



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 resilient to light changes

Master's Thesis

submitted by:
Georg Schneeberger

matriculation number:
3707914

Supervisor:
Dr. Thomas Burghardt
Martin Lorenz

© 2024

This thesis and its parts are **protected by copyright**. Any use outside the narrow limits of copyright law without the consent of the author is prohibited and punishable by law. This applies in particular to reproductions, translations, microfilming as well as storage and processing in electronic systems.

Abstract

This master thesis contributes to the domain of autonomous driving and reinforcement learning. The thesis solves a simplified autonomous driving task in simulation by training an agent that takes raw camera inputs in an end-to-end manner. 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 [1]. This thesis develops an agent that has to solve the same autonomous driving task. The agent from previous work was not able to reliably complete all parcours, its performance suffered further under changing light conditions, motivating the research goals of this thesis. The agent in this thesis differs fundamentally in implementation from the previous work.

The thesis answers three research questions:

- Question 1 - Is it possible to train an autonomous driving agent consisting of a convolutional neural network with end-to-end reinforcement learning to reliably solve the parcours of all difficulty levels?
- Question 2 - Is it possible to use an end-to-end trained CNN to make the agent robust to changing light conditions?
- Question 3 - Is it possible to use a neural network that can be transferred to a physical robot?

A reinforcement learning agent is developed to traverse simulated tracks of varying difficulty and light settings. The agent's policy consists of a convolutional neural network that is trained end-to-end using the Proximal Policy Optimization algorithm. The inputs to the neural network are camera images from the perspective of the agent. Preprocessing steps are applied to the images to reduce the impact of the different light conditions. The agent is trained using the proximal policy optimization algorithm. The thesis investigates different training approaches to develop a powerful agent, such as for example 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. The experiment for the third research question shows that in theory the policy can be transferred to a physical robot. The thesis shows that it is possible to train a convolutional neural network agent to solve the investigated autonomous driving task using end-to-end reinforcement learning.

What else can I say here? the used process of finding parameters?

Contents

List of Figures	V
List of Tables	VII
1. Introduction	1
2. Research Goals	2
2.1. Question 1 - Is it possible to train an autonomous driving agent consisting of a convolutional neural network with end-to-end reinforcement learning to reliably solve the parcours of all difficulty levels?	2
2.2. Question 2 - Is it possible to use an end-to-end trained CNN to make the agent robust to changing light conditions?	3
2.3. Question 3 - Is it possible to use a neural network that can be transferred to a physical robot?	3
3. Related Work	4
3.1. Self-Driving	4
3.1.1. Previous Work	5
3.2. Reinforcement Learning	6
3.3. Simulation for Reinforcement Learning and Self-Driving	9
3.4. Imitation learning	9
3.5. Light change resiliency	10
3.5.1. Domain Randomization	10
3.5.2. Data augmentation	10
3.5.3. Multi-Modal learning	11
3.5.4. Hierarchical Reinforcement Learning	11
3.6. Transfer to reality	11
3.6.1. Transfer Learning	11
3.7. Key Ideas	11
3.7.1. Fitting RL algorithms	11
3.7.1.1. CNN for feature extraction in RL	11
3.7.1.2. Memory mechanism	11
3.7.2. Reward shaping	11
3.7.3. Environment Implementation	11
3.7.4. Image preprocessing for resiliency to light changes	11
3.7.5. Domain Randomization	11
4. Methods	12
4.1. Environment Description	13
4.1.1. Environment Simulation	13
4.1.2. Arena Description	14
4.1.3. Agent Description	16

4.1.4.	Episode Design	17
4.1.4.1.	Step duration	17
4.1.4.2.	Episode Termination	18
4.1.5.	Reward Function	19
4.1.6.	Collision Mode	21
4.2.	Policy Description	24
4.2.1.	Preprocessing	24
4.2.1.1.	Downsampling	26
4.2.1.2.	Grayscale	26
4.2.1.3.	Histogram Equalization	26
4.2.2.	Memory Mechanism	27
4.2.3.	Neural Network	28
4.2.3.1.	Input Space	28
4.2.3.2.	Action Output Space	29
4.2.3.3.	Architecture	29
4.3.	Training Algorithm	31
4.3.1.	Collect Data	31
4.3.2.	Train Model	33
4.3.3.	Final Result of Training Algorithm	34
5.	Experiment Description	35
5.1.	Evaluation metrics	35
5.1.1.	success_rate	35
5.1.2.	goal_completion_rate	35
5.1.3.	collision_rate	35
5.2.	Basic evaluation algorithm	36
5.3.	Question 1 - Model evaluation track difficulties	37
5.4.	Question 2 - Model evaluation all light settings	37
5.5.	Question 3 - Investigating the feasibility of transferring the policy to a physical robot.	38
5.5.1.	Policy Replay Experiment	38
5.6.	Other Experiments	40
5.6.1.	Sampling mode performance test	40
5.6.2.	Identical start condition Test	41
5.6.3.	Fresh Observation Test	41
5.6.4.	JetBot generalization Test	42
6.	Finding appropriate hyperparameters for training	43
6.1.	General parameters	43
6.1.1.	Initial Agent position	43
6.1.2.	Agent Camera	43
6.1.3.	Observation Space	44
6.1.4.	Step Duration	44
6.1.5.	Remaining parameters	44
6.2.	Reward functions capability check	45

6.3.	Chosen Reward function	47
6.4.	Experiments fixed difficulty setting	47
6.5.	Experiments mixed difficulty setting	49
6.6.	Experiments mixed light setting hyperparameter	49
6.7.	Experiments on the importance of the histogram equalization preprocessing step . .	51
6.8.	Comment on the collision rate	52
6.9.	Hard Tracks, mixed light settings - the most successful policy	53
7.	Challenges	55
7.1.	Connecting the Python algorithm and Unity Simulation	55
7.2.	step duration	56
7.3.	Parameters for training	56
8.	Results	57
8.1.	Eval for question 1	57
8.1.1.	Experiment Results	57
8.1.2.	Conclusion	57
8.2.	Eval for question 2	58
8.2.1.	Experiment Results	58
8.2.2.	Conclusion	59
8.3.	Eval for Question 3 - Replays	59
8.3.1.	Experiment Results	60
8.3.2.	Discussion	60
8.4.	Other experiments	61
8.4.1.	Sampling mode performance test	61
8.4.2.	Test identical start conditions	62
8.4.3.	Fresh Observations Test	62
8.4.4.	Jetbot generalization	63
9.	Conclusion	65
Bibliography		66
Erklärung		68
A. Appendix		I
A.	Code Repository	I
B.	Most successful model	I
C.	Experiments for finding hyperparameters - Configuration files	I
C.1.	Reward functions capability check	I
C.2.	Most Successful Policy Configuration	I
D.	Example Video Files	II
D.1.	Video Files with FourWheelJetBot	II
E.	Pseudocode	II
F.	Neural Network Architecture	III

Contents

G.	Eval Model Track	III
H.	Replay on JetBot	VI
	H.1. Installation instructions for executing replays on the Jetbot	VI

List of Figures

2.1. Example image of a parcour and the agent's camera	2
3.1. Standard components in a modern autonomous driving systems pipeline listing the various tasks. The key problems addressed by these modules are Scene Understanding, Decision and Planning. Image from [4]	5
3.2. 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.	7
3.3. Taxonomy of RL algorithms from OpenAI's Spinning Up course [15]	7
4.1. Parallel simulation of multiple environments in one Unity instance	13
4.2. Communication between Python and Unity	14
4.3. Example evaluation tracks for each difficulty setting.	14
4.4. Arena and agent camera at different light settings.	15
4.5. Example Spawn Orientations and agent camera views for a hard track	15
4.6. Original Nvidia JetBot and simulated JetBot Designs	16
4.7. Timeline of steps in Unity for fixed and variable duration	18
4.8. Event Reward function	20
4.9. Complete reward function R with all its components	22
4.10. Event Reward only training with <i>collisionMode unrestricted</i>	22
4.11. Collision Modes	23
4.12. Histogram equalization of a grayscale image from standard light setting	27
4.13. Memory Mechanism	28
4.14. Neural Network Structure	29
4.15. Neural network layers and parameters for the best configuration	30
4.16. Environment parameters for training.	32
4.17. Metrics from collected episodes during a successful training	33
5.1. Difference in success rate and goal completion rate during early stages of training. .	35
5.2. Possible effects of slow policy computation on the performance.	39
6.1. Agent field of view at beginning of episode with <i>Random</i> spawnOrientation	43
6.2. Final reward function	47
6.3. Success and collision rates for all difficulties for an agent trained exclusively on hard tracks with standard light	48
6.4. All success and collision rates for an agent trained exclusively on hard tracks with standard light	48
6.5. Success and collision rates for all difficulties with standard light for an agent trained on hard tracks with all light settings	50
6.6. Success and collision rates for an agent trained on hard tracks with all light settings	50
6.7. Success rates for an agent trained on all light settings with histogram equalization .	51

6.8. Success rate comparison for an agent that is trained with or without histogram equalization on all light settings	52
6.9. Example of a collision from a successfully completed episode	52
6.10. Properties of the collected episodes over time for the most successful model	54
8.1. Success and collision rates for standard light setting.	57
8.2. Goal passing behaviour of the trained agent	58
8.3. Success and collision rate comparisons for light settings.	58
8.4. Replay times on jetbot hardware	59
8.5. Differences in policy outputs between recordings and replays on jetbot hardware . .	60
8.6. Results of the deterministic check across all evaluations during the experimentation phase	61
8.7. Success rates for policy evaluation using fresh and non-fresh observations	63
8.8. Evaluation of the DifferentialJetBot policy with both jetbot versions	64
A.1. Neural Network Architecture Action Head	IV
A.2. Neural Network Architecture Value Head	V

List of Tables

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.	25
5.1. Basic evaluation algorithm parameters	36
5.2. Collected and aggregate success_rate metrics	37
6.1. Agent movement with fixed step duration 0.3 seconds	45
6.2. Selected hyperparameters for the PPO algorithm	46
6.3. Agent capability check for individual reward functions.	46

1. Introduction

Recent advancements in artificial intelligence 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 notably over the last few years is the introduction of diffusion image models and large language models. These technologies were well received and moved artificial intelligence tools into public discourse. All over the world people have recognized the potential of artificial intelligence technologies and are now using them in their daily life and at work.

Artificial intelligence has already been of big 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 for example 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 [3] in their products. Due to the recent developments in artificial intelligence and the very high complexity of the task of autonomous driving, artificial intelligence 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 to autonomous driving agents in a simulated environment. This work builds upon the work of [1] and will use the same task and evaluation metrics. This thesis focusses on improving the agent's resiliency to changing light conditions by training a convolutional neural network end-to-end using reinforcement learning.

2. Research Goals

The goal of this thesis is to contribute in the domain of autonomous driving by investigating the use of reinforcement learning to train an autonomous driving agent that is resilient to changes in light conditions. The agent is evaluated on simulated parcours that consist of a series of goals indicated by two blocks, a parcour is successfully completed if the agents drives through all goals without collisions. This thesis builds upon previous work at the ScaDS.AI [1] and uses the same parcour and task specifications. The agent from previous work was not able to reliably complete parcours under changing light conditions, motivating the research goals of this thesis.

The self-driving agent is trained using reinforcement learning in a simulated environment, the training process will include changing light conditions and possibly data augmentation to help the agent generalize. Parcours of different difficulties and lighting settings are used to evaluate the agent's reliability and generalisation capabilities. The most important evaluation metric is the success rate. A parcour is considered a success when the autonomous driving agent passes all goals without any collisions.

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

The previous work [1] showed that it is possible to train an agent using reinforcement learning to solve the evaluation parcours, however the trained agents were not successful in reliably traversing the parcours of higher difficulty levels. Furthermore this work will implement the agent in a fundamentally different way, the agents developed in previous work utilized an extensive pre-processing pipeline to extract the relevant information from the camera images whereas the agents in this thesis will use a convolutional neural network to learn and extract the relevant information themselves.

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 an agent to reliably solve the parcours of all difficulty levels. This raises question 1: Is it possible to train an autonomous driving agent consisting of a

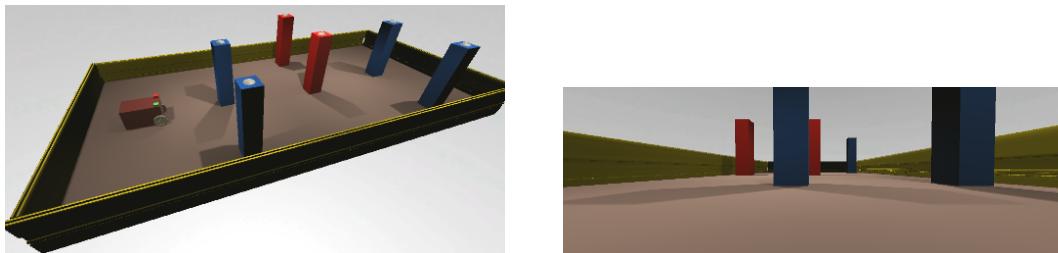


Figure 2.1.: Example image of a parcour and the agent's camera

convolutional neural network with end-to-end reinforcement learning to reliably solve the parcours of all difficulty levels?

The question will be answered by training agents that have been developed based on related work and analyzing their performance on the evaluation parcours. The evaluation parcours consist of different difficulty levels, the agent's success rate will be primarily used to answer the question.

