HAW
HAMBURG

BACHELOR THESIS
Michael Babic

Analysis and Evaluation of Reinforcement Learning Algorithms for a Continuous Control Problem

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Michael Babic

Analysis and Evaluation of Reinforcement Learning Algorithms for a Continuous Control Problem

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

Supervisor: Prof. Dr. Michael Neitzke
Supervisor: Prof. Dr. Martin Becke

Submitted on: 28. Juli 2022

Michael Babic

Thema der Arbeit

Analysis and Evaluation of Reinforcement Learning Algorithms for a Continuous Control Problem

Stichworte

Kontinuierliche Kontrolle, Agenten, Lernende Agenten, Verstärktes Lernen, Machinelles Lernen, Soft Actor Critic, Proximal Policy Optimization, Twin Delayed Deep Deterministic Policy Gradient, Truncated Quantile Critics, Trust Region Policy Optimization, Robot Operating System, Hyperparameter Optimization, Optuna, Stable Baselines 3, OpenAI

Kurzzusammenfassung

Die große Vielfalt an an Reinforcement Learning Algorithmen macht es schwer zu bestimmen, welcher Algorithmus für welche Aufgabe verwendet werden soll. Die wissenschaftlichen Arbeiten, die solche Algorithmen präsentieren, enthalten oft wiedersprüchliche Ergebnisse und machen es dadurch noch schwerer zu verstehen, ob die Erweiterungen der grundlegenden Algorithmen eine Leistungsverbesserung aufweisen. In dieser Arbeit wird ein Ansatz vorgestellt, um eine kontinuierliche Kontrollaufgabe für einen Agenten zu analysieren, eine Gruppe von Algorithmen auf der Grundlage der Merkmale auszuwählen und Sie strukturiert für das Problem zu konfigurieren. Die Konzepte der ausgewählten Algorithmen werden vorgestellt und es wird gezeigt, wie Sie die grundlegenden Reinforcement Learning Algorithmen verbessern, gefolgt von einer Vorhersage Ihrer Leistung in der erwähnten kontinuierlichen Kontrollaufgabe. Weitergehend präsentiert diese Arbeit einen Ansatz zum Finden von Parametern und einer Analyse der Algorithmen, wie schnell funktionierende Parameter gefunden werden konnten und inwiefern welche Veränderungen die Leistung am stärksten beeinflusst hat. Die Abweichung der einzelnen Durchläufen wird durch eine Reevaluierung der besten Parameter Konfigurationen für die Algorithmen reduziert und die stabilsten ausgewählt. Abschließend wird ein detaillierter Einblick in das Verhalten während des Trainings und der Evaluation mit den gewählten Parametern präsentiert, wie die Algorithmen lernen und die Umgebung erkunden, um die Vorhersagen über die Leistung der Algorithmen zu beantworten.

Michael Babic

Title of Thesis

Analysis and Evaluation of Reinforcement Learning Algorithms for a Continuous Control Problem

Keywords

Continuous Control, Agents, Learning Agents, Reinforcement Learning, Machine Learning, Soft Actor Critic, Proximal Policy Optimization, Twin Delayed Deep Deterministic Policy Gradient, Truncated Quantile Critics, Trust Region Policy Optimization, Robot Operating System, Hyperparameter Optimization, Optuna, Stable Baselines 3, OpenAI

Abstract

Due to the broad variety of Reinforcement Learning algorithms, it is difficult to determine which one to use for what task. Papers that present said algorithms often claim contradictory results which worsens this problem and makes it harder to understand if their extensions of the base algorithms bring an overall improvement in performance. This work presents an approach to analyze a custom created continuous control task, pick an algorithm or a group of algorithms based on found characteristics and provide a structure to configure the algorithms parameters. The concepts of chosen algorithms are shown and how they claim to improve on the basic Reinforcement Learning algorithms, followed by predictions about their performance on the environment. A structured way for finding suitable parameters for the algorithms is presented, and the algorithms are further analyzed, how fast a good set of parameters was found and what changes influenced the performance the most. The run to run variance is reduced by reevaluating the best parameter sets found for each algorithm multiple times and picking the most stable ones. Finally, an in depth view of the algorithms behavior while training and evaluating with found parameter sets is presented, how the algorithms learn and explore, to answer the predictions made about their performance.

