for our continuous control task.

A NN is used as for the policy here. The NN belongs to the class of CNNs. This architecture was chosen since CNNs are ideal for processing image data. The convolutional network's architecture follows the specifications from Mnih et al. [36]. This architecture has been used successfully in comparable RL control tasks.

The network takes the output of the observation preprocessing as input. This input is a three dimensional tensor, similar to an image. The network consists of convolutional and fully connected layers. The network produces two outputs, an action distribution and a value function. The action distribution is a probability distribution over the action space. The value function is a scalar value that estimates the expected return of the current state. The value function is not used during the action inference. The value is only used during the training of the policy network. It is used to compute the advantage function which is required for the loss in PPO Schulman et al. [12].

4.3.1. Observation Space

The input space of the NN is three dimensional. The input is structured like images with *width*, *heigth* and *channel* dimensions. The dimensions are determined by the agent's camera image dimensions, preprocessing parameters and the memory mechanism. The input space is a tensor with the following dimensions:

$$
\begin{aligned}
width &= \frac{agentImageWidth}{downsampling\_factor} \\
height &= \frac{agentImageHeight}{downsampling\_factor} \\
channels &= frame\_stacking \cdot num\_color\_channels
\end{aligned}
$$

The tensor values are integers in the range $[0, 255]$. Storing the pixel intensities as integers saves a lot of memory compared to floats.

4.3.2. Action Space

The environment's action space is a two dimensional scalar, the dimensions are called $leftAcceleration$ and $rightAcceleration$. The values are restricted to the range $[-1, 1]$, see 4.16. They dictate the agent movement 4.1.3.

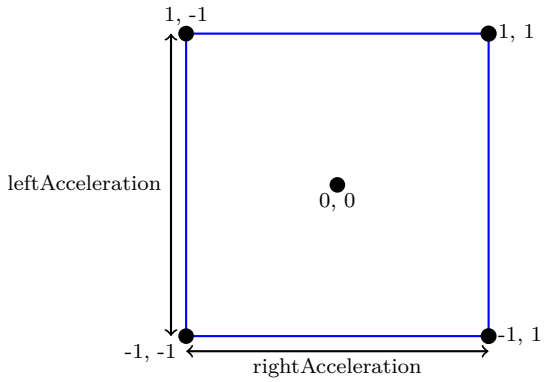


Figure 4.16.: the two-dimensional Action Space

4.3.3. Architecture

The NN produces two outputs, the action distribution and a value estimate. The outputs are produced by sections of the network called the action and value head. The action head produces the action distribution. The value head produces the estimate of the state's value.

The action and value head share the first part of the NN. They are built on top of the feature extractor. The full structure is shown in 4.17.

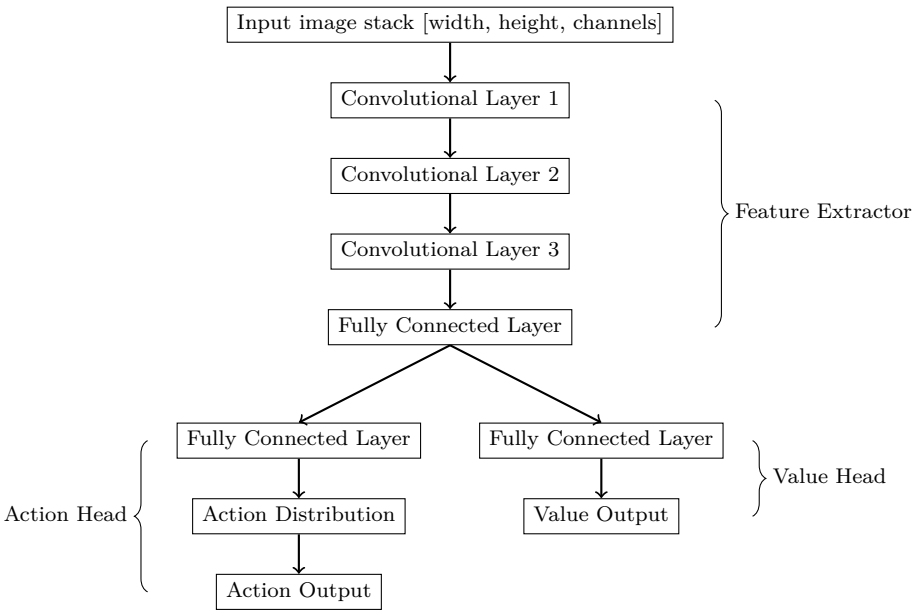


Figure 4.17.: NN Structure

4.3.3.1. Feature Extractor

The feature extractor comprises the first four layers of the network. As the name suggests, the feature extractor extracts features from the input for further processing by the network.

The feature extractor consists of three convolutional layers and one fully connected layer. The first convolutional layer takes the output of the memory buffer as input. The output of the feature extractor's fully connected layer is used by the action and value heads.

Component	Layer Type	Layer Specifications
Feature Extractor	Convolutional Layer	32 filters, 3 dimensional (10, 8, 8)
	Convolutional Layer	64 filters, 3 dimensional (32, 4, 4)
	Convolutional Layer	64 filters, 3 dimensional (64, 3, 3)
	Fully connected Layer	512 output neurons, (512 x 12096) matrix
Action Head	Fully connected Layer	2 output neurons, (2 x 512) matrix
Value Head	Fully connected Layer	1 output neuron, (1 x 512) matrix

$agentImageWidth$: 500
$agentImageHeight$: 168
$downsampling\_factor$: 2
$grayscale$: True
$equalize$: True
$frame\_stacking$: 10

Figure 4.18.: NN layers and parameters for the best configuration

4.3.3.2. Action Head

The action head consists of a single fully connected layer with two outputs. The two outputs represent the mean of the two-dimensional action distribution. The action distribution is a Gaussian Distribution. The action distribution can be sampled deterministically or stochastically to obtain an action for the agent. The most likely action is returned in determinsitc sampling. This is equal to the outputs of the action head. In stochastic sampling, the action is sampled from the action distribution.

The sampled actions are clipped to the action space's range of $[-1, 1]$. This is necessary as the sampling can return a value outside of the range.

4.3.3.3. Value Head

The value head consists of a single fully connected layer with one scalar output.

4.3.4. Parameters

The NN's input dimensions are determined by the agent's camera image dimensions and the observation processing. The network architecture follows the specifications from Mnih et al. [36]. However the observation space differ from their network. This results in a different number of parameters for the feature extractor's layers. The parameters and layer dimensions used network are shown in figure 4.18.

4.4. Policy Interaction with the Environment

This section summarizes the interaction of the policy with the environment. The full interaction is shown in figure 4.19.

As described in previous sections the agent's actions are controlled by the agent's policy. Given an observation the policy selects actions to execute. The observations represent the environment state. They are built from the agent's camera images. The observation processing steps transform these camera images to the observation space. The policy is a NN that takes the observation space as input and produces actions as output. The actions are applied to the environment. The environment processes the actions and returns a reward and a new observation.

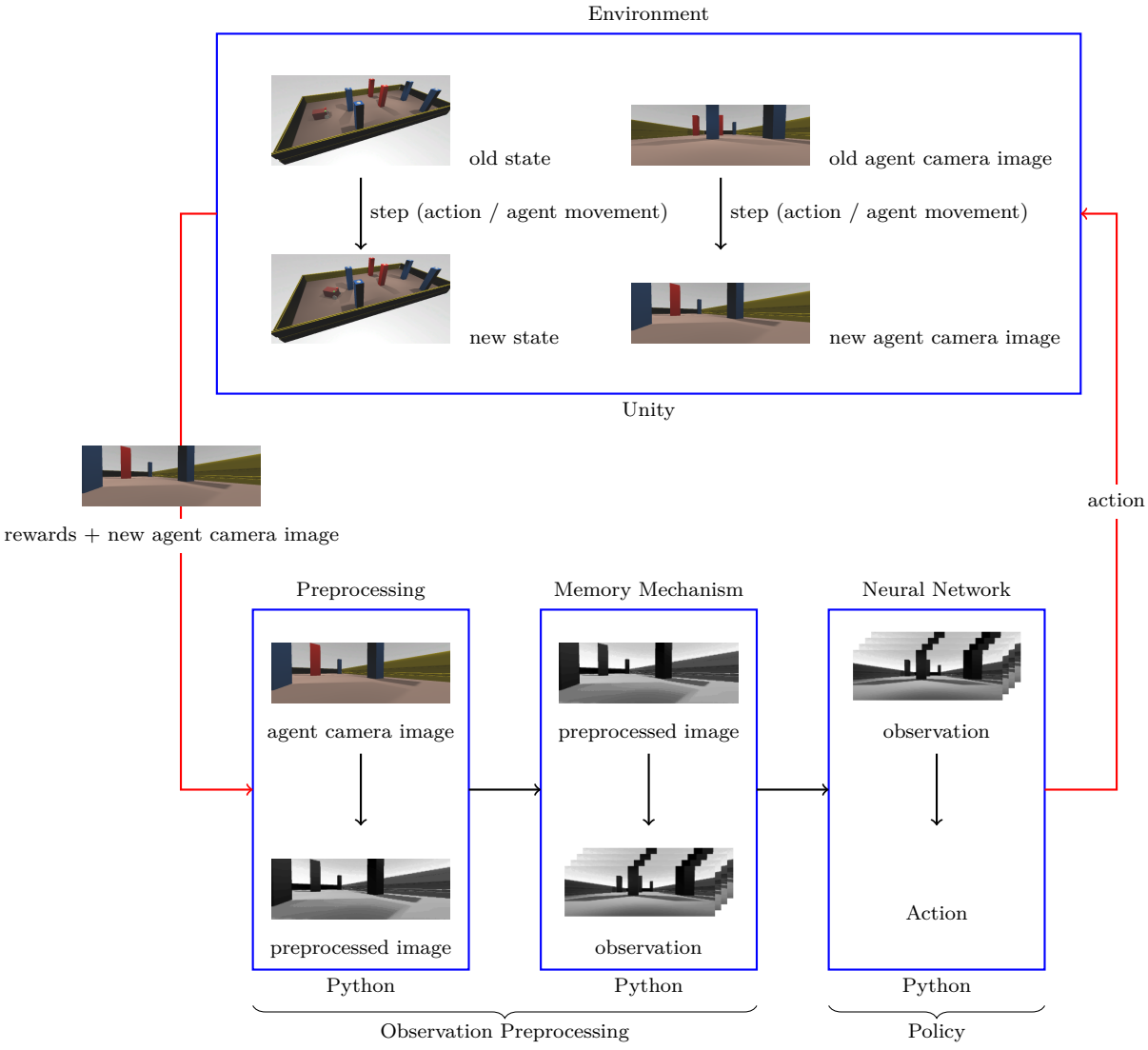


Figure 4.19.: Full policy interaction with the environment, RPC communication in red

4.5. Training Algorithm

The policy training process consists of a simple training loop. In each iteration, the algorithm collects data from the environment. The data is stored in a replay buffer. After collecting the data, the model is trained on the data in the replay buffer. The training process is repeated until a certain number of timesteps has been collected.

In most RL algorithms, the trained model's performance is evaluated at short intervals. The frequent evaluations serve as a way to monitor the model's performance during training and measure progress. It is not guaranteed in RL that a model improves during each model update. Therefore frequent evaluations also help in finding the best version of the model. In this training algorithm there is no dedicated evaluation process at short intervals. Instead the episodes from the data collection step are used to evaluate the model at every iteration of the training loop. This saves computational resources compared to a dedicated evaluation of the model.

Algorithm 4.5.1: TrainNetwork()

comment: Main Training Algorithm

main
$Env \leftarrow$ Environment($environment\_parameters$)
$RolloutBuffer \leftarrow$ RolloutBuffer($rollout\_buffer\_size$)
$Model \leftarrow$ Model($model\_parameters$)
$Optimizer \leftarrow$ Optimizer($optimizer\_parameters$)
$num\_timesteps \leftarrow 0$
while $num\_timesteps < total\_timesteps$
do
- $num\_collected\_steps \leftarrow$ CollectData()
- $num\_timesteps \leftarrow num\_timesteps + num\_collected\_steps$
- TrainModel()

4.5.1. Collect Data

The data is stored in a replay buffer for use by the training algorithm. The data collection process starts by resetting the replay buffer, this removes the collected sampled of previous iterations from memory. The data collection process is a simple loop that collects a fixed amount of samples from the environment. The PPO RL algorithm requires that the samples are collected on-policy. As a result the most recently updated model is used to chose actions during the data collection.

The data collection algorithm starts a new episode and applies the policy to the environment until the episode terminates. The observation is used to prompt the CNN policy for an action. The action is then applied to the environment, resulting in a reward and a new environment state. The observation, action and reward tuples are stored in the replay buffer. A new episode is started upon termination of the current episode. The data collection process continues until the replay buffer is full.

Parameter name	Options	Explanation
trainingMapType	randomEvalEasy	Selects a random easy track.
	randomEvalMedium	Selects a random medium track.
	randomEvalHard	Selects a random hard track.
	randomEval	Selects a random track from all difficulties. 20% easy, 40% medium, 40% hard
trainingLightSetting	bright	Bright illumination.
	standard	Standard illumination.
	dark	Dark illumination.
	random	Selects a random light setting from bright, standard and dark.
spawnOrientation	Fixed	Spawn JetBot with fixed coordinates and orientation.
	Random	Spawn JetBot with fixed coordinates and random orientation (-15 to 15 degrees).
	VeryRandom	Spawn JetBot with fixed coordinates and random orientation (-45 to 45 degrees).

Figure 4.20.: Environment parameters for training.

Actions are sampled non-deterministically from the policy. This ensures that the agent explores the environment and is able to learn from new data.

Environment settings for Data collection The environment settings define how the episodes are reset during the data collection process. The episodes are initialized according to the parameters trainingMapType, trainingLightSetting and spawnOrientation 4.20. The settings influence the difficulty and variety of the tracks. The settings define what data the agent policy can learn from during training.

Different configurations are used in the experimentation phase to gain insights into the network's capabilities.

Evaluation of the model There is no dedicated evaluation step in this training algorithm. The collected episodes are analysed to gain insight into the policy's performance during training. No episodes are executed solely for the purpose of measuring the policy's performance during training. The full evaluation of the model as described in 5.2 is time-consuming. The full evaluation process is only executed after the training process has finished, this speeds up training tremendously.

The collected episodes are used to evaluate the model during training. The success rate of the collected episodes is used to determine the best model. The model is saved to disk if the success rate of the collected episodes improved.

The success_rate and other statistics about the collected episodes are logged to tensorboard to monitor the training process. Example metrics are the goal_completion_rate and the collected

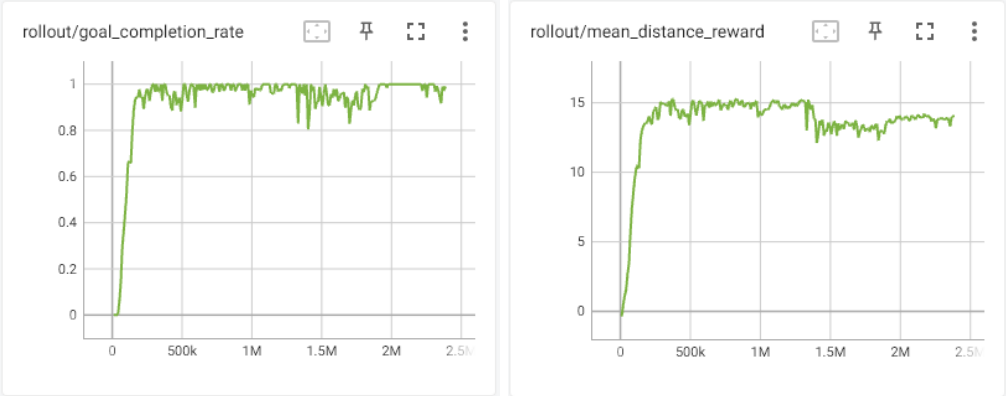


Figure 4.21.: Metrics from collected episodes during a successful training

weighted rewards. These metrics can show how the policy behaves over time. They show if the policy is able to learn from the collected episodes 4.21.