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

While question 1 simply investigates if it is possible to train an agent to reliably solve the parcours of all difficulty levels, question 2 investigates if it is possible to train an agent that is robust to changing light conditions in addition to being capable of solving parcours of all difficulty levels. The performance of agents from previous work [1] declined massively under changing light conditions. This raises question 2 - Is it possible to use an end-to-end trained CNN to make the agent robust to changing light conditions?

The question will be answered by training agents that have been specifically designed to be robust to changing light conditions, the agents will be trained using reinforcement learning in a simulated environment with changing light conditions and possibly further data augmentation to help the agent generalize and learn. The agents will be evaluated on the evaluation parcours used in question 1 with changing light conditions. Similarly the success rate will be primarily used to evaluate and compare the agent's performance. The difference in performance for different light conditions will be used to answer the question, if the performance is similar for all light conditions the agent is considered robust to changing light conditions.

2.3. Question 3 - Is it possible to use a neural network that can be transferred to a physical robot?

One goal of the ongoing research at the ScaDS.AI is to build real life robots for demonstration and research purposes [6], the robots are based on the NVIDIA JetBot platform. The robots are equipped with a camera, wheels and a small computer. A future goal is to transfer a trained agent onto these robots, however the limited processing power of these robots might not be sufficient for more complex agents that utilize neural networks. This raises question 3 - Is it possible to use a neural network that can be transferred to a physical robot?

The question will be answered by investigating the processing power required to run the pre-processing steps and neural networks used in the agents that are developed in this thesis. This will be evaluated empirically by creating replays of the agents in simulations and running these replays on the physical robots. If the robots are able to reproduce the behaviour from the replays, the agents can be considered transferable to the robots.

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. 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 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 equipped with a single camera sensor. The agent's behaviour is controlled by a convolutional neural network that processes the visual input. The agent has to learn a specific self driving task with reduced complexity. The self-driving task is similar to previous research at the ScaDS.AI [1]. However the agent design and training is very different. The task consists of a restricted 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. The environment is simulated with three light settings. The trained agent has to learn to navigate all difficulty levels and light settings.

Research relating to reinforcement learning algorithms, convolutional neural networks and self-driving will be reviewed.

3.1. Self-Driving

Neural networks have been used in the domain of self driving for a long time. Pomerleau [7] developed one for the first self driving systems with neural networks in 1989. The system was developed for road following. The system used a camera image and a laser range finder as input. Compared to current network sizes, they used a small fully connected neural network. 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 reinforcement learning. The system was able to follow roads in real-life under certain conditions. The paper highlights the advantages of neural networks for self-driving. The training data determines the salient image features and not the programmer. This has proven to be true with the success of convolutional neural networks in the domain of self-driving [8].

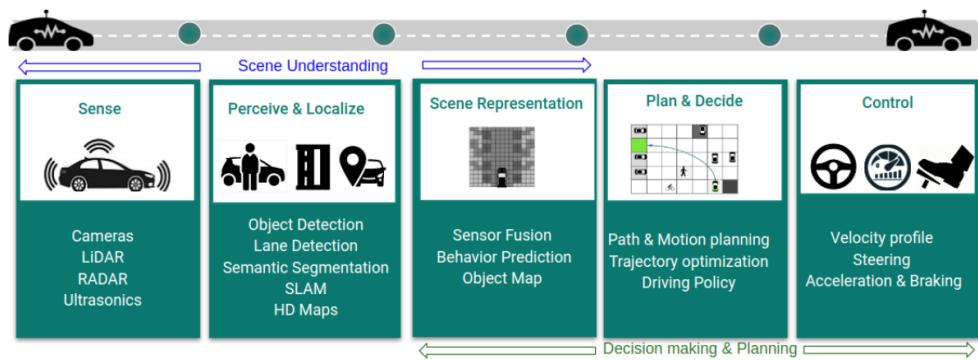


Figure 3.1.: Standard components in a modern autonomous driving systems pipeline listing the various tasks. The key problems addressed by these modules are Scene Understanding, Decision and Planning. Image from [4]

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 approaches such as Radar, Lidar and cameras. Thermometers 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 convolutional neural networks [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]. Components for Scene Understanding build a model of the current surroundings. The Decision making & Planning components use this model to decide and execute the next actions. Multiple interacting components are shown in figure 3.1.

An agent with similarly complex components is not feasible for this thesis. The agent in this thesis uses a single convolutional neural network to process the visual input and make decisions instead. I aim to contribute to the domain by expanding on previous research and focus on the training of a convolutional neural network agent.

TODO wohin diesen satz? Approaches from the domain of self-driving will be used to improve the training of the agent, for example reward shaping [4].

3.1.1. Previous Work

This thesis builds directly upon the work of Knig [10], Flach [6] and Schaller [1]. [10] 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 neural network. This network used the extracted features as inputs. It was trained using an evolutionary approach in simulation.

[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.

[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 [10]. The neural network policy was trained using the proximal policy optimization [11] reinforcement learning algorithm. 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 to 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 [8] motivate the choice of CNNs as the feature extraction method in this thesis.

3.2. Reinforcement Learning

Reinforcement Learning algorithms have been around for a long time, but only recently have they been able to achieve superhuman performance in games and control tasks [12]. Most reinforcement learning algorithms formalize the problem as consisting of an environment and an agent. 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 algorithm 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 reinforcement learning algorithms is to train an agent that interacts with its environment and maximizes the cumulative reward over time. The trained agent learns the desired behaviour and can be used to solve problems in the environment. The process of selecting an action is called the policy π [13]. The policy defines the agent's behaviour.

The agent interacts with the environment during the training phase of RL algorithms. The agent takes some representation of the environment as input and selects actions to execute in the environment. The selected actions are executed in the environment which results in a new state. The reward function assigns reward to these state transitions. The observed rewards are then used to update the policy 3.2.

Reinforcement learning 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

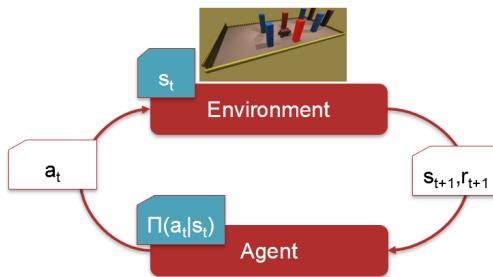


Figure 3.2.: RL Training Cycle: The agent selects action a_t based on policy $\pi(a_t|s_t)$ at state s_t and receives the next state s_{t+1} and rewards r_{t+1} from the environment. The states, actions and observed rewards are used to update the policy.

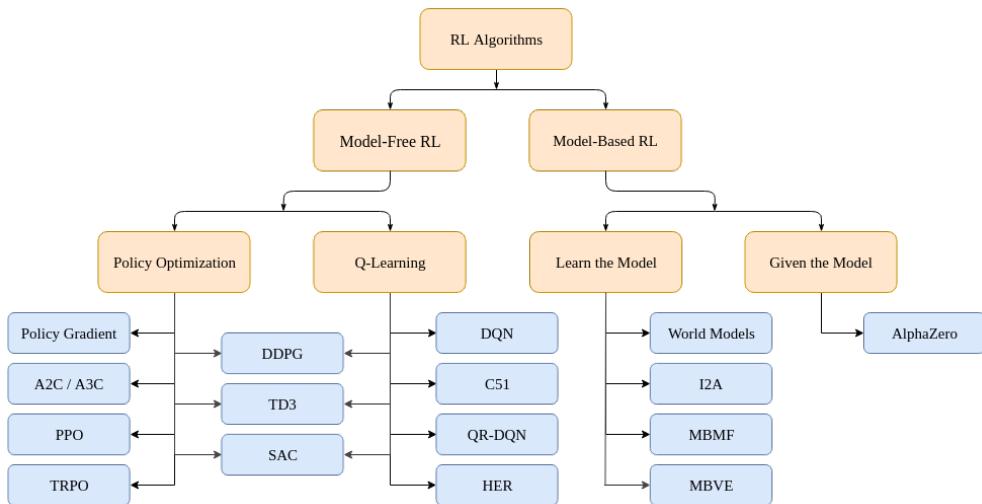


Figure 3.3.: Taxonomy of RL algorithms from OpenAI's Spinning Up course [15]

to be successful at many tasks that require planning [14]. Model-free approaches are often simpler and more flexible, they have shown great success in various control tasks [12].

Model free algorithms can be further divided into two families. The first family are value-based approaches. These algorithms learn a value function that assigns state and action pairs a value that represents the expected future reward. The policy selects actions based on this function. The state action pair with the highest Q-value is selected. The Q-learning algorithm is an early and common example of value-based algorithms [13]. The traditional Q-learning algorithm uses a table to store the 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, many extensions to the algorithms have been developed such as for example deep q-learning. Deep q-learning uses a deep neural network to approximate the Q-values [12]. The network learns to predict the Q-values for state-action pairs. Neural networks are general approximators. They can return accurate value predictions even for previously unseen states, compared to the state-action table. Although Deep q-learning have been used to great success for control tasks they will not be used in this thesis, as they require discrete action spaces and the environment in this thesis consists of a continuous action space.

3. Related Work

The other family are policy-based algorithms, instead of learning the values associated to state-action pairs these algorithms learn the policy directly. This allows for the use of continuous action spaces. The policy is often represented by a (convolutional) neural network. The policy's parameters are updated to maximize the expected future reward. 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.

Alternatively the policy can be updated using Actor-Critic approaches. These combine policy-based and value-based approaches. Actor-Critic approaches contain a policy and a value function. The value function represents the expected future reward of a state. The policy is updated using this value function.

The Proximal Policy Optimization algorithm is an Actor-Critic algorithm. It was developed to improve the stability of policy-based algorithms [11]. The PPO algorithm restricts the size of policy changes caused by parameter updates, which ensures the policy does not change drastically. This improves stability. PPO is currently one of the most popular algorithms for reinforcement learning, and has already been successfully used in the domain of autonomous driving [1].

Convolutional Neural Network for Reinforcement Learning

Convolutional neural networks are a neural network architecture specifically developed for processing image data, they consist of a number of filters and a fully connected neural network. The filters are applied to the image in a sliding window fashion, the filters detect patterns in the image such as for example edges and corners. Multiple successive applications of such filters enables the network to learn hierarchical information and recognize more complex structures. The fully connected neural network analyses the results of the filters and makes the final prediction [13].

CNNs are often used in Reinforcement Learning since RL problems often require an agent to process visual input. Furthermore CNNs can be trained end-to-end in Reinforcement Learning compared to other feature extraction methods, which means the CNN can learn what features are important for the task at hand. Therefore a convolutional neural network will be used to process the camera images instead of a hand-crafted feature extraction method.

CNNs typically do not take the raw camera/simulation images but rather preprocessed images, e.g. greyscaled images [12]. Preprocessing steps can help in reducing the complexity of the input space. Convolutional neural networks require a lot of data to learn, data augmentation can help increase the size of the training set and to make the agent more robust. Data augmentation generates new samples from already collected ones by applying transformations to the samples.

The PPO algorithm can be used to train agents that use convolutional neural networks as their policy. Convolutional neural networks are ideal for processing image data and can be trained end-to-end by the algorithm [11].

3.3. Simulation for Reinforcement Learning and Self-Driving

Simulations play a huge role in reinforcement learning and thus 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 reinforcement learning agents. Simulations also allow for the creation of scenarios that are not possible in the real world. This is especially useful for reinforcement learning 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 [16].

Simulated environments often serve as baselines for reinforcement learning algorithms, most famous are the atari games [12]. The Python Gymnasium API was developed for easy reuse and comparison of reinforcement learning algorithms for different problems [17], the Gymnasium API defines an interface that can be used to model tasks as reinforcement learning problems. A wide range of reinforcement learning frameworks support the Gymnasium API, for example Google's dopamine [18] and OpenAI's baselines [19]. Advanced simulations like the Unity engine [20], the physics simulator MuJoCo [21] and the driving simulator Carla [16] can be integrated with the Gymnasium API.

There are also dedicated frameworks for reinforcement learning that directly integrate with simulation engines. [1] used the ML-Agents framework [22] to train the self-driving agent in Unity directly.

In this thesis Unity will be used for the simulation, the simulation will be integrated with the Python Gymnasium API and PPO algorithm. This approach is chosen instead of the ML-Agents framework since it allows for more flexibility and control over the simulation and training process.

3.4. Imitation learning

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. Neural networks can be used to control the agent behaviour instead. The neural networks can be trained using reinforcement learning in simulation. Imitation learning is another approach to training an agent that interacts with its environment. Imitation learning requires a dataset that demonstrates the desired behaviour. The agent is trained on the dataset to mimic the expert behaviour.

In reinforcement learning the programmer has to define a reward function that the agent uses to learn and improve its behaviour. In imitation learning 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 approaches to training an agent with imitation learning, behavioural cloning and inverse reinforcement learning. Behavioural cloning is the simpler approach, the agent learns to mimic the expert behaviour directly. This is similar to supervised learning. Inverse reinforcement learning is more complex, the reward function from the expert demonstrations is learned first. The agent is then trained to maximize this reward function using reinforcement learning.

Bojarski [23] used imitation learning to train a self-driving lane following agent for real-world environments. This agent consists of a convolutional neural network that processes the camera images and predicts the steering angle directly. The agent is trained end-to-end to reproduce steering behaviour from recorded data. They demonstrate that imitation learning is a viable approach for developing an agent without the need for multiple components such as object detection and path planning.

Tesla used imitation learning 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 [24].

Imitation learning and reinforcement learning can be combined to improve the training of the agent. The reinforcement learning process generates samples via interaction between agent and environment. These samples and the expert behaviour dataset are used together to train the agent policy [25].