Contents

List of Figures

1 Introduction

With the growth in machine learning research, a lot of improvements were made in Reinforcement Learning. It is a field of machine learning that is concerned with solving sequential decision-making problems, where an agent learns to solve complex tasks like playing chess or controlling a robot.

The agent acts in an environment based on observations of it and learns from feedback signals that he receives for his actions. Through the course of multiple interactions, he improves his behavior in an effort to optimize the signals he receives.

Reinforcement Learning started with two basic ideas, learning through trial and error, that originated from animal learning behavior, and a research field about optimal control through dynamic programming, maximizing or minimizing a feedback signal from a system. These two, historically separated ideas, were followed by another idea that connected them, called temporal difference learning. It describes learning to be influenced by changes in temporally sequential predictions. This connection resulted in modern reinforcement learning, where an agent learns through said predictions to optimize a feedback signal from an environment while learning through trial and error [32, ch. 1.7].

Basic Reinforcement Learning algorithms, e.g., *Q-Learning* or *SARSA*, have a big issue. They have to store a value for each action in each possible observation, that represents how good an action is, which made them not applicable in many interesting tasks. Especially, learning from images was unthinkable, since the combination of pixels in reasonably sized images was too large to store. Even if it could be stored, an agent has to visit each observation to learn how his action impacts the environment at that state, making learning unfeasible.

DeepMind, an AI research company, presented the first deep learning model that used *Q-Learning* on image data. They created an agent that learned to play Atari 2600 games using Neural Networks as a function approximation for the stored values [20]. With many more so-called deep Reinforcement Learning algorithms surfacing afterwards, the company OpenAI, known for AI research and deployment, published a framework called baselines that includes stable implementations of the more popular and influential algorithms, as well as educational resources containing theories of said algorithms [3].

1.1 Motivation

Many real world applications are continuous control tasks with complex environment states, like driving an autonomous car or controlling a robot through their sensory representation of their surroundings, where an agent has to act very precisely to achieve good performance. Many newer Reinforcement Learning algorithms are designed for continuous control tasks, but the amount of algorithms makes it difficult to decide which one to use. Papers of some algorithms even claim contradictory performance, e.g., the *SAC* publication [13] claims that its algorithm outperforms the algorithm presented in the *TD3* publication [12] on almost every environment tested, while the second one claims the opposite. This problem can come from many things, like how they implemented the algorithm, the environments stochasticity, which random seeds they used, what parameters they used as well as the general randomness of training Neural Networks, as described in a case study about algorithm implementation variance [11].

The motivation of this work is to analyze continuous control Reinforcement Learning algorithms through their performance, stability, behavior, what their concepts are and to what extent these concepts matter for a continuous control task. We use a stable implementation framework that extends on OpenAI's baselines, to reduce the mentioned variance in the algorithms performances. Due to the contradictions in performances on the popular benchmark suits, we want to create a difficult custom continuous control task that we use for the analysis and evaluation, where the agent has to simultaneously control a robots linear and angular accelerations through images as his environment observation. Additionally, we want to present a structured way for deciding on an algorithm and searching for a working configuration of parameters.

1.2 Structure of the thesis

In chapter 2, theoretical foundations of reinforcement learning and frameworks used for the analysis and environment creation are presented.

In chapter 3, we describe our created environment, how we trained Reinforcement Learning agents on it and validated it to be suitable for the further analysis.

In chapter 4, the analysis of environment characteristics and Reinforcement Learning algorithms is done, providing a guideline to pick an algorithm through said characteristics and algorithm concepts. We end the chapter by making predictions about the performance of the analyzed algorithms on our environment.

In chapter 5, our experiments and their results are shown, giving instructions on how to configure algorithm parameters for an environment and presenting plots of the agents learning behavior and movement in the environment.

In chapter 6, we discuss our structured approach, analyze the results and make a conclusion about our predictions made in chapter 4.

The final chapter 7 provides a summary of the thesis and ends with related and future work.