4.5.2. Train Model

The model training part of the algorithm is very conventional and follows the standard implementation of PPO training. The model is trained on the collected data for a fixed number of epochs. In each epoch the replay buffer is sampled to create batches of data. The loss is computed for each batch. The gradient of the weights with respect to the loss is computed using backpropagation. The optimizer is then used to update the model parameters using the gradients.

The entire data from the replay buffer is sampled in every epoch. This ensure that the collected data is used efficiently. The data is shuffled before creating the batches. This ensures that the samples are less correlated and the model updates are more stable.

Loss function

The loss function follows the PPO algorithm by Schulman et al. [12]. The PPO loss function is a combination of the policy surrogate and value error. The policy surrogate and value error are combined to be able to update the weights of the entire NN together.

The policy surrogate objective function is responsible for improving the policy's output action distribution. The function behaves similarly to other policy update based aproaches. It restricts the size of parameter updates, increasing the stability during training. The ratio between the old and the current policy is clipped. The clipped ratio and non-clipped ratio are multiplied by the advantage. The minimum of the two is used as the policy surrogate loss. This prevents the updates from changing the policy drastically.

The advantage is an estimator for how much better (or worse) the policy performed than expected. The advantage $\hat{A}_t$ is computed using the value estimates produced by the NN. The multiplication of the ratio with the advantage leads to increased probabilities for good actions and decreased probabilities for bad actions.

The value error is responsible for improving the policy's output value estimates. The value error is the mean squared error between the predicted value and the target value. The parameter changes caused by the value error lead to a more accurate value prediction.

The policy loss can also include an entropy term. Similarly to the original paper Schulman et al. [12], no entropy term is used in this work. This results in the loss function (PPO Loss) with the value coefficient $c_1 = 0.5$.

$$L_t^{CLIP+VF} = \hat{\mathbb{E}}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta)] \quad \text{(PPO Loss)}$$

$$L_t^{CLIP}(\theta) = min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \quad \text{(Surrogate Objective)}$$

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \quad \text{(Ratio)}$$

$$L_t^{VF}(\theta) = (V_\theta(s_t) - V_t^{targ})^2 \quad \text{(Value Error)}$$

4.5.3. Final Result of Training Algorithm

Weight updates can lead to a decrease in performance during the training process. The model after the final weight update is not guaranteed to be the best observed version. The model performance is evaluated during training using the collected episodes. The model with the highest observed success_rate is considered to be the final output of the training algorithm. This model is later used for the full evaluations.

5. Experiment Description

5.1. Evaluation metrics

5.1.1. Success Rate

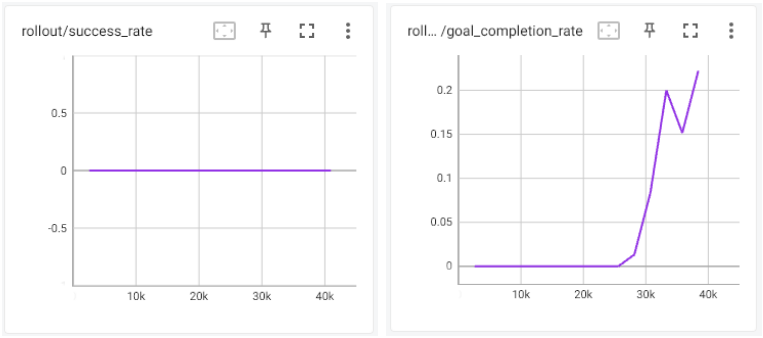
The primary metric of evaluation for the developed agents is the *success_rate*. The *success_rate* is defined as the ratio of successful episodes to the total number of episodes. An episode is considered successful if the agent passes through all three goals within the time limit 4.1.4.4. Collisions of the agent do not disqualify an episode from being succesful, as long as the agent passes all goals.

5.1.2. Goal Completion Rate

The *goal_completion_rate* is defined as the ratio of passed goals to the total number of goals in the episodes. The *goal_completion_rate* is a more fine-grained metric than the *success_rate*. However the two metrics are closely related as a high *success_rate* implies a high goal_completion_rate. The major advantage of the *goal_completion_rate* is that it can used to measure the progress of an agent during training more accurately. The *goal_completion_rate* would increase when the agent is able to pass more goals on average, whereas the *success_rate* would only increase when the agent is able to pass all goals in an episode more often. The *goal_completion_rate* captures learning progress earlier in training. This behaviour can be observed in training runs, shown in 5.1.

5.1.3. Collision Rate

The *collision_rate* is defined as the ratio of episodes with one or more collisions to the total number of episodes. The *collision_rate* is a measure of the agent's ability to avoid obstacles. The *collision_rate* is a secondary metric. The *collision_rate* is used in combination with the *success_rate* to determine the agent's performance. The collision rate is also very useful during



(a) success_rate (b) goal_completion_rate

Figure 5.1.: Difference in success rate and goal completion rate during early stages of training.

Parameter name	Options	Explanation
n_eval_episodes	any positive integer	Amount of episodes to use for evaluation
difficulty	easy	Determines tracks used for episodes
	medium	
	hard	
light_setting	bright	Determines light setting for episodes
	standard	
	dark	
jetbot_name	DifferentialJetBot	JetBot version for episodes
	FourWheelJetBot	
deterministic	True	If True, use deterministic sampling for policy actions
	False	
use_fresh_obs	True	If True, request new fresh observation from Unity before inference
	False	
record_videos	True	If True, record videos
	False	
log	True	If True, log all metrics to tensorboard
	False	
step	any positive integer	step for Tensorboard logging

Table 5.1.: Basic evaluation algorithm parameters

the training of the agent. If the agent's goal completion rate is low and there are no collisions, the agent might be stuck in a local optimum. In one such situation, the agent turned on the spot avoiding any collisions and did not move forward at all. The collision rate can be used to detect such situations, the training process can then be adjusted accordingly.

5.2. Basic Evaluation Algorithm

This section explains the most important component of the evaluation algorithms. It explains how the policy's performance is evaluated on a specific light and difficulty setting. The sampling mode, jetbot name and other policy parameters can also be specified 5.1. This basic component is widely used in the following tests.

The agent is evaluated by running a fixed number of episodes for the specific light and difficulty settings. The amount of episodes that the agent is evaluated on is defined by the config parameter *n_eval_episodes*. As described in 4.1, each difficulty setting includes a number of unique tracks. Furthermore the initial starting rotation is parameterized by the config parameter *spawn_point*.

The tracks and starting positions for the agent are generated by the algorithm shown in H.1. The algorithm divides the random interval specified by the spawn point parameter into *n_eval_episodes* equal parts. These spawn rotations are then each assigned a track in repeating order. This algorithm ensures that the agent is evaluated on the full range of unique tracks and spawn points. It also makes the evaluations comparable, as the same combinations of tracks and spawn points are used for each agent evaluation.

	bright	standard	hard	aggregate success_rates
easy	success_easy_bright	success_easy_standard	success_easy_dark	success_easy
medium	success_medium_bright	success_medium_standard	success_medium_dark	success_medium
hard	success_hard_bright	success_hard_standard	success_hard_dark	success_hard
aggregate success_rates	success_bright	success_standard	success_dark	total_success_rate

Table 5.2.: Collected and aggregate success_rate metrics

The evaluation algorithm initializes *n_eval_episodes* environments with the specified light settings. The obstacles and agents are then placed in the environments according to the generated map and rotation pairs. The evaluation episodes are started and the agents act in their environment until their episodes are terminated. The agent's actions are sampled deterministically or non-deterministically depending on the function call's parameter. The agent is executed in fresh or non-fresh observation mode depending on the function call's parameter.

The *success_rate* and *collision_rate* are calculated from the executed episodes, both metrics are returned to the caller. Many other metrics such as the *goal_completion_rate* and *average_episode_length* are calculated, they are logged to tensorboard if $log == True$.

5.3. Question 1 - Model Evaluation - Track Difficulties

The agent is evaluated on all three different difficulty settings to determine if the agent is successful in completing the tracks. The Basic evaluation algorithm is used with the standard light setting for each of the three difficulty settings. The policy is executed in non deterministic mode. This mode showed slight improvements over deterministic sampling, see 5.6.1

Test parameters 100 episodes are evaluated for every difficulty setting. The *spawnOrientation* from the training process is reused for the evaluation.

5.4. Question 2 - Model Evaluation - Light Settings

The agent is evaluated on every combination of light and difficulty settings. The Basic evaluation algorithm is used for these evaluations. The *success_rate* metric is collected for each combination of light and difficulty settings separately. The collected *success_rates* are then averaged to produce aggregate *success_rates* for each light setting and difficulty setting as well as the *total_success_rate*. All collected and aggregate *success_rates* are shown in 5.2. The experiment shwos if the agent is able to adapt to different light settings and if the agent's performance is influenced by the light settings.

5.5. Question 3 - Investigating the feasibility of transfering the policy to a physical robot.

Previous sections describe how a policy was developed that can be used to control an agent in a simulated environment. This section describes how I will investigate the feasibility of transfering the developed policy onto physical devices. The simulated agent was modeled after a Nvidia JetBot. The Nvidia JetBot is a small robot that is equipped with a camera, a processing unit and two motors that can be controlled independently. The Nvidia Jetbot is designed to be able to execute AI software. However the limited computational power of the JetBot raises the question whether the developed policy can be transfered to the JetBot.

The developed system takes a camera image from the front of the agent as input and applies preprocessing steps to the image. The preprocessed image is then processed by a CNN policy that outputs two acceleration values, one for each motor. The acceleration values are applied to the motors for a fixed amount of time. While the agent is moving, the camera image is constantly updated and the policy is applied to the new image. It is crucial that the agent's policy can be computed quick enough to be able act in real time.

5.5.1. Effects of Insufficiently Slow Policy Computation

If the jetbot is not able to compute the developed policy in the required time, there are two options that do not require changes to the developed/trained policy. The first option is to stop the motors until the policy is computed and then apply the acceleration values. This would make the jetbot movement overall slower and less smooth. The second option is to apply the last computed acceleration values to the motors until the new policy is computed. This could result in a degradation of the jetbot's performance since the actions would be less accurate. The two options are shown with visual representations in 5.2.

5.5.2. Policy Replay Experiment

This experiment examines the processing capabilities of the Nvidia Jetbot in the context of policy computation. The goal of this experiment is to determine if the policy can be computed on the jetbot hardware in real time. This would allow for a transfer of the developed policy to the physical agent without resorting to the two options highlighted in the previous section.

The experiment is conducted by recording the agent's actions in simulation and replaying them on the Nvidia Jetbot. The experiment measures the time it takes to replay the recorded actions on the jetbot.

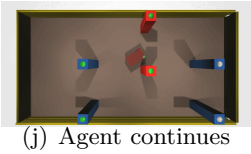
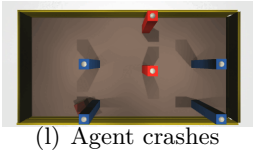
Policy computation in time	Option 1: Wait	Option 2: Apply previous outputs
 (a) Start	 (b) Start	 (c) Start
 (d) Agent turns right	 (e) Agent turns right	 (f) Agent turns right
 (g) Agent stops turning and goes strait	 (h) Agent waits	 (i) Agent continues to turn
 (j) Agent continues	 (k) Agent stops turning and goes strait	 (l) Agent crashes
Agent moves properly.	Agent overall speed is reduced.	Agent behaviour is changed.

Figure 5.2.: Possible effects of slow policy computation on the performance.

Recording Episodes Episodes are recorded in the simulation environment. The policy that is used for the recording is saved for later replay on the jetbot. The recordings consist of the agent's camera images and the corresponding policy outputs. The camera images are saved without the proprocessing steps applied. These images represent the raw camera input that the jetbot agent would receive in real time. The policy outputs are saved to verify the accuracy of the policy on the jetbot.

The episode recordings and policy are then transfered to the jetbot and executed.

Replaying Episodes The recorded episodes are replayed on the jetbot. The policy from the recording is used to execute the replays on the jetbot. All recorded images and policy outputs are loaded into memory in preparation of an episode replay. The episodes are replayed step by step. The replay of a step consists of all the processing to obtain new acceleration values from a camera image. Image preprocessing and memory mechanism are executed for each step according to the specifications of the saved policy. The policy outputs are then computed. The computed policy outputs and the recorded policy outputs are used to verify the accuracy of the saved policy on the jetbot.

The transfer of a NN to a new device can result in different outputs due to different hardware. The computed and saved policy outputs are compared to determine the accuracy of the policy on the jetbot.

The test measures the time it takes to replay the recorded steps on the jetbot. The measured times are compared against the $fixedTimestepsLength$ parameter of the recorded policy.

Test Dataset Episodes of all difficulty and light setting combinations are recorded to have a diverse set of data to evaluate. The episodes are recorded with the same policy as question 1 and 2, the most successful policy.

5.6. Other Experiments

5.6.1. Sampling Mode Performance Test

The PPO policy produces an action distribution when given an input observation. The action distribution is sampled to obtain an action for the agent to execute in the environment. The distribution can be sampled deterministically or non-deterministically. Deterministic sampling returns the most likely action, indicated by the mean of the distribution. Non-deterministic sampling returns a sample from the distribution according to the distribution's probabilities. This results in different output actions for the same observations. The PPO policy uses non-deterministic sampling during training to explore the action space.

While the exploration caused by non-deterministic sampling is beneficial during training, it could be detrimental during evaluation. This exploration could lead the policy to take actions that are not optimal for the given observation and result in a lower $success\_rate$. Non-deterministic sampling can also be used during evaluation to reduce overfitting. This has been used in the Atari paper

Mnih et al. [14] and the human-level control paper Mnih et al. [36]. The environments examined in these papers are deterministic with identical starting states. Therefore a deterministic policy would result in identical results for the individual evaluation episodes.

Due to the difference in environment properties between the JetBot environment and the Atari environment, it is not clear if deterministic or non-deterministic sampling is better for the JetBot environment. The sampling mode performance test evaluates the policy with deterministic and non-deterministic sampling to determine if the performance is affected by the sampling method.

The sampling mode performance test uses the Basic evaluation algorithm to evaluate the policy with derministic and non-deterministic sampling for each difficulty level. The test evaluates the policy only using the standard light setting to save evaluation time. The success rates for the two sampling modes are compared to determine if the policy's performance is influenced by the sampling method. The better sampling mode is used for the experiments for question 1 and 2.

5.6.2. Identical Start Condition Test

The environment is simulated in Untiy. The unity game engine is not fully deterministic [37]. This means identical actions in identical environment states can result in slightly different environment states. In addition the policy evaluation may use non-deterministic sampling for selecting actions 4.3.3.2. This results in different actions for identical observations.

Therefore, episodes with identical starting conditions could result in different outcomes. This would be problematic when evaluating the agent since the evaluation results would not be reliable.

The identical start condition test evaluates the agent on multiple episodes with identical starting conditions. The episode results are analysed to see if the policy is consistent when given identical starting conditions. If the policy is inconsistent given identical starting conditions, the evaluation results according to 5.2 are not reliable. The evaluation algorithm described in 5.2 evaluates the agent on a series of different starting conditions, each starting condition is unique and only evaluated once.