3.5. Light change resiliency

3.5.1. Domain Randomization

useful for transfer sim-to-real <https://arxiv.org/html/1703.06907>

domain randomization for sim-to-real tries to teach the agent to solve a task under a big variety of conditions in simulation, the hope is that the agent will be able to generalize this variation and learn in this environment. The real world is simply one variety that the agent might have already learned to generalize.

The randomization can include textures, image resolutions, ..., motor power of agent in sim, ... (physical properties)

domain randomization can lead to a high variance of policy performance for different environment randomizations. The paper <https://arxiv.org/html/1910.10537> introduces a regularization approach for the policies. visual vs dynamics randomization (camera changes vs physics e.g. friction of wheels)

3.5.2. Data augmentation

Data is collected in simulation and then augmented

3.5.3. Multi-Modal learning

Provide the agent with more than just image data, e.g. radar...

3.5.4. Hierarchical Reinforcement Learning

combination of multiple levels. higher level policy controls the actions, the lower level policy learns to build representations for the higher level policy

Hierarchical Reinforcement Learning: Using a hierarchical approach where high-level policies govern low-level policies can help the agent adapt to changing conditions more effectively. High-level policies can decide on strategies based on the overall environment, while low-level policies handle specific tasks like adjusting to lighting changes

TODO relate the previous work that used a hand-crafted feature extraction pipeline to hierarchical rl

3.6. Transfer to reality

3.6.1. Transfer Learning

learn on diverse data and then fine-tune on real data

3.7. Key Ideas

3.7.1. Fitting RL algorithms

3.7.1.1. CNN for feature extraction in RL

3.7.1.2. Memory mechanism

3.7.2. Reward shaping

3.7.3. Environment Implementation

speed of environment is critical...

3.7.4. Image preprocessing for resiliency to light changes

3.7.5. Domain Randomization

4. Methods

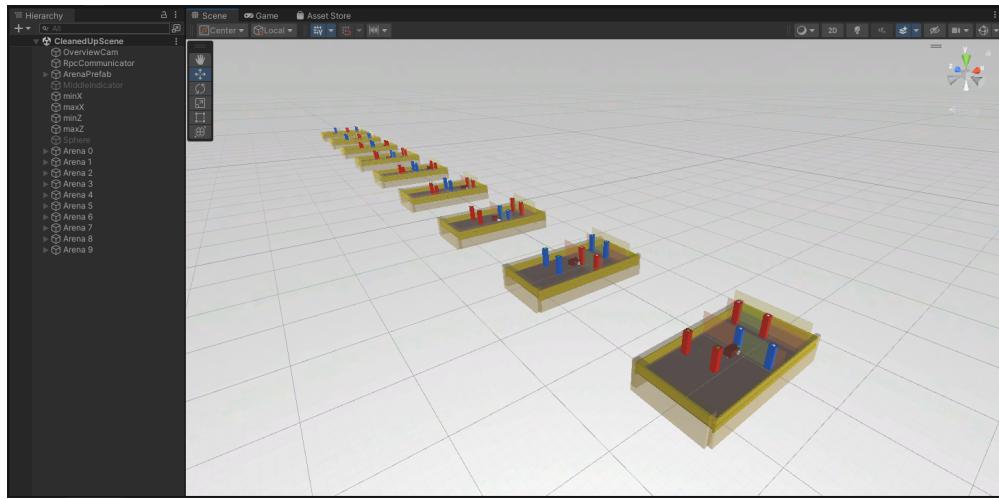


Figure 4.1.: Parallel simulation of multiple environments in one Unity instance

4.1. Environment Description

4.1.1. Environment Simulation

The environment is simulated in the Unity game engine. The reinforcement learning algorithm is implemented in Python and builds upon the stable-baselines3 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 reinforcement learning algorithm and the Unity engine. The communication is implemented using JsonRPC and the Peaceful Pie [26] library.

Parallel environment processing The stable-baselines3 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 4.1. The separation between the Unity engine and the Python algorithm requires JsonRPC calls for every environment step and reset. This can slow the training process. The overhead of sending calls to the Unity engine can be reduced by bundling the calls for all environments in one JsonRPC call. The *use_bundled_calls* config parameter enables the bundling of calls.

Non-blocking step calls The step function of the Unity environment is non-blocking. This means that Unity returns the call before the entire step transition is complete. This saves time by reducing the amount of JsonRPC calls. In conventional reinforcement learning each step transition results in a new observation that is used to predict the next action. 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 after the next step has completed. Essentially the returned observation is delayed by one step or $fixedTimestepsLength\text{seconds}$. A shorter $fixedTimestepsLength$ results in smaller changes of the environment during the step transitions. In this case the delay of observations

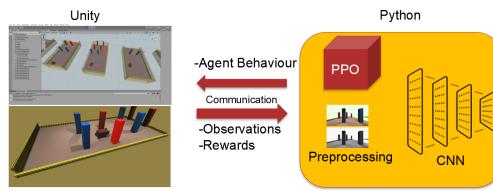
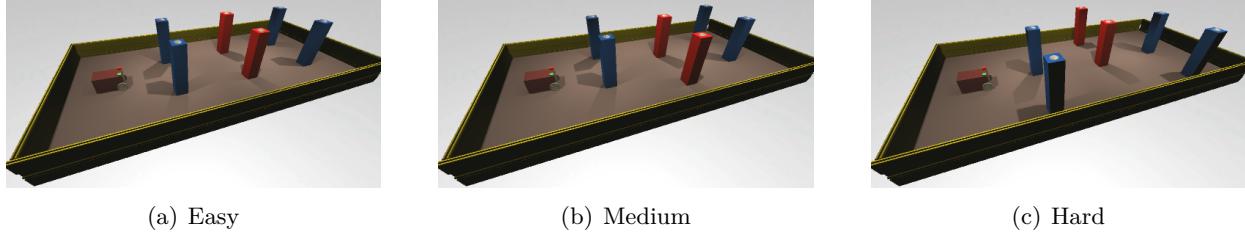


Figure 4.2.: Communication between Python and Unity



(a) Easy

(b) Medium

(c) Hard

Figure 4.3.: Example evaluation tracks for each difficulty setting.

has less of an impact on the accuracy of the observations. Experiments show that the policy can learn to deal with this delay.

Alternatively there is the policy parameter `use_fresh_obs`. If this parameter is set to true, the policy will request a new observation from Unity via another JsonRPC 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 JsonRPC 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 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 4.3. The tracks in each setting are structurally very similar to each other. They differ in goal coloring and the orientation of the shift from the center line.

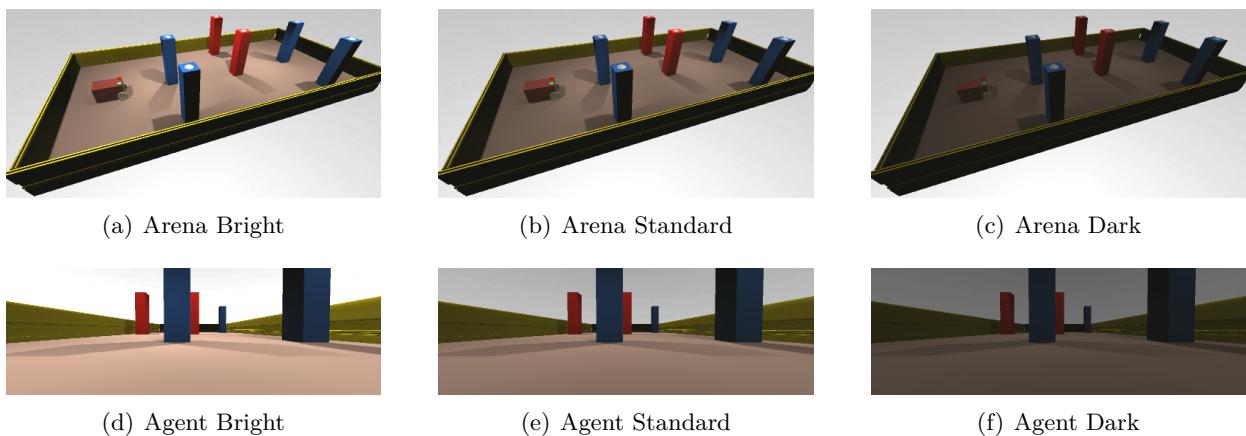


Figure 4.4.: Arena and agent camera at different light settings.

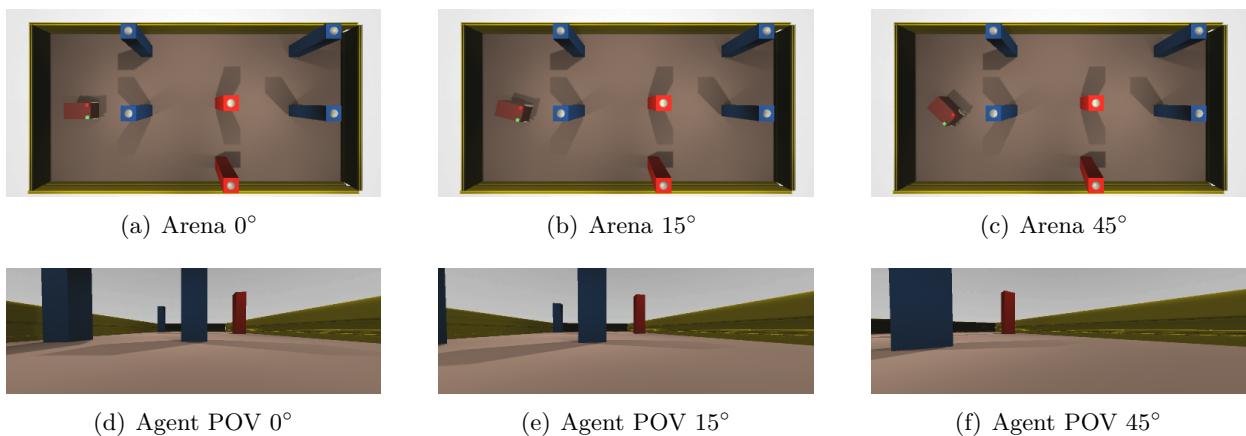


Figure 4.5.: Example Spawn Orientations and agent camera views for a hard track

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 initial 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 4.5.

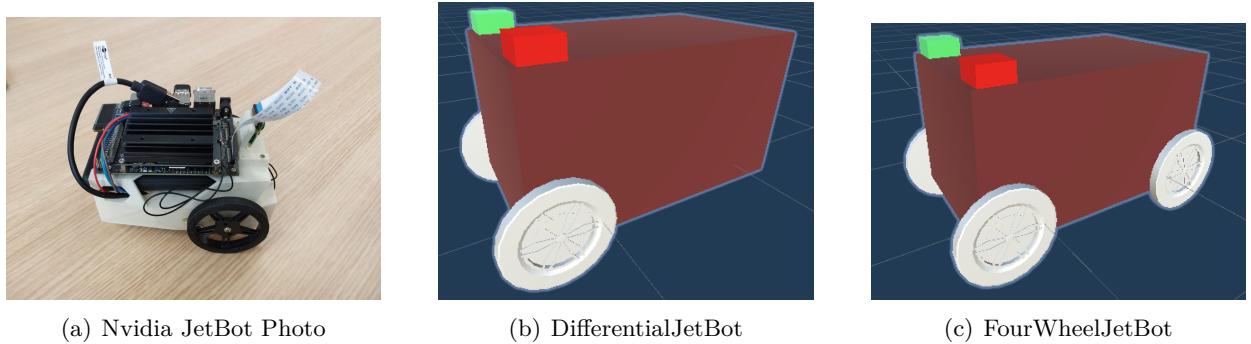


Figure 4.6.: Original Nvidia JetBot and simulated JetBot Designs

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 a *leftAcceleration* and *rightAcceleration* value from the Python algorithm. The values range from -1 to 1 and control the agent's wheels and movement. 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 FourWheelJetBot has 2 steering driving wheels at the front and two non-driving wheels in the back. The DifferentialJetBot steers by applying 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 steers by turning the front wheels in the desired direction and applying equal torques to both 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. The FourWheelJetBot was used in the work by [1]. The DifferentialJetBot was developed to match the physical NVIDIA JetBot more closely. The two JetBot designs are shown in figure 4.6.

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 during each step. This includes things like agent movement, collision detection and reward calculation.

4.1.4.1. Step duration

The duration of each step is defined by the environment settings. There are two distinct 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 6.1.4. 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 4.7. The figure shows how the Unity environment waits for new steps to start. The figure also 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 transferred 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 4.7. The figure shows how the Unity environment waits only for the first step. The figure also 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.