2 Basics

The goal of this chapter is to provide a basic understanding of reinforcement learning and their basic algorithms, which are the foundation of every following algorithm discussed and analyzed in this work. Secondly we present frameworks used to build an environment to run reinforcement learning algorithms on. Finally, we present an optimization framework to find optimal parameters in a given time frame for each algorithm on the build environment.

2.1 Reinforcement Learning

Reinforcement Learning is learning from interaction [32, ch. 1]. An Agent acts in an Environment, which provides him with a feedback signal about his behavior and the environments current representation, called an observation or state. The feedback signal is a scalar and is called the reward. An agent gets a starting state and acts based on it, getting a reward and a following state afterwards. He then acts again based on the new state, and receives a new reward and state. This loop continues until the environment terminates, e.g. reaching a goal. A run till termination is called an episode. A discount factor gamma is introduced for the agent, that weighs down each following reward per step taken. A discount factor getting smaller makes the agent more short-sighted, since future rewards are getting less valuable. It makes the agent learn to collect good rewards faster, e.g. getting faster to a goal. The agent's objective is to maximize the discounted long term reward [32, ch. 1.3].

2.1.1 Markov Decision Process

To be able to make a decision through a state, the environment has to be a Markov Decision Process. It is a mathematical framework, which defines the feedback loop of an agent acting in an environment, getting a state and a reward after acting:

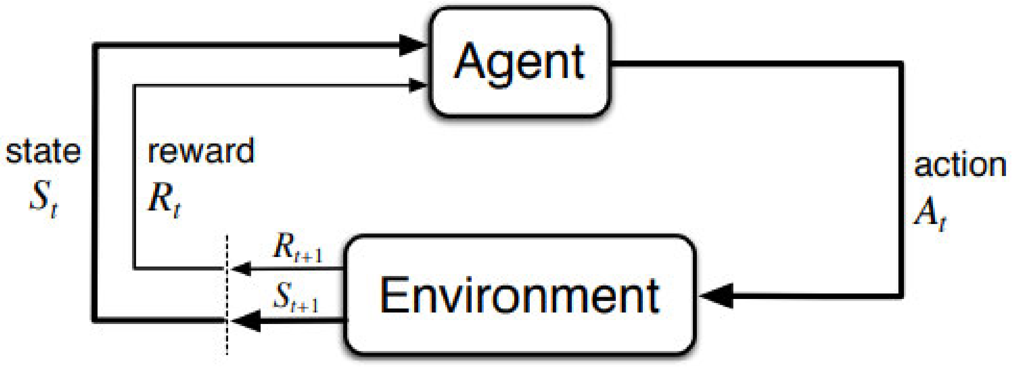


Figure 2.1: The agent-environment interaction in a Markov decision process [32, ch. 3.1]

Figure 2.1 shows the agent and environment loop. The agent acts with his action A on a step *t*, and the environment replies with a following state S and R on the following step *t+1*. The following equation (2.1) describes the dynamics of an MDP:

$$p(s', r|s, a) \doteq Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{2.1}$$

It shows the probability of transitioning to state *s'* and getting a reward *r* by taking action *a* in state *s*. The probability of a following state depends only on the action and state taken, not on the history of states [32, ch. 3.1].

2.1.2 Value Functions and Policies

An agent wants to learn a policy which maximizes his discounted long term reward. The policy π is the agent's behavior policy, that maps a state to a probability for each possible action in that state.

There are two major ways of learning a good policy and a combined way: Learning a value representation of the states in the environment, learning a policy directly or learning both.

The algorithms presented in this chapter are the foundations of the algorithms picked in chapter 4.

Value based Reinforcement Learning

In Value based RL, the agent learns a function that assign a value to a state v_π*(s)*, called the state value function, or to a state action pair q_π*(s,a)*, called the q function under the agent's policy π. The value of a state is the expected reward for the agent if he transitions into that state and follows his policy (or for the q function, the expected reward for taking the action in the current state and following his policy). Transitioning into a state can have a negative reward, but from there on the agent could get a lot of positive rewards, so the immediate reward for going into that state may be low, but its value high [32, ch. 1.3]. This makes it a sequential decision-making problem.