This test runs multiple episodes with identical start conditions and analyses the episodes. The episode results are characterized and grouped. The groups are then analysed to determine if the agent's performance is consistent given identical starting conditions. Ideally all episodes would result in the same outcome. The episode results are characterized by the endEvent, collision and the three goals' completion.

5.6.3. Fresh Observation Test

In the standard RL algorithms each step transition results in a new observation that represents the environment's new state. Calls to the Unity environment and the step transitions in the environment take time. In order to speed up the training process, the step calls to the simulated environment have been implemented in a non-blocking way. As a result the observations are delayed by one step's duration. This is explained in 4.1.1.

The option *use_fresh_obs* controls what observations the policy uses to produce the next actions. If *use_fresh_obs* is set to false, the observation returned by the last step call is used. If *use_fresh_obs* is set to true, a new observation is requested from the Unity simulation.

The fresh observation test evaluates a trained policy with both *use_fresh_obs* settings to determine if the agent's performance is influenced by the freshness of the observations. The trained policy used one setting of the *use_fresh_obs* parameter during training. The test shows if there is a difference in the agent's performance when using the other setting. The test evaluates the agent on light and difficulty settings following the evaluation algorithm described in 5.2.

5.6.4. JetBot Generalization Test

There are two versions of the agent in simulation, the DifferentialJetBot and FourWheelJetBot 4.8. The DifferentialJetBot has two front wheels that are accelerated independently for steering. The FourWheelJetBot angles its two front wheels for steering. These differences can result in different agent movement for the same acceleration inputs. The DifferentialJetBot is used during training as this is closer to the physical jetbot availible at the Scads.AI. The agent's movement behaviour influences the developed policy. The agent learns from the changes in the environment that result from its actions. The policy might perform worse when the agent's movement behaviour changes.

A policy trained on the DifferentialJetBot might not generalize well to the FourWheelJetBot and vice versa. The JetBot generalization test evaluates the trained policy on both JetBot versions to determine if the policy can be transfered to the FourWheelJetBot without retraining. The test evaluates the agent on the different difficulty levels using the basic evaluation algorithm described in 5.2. The test is only executed for the standard light condition to save time.

6. Parameter Search for the RL Training

This chapter explains how suitable parameters were found for training a successful policy. All parameters can be set in the training configuration file. The configurations that were used for the experiments are referenced in the appendix C.

6.1. General Parameters

There are a number of training parameters that were kept constant throughout the later training and evaluation runs. These parameters were chosen based on initial experimentations and improvements during the development phase.

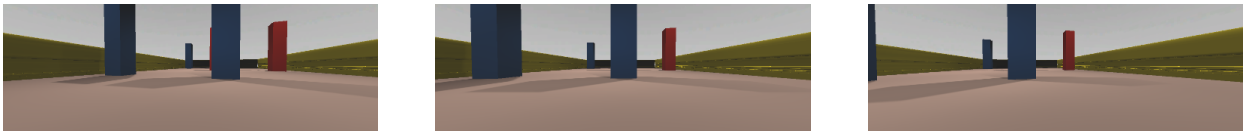
6.1.1. Initial Agent position

The agent always starts the episodes at the same position, only the agent orientation can change. There are three options for the spawnOrientation. The experiments showed that it is possible to train the policy very reliably with *Fixed* and *Random* agent orientation. It was not possible to train the policy with *VeryRandom* spawnOrientation.

The *Random* orientation was chosen for all further experiments. The agent is thus spawned with an orientation in the range of $[-15°, 15°]$ during training and evaluation. This increases the difficulty of the task compared to *Fixed* orientation. The policy has to learn more diverse starting conditions.

6.1.2. Agent Camera

The agent camera is used to obtain observations from the environment. The agent's camera resolution is important and can heavily impact the agent's policy. The width and height of the agent camera image were set to $500x168$. This ensures that the agent has a realistic field of view. The agent's field of view is wide enough to partially observe the first goal at the beginning of the episode with *Random* spawnOrientation 6.1.



(a) minimum $-15°$ (b) neutral $0°$ (c) maximum $15°$

Figure 6.1.: Agent field of view at beginning of episode with *Random* spawnOrientation

6.1.3. Observation Space

The observation space defines the input dimensions of the policy NN. The dimensions are determined by the agent camera image width and height, the preprocessing steps and the memory mechanism. All further experiments used the same settings for downsampling, grayscaling and memory mechanism. This results in the same observation space for all trained models.

Preprocessing steps A downsampling factor of 2 was used. The image was then converted to grayscale. This means that the agent camera image was downsampled by a factor of 2 along each axis and the channel dimension was reduced from 3 to 1.

The histogram equalization preprocessing step was not used in every experiment. The histogram equalization step does not change the shape of the observation space. THerefore it can be added or removed without requiring a change in the policy.

Memory Mechanism The $frame\_stacking$ parameter was set to 10. This means the policy can use the last 10 steps for its decision making. Given the fixed step duration of 0.3 seconds, the policy can use information from the last 3 seconds.

Result The agent camera image had a resolution of $500x168$ which was downsampled to $250x84$. The grayscaling step removes the three color channels. The memory mechanism stacks the last 10 agent camera images along the channel dimension. This results in an observation space of shape $[84, 250, 10]$ with the range of $[0, 255]$.

6.1.4. Step Duration

A fixed step duration of 0.3 seconds was used for all experiments. This means that each action is applied to the environment for 0.3 seconds in the Unity simulation. New steps are only started when this duration has passed. The fixed step duration was chosen to ensure that the agents behave similar on different devices, regardless of the processing speed. This was usefull as the policy training and evaluation runs were executed on multiple devices.

The fixed step duration of 0.3 seconds results in very small movements for each step. These frequent and small steps allow the policy to make precise movements. Single steps are almost not noticeable as shown in table 6.1. Successful episodes for hard tracks require about 200 steps for completion.

6.1.5. Remaining Parameters

The remaining hyperparameters concern mostly the PPO training algorithm. The training algorithm was adapted from the SB3 library by Raffin et al. [31]. The parameters were chosen experimentally to ensure that the training process was successful and could be executed on the different training devices. The parameters and their tradeoffs are quickly summarized in the table 6.2.

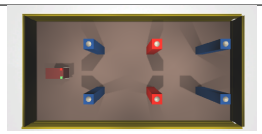
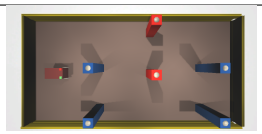
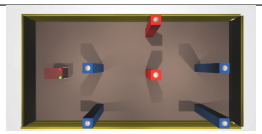
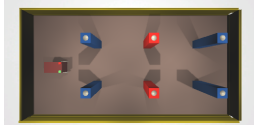
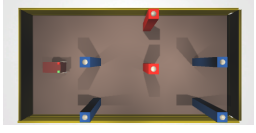
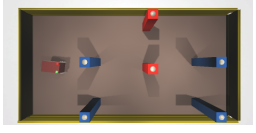
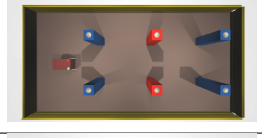
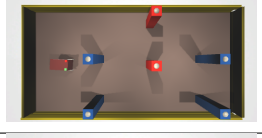
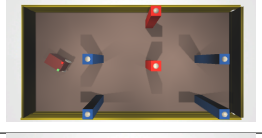
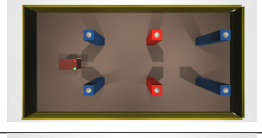
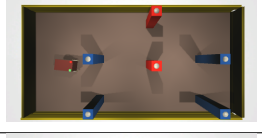
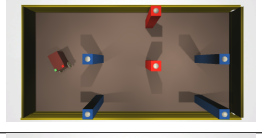
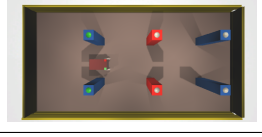
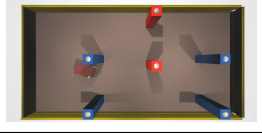
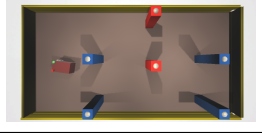
behaviour description	full speed ahead	steer right	turn on the spot
action	(1, 1)	(1, 0)	(1, -1)
initial position			
position after step 1			
position after step 5			
position after step 15			
position after step 30			

Table 6.1.: Agent movement with fixed step duration 0.3 seconds

parameter	function	value	tradeoffs
n_steps	amount of samples to collect per environment during data collection	256	+diversity of collected samples +stability of policy -data collection time -memory requirement
batch_size	amount of samples for loss function calculation	64	
n_epochs	amount of NN weight updates for the collected samples	5	+sample efficiency -stability of policy
n_envs	amount of environments to simulate in parallel	10	+parallel episode simulation -performance Unity editor
use_fresh_obs	request fresh obs before inference	False	-increased communication -data collection time
use_bundled_calls	bundle calls for parallel environments	True	+reduced communication
seed	seed for NN initialization and random number generators	2048	+fixed NN initialization

Table 6.2.: Selected hyperparameters for the PPO algorithm

function name	encouraged behaviour	learned behaviour	expected behaviour learned?
eventReward	agent drives through the track without collisions	agent turns on the spot continuously	no
distanceReward	agent drives towards the next goal	agent drives towards the next goal	yes
orientationReward	agent turns towards closest goal	agent turns around on the spot continuously	no
velocityReward	full speed ahead (no turning)	full speed ahead (no turning)	yes

Table 6.3.: Policy capability check for individual reward functions.

6.2. Reward Functions Capability Check

Previous sections introduced the composite reward function. The composite reward function consists of a weighted sum of individual reward functions. The individual reward functions are designed to encourage the policy to learn the desired behaviour. The goal is to train a policy that guides the agent through the track without collisions. This is encapsulated in the event reward function. However the event reward function is a very sparse signal, which makes it hard for the policy to learn from. The other individual reward functions are designed to be dense reward functions, they give rewards in every episode step.

It is important to find appropriate weights of the individual reward functions for the composite reward function. Experiments are conducted to analyse the usefullness of the individual reward functions. The experiments test if the policy is capable of learning the behaviour encouraged by the individual reward functions. This is done by training the policy with only one reward function at a time. The reward function's coefficient is set to one. The coefficients of the other reward functions are set to zero. The policy is trained on easy tracks with a Random spawnOrientation and standard light setting to reduce the difficulty of learning the encouraged behaviour.

The experiment results are shown in 6.3 and appendix D.2. The policy is capable of learning the behaviour encouraged by the distanceReward and velocityReward functions. However the policy is not capable of learning the behaviour encouraged by the eventReward and orientationReward functions.

6.3. Chosen Reward Function

As a result of the capability check, the distanceReward function was determined to be most suitable for the training process. The distanceReward function was also tested on hard tracks and showed promising results. However the distanceReward alone was not able to train an policy that avoids collisions entirely.

$$R(s_t, a_t) = \Delta distance(Agent, NextGoalPosition) \cdot \Delta T$$

Figure 6.2.: Final reward function
s_t: state t a_t: action in state t

Further tests of combining the distanceReward with the eventReward function were carried out. The goal was to find parameters for the composite reward function that would allow the policy to learn to complete the tracks without collisions. The distanceReward function alone does not penalize collisions as heavily as the eventReward. Experiments were executed using both functions together with different coefficients. The behaviour of the policy and the obtained rewards were monitored during training. The results showed that the policy was not capable of leaning from the combined rewards, regardless of the combinations of coefficients.

The distanceReward showed the best results when used alone. The policy was able to learn to complete the tracks reliably. The policy did not learn to avoid collisions completely. This is further dicussed in 6.8.

The distanceReward function was chosen as the only reward function for the final training process. The other reward functions function were not used in the training process. The coefficient for the distanceReward was set to 1, the others were set to 0. As a result the total reward function is defined as 6.2.

6.4. Experiments Training with Fixed Difficulty Settings

In the experimentation phase of this project the policies were trained exclusively on tracks of a specific difficulty setting. This was done to analyse the capability of the policies and to find appropriate hyperparameters that allow the policies to learn. These hyperparameters include the agent camera image dimensions, the $fixedTimestepLength$, the amount of steps to collect in each iteration and more.

The experiments showed that it is possible to train the policy with the right hyperparameters to solve tracks of a particular difficulty level when the policy is trained exclusively on that difficulty level. For example the policy could be trained to solve the medium tracks very successfully without needing to encounter easy or hard tracks during training. The light settings were restricted to standard during this experimentation to focus on the difficulty levels.

The experiments produced a set of hyperparameters that could be used for training the policy on the different difficulty levels exclusively. The policies were able to reach success_rates of 100% for the collected episodes. The policies generated in this section are called Easy Tracks Standard Light Policy (ES-P), Medium Tracks Standard Light Policy (MS-P) and Hard Tracks Standard Light Policy (HS-P) for easy, medium and hard tracks respectively.

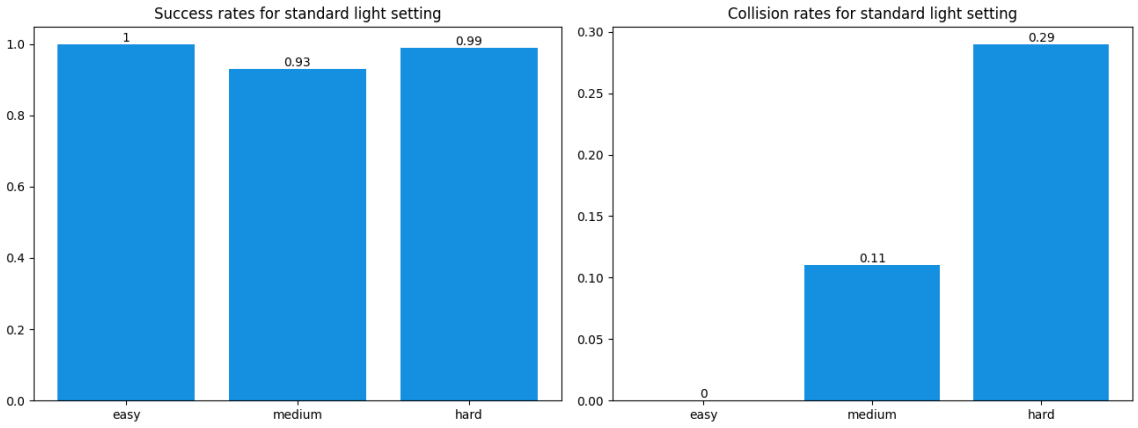


Figure 6.3.: Success and collision rates for all difficulties for an agent with HS-P

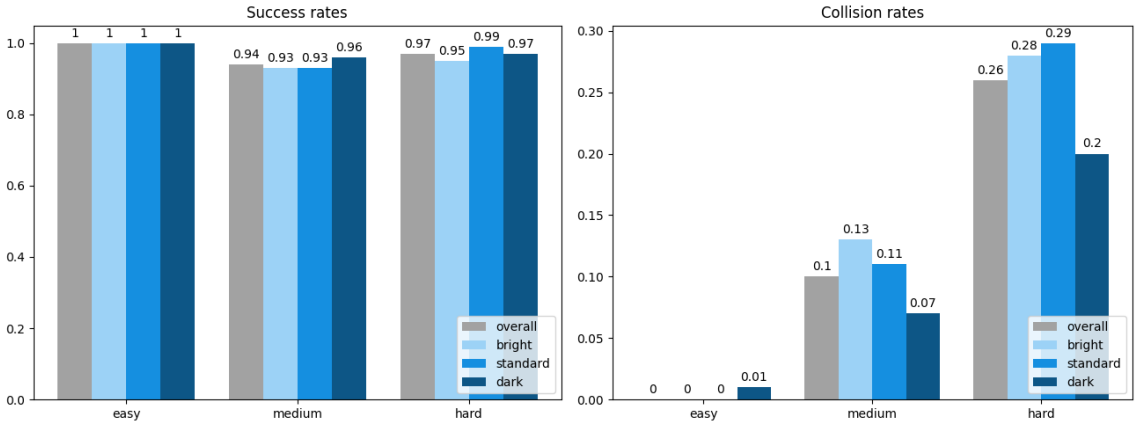


Figure 6.4.: All success and collision rates for an agent with HS-P

Generalization to lower difficulty levels The evaluation of these policies showed that they were able to generalize to the tracks of lower difficulty levels. For example the policy that was only trained on hard tracks with standard light conditions could solve the easy and medium tracks as well. The agent trained on hard tracks and standard light setting was able to solve hard tracks with a success_rate of 99%. It achieved a success_rate of 100% and 93% on easy and medium tracks without being trained on these difficulty settings 6.3.