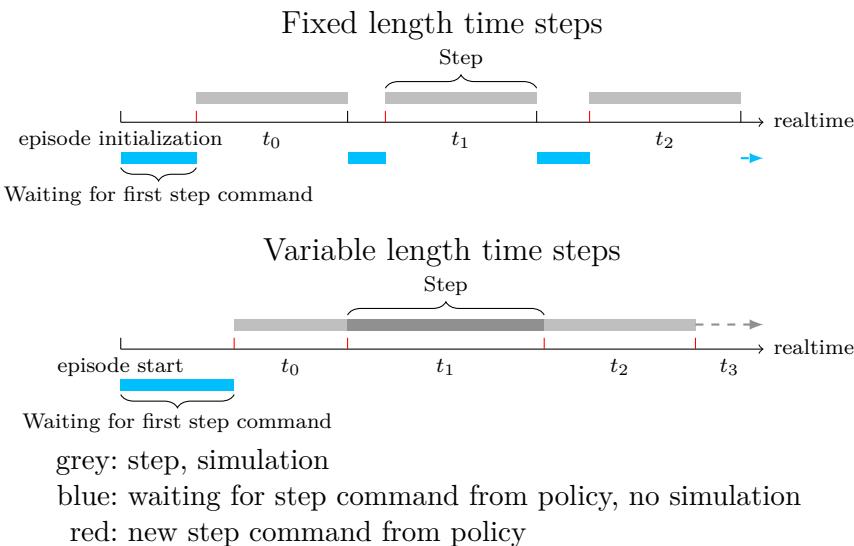


Figure 4.7.: Timeline of steps in Unity for fixed and variable duration

4.1.4.2. 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. The reward function assigns rewards to the agent based on its actions in the environment. The reinforcement learning algorithm trains the agent to maximise the cumulative reward over an episode. The reward function is a critical component of the reinforcement learning process. It is important to design a reward function that is closely aligned 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 4.8. 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

$$\text{num_goals} * \text{goal_reward} + \text{parcour_reward} = 3 * 100 + 100 = 400$$

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

Figure 4.8.: Event Reward function

s_t : state t a_t : action in state t

. 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.

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 exacerbate the issue of the agent learning unintended behaviour. The dense rewards can be opinionated and encourage the agent to follow a very specific path. This can lead to the agent not exploring the environment enough to find the very best policy. 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 [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 awarded 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.

4. Methods

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 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.

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 agent 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 the eventReward was combined with the velocityReward to train the agent. In this work we combine the individual reward functions in the composite reward function 4.9. 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.

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 4.11. In previous research the episodes were terminated upon collision and when missing a goal [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 initial 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 4.10. The policy did not learn to complete the parcour during the remainder of training. The unrestricted mode

4. Methods

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

Figure 4.9.: 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

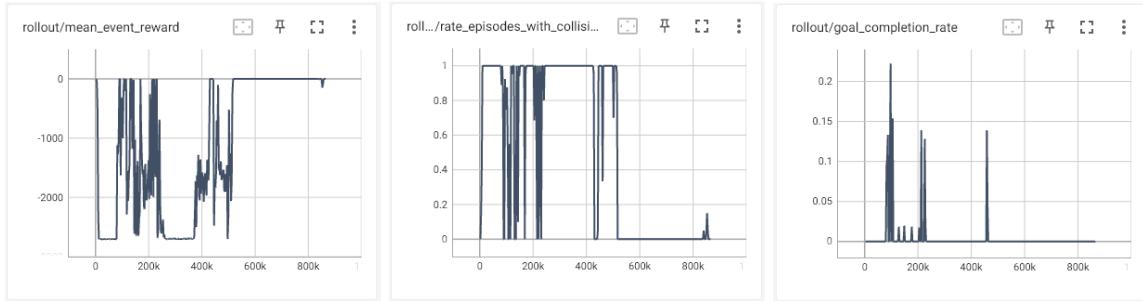


Figure 4.10.: Event Reward only training with *collisionMode unrestricted*

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 successful policy.

4. Methods

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 neural network training.
ignore	Collisions are ignored, no negative reward is given.	The agent might learn to avoid collisions based on other rewards.

Figure 4.11.: Collision Modes

4.2. Policy Description

The policy dictates what action the agent takes. The policy consists of preprocessing steps, a memory mechanism and a neural network. The policy is applied to images from the agent camera. The resulting actions are then applied to the agent in the Unity environment. The preprocessing steps and memory mechanism are considered as part of the policy as they are applied to the input images before the neural network. The preprocessing steps cannot be changed for a trained policy, as they define dimensions and contents of the neural network's inputs.

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's neural network. The network then produces an action.

4.2.1. Preprocessing

The images from the agent camera are preprocessed before they are fed into the neural network. The preprocessing steps are applied to reduce the size of the neural network'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 neural network 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 neural network's inference time. The neural network 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 neural network 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.

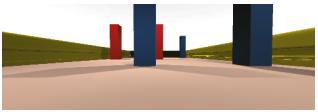
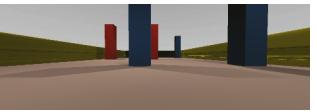
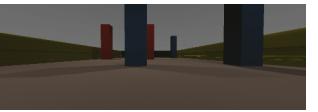
light Setting	Bright	Standard	Dark
Agent Camera Image from Unity			
Downsampled			
Grayscale			
Equalized			
Final Input Image			

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 neural network 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 settings 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 downsampled. 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 distinguishable. The advantage of grayscale images is that the policy's neural network 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

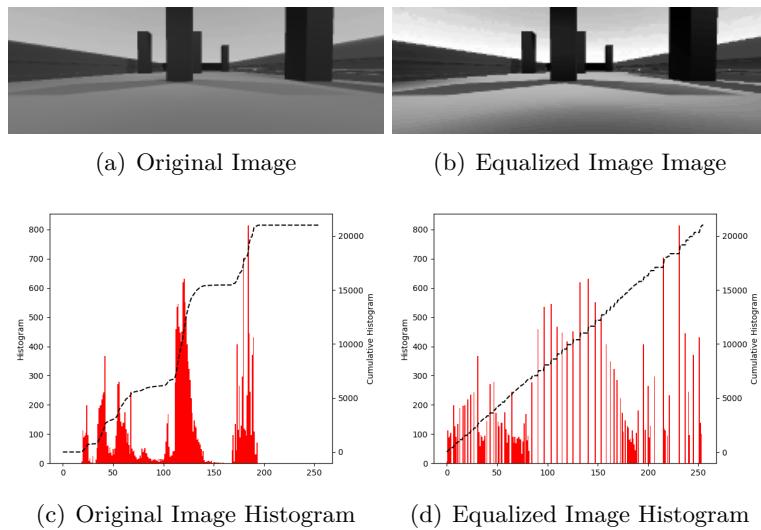


Figure 4.12.: Histogram equalization of a grayscale image from standard light setting

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 4.12.

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 neural network to detect the objects.

4.2.2. Memory Mechanism

Memory mechanisms are a common approach to enhancing a reinforcement learning agent. Memory mechanisms can be implemented in many different ways. Our memory mechanism uses frame stacking, similar to [1] and [12]. The memory is configured with the *framestacking* parameter. The memory is a simple first-in-first-out buffer that stores the last *framestacking* 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 neural network input. This neural network 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 *framestacking* 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.

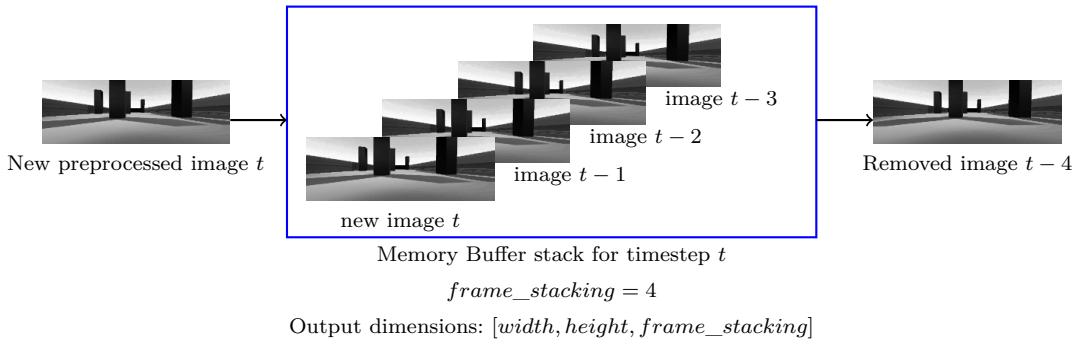


Figure 4.13.: Memory Mechanism

4.2.3. Neural Network

The neural network is the core of the policy. It is responsible for producing the actions from the image observations. The neural network belongs to the class of convolutional neural networks. This architecture was chosen since convolutional neural networks are ideal for processing image data. The convolutional network's architecture follows the specifications from [27]. This architecture has been used successfully in comparable reinforcement learning problems.

The network takes the output of the memory mechanism 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 [11].

4.2.3.1. Input Space

The input space of the neural network is three dimensional. The input is structured like images with *width*, *height* 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.

4.2.3.2. Action Output Space

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

4.2.3.3. Architecture

The neural network has an action and a value head. The action head produces the action distribution. The value head produces an estimate of the state's value. The action and value head share a part of the neural network. The shared part is called the feature extractor and constitutes the first four layers. 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.

The action head consists of a single fully connected layer with two outputs. The two outputs represent the mean of the 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 deterministic 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 output space's range $[-1, 1]$. This is necessary as the sampling can return a value outside of the range.

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

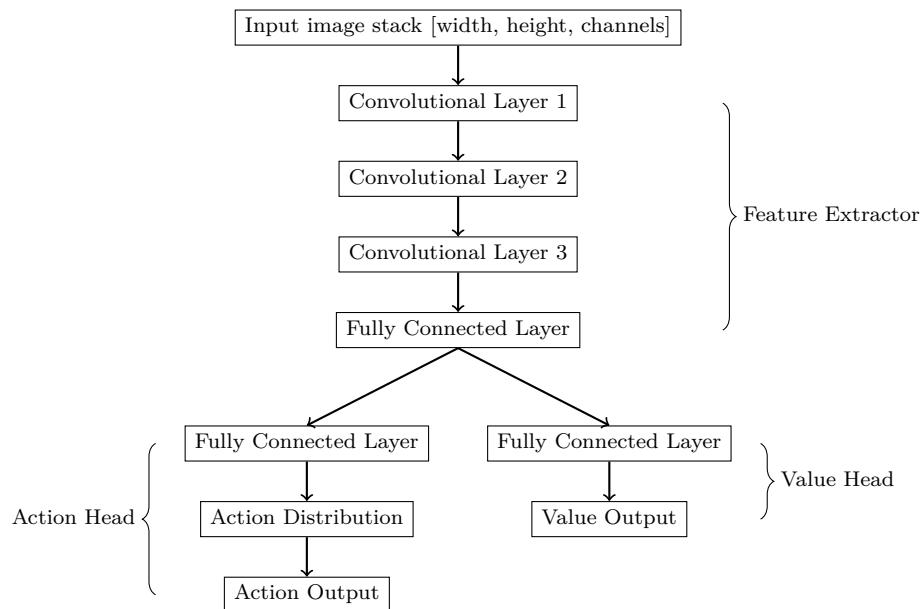


Figure 4.14.: Neural Network Structure

The neural network's input dimensions are determined by the agent's camera image dimensions, preprocessing parameters and the memory mechanism. The network architecture follows the specifications from [27]. However the input dimensions differ from the original network. This results

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

$agentImageWidth: 500$
 $agentImageHeight: 168$
 $downsampling_factor: 2$
 $greyscale: \text{True}$
 $equalize: \text{True}$
 $frame_stacking: 10$

Figure 4.15.: Neural network layers and parameters for the best configuration

in a different number of parameters for the feature extractor's layers. The parameters and layer dimensions for the most successful network configuration are shown in figure 4.15.

4.3. Training Algorithm

The neural network 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 reinforcement learning 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 reinforcement learning 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 full evaluation of the model.

Algorithm 4.3.1: TRAINNETWORK()

```

comment: Main Training Algorithm

main
  Env ← ENVIRONMENT(environment_parameters)
  RolloutBuffer ← ROLLOUTBUFFER(rollout_buffer_size)
  Model ← MODEL(model_parameters)
  Optimizer ← OPTIMIZER(optimizer_parameters)
  num_timesteps ← 0
  while num_timesteps < total_timesteps
    do {  

      num_collected_steps ← COLLECTDATA()  

      num_timesteps ← num_timesteps + num_collected_steps  

      TRAINMODEL()
    }
  
```

4.3.1. 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 samples of previous iterations from memory. The data collection process is a simple loop that collects a fixed amount of samples from the environment. The reinforcement learning algorithm Proximal Policy Optimization requires that the samples are collected on-policy. As a result the most recently updated model is used to choose 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 model 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 4.16.: Environment parameters for training.

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` 4.16. 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 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 4.17.

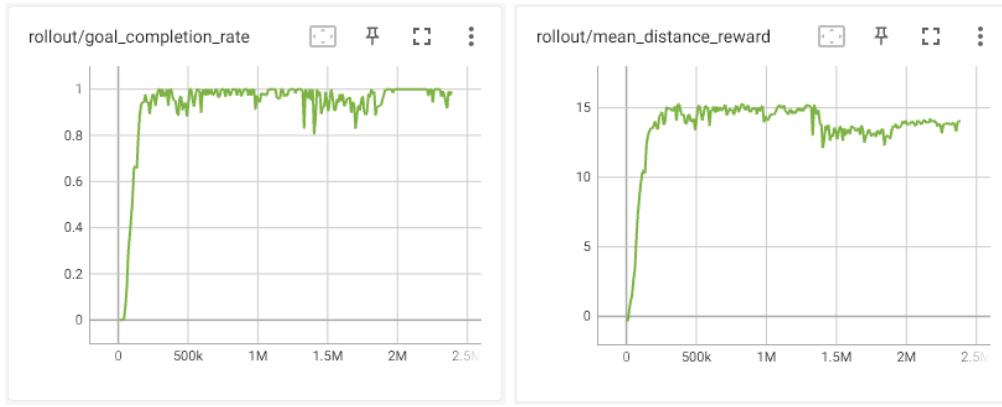


Figure 4.17.: Metrics from collected episodes during a successful training

4.3.2. 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 Proximal Policy Optimization algorithm [11]. 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 neural network 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 approaches. 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 neural network. 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.