With a given value function, an agent's policy can be to just act greedy in respect to the value function. This means, that he always takes the action in each state that transitions into a state with the highest value from the current value function. In an environment where it is not clear which action leads to which state, the q function is used, since it directly assigns a value to each action in a state. The agent would start in a state s and follow his current q function, getting a value for each action in his current state, picking the highest valued one and repeats until termination. Value based agents do not need to learn a policy directly, they only learn the state value or q function and derive the policy from it.

To act optimally in an environment through value functions, an agent seeks to learn one of the following two functions:

$$\begin{aligned} q_*(s,a) &= E[R_{t+1} + \gamma max_{a'} q^*(S_{t+1}, a') | S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r|s, a)[r + \gamma max_{a'} q_*(s', a')] \end{aligned} \tag{2.2}$$

$$\begin{aligned} v_*(s) &= max_a E[R_{t+1} + \gamma v^*(S_{t+1}) | S_t = s, A_t = a] \\ &= max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')] \end{aligned} \tag{2.3}$$

Equation (2.2) and equation (2.3) are the bellman optimality equations and assign the maximum expected return on a state or a state-action pair. Acting greedy in respect to the optimal value function results in having an optimal policy [32, ch. 3.6]. The value of a state-action pair is the possibility of a following state and reward occurring after taking the action times the reward with the discounted q value of the following state, choosing the optimal action in it. This gets summed for each possible following state and reward.

Learning and Approximation

Since the environment dynamics are not given in most environments, the $v*$ or $q*$ function have to be learned by interacting with the environment. A common way of learning the $q*$ function is the *Q-Learning* Algorithm, where the values are updated towards the optimal function by taking a sample in the environment:

$$\begin{aligned} \delta_t = R_{t+1} + \gamma max_a Q(S_{t+1}, a) - Q(S_t, A_t) \\ Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\delta_t \end{aligned} \tag{2.4}$$

Equation (2.4) [32, ch. 6.5] shows how *Q-Learning* updates Q values. The Q value of a state S and action A on timestamp t gets updated by the summation of the old Q value and the temporal difference error (TD error) times a learning rate *alpha*, that indicates how much an update changes the current value. The TD error is the reward the agent gets from the environment plus the difference of the Q value of the next state, taking the maximal action, and the value of the current state. It is called bootstrapping, since we update the current Q value on the estimate of the Q values for the current and next state [32, ch. 6.5].

For this algorithm, we need to store an entry for each state and action pair, which is not feasible for bigger state spaces. An environment with $30 * 30$ mono images as its observation, with a byte per pixel, would already have 256^{900} different states. Function approximation is needed instead of the tabular case, for example a neural network, which is often used when having an image as a state. Another benefit of using function approximation is to generalize in the state space, learning behavior in not visited states, so that not each state has to be visited at least once to have a value estimate for it [32, ch. 9]. Instead of a table that returns a Q value by indexing a state and action, a neural network gets a state as an input and outputs a Q value for each possible action.

Policy Gradient based Reinforcement Learning

In Policy Gradient based Reinforcement Learning, the agent learns a policy directly, which approximates the value function through a parameter vector that has to be differentiable with respect to its parameters [32, ch. 9]:

$$\pi(a|s,\theta) = Pr\{A_t = a|S_t = s, \theta_t = \theta\} \tag{2.5}$$

Equation (2.5) [32, ch. 13] shows that the probability of an action a in a state s is determined by the parameter vector θ. Given a θ and a state, the function returns a probability distribution for the actions in that state. An agent samples an action directly from the policy. In this work, θ represents a Neural Network.

The *REINFORCE* algorithm updates the policy towards the optimal policy:

$$\theta \leftarrow \theta + \alpha G_t \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)} \tag{2.6}$$

Equation (2.6) updates after each episode for each step t taken [32, ch. 13.4]. G_t is the discounted sum of rewards until the current step. G_t is an unbiased value of v_π, since there is no bootstrapping and the true returns experienced are used. G_t gets multiplied by a step-size and a vector of the gradient of the probability of taking the action a under the current policy divided by the probability of the action. This vector points towards the parameter space that increases the probability of the action taken. The intuition is to update the action towards the direction of the vector by the reward experienced. It increases the probabilities of actions that had a more positive reward, which decreases the probabilities of actions that had a less positive reward [32, ch. 13.3].