Generalization to different light settings Further evaluation of ES-P, MS-P and HS-P on different light settings showed that the policies were able to generalize to the different light settings as well. The policies were trained on the standard light setting only. However the policies were able to solve episodes with the bright and dark light settings with only slight decreases in performance. The success rates for the different light settings are shown in figure 6.4.

The histogram equalization preprocessing step was used during the training and evaluation of this policy. Experiemnt show that this step is mostly responsible for the good performance at different light settings, see 6.7.

6.5. Experiments Training with Mixed Difficulty Settings

The next step was to train the policy on all difficulty levels at the same time. The goal was to find hyperparameters that would allow the policy to learn to solve all difficulty levels better than with the previous exclusive training. The policy was trained on all difficulty levels at the same time with the same hyperparameters that were used before. The *trainingMapType* parameter was set to *randomEval*, this results in split of 20% easy, 40% medium and 40% hard tracks.

The training with mixed tracks for the data collection episodes failed. The policy was not able to learn to solve the tracks as successfully as the isolated training on hard tracks. Given HS-P performed very well on all tracks, the following policies are trained exclusively on hard tracks as well.

6.6. Experiments Training with Mixed Light Settings

In the previous experiments, the complexity of the task was reduced by focusing on single light settings. The goal was to test the parameters and find configurations that enable the policy to learn the tasks with increasing complexity. The experiments showed that a policy that is trained on hard tracks exclusively is able to generalize to the lower difficulty tracks as well. These policies were also able to solve the tracks of differenct light settings with only slight differences in performance.

The next step was to train the policy on all light settings at the same time. The goal was to find parameters that would allow the policy to learn to solve the tracks with different light settings better than with the previous exclusive training. The new Hard Tracks Mixed Light Policy (HX-P) was trained on difficult tracks with all light settings. The *lightSetting* parameter was set to *random*, this results in split of 33% bright, 33% standard and 33% dark illumination for the data collection episodes.

Resuts for standard light setting HX-P was able to learn to solve the tracks of different difficulties. Similarly to the previous experiments, the policy was able to generalize to the easy and medium tracks. The policy was able to solve the easy, medium and hard tracks with success rates of 100%, 100% and 99% 6.5.

This is an improvement over HS-P which was policy trained exclusively on hard tracks with standard light. This improvement may be due to the increased variety of data encountered during the training phase.

Performance on all light settings HX-P was evaluated on the different light settings. The policy was able to solve the episodes reliably for all light settings. There were only slight differences in performance between the light settings for hard tracks 6.6.

This policy performed much better in episodes with bright or dark light setting compared to HS-P. This was to be expected as this policy was trained on all light settings.

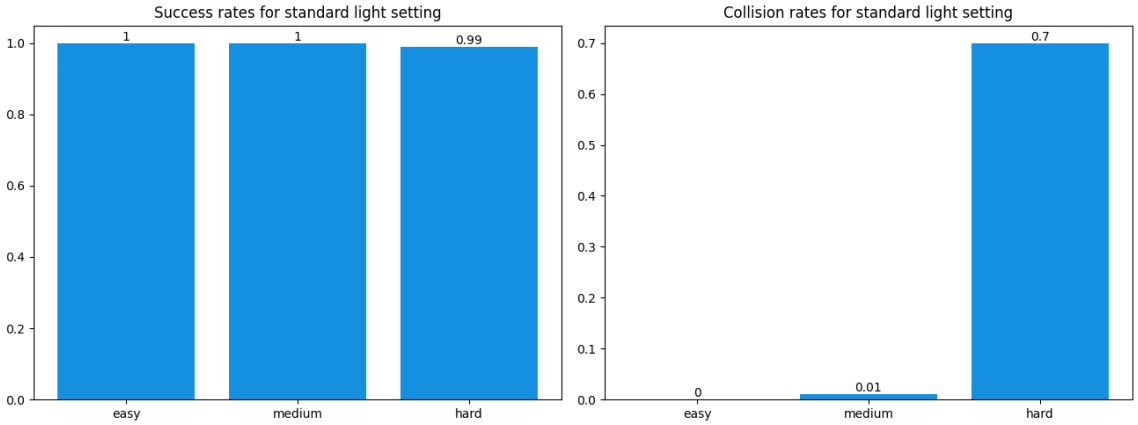


Figure 6.5.: Success and collision rates for all difficulties with standard light for HX-P

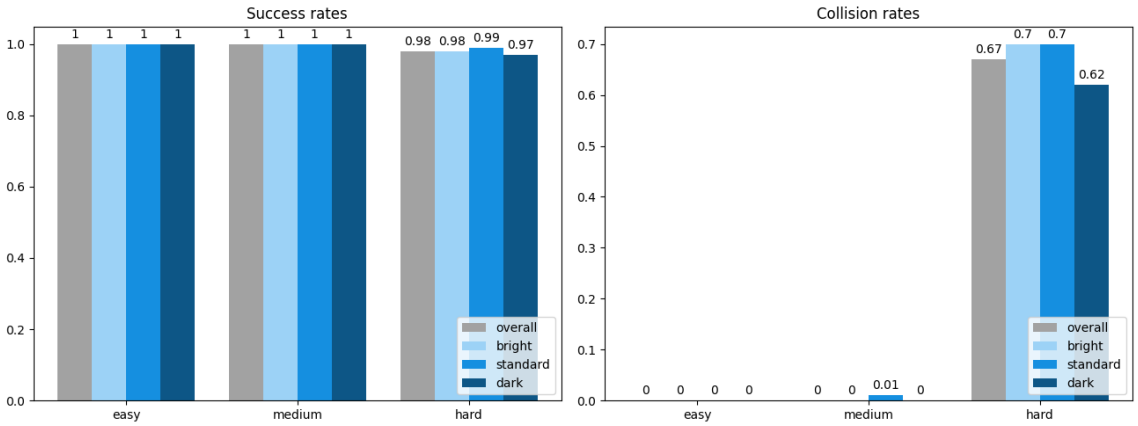
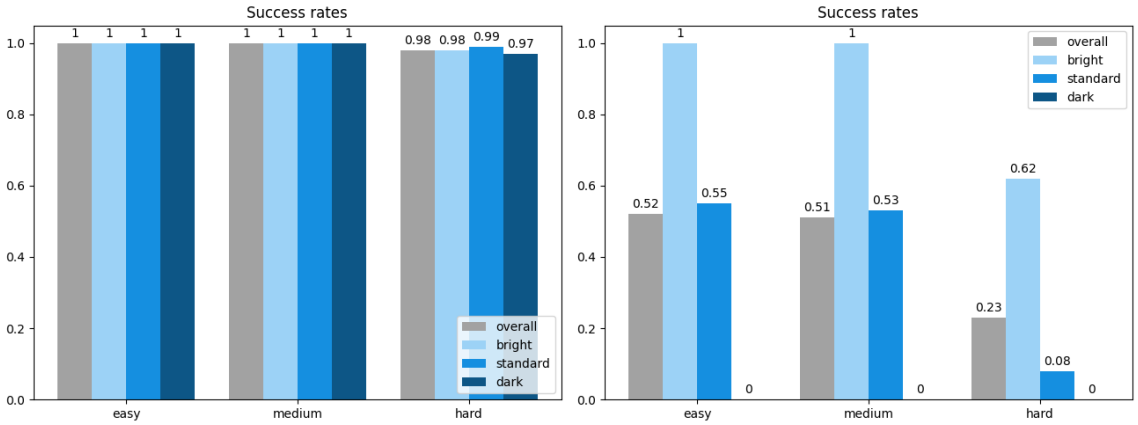


Figure 6.6.: Success and collision rates for a policy trained on hard tracks with all light settings HX-P



(a) Success rate with histogram equalization (b) Success rate without histogram equalization

Figure 6.7.: Success rates for HX-P

6.7. Experiments on the Importance of the Histogram Equalization Preprocessing Step

The previous training results showed that HX-P was able to learn to solve the tracks with different light settings. The histogram equalization preprocessing step was applied to the preprocessed image 4.2.1.3 to increase the policy's resilience towards light changes.

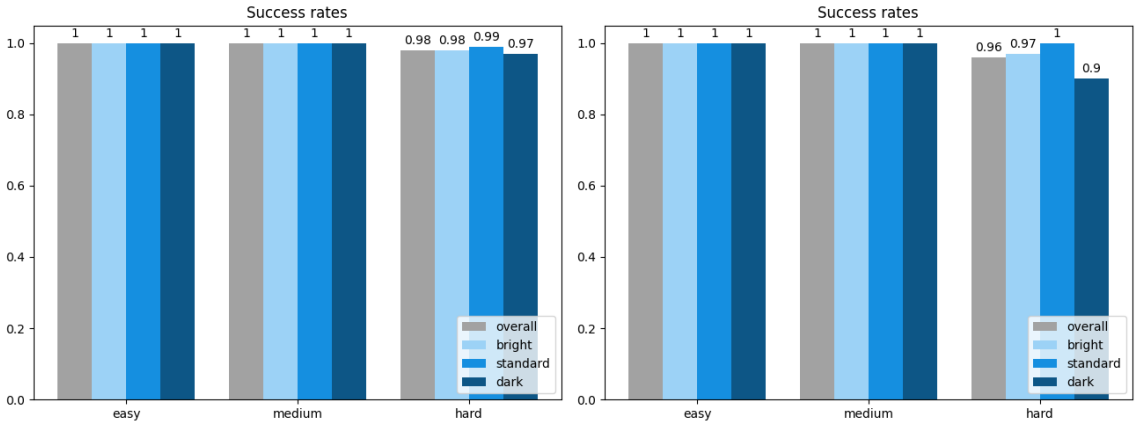
HS-P was trained on the standard light setting and was able to solve tracks under bright and dark light conditions as well. This implies the histogram equalization plays a big role.

Evaluating a policy without the histogram equalization preprocessing step The policy from the previous section about mixed light settings during training HX-P was trained with the histogram equalization preprocessing step. This policy was now evaluated without the histogram equalization step.

The removal of the preprocessing step leads to a performance collapse for the standard and dark settings 6.7. The success rate drops to 0% for the dark light setting. The performance for the bright light setting is not affected as severely. Nevertheless the success rate drops from 98% to 62% for the hard tracks.

The decrease in performance was to be expected, as the policy was trained with the histogram equalization preprocessing step. The histogram equalization step changes the image content substantially 4.1. The policy has learned to use the output of the preprocessing step. As a result, the policy is not able to solve the tracks without the preprocessing step.

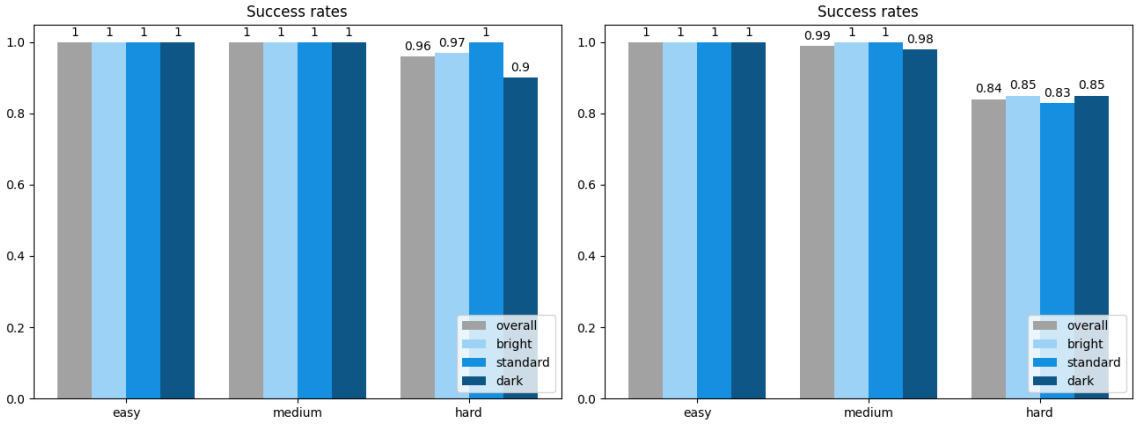
Training a policy without the histogram equalization preprocessing step The Hard Tracks Mixed Light without Histogram Equalization Policy (HX-noHist-P) was trained without the histogram equalization preprocessing step on all light settings. The goal was to determine if the preprocessing step is required for high performance. The results are shown in the second graph in 6.8.



(a) Policy trained with histogram equalization (HX-P)

(b) Policy trained without histogram equalization (HX-noHist-P)

Figure 6.8.: Success rate comparison for policies trained with and without histogram equalization on all light settings



(a) Policy evaluation without histogram equalization

(b) Policy evaluation with histogram equalization

Figure 6.9.: Success rates for HX-noHist-P

The policy was not able to learn to solve the hard tracks as successfully as the policy trained with the histogram equalization preprocessing step. The performance difference is concentrated on the hard tracks.

The HX-noHist-P was trained without the histogram equalization preprocessing step. Adding the histogram equalization step after the training results in a decrease in performance, shown in 6.9.

Conclusion The histogram equalization step contributes strongly to the resilience to light changes for the trained policies. It is possible to train a resilient policy without the histogram equalization preprocessing step. However the best performance is achieved when the policy is trained with the histogram equalization preprocessing step and all light settings.

The histogram equalization preprocessing step should never be removed or added after the training is complete, this results in a performance decrease.

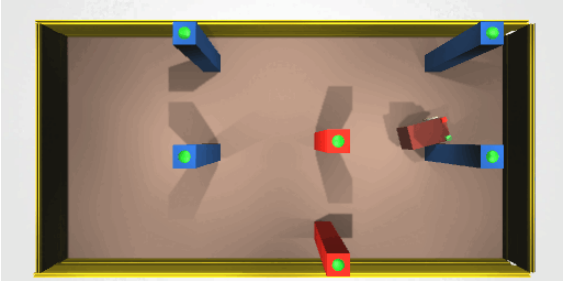


Figure 6.10.: Example of a collision from a successfully completed episode

6.8. Comment on the Collision Rate

The trained policies are able to complete the tracks very reliably. However the policies do not avoid collisions entirely. The collision rate generally increases for higher difficulty tracks. The collision rates vary quite strongly between the different trained policies. The HX-P policy has a collision rate of 70% for the hard tracks with standard light setting. Whereas the HS-P policy has a collision rate of only 29%.