4. Methods

The policy loss can also include an entropy term. Similarly to the original paper [11], no entropy term is used in this work. This results in the loss function (PPO Loss) with the value coefficient $c_1 = 0.5$.

$$\begin{aligned}
 L_t^{CLIP+VF} &= \hat{\mathbb{E}}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta)] && \text{(PPO Loss)} \\
 L_t^{CLIP}(\theta) &= \min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t) && \text{(Surrogate Objective)} \\
 r_t(\theta) &= \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} && \text{(Ratio)} \\
 L_t^{VF}(\theta) &= (V_\theta(s_t) - V_t^{targ})^2 && \text{(Value Error)}
 \end{aligned}$$

4.3.3. 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 4.1.4.2. Collisions of the agent do not disqualify an episode from being successful, 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 be 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

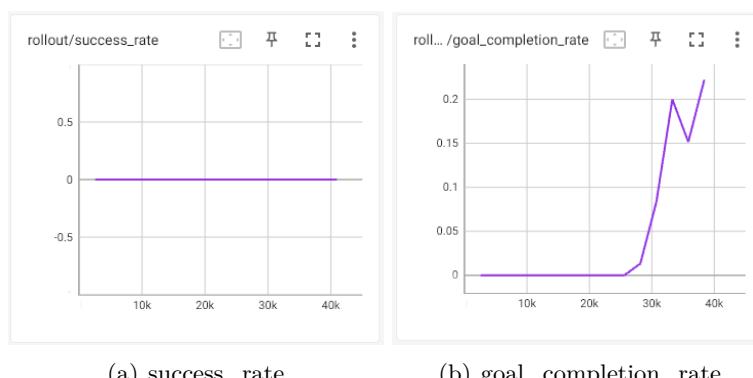


Figure 5.1.: Difference in success rate and goal completion rate during early stages of training.

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	
log	True	If True, log all metrics to tensorboard
	False	
record_videos	True	If True, record videos
	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 G.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 policy 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 for further analysis. 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 all 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 shows 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 transferring 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 transferring 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 transferred to the JetBot.

The developed policy 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 convolutional neural network 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 quickly enough to be able to act in real time.

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.

5.5.1. 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 turn- ing 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 preprocessing 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 transferred 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 neural network 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 agent 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 [12]

and the human-level control paper [27]. 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 agent with deterministic and non-deterministic sampling to determine if the agent's performance is affected by the sampling method.

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

5.6.2. Identical start condition Test

The environment is simulated in Unity and was described in 4.1 to not be fully deterministic. This means identical actions in identical environments could result in different outcomes. These changes are small, such as changes to a few pixels in the agent camera. However these small changes might result in different agent actions. These actions further influence the environment and following actions. This means that episodes with identical starting conditions could result in different outcomes. This could be problematic when evaluating the agent, as the agent's performance could be influenced by the environment's randomness.

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. A low number of groups indicates that the agent's performance is consistent. Ideally there is only one group that all episode results belong to. The episode results are characterized by the endEvent, collision and the three goals' completion.

5.6.3. Fresh Observation Test

In the standard reinforcement learning 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. This means the step calls return before the entire step transition has been completed. The observation after the step transition is not yet available when the call returns. The observation returned by the step call is the observation of the environment at the beginning of the step transition. 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 after the next step has completed.

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 4.6. 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 available 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 transferred 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. Finding appropriate hyperparameters for 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 is 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 agent with *Fixed* and *Random* orientation. It was not possible to train the agent with *VeryRandom* spawnOrientation. The *Random* orientation was chosen for all further experiments. The agent is spawned thus spawned with an orientation in the range of $[-15^\circ, 15^\circ]$ 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 500×168 . 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.

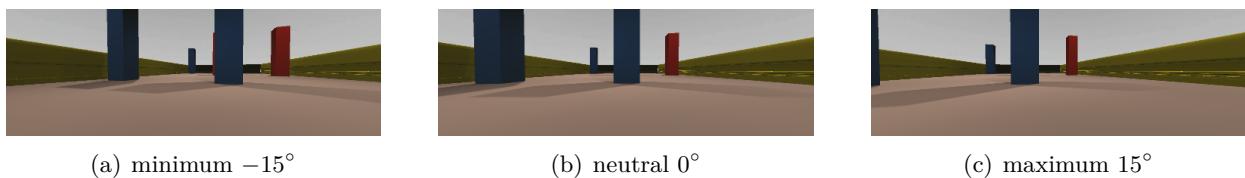


Figure 6.1.: Agent field of view at beginning of episode with *Random* spawnOrientation

6.1.3. Observation Space

The observation space defines the input dimensions of the agent's neural network. 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, greyscaling 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 greyscale. 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.

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 500×168 which was downsampled to 250×84 . The greyscaling 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 policies behave similar on different devices, regardless of the processing speed. This was useful 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 training algorithm. The training algorithm was adapted from the stable-baselines3 library [19]. 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 quickly summarized in the table 6.2.

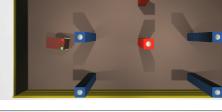
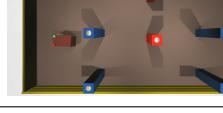
behaviour description	full speed ahead	steer right	turn on the spot
action	(1, 1)	(1, 0)	(1, -1)
initial position			
position after step 1			
position after step 5			
position after step 15			
position after step 30			

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

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 agent to learn the desired behaviour. The goal is to achieve an agent that completes the parcour 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 agent 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. We are conducting experiments to analyse the usefulness of the individual reward functions. First we analyse if the agent is capable of learning the behaviour encouraged by the reward function. This is done by training the agent 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 agent 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. The agent is capable of learning the behaviour encouraged by the distanceReward and velocityReward functions. However the agent is not capable of learning the behaviour encouraged by the eventReward and orientationReward functions.

6. Finding appropriate hyperparameters for training

parameter	function	value	tradeoffs
n_steps	amount of samples to collect per environment during data collection	64	+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 neural network 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 neural network initialization and random number generators	2048	+fixed neural network initialization

Table 6.2.: Selected hyperparameters for the PPO algorithm

function name	encouraged behaviour	learned behaviour	expected behaviour learned?
eventReward	agent drives through the parcour 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.: Agent capability check for individual reward functions.

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

Figure 6.2.: Final reward function

s_t : state t a_t : action in state t

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 build an agent that avoids collisions entirely.

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 agent 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 agent and the obtained rewards were monitored during training. The results showed that the agent 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 agent was able to learn the desired behaviour and complete the tracks reliably. 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 fixed difficulty setting

In the experimentation phase of this project the agents were trained exclusively on tracks of a specific difficulty setting. This was done to analyse the capability of the jetbot agent and to find appropriate hyperparameters that allow the agents 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 agent with the right hyperparameters to solve tracks of a particular difficulty level when the agent is exclusively trained on that difficulty level. For example the agent could be trained to solve the medium tracks very successfully without needing to encounter easy 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 agent on the different difficulty levels exclusively. The agents were able to reach success_rates of 100% for the collected episodes.

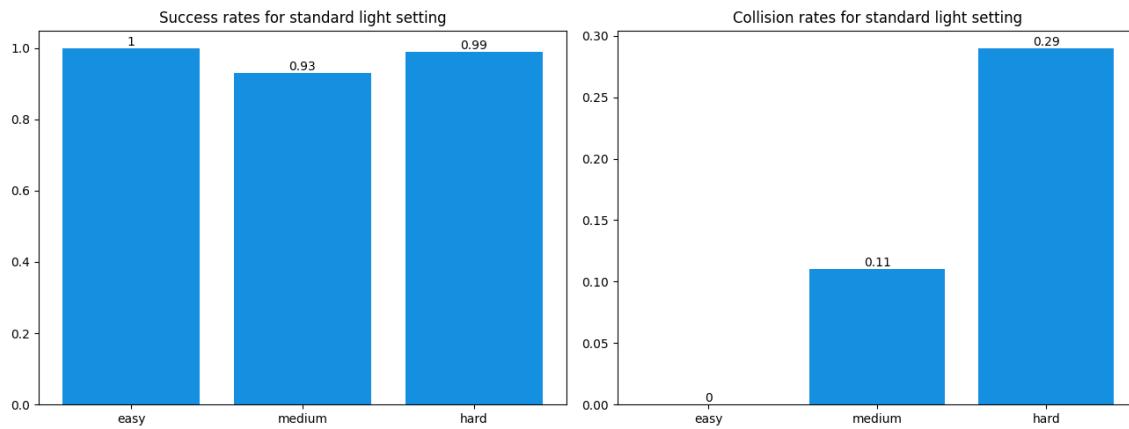


Figure 6.3.: Success and collision rates for all difficulties for an agent trained exclusively on hard tracks with standard light

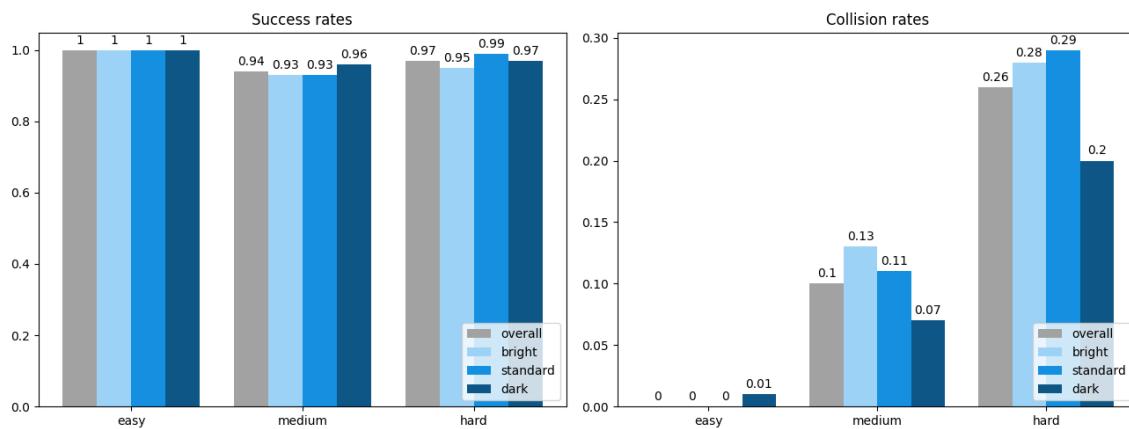


Figure 6.4.: All success and collision rates for an agent trained exclusively on hard tracks with standard light

Generalization to lower difficulty levels The evaluation of these agents showed that they were able to generalize to the tracks of lower difficulty levels. For example the agent 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 these agents on different light settings showed that the agents were able to generalize to the different light settings as well. The agents were trained on the standard light setting only. However the agents 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.

6.5. Experiments mixed difficulty setting

The next step was to train the agent on all difficulty levels at the same time. The goal was to find hyperparameters that would allow the agent to learn to solve all difficulty levels better than with the previous exclusive training. The agent 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 agent was not able to learn to solve the tracks as successfully as the isolated training on hard tracks. Further changes to the hyperparameters did not improve the learning process. For example the amount of data collected in each iteration was increased to balance out the increased variety of tracks.

6.6. Experiments mixed light setting hyperparameter

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

The next step was to train the agent on all light settings at the same time. The goal was to find hyperparameters that would allow the agent to learn to solve the tracks with different light settings better than with the previous exclusive training. The agent 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.

General Results The agent was able to learn to solve the tracks of different difficulties. Similarly to the previous experiments, the agent was able to generalize to the easy and medium tracks. The agent 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 the agent trained exclusively on hard tracks with standard light 6.4.

Performance on all light settings The agent was evaluated on the different light settings. The agent 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. The agent performed much better in episodes with bright or dark light setting compared to the agent trained exclusively on tracks with standard light 6.4.