Actor Critics

Actor critic methods combine Value based Reinforcement Learning with Policy Gradient based Reinforcement Learning. The actor is the behavior policy, that the agent uses to sample an action from and the critic is the value function, that the agent uses to approximate the expected return of a state. The basic idea is to have an actor pick an action and a critic that informs the actor about how good the action was.

An extension of the *REINFORCE* algorithm replaces the episodic returns using the critic, called *A2C*, is shown in the following equation (2.7) [32, ch. 13.5]:

$$\begin{aligned} \delta_t = R_{t+1} + \gamma v(S_{t+1}, w_t) - v(S_t, w_t) \\ w \leftarrow w_t + \alpha^w \delta \nabla v(S_t, w_t) \\ \theta \leftarrow \theta_t + \alpha^\delta \delta_t \frac{\nabla \pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)} \end{aligned} \tag{2.7}$$

The exponents on the step size parameters' α indicate that a different step size can be used for the critic and actor update. The TD error gets calculated using the critic that approximates the value function. This specific version of the TD error is called the advantage, hence the name *A2C*, meaning advantage actor critic. It adds the reward sampled with the next state value and subtracts the current state value, representing how our action taken compares to the mean value of the current state. It shows how much more reward the sampled action has received over the mean value, what we would expect to receive. The critic has an added differentiable parameter vector w, in our case a neural network, for function approximation. If δ_t was positive, we update our value function to increase its value for the current state, and if it was negative, we decrease it. The actor update, the update of the policy's parameter vector θ, now includes delta, which enables updates on each step instead of sampling a whole episode and using G_t. As in *REINFORCE*, we push the policy towards picking actions that maximize the expected reward, but through the critics estimate of the expected reward.

Another approach for actor critics is to approximate the $q*$ function and have the actor learn to exploit the Q function, called Deterministic Actor Critic (*DPG*), as shown in equation (2.8) [31]:

$$\begin{aligned} \delta_t = R_{t+1} + \gamma Q(S_{t+1}, \mu(S_{t+1}, \theta)) - Q(S_t, A_t, w_t) \\ w \leftarrow w_t + \alpha^w \delta_t \nabla_w Q(S_t, A_t, w_t) \\ \theta \leftarrow \theta_t + \alpha^\theta \nabla_\theta \mu(S_t, \theta) \nabla_{\mu(S_{t+1}, \theta)} Q(S_t, A_t, w_t) \end{aligned} \tag{2.8}$$

The TD error gets calculated by the Q function. The difference between the *Q-Learning* approach is that the actor approximates the max action, instead of choosing the max action through the Q function directly. The value functions parameters w will get updated as in the *A2C* algorithm, but with the Q function instead of the value function. The policy here is depicted as μ and gets updated to pick an action that maximizes the current Q function. In this algorithm, μ is deterministic and returns an action given a state, instead of a distribution over actions, as shown in equation (2.9).

$$\mu(s, \theta) \approx a^*(s) = \arg\max_a Q^*(s, a) \tag{2.9}$$

The idea is to learn the q^* function and let an actor learn to exploit it.

2.1.3 Continuous Actions

Policy gradient based algorithms, and therefore actor-critic algorithms, can be extended for continuous control. In continuous control tasks, the actor returns not a chosen action from a set of actions, but a continuous value for an action, e.g., the velocity of a robot. In this work, the robots actions are the angular and linear velocity, that the agent sets through two continuous values. The *REINFORCE* algorithms and its extensions like *A2C* do this by returning a mean and a standard deviation for each action, instead of a probability for discrete actions.

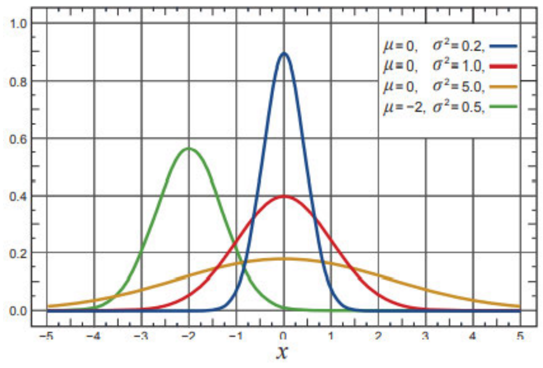


Figure 2.2: Continues Actions visualized [32, ch. 13.7]

Figure 2.2 shows four actions with their distribution, given by the outputs of an actor for each action. An action gets sampled from the distribution for each pair of the mean and standard deviation. In this work, the agent would return the mean and standard deviation pair for the angular and linear velocity.