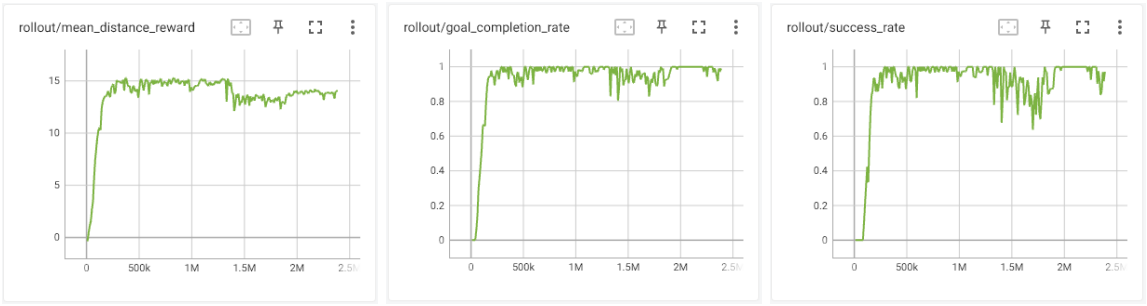
The agents are able to complete the tracks reliably despite the collisions. The analysis of successful episodes with collisions shows that these collisions are only minor collisions. The agent collides with the goal obstacles at its side. The agent basically scapes against the goal posts. The agent does not collide with the goal obstacles head on 6.10.

The reason for the high collision rates is the selected reward function. The policies were trained with the distance reward function exclusively following the first experimentations. The distance reward function does not penalise collisions directly. The distance reward function only discourages collisions indirectly. Collisions that prevent an agent from moving towards the next goal result in less collected reward. For example a frontal collision of the agent with a goal post object requires the agent to then move backwards and around the goal post. The backwards movement is penalized. The policy learns to avoid frontal collisions.

The minor collisions that occur are not penalized as the agent is still able to move towards the next goal. As a result the policy does not learn to avoid these collisions entirely.

6.9. The Most Successful Policy HX-P

This section summarizes how the most successful policy HX-P was implemented and trained. The parameters were chosen following the previous experiments. This policy is used for the evaluations of the three main research goals. The policy was trained using the distance reward alone. It was trained on hard tracks with all light settings. The policy used the histogram equalization preprocessing step.



(a) Distance Reward (b) Goal completion rate (c) Success rate

Figure 6.11.: Properties of the collected episodes over time for the most successful model

Policy Settings The policy was configured to use the three preprocessing steps downsampling, grayscaling and histogram equalization. The policy used a frame stacking of 10. The agent camera image had a resolution of $500x168$ which was downsampled to $250x84$. This results in an observation space of shape $[84, 250, 10]$. The duration of steps was fixed to 0.3 seconds. The NN follows the earlier descriptions in 4.3.3 exactly. It consisted of a feature extractor part with 3 convolutional layers and one fully connected layer. The feature extractor takes the observation as input, its outputs are used by the action and value head. The action and value head each consist of a single fully connected layer. The entire network is trained together end-to-end using backpropagation of the loss defined by the PPO algorithm.

Reward function The policy was trained exclusively using the distance reward. The other reward functions were not used, see 6.2.

Training data The episodes for data collection used hard tracks with all light settings. The agent was initialized with random orientations. The episodes were not reset upon collision.

Training algorithm settings The training was configured to use 10 environments in parallel to collect data. 2560 samples were collected in each data collection step. The training was terminated after a total of 2385920 steps. 8423 full episodes were executed to collect these samples. 233 data collection and model training iterations were executed in total. The total training duration was 1.02 days. The training progress can be seen in figure 6.11. The model reached a success rate of 100% for the collected episodes.

The full training config can be found in the appendix C.2.

7. Challenges

7.1. Connecting the Python Algorithm and Unity Simulation

The first challenge was to connect the Python algorithm with the Unity simulation. The Python algorithm is responsible for training and evaluating the agent. The agent's decisions are also executed in Python, this includes the preprocessing steps, NN policy and action sampling. The Unity simulation is responsible for rendering the environment and providing the agent with observations and rewards. The environment is wrapped in a gymnasium environment [29], this way it can be integrated with many existing RL libraries and algorithms. The unity simulation acts as a RPC server, the python gymnasium environment acts as the client.

Data exchange The first challenge was to transfer the image data from unity to python. This was solved by encoding the image data as a base64 string and sending it back to python over the network via RPC. The image data was then decoded in python and converted to a numpy array.

Communication Speed The seperation of the simulation and the RL algorithm introduces a delay for all interactions with the simulated environments. This is most noticeable in the data collection part of the training loop, since the environment has to execute the actions of the agent and send back the new observations and rewards. The RL library SB3 by Raffin et al. [31] was used for the training of the agent. This library is built to enable the parallel execution of multiple environments. However the library requires that actions are performed on all environments at the same time. Together with the delay caused by communication, this parallel execution would slow down significantly.

The solution to this problem was bundling the actions for the different environments and sending them to Unity in one batch. The unity server then executes the actions on the individual environments. This way the delay caused by the communication overhead is only introduced once per batch of actions.

Seed The PPO algorithm from the SB3 library was used to train the agents. This algorithm can be made deterministic by setting a seed. However the Unity game engine is not built to be fully deterministic. The RL algorithm and the environment simulation in Unity interact during the training and evaluations.

As a result, the training progress and all agent evaluations cannot be made fully reproducible.

The configuration files include a seed parameter. This seed is used to seed the random number generators of python and the RL library. This way at least the model initialization is reproducible.

7.2. Parameters for training

The policy and the policy training process use extensive parameterizations. The parameters can have an impact on the agent's overall capability (e.g. histogram equalization) and the training progress (e.g. reward function parameters).

The search for the optimal parameters is a time-consuming process. Chapter 6 describes how the parameters for the training were found via experiments.

8. Results

This chapter presents the results of the experiments conducted in this project. The first three experiments were designed to answer the research questions posed in chapter 2 specifically.

The experiments were conducted using the policy that was developed as a result of the parameter experimentation 6.9. This policy was trained exclusively on hard tracks with all light settings. The histogram equalization preprocessing step was used during training.

8.1. Eval for question 1 - Is it possible to train an autonomous driving agent consisting of a CNN policy with end-to-end reinforcement learning to reliably solve the tracks of all difficulty levels?

The trained policy was evaluated on tracks of all difficulty settings with the standard light setting using the Basic Evaluation algorithm 5.2 with 100 episodes per setting. The success rates for the different difficulty settings are shown in figure 8.1. Example videos of the agent's behaviour during evaluation can be found in the appendix D.

8.1.1. Experiment Results

The policy completed the easy, medium and hard tracks with a succes_rate of 100%, 100% and 99%. The collision_rates were 0%, 1% and 70%. Especially for the hard difficulty setting, the agent does not avoid collisions completely.

The analysis of successful episodes with collisions shows that these collisions are only minor collisions. The agent passes the goals very close to the goal post that is closer to the arena middle 8.2. This often results in collisions with the goal obstacles at its side.

The unsuccessful episodes of hard tracks were analysed to determine why the agent does not reach 100% success rates. The analysis showed that the agent has frontal collisions with the first goal post

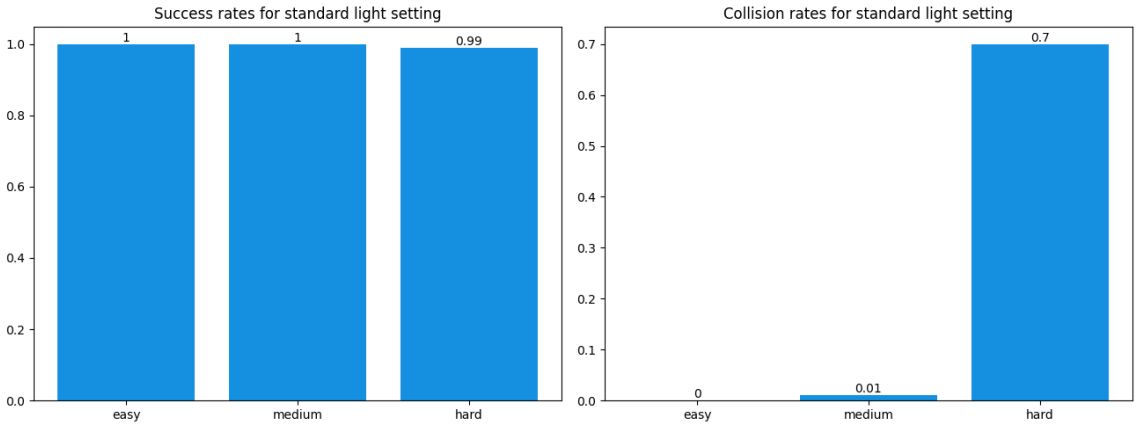
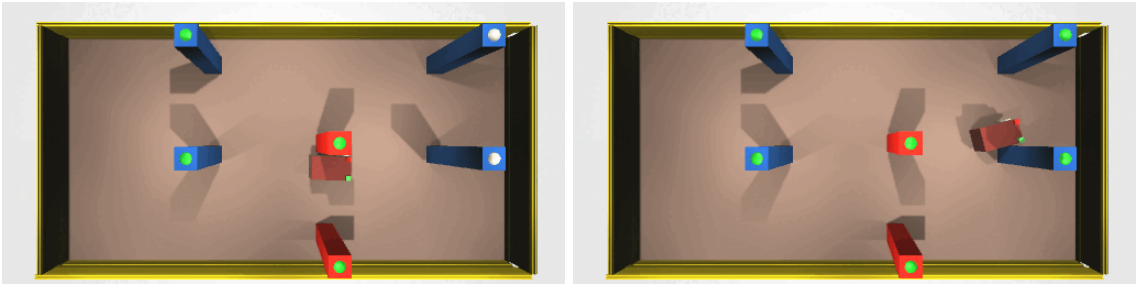
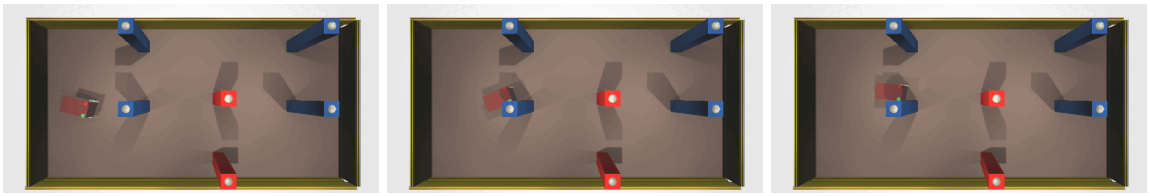


Figure 8.1.: Success and collision rates for standard light setting.



(a) Agent passes the goal close to the arena middle (b) Agent may collide with goal post at the side

Figure 8.2.: Goal passing behaviour of the trained agent



(a) Agent is spawned with extreme rotation (b) Agent approaches first goal post (c) Agent collides with first goal post and remains stuck

Figure 8.3.: Analysis of unsuccessful episodes

at the agents side 8.3. This occurs only when the agent is spawned with an extreme rotation. The agent cannot complete the tracks in about 40% of episodes with the maximum rotation of 15°.

8.1.2. Conclusion

The trained policy is able to complete all difficulty levels very reliably. The policy reached success rates of 100%, 100% and 99% for easy, medium and hard tracks. The end-to-end trained CNN policy proves to be very effective at solving the autonomous driving task.

The policy outperforms previous work at the ScaDS.AI by Schaller [1]. The policies developed by Schaller [1] reached success rates of 99%, 89% and 64% for easy, medium and hard tracks.

The policy was trained on the difficult setting only. The policy did not see easy and medium difficulty tracks before the evaluation. The policy is able to generalize to tracks of lower difficulty.

8.2. Eval for Question 2 - Is it possible to use an end-to-end trained CNN policy to make the agent robust to changing light conditions?

The most succesfull model was used to evaluate the agent's performance under different light settings. The agent was evaluated on the standard, dark and bright light settings. Each combination of difficulty and light setting was evaluated with the Basic Evaluation Algorithm for 100 episodes. The success rates for the different light settings are shown in figure 8.4.

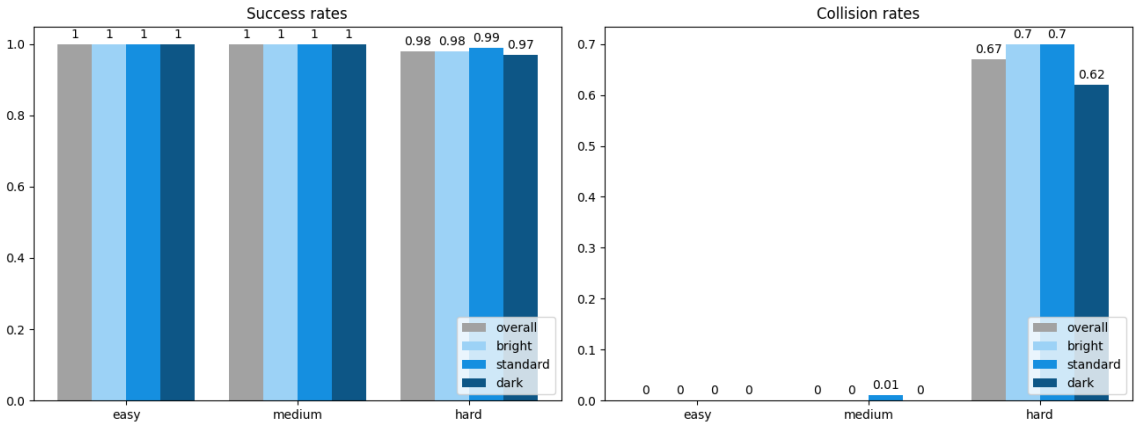


Figure 8.4.: Success and collision rate comparisons for light settings.

8.2.1. Experiment Results

The success rates for all light and difficulty settings are very high at above 95%. The agent completed the easy and medium tracks with a success rate of 100%. The light setting had no influence on the success_rate for the easy and medium tracks. The evaluations for the bright and dark light settings show slight decreases in performance compared to the standard light setting for hard tracks. The success_rate of hard tracks in the bright and dark light setting was 98% and 97%.

The collision rates are very low for the easy and medium tracks. All tracks in these settings were solved without collisions except for the medium tracks with the standard light setting. This medium standard evaluation had a collision rate of 1%. The collision rates for the hard tracks are quite high at 70% for the standard light setting. The bright and dark light settings are very high as well with 70% and 62%.

Across all light and difficulty settings the overall success rate is almost 100% and the overall collision rate is 22%.

8.2.2. Conclusion

The policy was able to complete the tracks very reliably under the different light settings. The light setting only had a minor impact on the success rate. Furthermore the different light settings did not lead to an overall increase in collisions.

The developed policy was trained following the experiments in the previous chapter. The agent used the histogram equalization preprocessing step and mixed light settings during training. Using these settings the policy was able to learn to navigate the tracks with all light conditions.