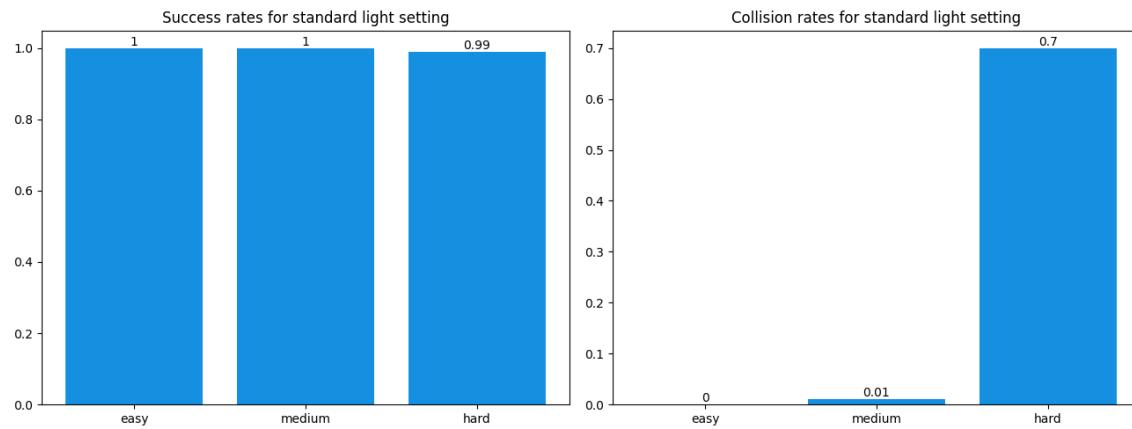


Figure 6.5.: Success and collision rates for all difficulties with standard light for an agent trained on hard tracks with all light settings

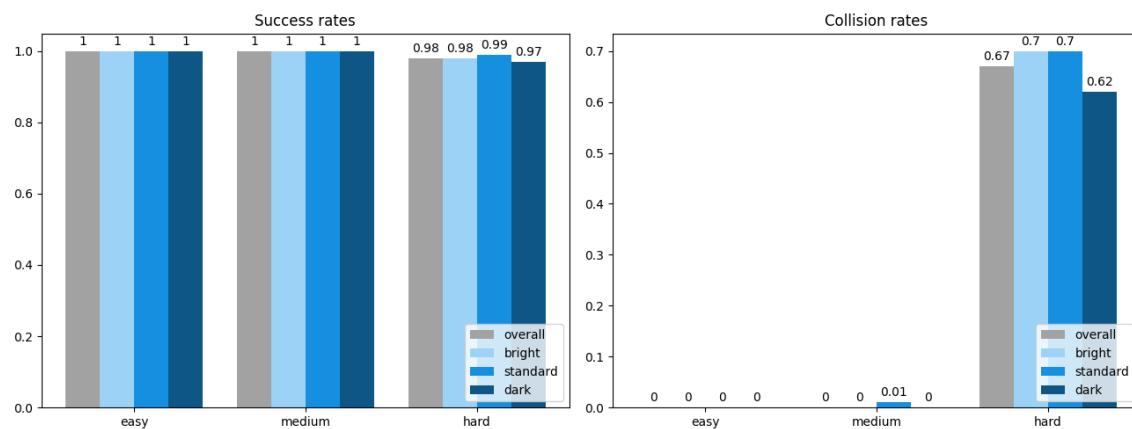


Figure 6.6.: Success and collision rates for an agent trained on hard tracks with all light settings

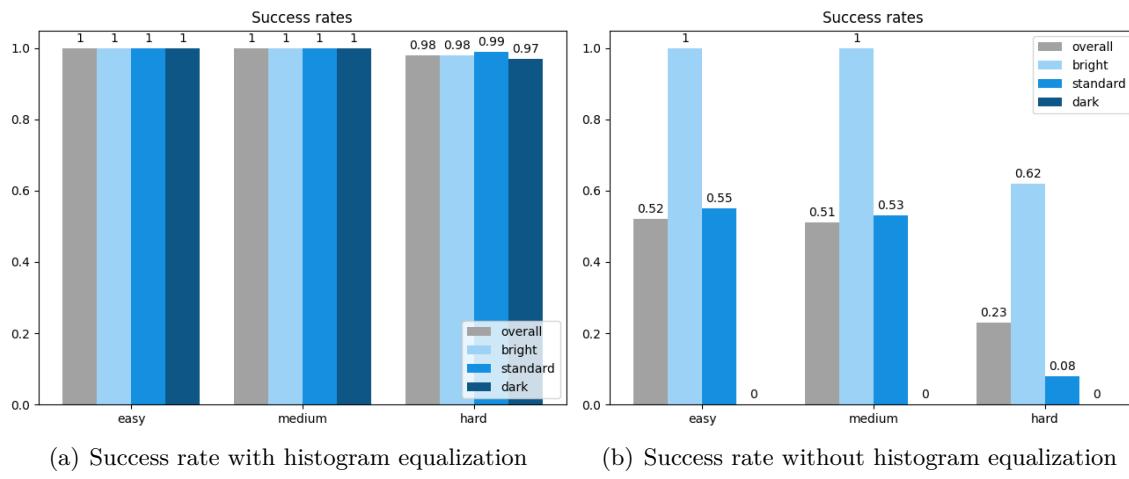


Figure 6.7.: Success rates for an agent trained on all light settings with histogram equalization

6.7. Experiments on the importance of the histogram equalization preprocessing step

The previous training results showed that the agent was able to learn to solve the tracks with different light settings. The agents used the histogram equalization preprocessing step 4.2.1.3 to increase the resiliency towards light changes. Agents that were trained on the standard light setting were able to solve tracks under bright and dark light conditions as well. This implies the histogram equalization plays a big role.

Evaluating an agent without the histogram equalization preprocessing step The agent from the previous section about mixed light settings during training was evaluated without the histogram equalization preprocessing step. This agent was trained with 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 agent was trained with the histogram equalization preprocessing step. The histogram equalization step changes the image content substantially 4.1. The agent has learned to use the output of the preprocessing step. As a result, the agent is not able to solve the tracks without the preprocessing step.

Training an agent without the histogram equalization preprocessing step An agent without the histogram equalization preprocessing step was trained on the mixed light setting to determine if the preprocessing step is required for high performance. The results of training an agent without the preprocessing step are shown in the second graph in 6.8. The agent was not able to learn to solve the hard tracks as successfully as the agent trained with the histogram equalization preprocessing step. The difference is noticeable but quite small at 2% for the success rate of hard tracks.

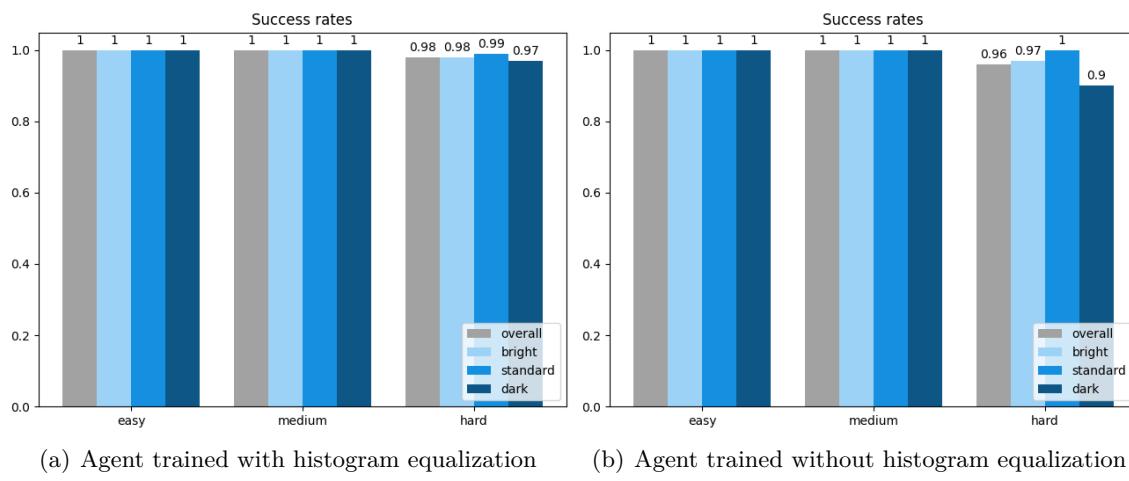


Figure 6.8.: Success rate comparison for an agent that is trained with or without histogram equalization on all light settings

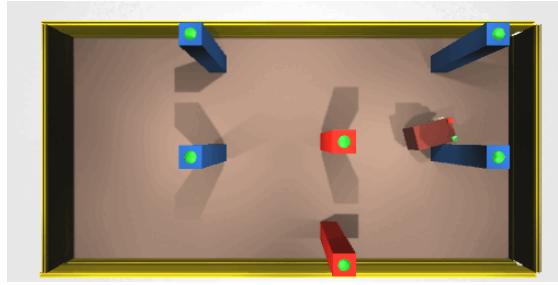


Figure 6.9.: Example of a collision from a successfully completed episode

Conclusion The histogram equalization step contributes strongly to the resiliency to light changes for the constructed agent. It is possible to train a resilient agent without the histogram equalization preprocessing step. However the best performance is achieved when the agent 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.

6.8. Comment on the collision rate

The trained agents are able to complete the parcours very reliably. However the agents do not avoid collisions entirely. The collision rate generally increases for higher difficulty tracks. The collision rates vary quite strongly between the different trained models. The agent trained on the mixed light setting has a collision rate of 70% for the hard tracks with standard light setting 6.6. Whereas the agent trained on hard tracks with standard light setting has a collision rate of only 29% 6.3.

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 6.9.

The reason for the high collision rates is the selected reward function. The agents 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 agent 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 agent does not learn to avoid these collisions entirely.

6.9. Hard Tracks, mixed light settings - the most successful policy

This section summarizes how the most successful policy 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.

Policy Settings The policy was configured to use the three preprocessing steps downsampling, greyscaling and histogram equalization. The policy used a frame stacking of 10. The agent camera image had a resolution of 500×168 which was downsampled to 250×84 . This results in an observation space of shape $[84, 250, 10]$. The duration of steps was fixed to 0.3 seconds. The neural network follows the earlier descriptions exactly 4.2.3.3. 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 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 6.10. The model reached a success rate of 100% for the collected episodes.

6. Finding appropriate hyperparameters for training

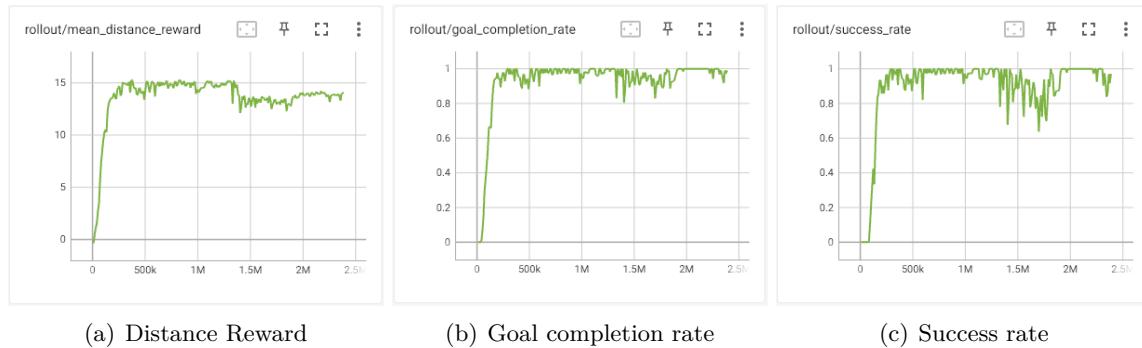


Figure 6.10.: Properties of the collected episodes over time for the most successful model

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 policy decisions are also executed in Python, this includes the preprocessing steps, neural network 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 [17], this way it can be integrated with many existing reinforcement learning libraries and algorithms. The unity simulation acts as a JsonRPC 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. The image data was then decoded in python and converted to a numpy array.

Communication Speed The separation of the simulation and the reinforcement learning 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 reinforcement learning library stable-baselines3 [19] 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 Proximal Policy Optimization algorithm from the stable-baselines3 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 reinforcement learning 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 reinforcement learning library. This way at least the model initialization is reproducible.

7.2. step duration

non blocking steps

TODO explain that a fixed step duration makes the policy transferable to other devices (eval runs with fixed step duration should have the same result for a particular trained policy)

7.3. 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 ?? specifically. The experiments were conducted using the policy that was developed as a result of

8.1. Eval for question 1

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.

8.1.2. Conclusion

The trained policy is able to complete all difficulty levels very reliably.

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

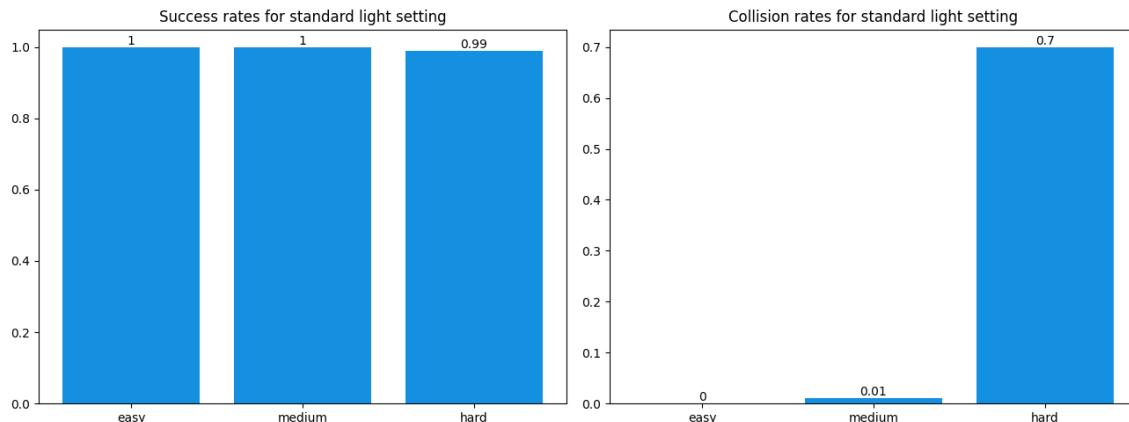


Figure 8.1.: Success and collision rates for standard light setting.