The problem with *Q-Learning* is that we have to calculate the action that maximizes the Q function in the next state. Normally, one can just calculate it for each action by the function *Q(s,a)*, but with a continuous action, we can't exhaustively evaluate the Q function, since an action has no discrete set of values. *DPG* addresses this problem by having an actor directly approximate the max action and returns a continuous value directly [2].

2.1.4 On Policy and Off Policy

On-Policy means sampling data from the agent's policy and only using that to update the policy. After the update, the sampled data can not be reused, since the agent's policy changed and will now behave differently from the one that gathered the data. *REINFORCE* and *A2C* are *On-Policy* for example, since they update the current policy based on the data they collected and use the new policy to collect new data. They have no separate policy to collect data.

Off-Policy means updating the agent's policy by data gathered from a different policy [32, ch. 5.4]. This is used in *Q-Learning* and *DPG* for example, since the Q value of a state gets updated by the maximum valued action in the next state. It allows sampling data with a random policy and update the state action estimates by using the greedy policy in the Q value updates. This enables reusing old data for updating the trained policy.

2.1.5 Exploration and Exploitation

Exploring is a big part of Reinforcement Learning algorithms. To learn a good policy or value representation of the states, an agent must explore the environment and sample from a lot of different states. Exploitation means to maximize the expected reward for the agent based on his current knowledge [32, ch. 1.1]. In the *On-Policy* case, the learned policy is stochastic and the agent explores naturally, since he samples from the distribution over the actions.

In the *Off-Policy* case, the policy to sample data can be a randomized policy to improve the exploration. *Q-Learning* introduces a random variable ϵ, that makes the agent pick a random action with a probability of ϵ and otherwise picks the action that maximizes the current q function. *DPG* is used in continuous action spaces and adds a random noise to the value for exploration [31].

2.2 Robot Operating System

The Robot Operating System (*ROS*) is an open source development kit for robotics applications and offers a broad variety of software, e.g., many Simultaneously Localization and Mapping (SLAM) algorithms [25]. In this work, we use a SLAM algorithm called Gmapping, that creates a 2D map from a LiDAR source by detecting walls and adding them to a map [23]. *ROS* is a distributed system where multiple nodes communicate through topics by publishing and subscribing or through direct function calls, called services. A *ROS-Master* manages the connection between the nodes and acts as a name- and parameter server for other nodes to look up available services, topics, nodes and globally available parameters. When a node subscribes a topic, the *ROS-Master* looks for another node publishing on the same topic and signals both to create a TCP connection between them. The publishing node sends the data through that TCP connection to the subscribing node. Services are distributed similarly, a node registers a function and if another node wants to call that function, they create a TCP connection and the function gets called [24]. *ROS* includes a way to start multiple nodes at once through an XML file with *RosLaunch*, which is used in this work to start parts of our environment. Using *ROS*, one can replace nodes in the distributed systems with other nodes that provide the same topics and services. This opens up the possibility to exchange a simulation with a real world robot, as long as the data provided by the nodes is from the same type.

2.3 Unity

Unity is a real time development platform that has a physics and rendering engine and is used for developing games and creating simulations. A *Unity* project contains assets, files for the project, e.g., written code, and contains objects in a scene which defines the environment [17]. Objects interact with the scene through the physics engine and are visualized through the rendering engine.

2.3.1 ZeroSim

ZeroSim is a *Unity* project that provides a connection to *ROS* and tools for building robots and environments for robots. A robot can have a variety of sensor, a 2D or 3D LiDAR, cameras, IMU's and more. These sensors acquire data in the *Unity* scene and publish it directly to *ROS* topics. A robot can be controlled by using a differential driver, that moves the robot through publishing on a velocity topic in *ROS*. It provides a bridge from *ROS* to *Unity* and enables one to create fast and stable environments [33].