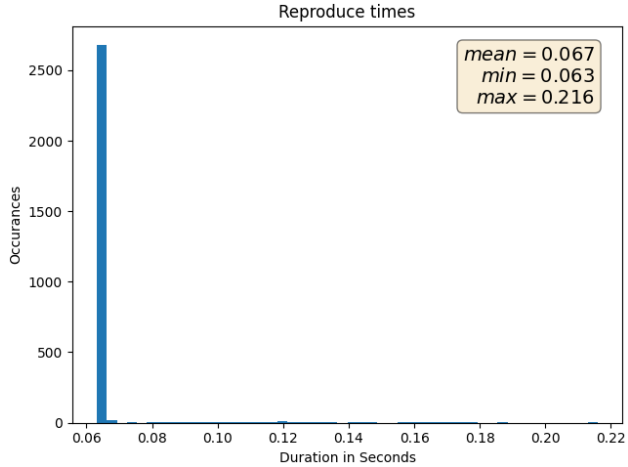


Figure 8.5.: Replay times on jetbot hardware

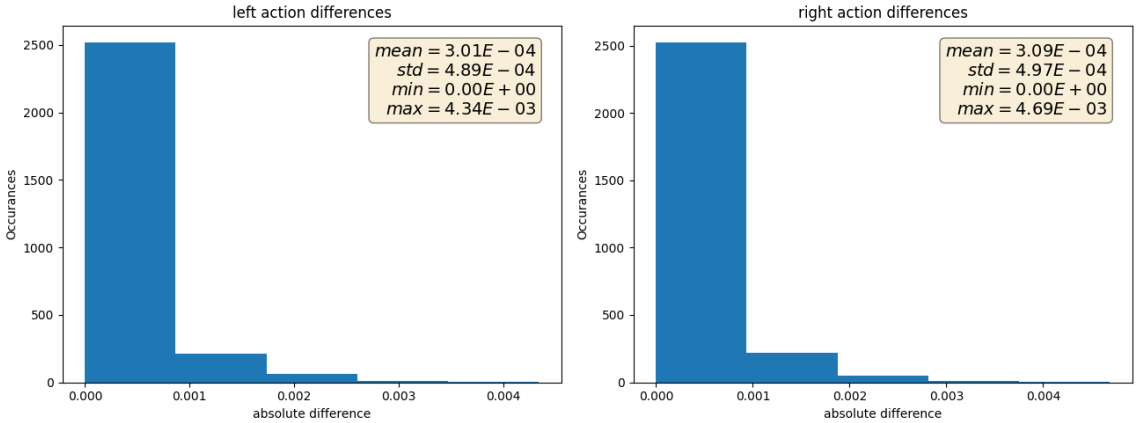


Figure 8.6.: Differences in policy outputs between recordings and replays on jetbot hardware

8.3. Eval for Question 3 - Is it possible to use a NN policy that can be executed in real-time on a physical robot?

The most successful model was used to record replays of the agent's behaviour. This most successful model used a $fixedTimestepLength$ of 0.3 seconds. This means the hardware has to able to make a decision every 0.3 seconds to be considered fast enough.

Five episodes were recorded for each difficulty and light setting combination. As expected from the previous evaluations of the policy, the replay episodes had a very high success_rate of 100%. The recorded episodes were then replayed on the Nvidia Jetbot. The time it took to replay the episodes was measured 8.5. The policy outputs from the replay were compared to the policy outputs from the recordings 8.6. The differences are caused by the hardware and software differences between the training and the replay environment.

8.3.1. Experiment Results

Replay times The replay times for the recordings on jetbot hardware are shown in figure 8.5. The maximum duration was 0.216 seconds. The mean is much lower at 0.067 seconds. The plot

shows that the maximum duration was an extreme outlier. Given the $fixedTimestepLength$ of 0.3 seconds and the maximum duration of 0.216 seconds, the hardware is fast enough to replay the episodes. This leaves at least 0.084 seconds for the agent to receive an image from the camera and send the new instruction to the motors.

The cameras used in the nvida jetbot are capable of capturing images with a resolution of $1280x720$ pixels at 60 frames per second. This means the camera can capture an image every 0.0166 seconds. The hardware is quick enough to compute actions in real time.

Policy Outputs The policy outputs from the recordings and the replays on jetbot hardware are nearly identical. The differences are shown in figure 8.6. The outputs were reproduced very closely. The maximum difference was $4.69e-03$. This is difference is negligeable compared to the range of policy outputs $[-1, 1]$.

8.3.2. Discussion

The jetbot hardware is capable enough to compute the policy in real time. The differences of the policy outputs between the recordings and the replays are very small, the policy outputs were reproduced very closely. This suggests the differences in hardware and software do not impact the policy significantly.

8.4. Other Experiments

8.4.1. Sampling Mode Performance Test

The sampling mode performance test uses the Basic evaluation algorithm to evaluate an agent with derministic and non-deterministic sampling for each difficulty level. The success rates for the two sampling modes are compared to determine if the agent's performance is influenced by the sampling method. The sampling mode test was executed for all agents during the experimentation phase of this project using standard light.

Results The tests showed that the difference in performance for the sampling modes was very small for the trained policies. The results showed empirically that non-deterministic sampling leads to better success rates. Nevertheless some policies performed slightly worse using the non-deterministic sampling mode. The success rate was on average 1.7% higher when using non-deterministic sampling. The biggest difference was 2.6% for the medium difficulty setting 8.7.

Discussion The test indicates that the non-deterministic sampling mode performes better, although the differences in performance are quite small. Given these results, the main evaluations for question 1 and 2 were conducted with non-deterministic sampling.

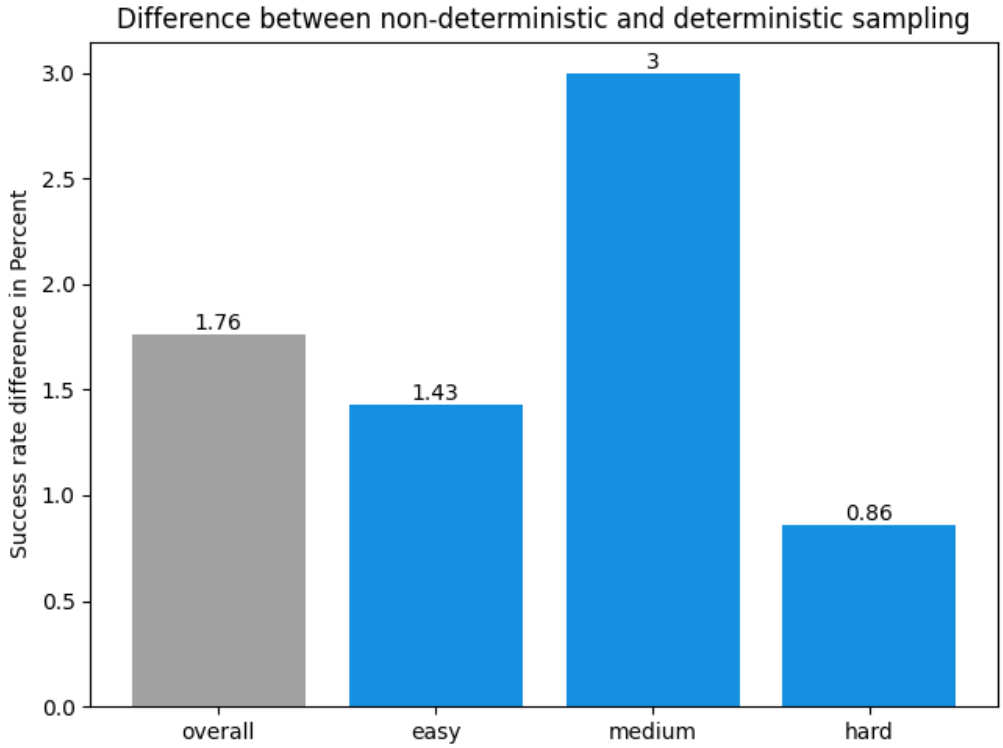
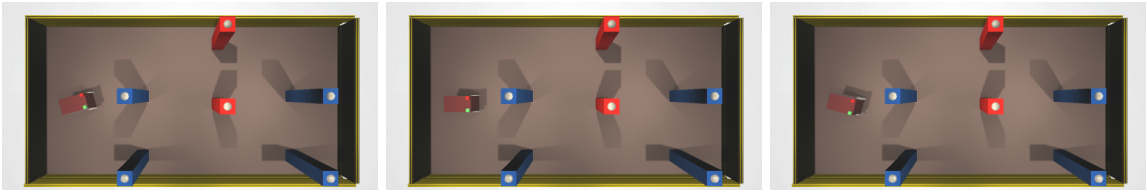


Figure 8.7.: Results of the deterministic check across all evaluations during the experimentation phase



(a) -15° (b) 0° (c) 15°

Figure 8.8.: tested starting rotations

8.4.2. Identical Start Condition Test

The environment is not 100% deterministic. This is due to the environment implementation in Unity [37]. The environment uses physics simulations which are not fully deterministic. In addition the policy's actions are sampled non-deterministically during the evaluation. This means that identical start conditions of an episode may result in different trajectories and results. This test evaluates the impact of the non-deterministic environment and sampling on the agent's behaviour.

The agent with HX-P is placed in the same start conditions for 100 episodes. The starting conditions entail the selected track, light condition and the agent's starting rotation. The test was executed for the hardBlueFirstRight trach with the standard lighting. Three starting rotations were used 8.8. The episode results are compared.

Results The results show that the policies's performance is very consistent for the different starting rotations regarding the *success_rate*. The agent successfully completes the tracks with −15ř and 0ř

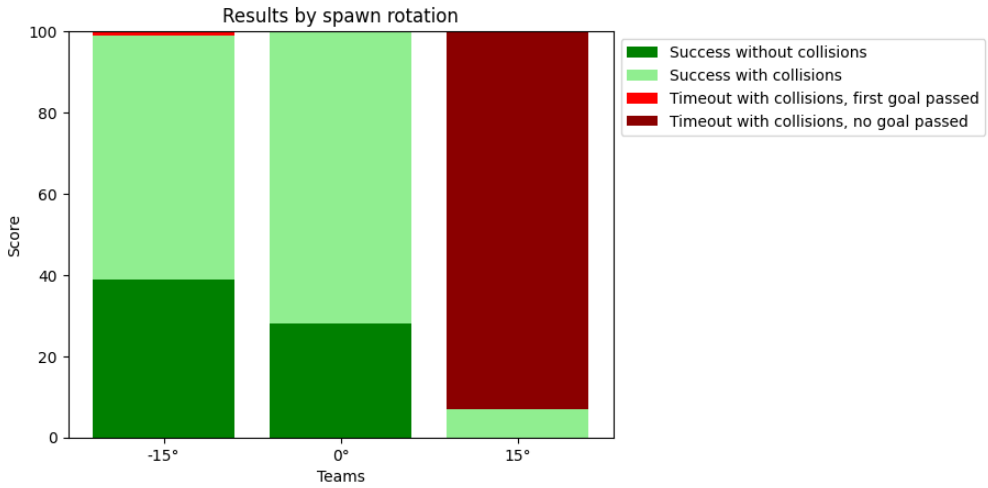


Figure 8.9.: Results of 100 episodes for different starting rotations

starting rotation. The agent completes the track in only 7% of episodes with 15ř starting rotation. For the different starting rotations, the agent either completes the track very reliably or very rarely.

The *collision_rate* is not consistent for the episodes with the same starting rotation. The *collision_rate* is 60% and 72% for the −15ř and 0ř starting rotations.

Discussion The test shows that identical starting conditions do not always lead to the same episode result. However the success rate is very consistent for the identical starting conditions. The collision rate is significantly less consistent.

The results confirm that the *success_rates* of the Basic Evaluation Algorithm are reliable.

8.4.3. Fresh Observations Test

HX-P was trained to not use fresh observations with a fixed step duration of 0.3 seconds. The policy input receives the observation from the start of the previous step as input. The policy input essentially lags 0.3 seconds behind the environment state. This saves processing time during training and evaluation. Alternatively a policy can be run with the *use_fresh_obs* parameter. This makes the policy request a fresh observation from the Unity simulation, see 4.1.1.

The Fresh Observations Test analyses whether switching the trained policy to use fresh observations improves the performance. The policy is evaluated with fresh observations and without fresh observations for each difficulty level using the Basic Evaluation Algorithm 5.2. The success rates are compared to determine if fresh observations improve the agent's performance.

Results The results show that the policy performs similar for the easy and medium settings. The success rates are 100% for both fresh and non-fresh observations. The hard setting shows a strong decrease in performance when using fresh observations. The success rate drops from 98% to 68% 8.10.

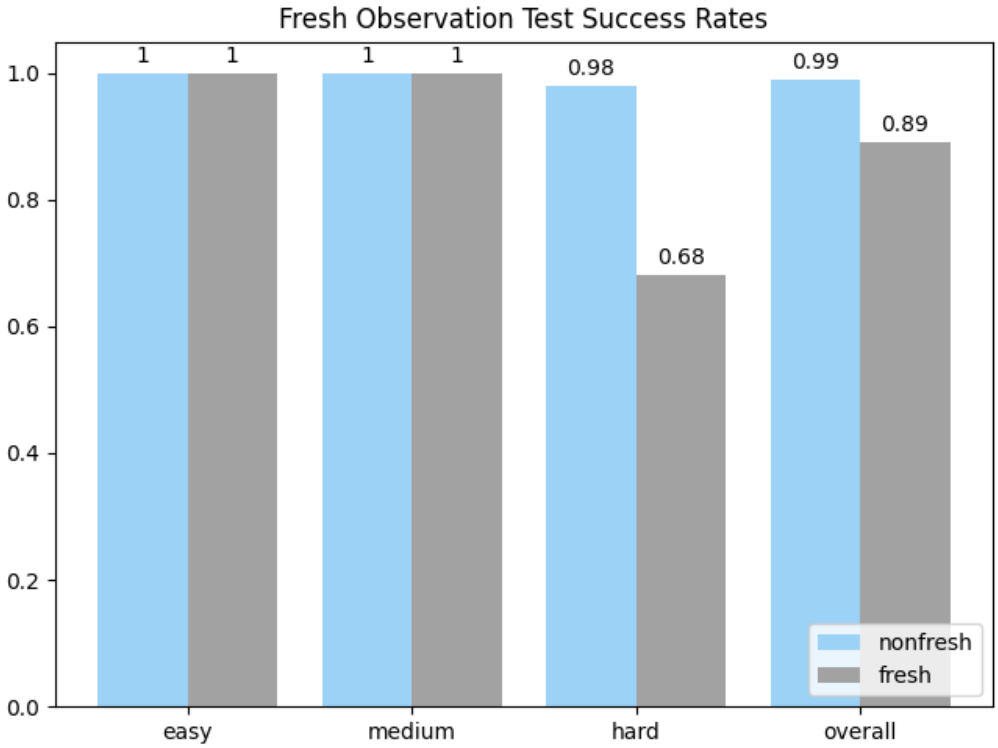


Figure 8.10.: Success rates for HX-P using fresh and non-fresh observations

The evaluations using fresh observations also take longer. The Basic Evaluation Algorithm's per call time for the fresh observations is about 20 minutes compared to 17 minutes for the non-fresh observations. The evaluation using fresh observations is slower because the policy has to request a fresh observation from the Unity simulation. This takes additional time.

Discussion The policy was trained using non-fresh observations. The policy's performance decreases for hard tracks when it is evaluated using fresh observations. This suggests that the policy has learned to work with the lag of 0.3 seconds between the environment state and input to the policy. The policy's performance decreases when the lag is removed.

The test confirms that the *use_fresh_obs* parameter should not be changed after the training has been completed.

8.4.4. Jetbot Generalization Test

HX-P was trained using the DifferentialJetBot shown in figure 4.8. The Jetbot Generalization test evaluates the same policy using the FourWheelJetBot. The policy is only evaluated on the standard light setting to save time.