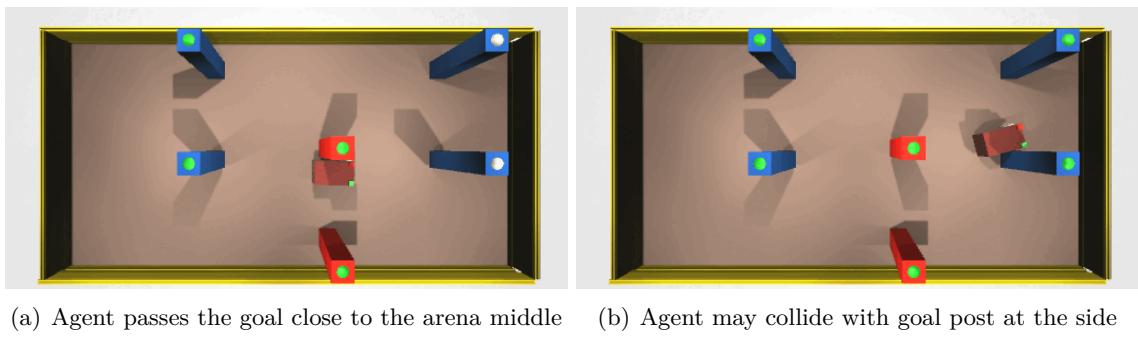


Figure 8.2.: Goal passing behaviour of the trained agent

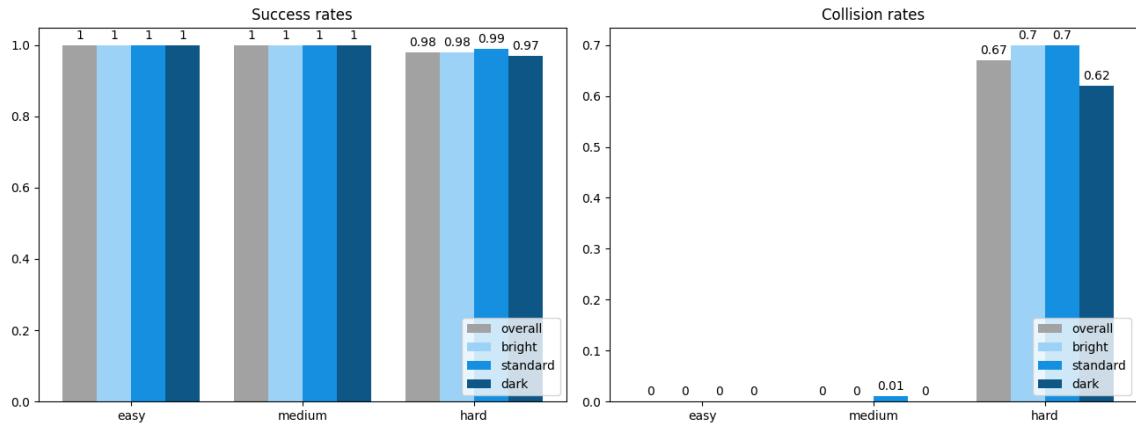


Figure 8.3.: Success and collision rate comparisons for light settings.

8.2. Eval for question 2

The most successful 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 8.3.

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%.

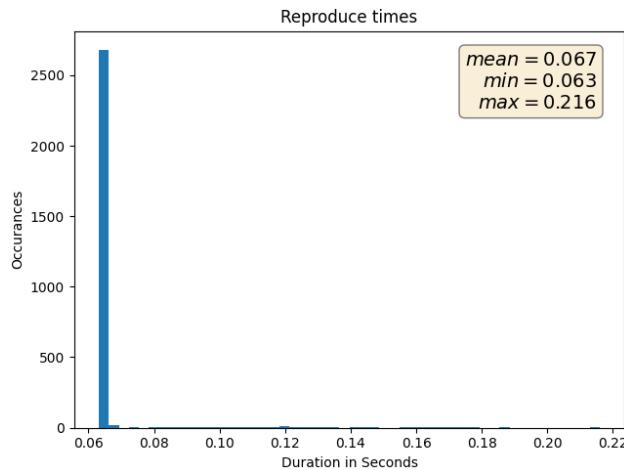


Figure 8.4.: Replay times on jetbot hardware

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 parcours 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 policy 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.

8.3. Eval for Question 3 - Replays

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 be 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 8.4. The policy outputs from the replay were compared to the policy outputs from the recordings 8.5. The differences are caused by the hardware and software differences between the training and the replay environment.

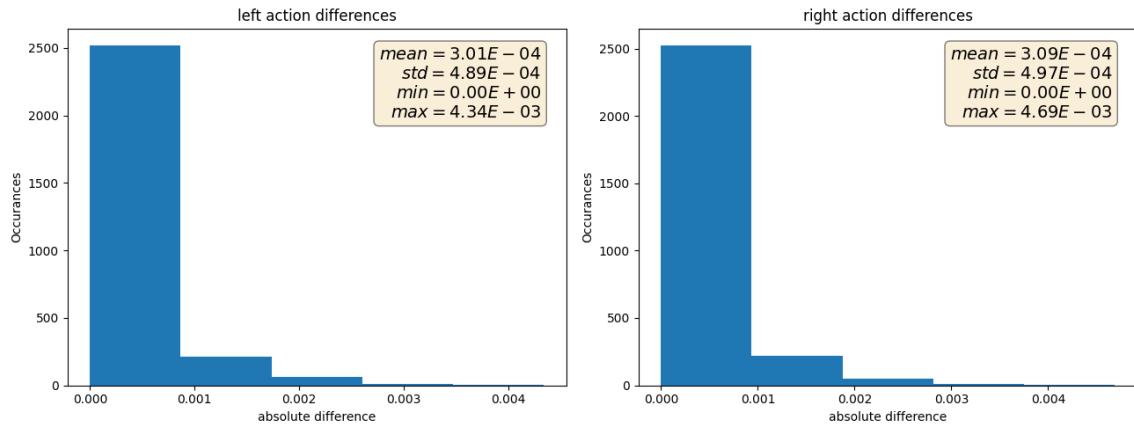


Figure 8.5.: Differences in policy outputs between recordings and replays on jetbot hardware

8.3.1. Experiment Results

Replay times The replay times for the recordings on jetbot hardware are shown in figure 8.4. 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 nvidia 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 8.5. The outputs were reproduced very closely. The maximum difference was $4.69e - 03$. This difference is negligible 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.

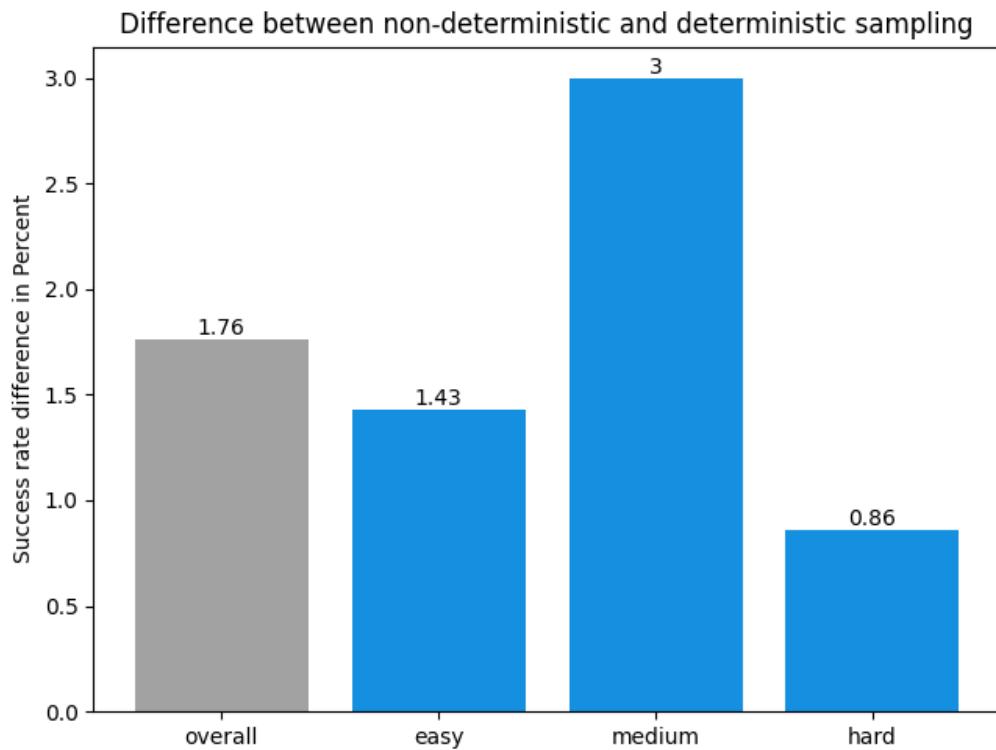


Figure 8.6.: Results of the deterministic check across all evaluations during the experimentation phase

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 deterministic 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 8.6.

Discussion The test indicates that the non-deterministic sampling mode performs 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.

8.4.2. Test identical start conditions

The environment is not 100% deterministic. This is due to the environment implementation in Unity. The environment uses physics simulations which are not deterministic. In addition the agent's actions are sampled non-deterministically during the evaluation. This means that the same 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 trained agent is placed in the same start conditions for 100 episodes and the trajectories are compared. The starting conditions entail the selected track, light condition and the agent's starting rotation.

Results identical start conditions result in over 90% identical trajectories for hard tracks

Discussion -> nice

8.4.3. Fresh Observations Test

The policy was trained to not use fresh observations 4.1.1 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.

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 goes from 98% to 68% 8.7.

The evaluations using fresh observations also take longer. The Basic Evaluation Algorithm's per call time for the fresh observations is 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.

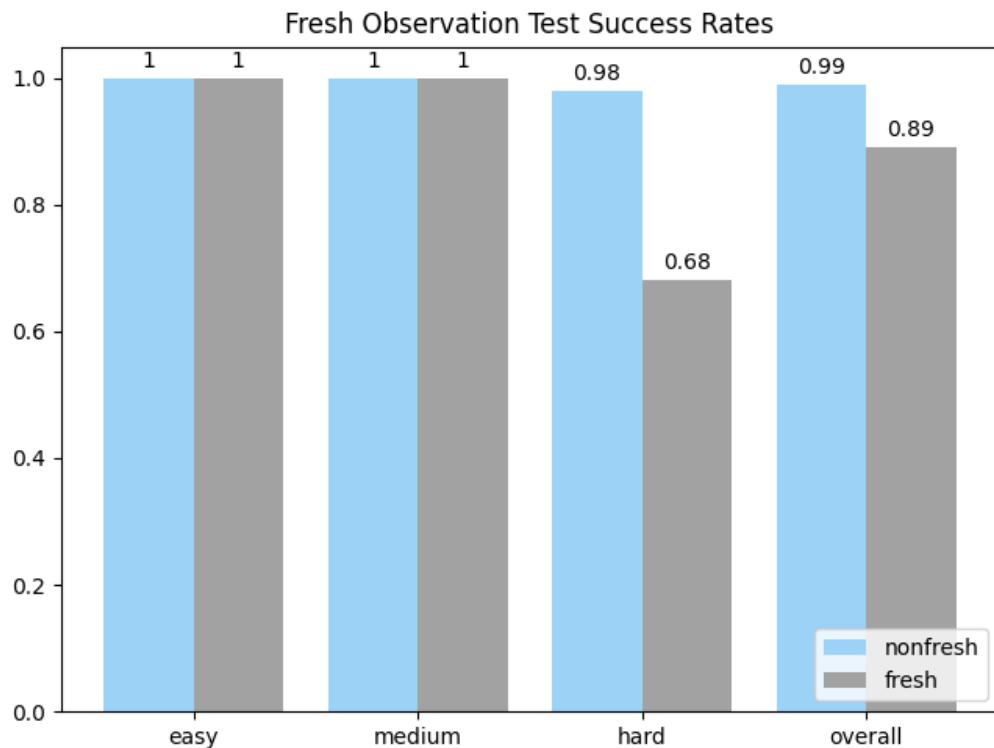


Figure 8.7.: Success rates for policy evaluation using fresh and non-fresh observations

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

The policy was trained using the DifferentialJetBot shown in figure 4.6. 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 8.8. The collision rates increase for higher difficulties with 0%, 4% and 94%. The success and collision rates are slightly worse than on 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

8. Results

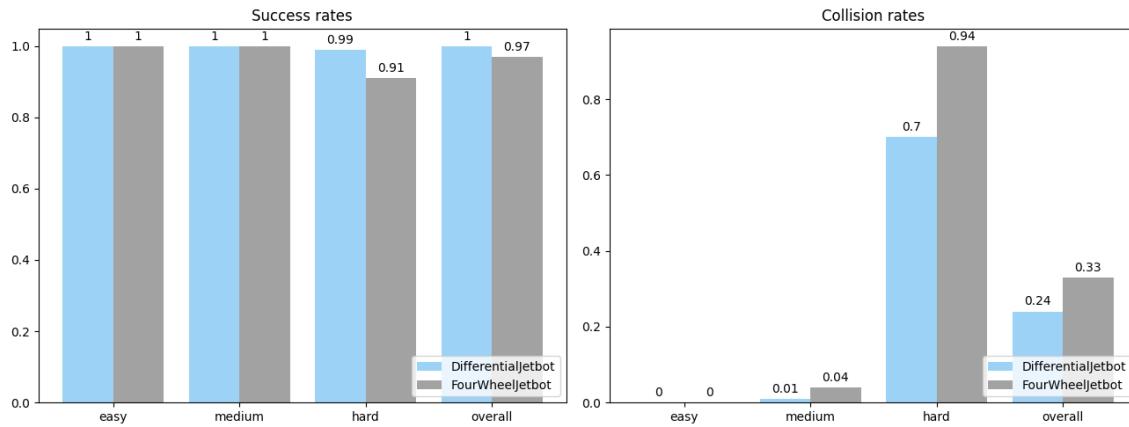


Figure 8.8.: Evaluation of the DifferentialJetBot policy with both jetbot versions

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 transferred to the Four-WheelJetBot with only a slight decrease in performance. The policy does not have to be retrained in this case.

9. Conclusion

TODO conclusion

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] Dean A. Pomerleau. “ALVINN, an autonomous land vehicle in a neural network”. In: 2015. URL: <https://api.semanticscholar.org/CorpusID:18420840>.
- [8] 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).
- [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] Jonas König. “Model training of a simulated self-driving vehicle using an evolution-based neural network approach”. Bachelor’s Thesis. Universität Leipzig, 2022.
- [11] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG].
- [12] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG].
- [13] 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>.
- [14] 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>.
- [15] 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).
- [16] Alexey Dosovitskiy et al. “CARLA: An Open Urban Driving Simulator”. In: *Proceedings of the 1st Annual Conference on Robot Learning*. 2017, pp. 1–16.