2.4 Hyperparameter Optimization

In Hyperparameter Optimization, the task is to optimize an objective function with a set of parameters and their respective ranges. Most machine learning algorithms, e.g., Reinforcement Learning algorithms, have a lot of parameters that require tuning, since they considerably impact the performance [7]. To evaluate Reinforcement Learning algorithms in an unbiased way, without manually testing values, a Hyperparameter Optimization framework is used for multiple tuning runs, called trials.

2.4.1 Optuna

In *Optuna*, the objective function contains the range of values for each parameter, where a value for each parameter is chosen in a trial of the objective function. The return of the objective function indicates how good the current trial with its chosen parameters is. The objective function in this work is to train the agent on a map with a given Reinforcement Learning algorithm and return his score after evaluating the trained agent [4]. *Optuna* tries to improve the parameters for the next trial based on the returned value of the objective function using a Tree of Parzen Estimators (TPE) by default. It approximates the performance of hyperparameters based on past measurements and models the probability of a score for a given hyperparameter [6].

3 Approach

In this chapter, we present the environment used to analyze reinforcement algorithms. We present the basics of the environment, our implementation, followed by our rewarding metrics and how we integrated Reinforcement Learning agents. After that, we validate our environment as a suitable environment for reinforcement learning algorithms in the continuous control task for chapter 4.

3.1 Environment

The task is to control the Loomo Segway, a robot at our lab, from a starting point to a goal point.

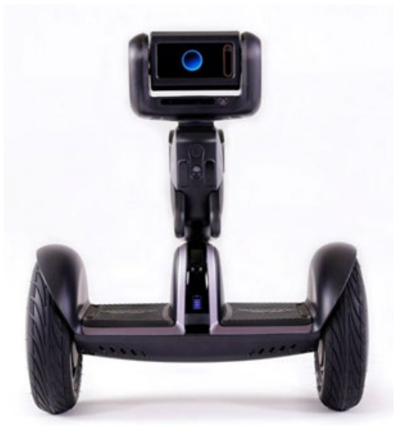


Figure 3.1: Loomo Segway [1]

The agent controls the robot by giving angular and linear velocity commands and has to avoid driving the robot into a wall or other obstacles. The agent receives a pixelated image of him and his surroundings, containing the goal if in reach and an added a-star path from him to the goal as the environments state.

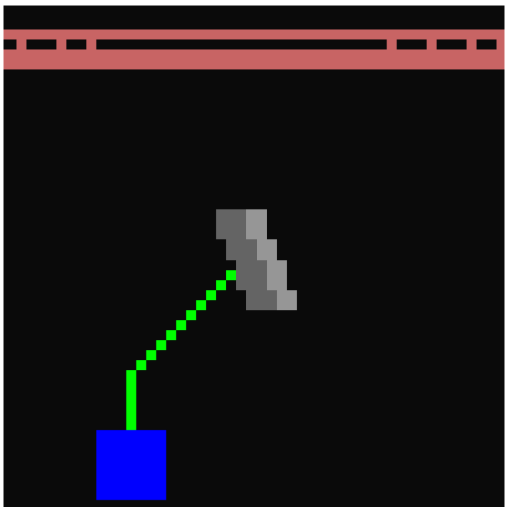


Figure 3.2: Environment State Representation

Figure 3.2 shows the visualized environment state. The walls are colored in red and the robot in gray, with the darker gray being the front of the robot. The green path is the a-star path that ends on the goal. Each pixel represents $6.5cm^2$. In this example, the state is a $51 * 51$ image and each pixel can have 6 different values, meaning that the state space is 6^{2601}.

SLAM creates the map through a 2D LiDAR on top of the robot and the goal and path are added to the state. The state could be created from other sources, for example cameras or a satellite, without needing to know the robots internal state, its current linear and angular velocity. It removes the need for a bidirectional communication from an actor to the robot, only needing the actor to send commands to the robot.