Results The policy evaluation achieves very high success rates for the FourWheelJetBot with 100%, 100% and 91% for the easy, medium and hard tracks 8.11. The collision rates increase for higher difficulties with 0%, 4% and 94%. The success and collision rates are slightly worse than on

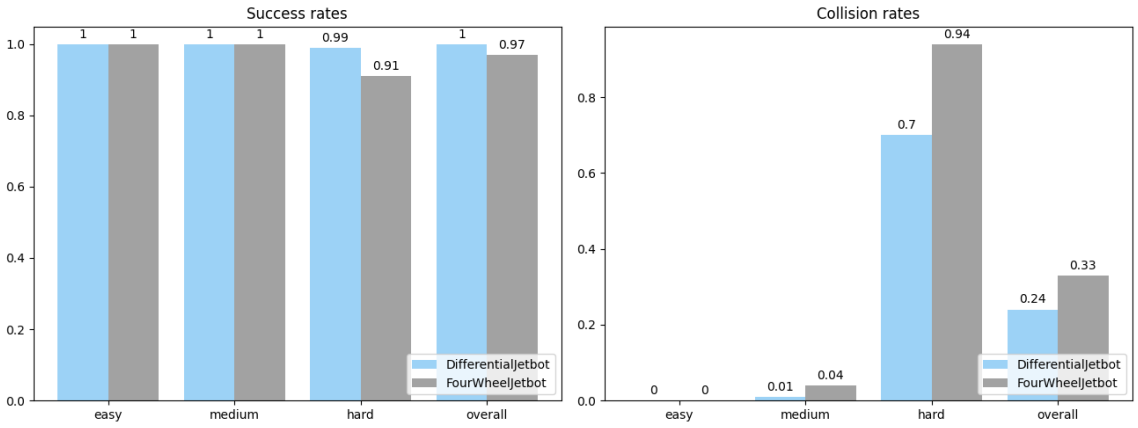


Figure 8.11.: Evaluation of the DifferentialJetBot policy HX-P with both jetbot versions

the DifferentialJetBot. The overall success rate of the FourWheelJetBot is 97% compared to 100% for the DifferentialJetBot.

The overall collision rate is 33% compared to 24% for the DifferentialJetBot. Especially for the hard tracks the collision rate is much higher for the FourWheelJetBot with 94% compared to 70% for the DifferentialJetBot. The collisions of the DifferentialJetBot were previously described as mainly being minor collisions where the agent scrapes the goal posts with its side. Analysis of the recorded videos shows that the collisions of the FourWheelJetBot are similar. However the agent scrapes the goals on its side for longer durations D.1.

Discussion The policy that was trained on the DifferentialJetBot can be transfered to the FourWheelJetBot with only a slight decrease in performance. The policy does not have to be retrained in this case.

9. Conclusion

This research has investigated the development of an autonomous driving agent capable of adapting to varying light conditions. By leveraging end-to-end trained Convolutional Neural Networks, the results show that it is possible to train an agent that solves the autonomous driving task very reliably. The task consists of a series of goals that the agent has to drive through. There are three difficulty levels that position the goals in increasingly challenging configurations. The policy is evaluated on the three light settings bright, standard and dark. The results highlight the effectiveness of our approach, demonstrating significant robustness towards changing light conditions. This work advances the field of autonomous driving and opens avenues for future research to transfer the policy to real-world jetbot agents.

The developed autonomous driving agent can move in its environment and has a front facing camera. The images from this camera are used for the agent's policy. The camera images are preprocessed to improve the policy's performance and resilience towards light changes. The policy is trained using the Proximal Policy Optimization (PPO) algorithm in simulation.

To summarize our key findings, the primary research questions are revisited:

- Question 1 - Is it possible to train an autonomous driving agent consisting of a CNN policy with end-to-end reinforcement learning to reliably solve the tracks of all difficulty levels?
- Question 2 - Is it possible to use an end-to-end trained CNN policy to make the agent robust to changing light conditions?
- Question 3 - Is it possible to use a NN policy that can be executed in real-time on a physical robot?

Question 1 The Hard Tracks Mixed Light Policy (HX-P) proved to be very suitable for the autonomous driving task. The policy was trained successfully end-to-end using only the most difficult tracks. The policy was able to solve the task reliably, even on the most challenging tracks. It was able to achieve a *success_rate* of 100% on the easy and medium difficulty tracks and 99% in on the difficult tracks.

The policy was able to outperform the previous work on the same task. Schaller [1] used a rather complex preprocessing pipeline to extract features for a Neural Network (NN) policy.

Question 2 The developed HX-P was robust to the tested light changes. The policy was able to solve the task with a *success_rate* of 100% on all light settings for easy and medium tracks. The *success_rate* decreased slightly for the difficult tracks. The policy achieved 98% on the bright and 97% on the dark light setting, compared to a *success_rate* of 99% for the standard setting.

The experiments show that the histogram equalization preprocessing step plays a major role in the policy's performance and robustness towards light changes. Furthermore the histogram equalization

allows a policy to be trained exclusively on the standard light setting and generalize to the other settings with a small performance decrease 6.4.

Question 3 Episode replays of the HX-P were created and then replayed on the pyhsical robot's hardware. The hardware was able to compute the policy in real-time. The increased processing requirements of the Convolutional Neural Network (CNN) policy are no issue for the robot's hardware. This shows that the policy can be transferred to the physical robot in theory.

To summarize this work, successful autonomous driving agents were developed. The agents' CNN policies were trained end-to-end using the PPO Reinforcement Learning (RL) algorithm. The agents were able to solve the task reliably for all light settings and difficulty levels with success rates above 95%. The image preprocessing steps proved crucial in the policy's performance and robustness towards light changes. The preprocessing steps and policy computation was evaluated on the physical robot's hardware and found to be feasible for real-time computation.

The thesis contributes to the further development of autonomous driving agents. The work shows that CNN policies can be trained end-to-end to solve the autonomous driving task. The developed policy's resistance to different light changes makes this approach a promising candidate for transfer to the real-world robot hardware.

10. Future Research Directions

There are two interesting main directions for future work based on the results of this thesis. The first point arises from the fact that the agents were able to complete the tracks with a high success rate. However the agent's HX-P resulted a high rate of collisions with the goal objects. The second point is the transfer of the policy to real world robots 4.8.

Decreasing the collision rate HX-P and all other developed policies did not learn to avoid collisions entirely. The reward function that was used for the policy training did not punish collisions enough, this is discussed in chapter 6.8. Collision avoidance is especially important for real-world applications.

Possible solutions to this problem could include finding better reward function parameterizations that leads to a policy that avoids collisions. Another approach could be to modify the policy's inputs to include additional information. This could include a wider field of view or additional sensors, such as a depth sensor.

Transfering the policy to real world robots The agents developed in this thesis were trained in a simulated environment. The goal of the ongoing research at the ScaDS.AI is to transfer the agents to real world robots. The robots are based on the NVIDIA JetBot platform and are equipped with a camera, wheels and a small computer. The processing power of these robots is limited. The research in this thesis shows that the hardware is sufficient to compute the preprocessing steps and policy in real-time.

The policies developed in this thesis could be transfered to the physical robots. Evaluating the policy in a real-world environment would be an interesting direction for future research. Previous research at the ScaDS.AI by Flach [6] has shown this transfer is difficult. However the agents evaluated by Flach [6] used completely different preprocessing steps and policies compared to the CNN light-change resistant policies developed in this thesis.

Bibliography

[1] Maximilian Schaller. "Train an Agent to Drive a Vehicle in a Simulated Environment Using Reinforcement Learning". MA thesis. Universität Leipzig, 2023.

[2] Johannes Deichmann et al. *Autonomous driving's future: Convenient and connected*. 2023. URL: https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/autonomous-drivings-future-convenient-and-connected (visited on 12/09/2023).

[3] Mike Monticello. *Ford's BlueCruise Remains CR's Top-Rated Active Driving Assistance System*. 2023. URL: https://www.consumerreports.org/cars/car-safety/active-driving-assistance-systems-review-a2103632203/ (visited on 10/24/2023).

[4] B Ravi Kiran et al. *Deep Reinforcement Learning for Autonomous Driving: A Survey*. 2021. arXiv: 2002.00444 [cs.LG].

[5] Collimator. *The State of Autonomous Vehicles: Seeking Mainstream Adoption*. 2023. URL: https://www.collimator.ai/post/the-state-of-autonomous-vehicles-in-2023 (visited on 02/16/2023).

[6] Merlin Flach. "Methods to Cross the simulation-to-reality gap". Bachelor's Thesis. Universität Leipzig, 2023.

[7] NVIDIA-AI-IOT. *JetBot*. 2023. URL: https://jetbot.org/master/ (visited on 11/23/2023).

[8] Dean A. Pomerleau. "ALVINN, an autonomous land vehicle in a neural network". In: 2015. URL: https://api.semanticscholar.org/CorpusID:18420840.

[9] Think Autonomous. *Tesla's HydraNet - How Tesla's Autopilot Works*. 2023. URL: https://www.thinkautonomous.ai/blog/how-tesla-autopilot-works/ (visited on 09/15/2023).

[10] Tesla. *Tesla Autonomy Day*. 2019. URL: https://www.youtube.com/watch?v=Ucp0TTmvqOE (visited on 05/23/2024).

[11] Jonas König. "Model training of a simulated self-driving vehicle using an evolution-based neural network approach". Bachelor's Thesis. Universität Leipzig, 2022.

[12] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG].

[13] Nilesh Barla. *Self-Driving Cars With Convolutional Neural Networks (CNN)*. 2023. URL: https://neptune.ai/blog/self-driving-cars-with-convolutional-neural-networks-cnn (visited on 12/09/2023).

[14] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG].

[15] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[16] Julian Schrittwieser et al. "Mastering Atari, Go, chess and shogi by planning with a learned model". In: *Nature* 588.7839 (Dec. 2020), pp. 604–609. DOI: 10.1038/s41586-020-03051-4. URL: https://doi.org/10.1038%2Fs41586-020-03051-4.

[17] OpenAI. *OpenAI Spinning Up Part 2: Kinds of RL Algorithms*. 2018. URL: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html (visited on 12/09/2023).

[18] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529 (Jan. 2016), pp. 484–489. DOI: 10.1038/nature16961.

[19] Abderrezzaq Sendjasni, David Traparic, and Mohamed-Chaker Larabi. "Investigating Normalization Methods for CNN-Based Image Quality Assessment". In: *2022 IEEE International Conference on Image Processing (ICIP)*. 2022, pp. 4113–4117. DOI: 10.1109/ICIP46576.2022.9897268.

[20] Connor Shorten and Taghi M. Khoshgoftaar. "A survey on Image Data Augmentation for Deep Learning". In: *Journal of Big Data* 6 (2019), pp. 1–48. URL: https://api.semanticscholar.org/CorpusID:195811894.

[21] D. Vijayalakshmi et al. "A Comprehensive Survey on Image Contrast Enhancement Techniques in Spatial Domain". In: *Sensing and Imaging* 21 (2020). URL: https://api.semanticscholar.org/CorpusID:224927531.

[22] Pin Wang and Ching-Yao Chan. "Formulation of Deep Reinforcement Learning Architecture Toward Autonomous Driving for On-Ramp Merge". In: *CoRR* abs/1709.02066 (2017). arXiv: 1709.02066. URL: http://arxiv.org/abs/1709.02066.

[23] Ali Ghadirzadeh et al. "Deep Predictive Policy Training using Reinforcement Learning". In: *CoRR* abs/1703.00727 (2017). arXiv: 1703.00727. URL: http://arxiv.org/abs/1703.00727.

[24] Josh Tobin et al. *Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World*. 2017. arXiv: 1703.06907 [cs.RO]. URL: https://arxiv.org/abs/1703.06907.

[25] Mariusz Bojarski et al. "End to End Learning for Self-Driving Cars". In: (Apr. 2016).

[26] Dianzhao Li and Ostap Okhrin. "Modified DDPG car-following model with a real-world human driving experience with CARLA simulator". In: *Transportation Research Part C: Emerging Technologies* 147 (Feb. 2023), p. 103987. ISSN: 0968-090X. DOI: 10.1016/j.trc.2022.103987. URL: http://dx.doi.org/10.1016/j.trc.2022.103987.

[27] Alexey Dosovitskiy et al. "CARLA: An Open Urban Driving Simulator". In: *Proceedings of the 1st Annual Conference on Robot Learning*. 2017, pp. 1–16.

[28] Shital Shah et al. "AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles". In: *Field and Service Robotics*. 2017. eprint: arXiv:1705.05065. URL: https://arxiv.org/abs/1705.05065.

[29] Mark Towers et al. *Gymnasium*. Mar. 2023. DOI: 10.5281/zenodo.8127026. URL: https://zenodo.org/record/8127025 (visited on 07/08/2023).

[30] Pablo Samuel Castro et al. "Dopamine: A Research Framework for Deep Reinforcement Learning". In: (2018). URL: http://arxiv.org/abs/1812.06110.

[31] Antonin Raffin et al. "Stable-Baselines3: Reliable Reinforcement Learning Implementations". In: *Journal of Machine Learning Research* 22.268 (2021), pp. 1–8. URL: http://jmlr.org/papers/v22/20-1364.html.

[32] Unity Technologies. *Unity Engine*. 2023. URL: https://unity.com.

[33] Emanuel Todorov, Tom Erez, and Yuval Tassa. "MuJoCo: A physics engine for model-based control". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2012, pp. 5026–5033. DOI: 10.1109/IROS.2012.6386109.

[34] Andrew Cohen et al. "On the Use and Misuse of Absorbing States in Multi-agent Reinforcement Learning". In: *RL in Games Workshop AAAI 2022* (2022). URL: http://aaai-rlg.mlanctot.info/papers/AAAI22-RLG_paper_32.pdf.

[35] Hugh Perkins. *Peaceful Pie, Connect Python with Unity for reinforcement learning!* 2023. URL: https://github.com/hughperkins/peaceful-pie (visited on 10/23/2023).

[36] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518 (2015), pp. 529–533. URL: https://api.semanticscholar.org/CorpusID:205242740.

[37] Unity Technologies. *MonoBehaviour.FixedUpdate()*. 2024. URL: https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html (visited on 05/24/2024).

Declaration of Autonomy

I hereby declare that I have written the Master's thesis with the title:

„End-To-End Reinforcement Learning Training of a Convolutional Neural Network to Achieve an Autonomous Driving Agent on Tracks in Simulation“

independently, that I have not used resources other than those specified and that all statements taken verbatim or meaningfully from other works have been marked as such. I am aware that any violation can also lead to the revocation of the degree retroactively. I further declare, that the electronic copy coincides exactly with the printed exemplar.

Leipzig, 28.06.2024

_____

Georg Schneeberger

A. Appendix

A. Code Repository

The code repository is publically availible on GitHub:

carsim-rl-cnn

The readme file contains instructions on how to install and run the code.

B. HX-P

HX-P, the most successful policy model is publically availible on Huggingface:

hardDistanceMixedLight.zip

The episode recordings for question 3 are also hosted there.