- [17] Mark Towers et al. *Gymnasium*. Mar. 2023. DOI: [10.5281/zenodo.8127026](https://doi.org/10.5281/zenodo.8127026). URL: <https://zenodo.org/record/8127025> (visited on 07/08/2023).
- [18] Pablo Samuel Castro et al. “Dopamine: A Research Framework for Deep Reinforcement Learning”. In: (2018). URL: <http://arxiv.org/abs/1812.06110>.
- [19] 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>.
- [20] Unity Technologies. *Unity Engine*. 2023. URL: <https://unity.com>.
- [21] 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](https://doi.org/10.1109/IROS.2012.6386109).
- [22] 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.
- [23] Mariusz Bojarski et al. “End to End Learning for Self-Driving Cars”. In: (Apr. 2016).
- [24] Tesla YouTube. *Tesla Autonomy Day*. 2019. URL: <https://www.youtube.com/watch?v=Ucp0TTmvqOE> (visited on 05/23/2024).
- [25] 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](https://doi.org/10.1016/j.trc.2022.103987). URL: <http://dx.doi.org/10.1016/j.trc.2022.103987>.
- [26] Hugh Perkins. *Peaceful Pie, Connect Python with Unity for reinforcement learning!* 2023. URL: <https://github.com/hughperkins/peaceful-pie> (visited on 10/23/2023).
- [27] 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>.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit mit dem Thema:

„End-to-End Reinforcement Learning Training of a Convolutional Neural Network to achieve an autonomous driving agent resilient to light changes“

selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann. Ich versichere, dass das elektronische Exemplar mit den gedruckten Exemplaren übereinstimmt.

Leipzig, den xx.xx.xxxx

GEORG SCHNEEBERGER

A. Appendix

A. Code Repository

The code repository is publically available on GitHub:

[carsim-rl-cnn](#)

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

B. Most successful model

The most successful model is publically available on Huggingface:

[carsim-rl-cnn](#)

[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 available on GitHub: [hardDistanceMixedLight.yaml](#)

D. Example Video Files

Video Files are available on Github: example_videos

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

E. Pseudocode

TODO pseudocode für eval_model_track?

Algorithm E.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_successful_episodes  $\leftarrow 0
    RolloutBuffer.RESET(trainingMapType, trainingLightSetting)
    Env.RESET()
    while RolloutBuffer.NOTFULL()
        do  $\left\{ \begin{array}{l} obs \leftarrow Env.GETOBSERVATION() \\ action \leftarrow Model.GETACTION(obs) \\ reward \leftarrow Env.STEP(action) \\ num\_steps \leftarrow num\_steps + 1 \\ ADDTOROLLOUTBUFFER(obs, action, reward) \\ \text{if } Env.ISFINISHED() \\ \quad \text{then } \left\{ \begin{array}{l} num\_episodes \leftarrow num\_episodes + 1 \\ \text{if } Env.FINISHEDSUCCESSFULLY() \\ \quad \text{then } \left\{ \begin{array}{l} num\_successful\_episodes \leftarrow num\_successful\_episodes + 1 \\ Env.RESET(trainingMapType, trainingLightSetting) \end{array} \right. \\ rollout\_success\_rate \leftarrow \frac{num\_successful\_episodes}{num\_episodes} \\ \text{if } rollout\_success\_rate \geq best\_rollout\_success\_rate \\ \quad \text{then } \left\{ \begin{array}{l} best\_success\_rate \leftarrow rollout\_success\_rate \\ Model.SAVETOFILE() \\ best\_model \leftarrow Model \end{array} \right. \end{array} \right. \end{array} \right\}$ 
    return (num_steps)$$$ 
```

A. Appendix

Algorithm E.2: TRAIN MODEL()

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

```

procedure TRAINMODEL()
    amount_of_batches  $\leftarrow \frac{\text{rollout\_buffer\_size}}{\text{batch\_size}}$ 
    for  $i \leftarrow 0$  to  $n\_epochs$ 
        do  $\left\{ \begin{array}{l} \text{RolloutBuffer.SHUFFLE()} \\ \text{RolloutBuffer.CREATEBATCHES(batch\_size)} \\ \text{for } m \leftarrow 0 \text{ to } amount\_of\_batches \\ \quad \text{do } \left\{ \begin{array}{l} \text{batch} \leftarrow \text{RolloutBuffer.GETBATCH}(m) \\ \text{loss} \leftarrow \text{COMPUTELOSS}(batch) \\ \text{Model.BACKPROPAGATE}(loss) \\ \text{Optimizer.STEP()} \end{array} \right. \end{array} \right.$ 
```

F. Neural Network Architecture

G. Eval Model Track

Algorithm G.1: GENERATE MAP AND ROTATION COMBINATIONS()

```

procedure GENERATE__MAP__AND__ROTATIONS( $difficulty, n\_eval\_episodes, env$ )
    rotationMode  $\leftarrow env.\text{GETSPAWNMODE}()$ 
    rotation_range_min, rotation_range_max  $\leftarrow \text{Spawn.GETROTATIONRANGE}(rotationMode)$ 
    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  $\left\{ \begin{array}{l} step \leftarrow range\_width/(n\_eval\_episodes - 1) \\ \text{for } i \leftarrow 0 \text{ to } n\_eval\_episodes - 1 \\ \quad \text{do } \left\{ rotations.\text{APPEND}(rotation\_range\_min + i * step) \end{array} \right. \right.$ 
    track_numbers  $\leftarrow MapType.\text{GETALLTRACKNUMBERSOFFDIFFICULTY}(difficulty)$ 
    tracks  $\leftarrow []$ 
    for  $i \leftarrow 0$  to  $n\_eval\_episodes - 1$ 
        do  $\left\{ tracks.\text{APPEND}(i \bmod \text{LEN}(track\_numbers)) \right. \right.$ 
    combinations  $\leftarrow []$ 
    for  $i \leftarrow 0$  to  $n\_eval\_episodes - 1$ 
        do  $\left\{ combinations.\text{APPEND}((tracks[i], rotations[i])) \right. \right.$ 
    return (combinations)

```

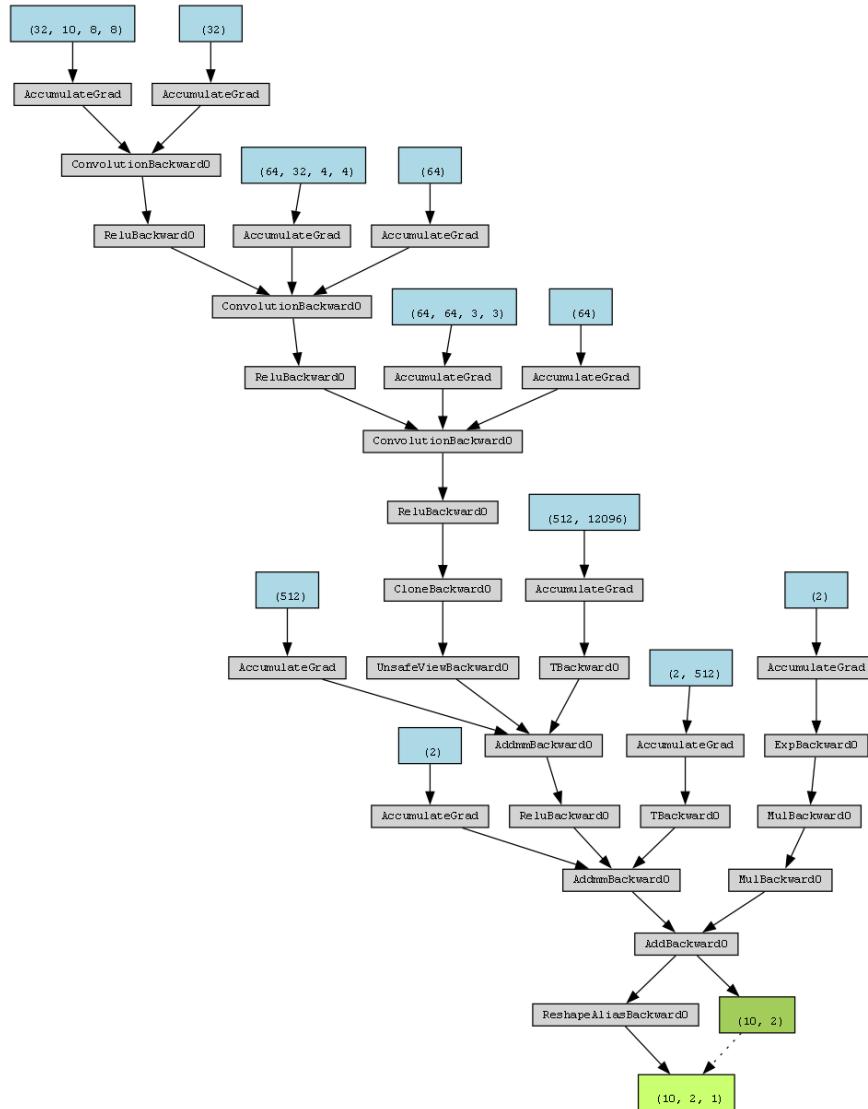


Figure A.1.: Neural Network Architecture Action Head

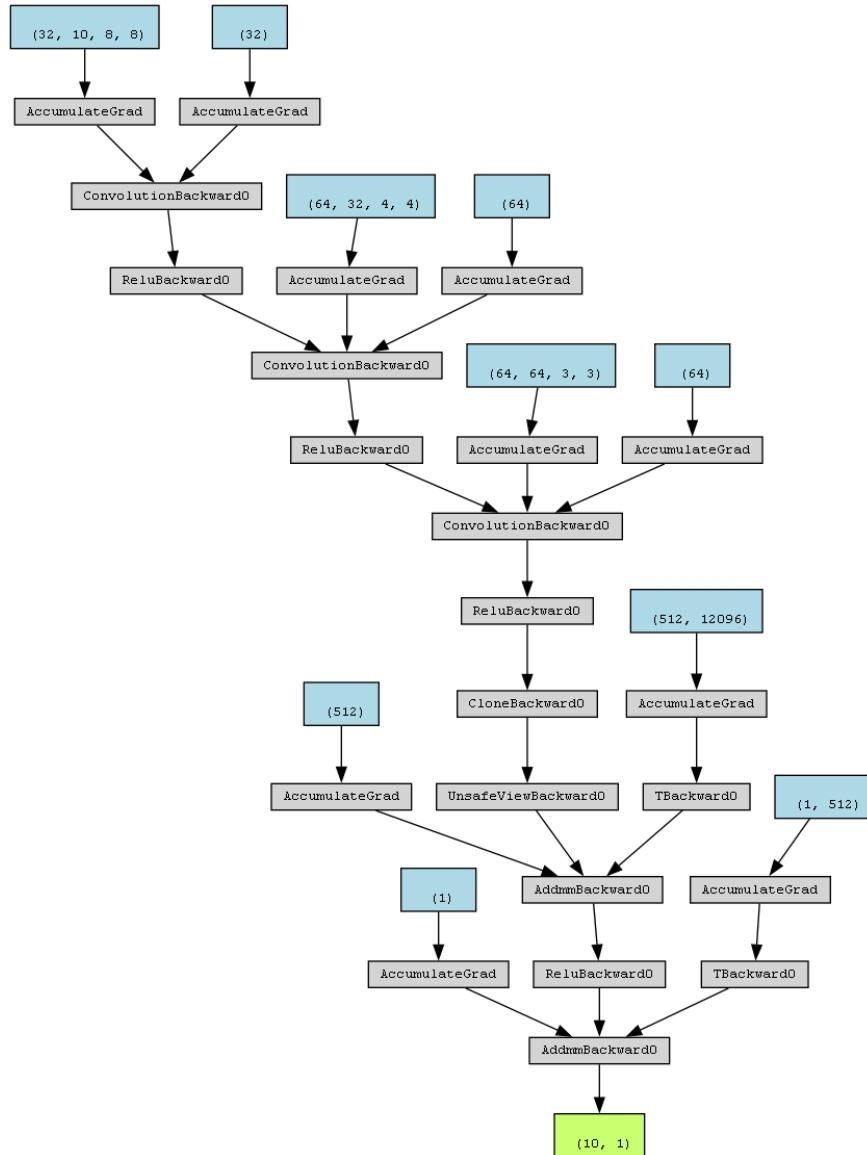


Figure A.2.: Neural Network Architecture Value Head

H. Replay on JetBot

H.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 available in the replays_on_jetbot.md file in the code repository.

Algorithm H.1: RECORD EPISODE()

```
comment: Record episode

procedure RECORD_EPISODE(policy, env, directory)
    env.RESET()
    sampled_actions  $\leftarrow \emptyset$ 
    infer_obsstrings  $\leftarrow \emptyset$ 
    step_obsstrings  $\leftarrow \emptyset$ 
    done  $\leftarrow \text{false}$ 
    while !done
        do
            obs, obsstring  $\leftarrow \text{env.GETOBSERVATION}()$ 
            action  $\leftarrow \text{policy.INFER}(\textit{obs})$ 
            step_obsstring, done  $\leftarrow \text{env.STEP}(\textit{action})$ 
            infer_obsstrings.APPEND(obsstring)
            step_obsstrings.APPEND(step_obsstring)
            sampaled_actions.APPEND(action)
        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(sampaled_actions)
            do SAVETOFILE(sampaled_actions[i], directory + "/sampaled_action" + i + ".npy")
```

Algorithm H.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  $\leftarrow$  LOADRECORDED ACTIONS(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)
```