C. Experiments for finding hyperparameters - Configuration files

how to run a specific config? link readme

C.1. Reward functions capability check

The used configs are the following:

- ppo_rewardFunction_capability_check_orientationReward.yaml
- ppo_rewardFunction_capability_check_distanceReward.yaml
- ppo_rewardFunction_capability_check_velocityReward.yaml
- ppo_rewardFunction_capability_check_eventReward.yaml

C.2. Most Successful Policy Configuration

The training config of the model for the final evaluations is availible on GitHub: hardDistanceMixed-Light.yaml

D. Example Video Files

Video Files are availible on Huggingface: carsim-rl-cnn

D.1. Video Files with FourWheelJetBot

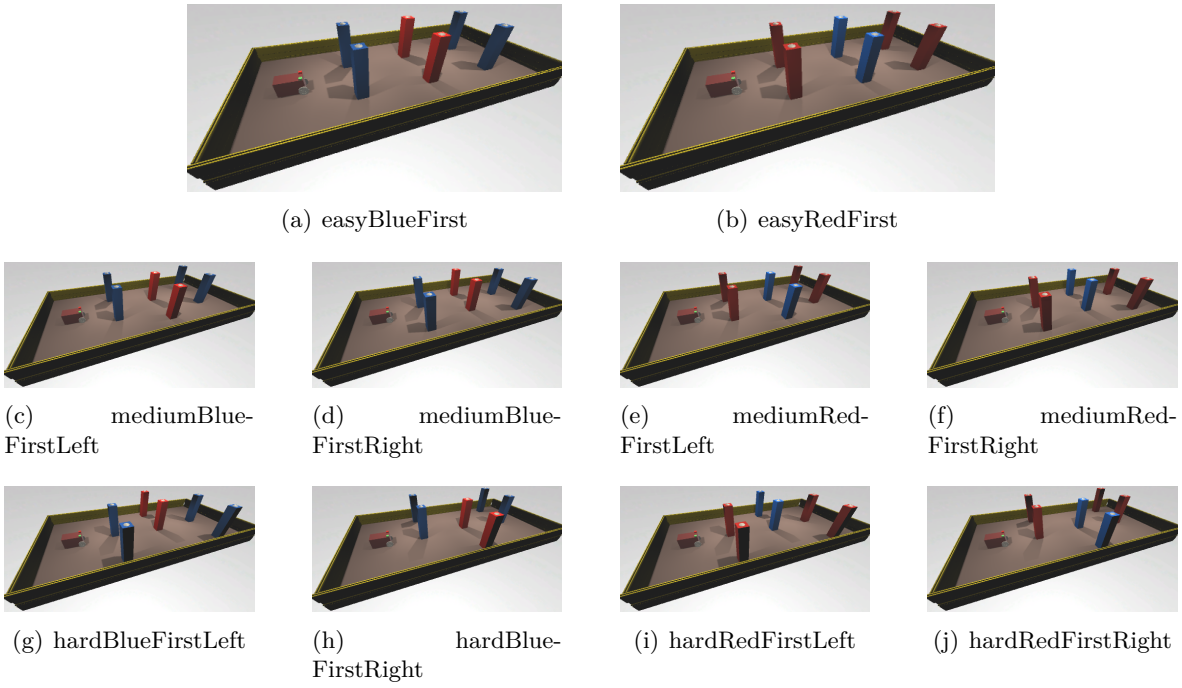
Collisions The collisions are generally more severe for the FourWheelJetBot:

- example_videos/hard_standard_FourWheelJetBot_env_0_video_1_topview.gif
- example_videos/hard_standard_FourWheelJetBot_env_0_video_2_topview.gif

D.2. Reward Function Capability check videos

Video Files are availible on HuggingFace: rewardCapabilityCheck_videos

E. All Tracks



(a) easyBlueFirst (b) easyRedFirst

(c) mediumBlue-FirstLeft (d) mediumBlue-FirstRight (e) mediumRed-FirstLeft (f) mediumRed-FirstRight

(g) hardBlueFirstLeft (h) hardBlue-FirstRight (i) hardRedFirstLeft (j) hardRedFirstRight

Figure A.1.: All tracks

F. Pseudocode

Algorithm F.1: COLLECT DATA()

comment: Fill RolloutBuffer with samples obtained by current model

procedure COLLECTDATA($trainingMapType, trainingLightSetting$)
 $num\_steps \leftarrow 0$
 $num\_episodes \leftarrow 0$
 $num\_succesful\_episodes \leftarrow 0$
 $RolloutBuffer$.RESET($trainingMapType, trainingLightSetting$)
 Env.RESET()
 while $RolloutBuffer$.NOTFULL()
 do
 $obs \leftarrow Env$.GETOBSERVATION()
 $action \leftarrow Model$.GETACTION(obs)
 $reward \leftarrow Env$.STEP($action$)
 $num\_steps \leftarrow num\_steps + 1$
 ADDTOROLLOUTBUFFER($obs, action, reward$)
 if Env.ISFINISHED()
 then
 $num\_episodes \leftarrow num\_episodes + 1$
 if Env.FINISHEDSUCCESSFULLY()
 then $num\_succesful\_episodes \leftarrow num\_succesful\_episodes + 1$
 Env.RESET($trainingMapType, trainingLightSetting$)
 $rollout\_success\_rate \leftarrow \frac{num\_succesful\_episodes}{num\_episodes}$
 if $rollout\_success\_rate >= best\_rollout\_success\_rate$
 then
 $best\_success\_rate \leftarrow rollout\_success\_rate$
 $Model$.SAVETOFILE()
 $best\_model \leftarrow Model$
 return (num_steps)

Algorithm F.2: TRAIN MODEL()

comment: Sample from replay buffer and update the model based on the loss

procedure TRAINMODEL()
$amount\_of\_batches \leftarrow \frac{rollout\_buffer\_size}{batch\_size}$
for $i \leftarrow 0$ **to** n_epochs
do
- $RolloutBuffer$.SHUFFLE()
- $RolloutBuffer$.CREATEBATCHES($batch\_size$)
- **for** $m \leftarrow 0$ **to** $amount\_of\_batches$
  do
  - $batch \leftarrow RolloutBuffer$.GETBATCH($m$)
  - $loss \leftarrow$ COMPUTELOSS($batch$)
  - $Model$.BACKPROPAGATE($loss$)
  - $Optimizer$.STEP()

G. NN Architecture

Full loss graph for the action A.2 and value head A.3 are availible here.

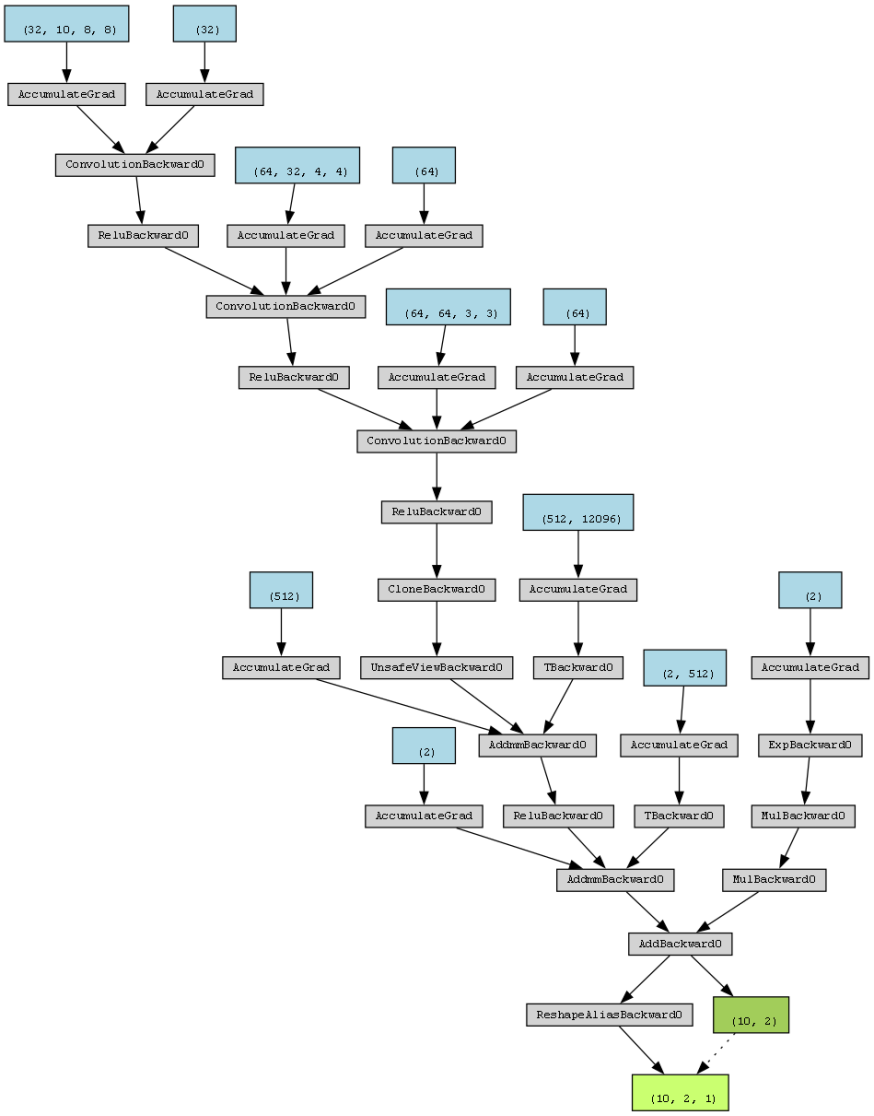


Figure A.2.: Action Head Loss graph

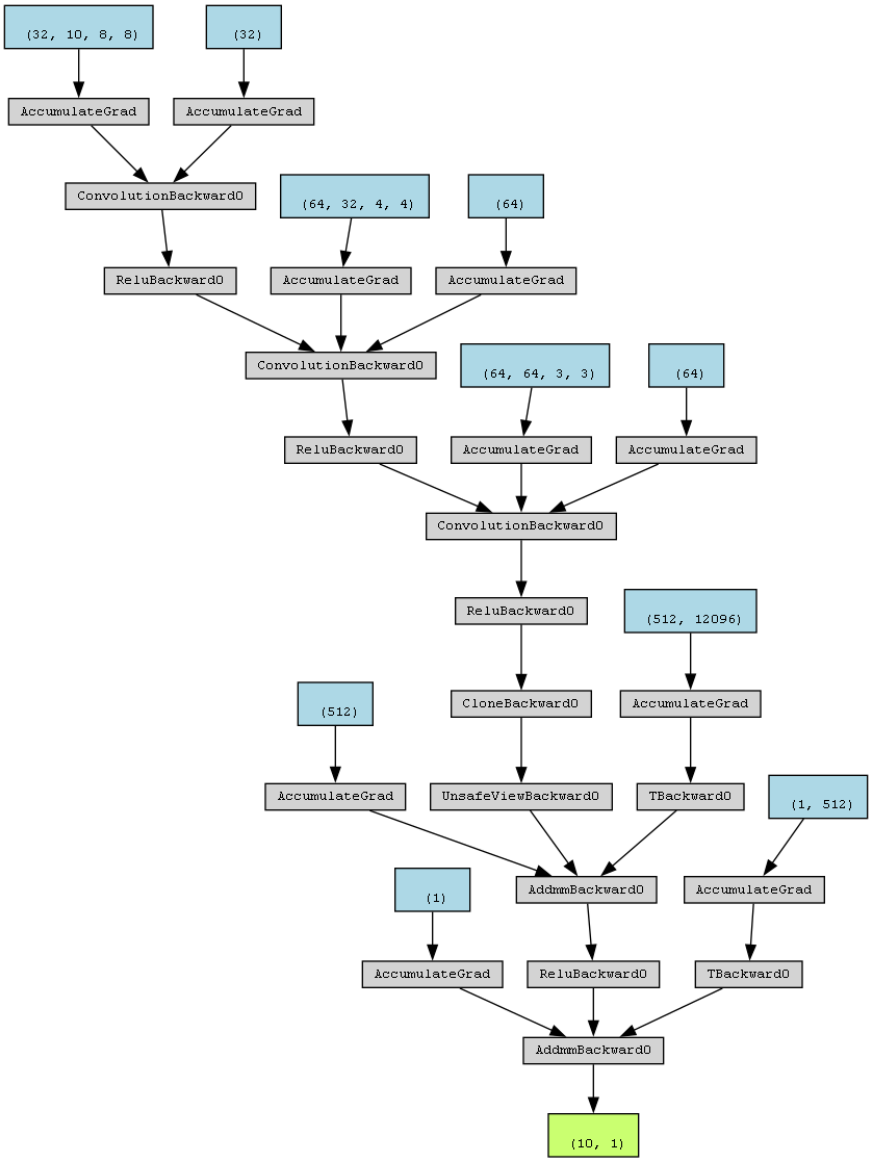


Figure A.3.: Value Head Loss graph

H. Eval Model Track

Algorithm H.1: Generate Map and Rotation Combinations()

procedure GENERATE_MAP_AND_ROTATIONS($difficulty, n\_eval\_episodes, env$)
 $rotationMode \leftarrow env$.GETSPAWNMODE()
 $rotation\_range\_min, rotation\_range\_max \leftarrow Spawn$.GETROTATIONRANGE($rotationMode$)
 $range\_width \leftarrow rotation\_range\_max - rotation\_range\_min$
 $rotations \leftarrow []$
 if $n\_eval\_episodes == 1$
  then $rotations$.APPEND($(rotation\_range\_min + range\_width)/2$)

  else $\begin{cases} step \leftarrow range\_width/(n\_eval\_episodes - 1) \\ \textbf{for } i \leftarrow 0 \textbf{ to } n\_eval\_episodes - 1 \\ \quad \textbf{do } \{rotations.\text{APPEND}(rotation\_range\_min + i * step) \end{cases}$
 $track\_numbers \leftarrow MapType$.GETALLTRACKNUMBERSOFDIFFICULTY($difficulty$)
 $tracks \leftarrow []$
 for $i \leftarrow 0$ **to** $n\_eval\_episodes - 1$
  do $\{tracks$.APPEND($i \mod$ LEN($track\_numbers$))
 $combinations \leftarrow []$
 for $i \leftarrow 0$ **to** $n\_eval\_episodes - 1$
  do $\{combinations$.APPEND($(tracks[i], rotations[i])$)
 return ($combinations$)

I. Replay on JetBot

I.1. Installation instructions for executing replays on the Jetbot

The replays are executed on NVIDIA Jetbot hardware. The test does not make use of jetbot specific hardware and software features such as the jetbot camera. The test runs on the jetbot's standard ubuntu installation. Instructions for the installation and execution of the replay test are availible in the replays_on_jetbot.md file in the code repository.

Algorithm I.1: RECORD EPISODE()

comment: Record episode

procedure RECORD_EPISODE($policy, env, directory$)
 env.RESET()
 $sampled\_actions \leftarrow []$
 $infer\_obsstrings \leftarrow []$
 $step\_obsstrings \leftarrow []$
 $done \leftarrow$ **false**
 while !$done$
  do $\begin{cases} obs, obsstring \leftarrow env.\text{GETOBSERVATION}() \\ action \leftarrow policy.\text{INFER}(obs) \\ step\_obsstring, done \leftarrow env.\text{STEP}(action) \\ infer\_obsstrings.\text{APPEND}(obsstring) \\ step\_obsstrings.\text{APPEND}(step\_obsstring) \\ sampled\_actions.\text{APPEND}(action) \end{cases}$
 for $i \leftarrow 0$ **to** $len(step\_obsstrings)$
  do SAVETOFILE($step\_obsstrings[i], directory + '' /step\_image'' + i + '' .png''$)
 for $i \leftarrow 0$ **to** $len(infer\_obsstrings)$
  do SAVETOFILE($infer\_obsstrings[i], directory + '' /infer\_image'' + i + '' .png''$)
 for $i \leftarrow 0$ **to** $len(sampled\_actions)$
  do SAVETOFILE($sampled\_actions[i], directory + '' /sampled\_action'' + i + '' .npy''$)

Algorithm I.2: Replay Episode()

comment: Replay episode and record processing + inference time

procedure REPLAY_EPISODE($policy, env, directory$)
 $recorded\_episode\_length \leftarrow$ LoadRecordedEpisodeLength($directory$)
 $recorded\_actions \leftarrow$ LoadRecordedActions($directory$)
 $infer\_obs\_unity\_images \leftarrow$ LoadInferImages($directory$)
 $step\_obs\_unity\_images \leftarrow$ LoadStepImages($directory$)
 $reproduce\_times \leftarrow []$
 $replay\_time\_start \leftarrow$ TIME()
 for $i \leftarrow 0$ **to** $recorded\_episode\_length$
 do
 $obs \leftarrow$ ProcessImage($env, infer\_obs\_unity\_images[i]$)
 $action \leftarrow policy$.Infer(obs)
 $reproduce\_times$.append(TIME() $-$ $replay\_time\_start$)
 AssertClose($action, recorded\_actions[i]$)
 $replay\_time\_start \leftarrow$ TIME()
 ProcessImage($env, step\_obs\_unity\_images[i]$)
 return ($reproduce\_times$)