3.1.1 Implementation

To implement this environment as an MDP, to build an agent-environment loop, we use the OpenAi framework *gym*. It represents an interface standard for creating environments, shown in equation (3.1). The two basic functions are step and reset, where reset is called at the start of an episode to get a starting state and step is called subsequently until termination.

$$\begin{aligned} state &= environment.reset() \\ next\_state, reward, done, \_ &= environment.step(action) \end{aligned} \tag{3.1}$$

The step function takes an action as an argument and returns a following state, a reward for the action taken in the last state,

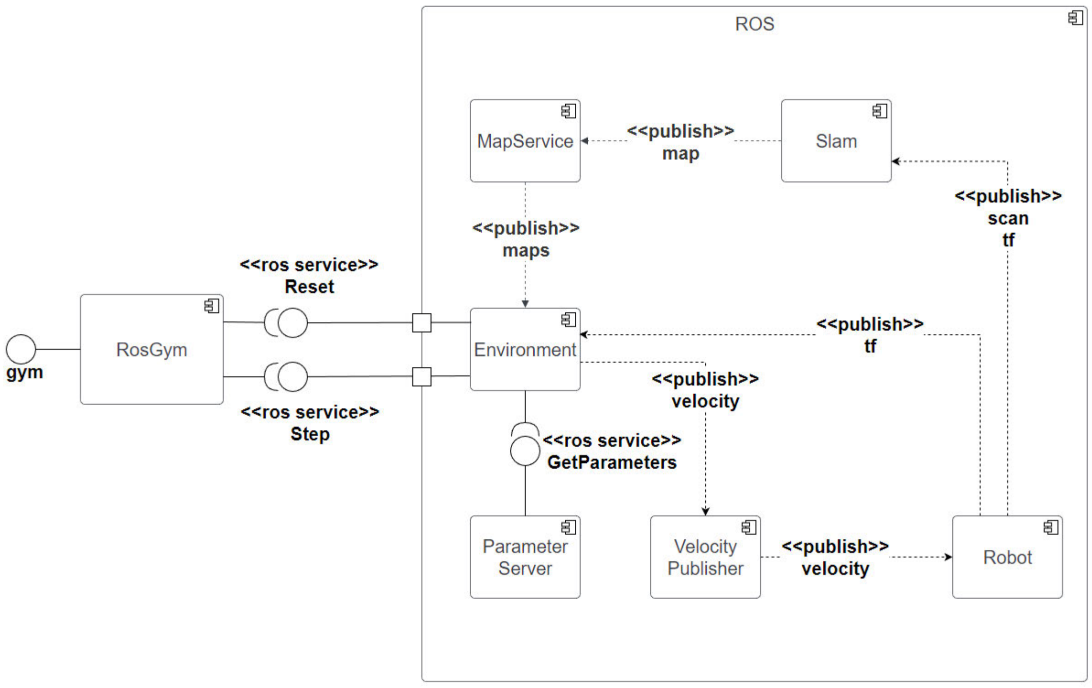


Figure 3.3: Component Diagram Environment

Our *gym* implementation, named *RosGym* throughout this work, starts a *RosLaunch* file containing multiple nodes and is visualized by figure 3.3. All nodes have their own function: The *Environment* node implements the reset and step functions from the agent-environment loop as a *ROS-Service* that the *RosGym* uses on his step and reset functions. It needs the robots position for drawing the robot into a state and gets it from the robot by subscribing to his position data, named *tf* or *transform frames*. It subscribes to the maps by the *MapService* node to draw the walls surrounding the robot, the a-star path and the goal. The *MapService* node subscribes to the published maps from the *Slam* node that runs *Gmapping*, and adds weights for the a-star pathing algorithm around walls, so that the a-star path does not go near walls, to keep a set distance to them. The *Slam* node itself uses the robots *transform frames* and *LiDAR* data to create the maps for the *MapServer*. A *ParameterServer* node keeps a config file that lets us change the behavior of the *Environment* node, e.g., how much distance the a-star path keeps from walls, how fast the robot can drive or how much reward the robot gets on reaching the goal. The *Environment* node publishes a velocity, that it receives from the agent as an action, and a *VelocityPublisher* node subscribes to the message and publishes it cyclically to the robot. The robot takes these velocity messages and sets it as his linear and angular velocity to drive.

On the reset function, the *RosGym* calls the *ROS-Service reset* from the *Environment* node. Figure 3.4 represents the behavior of a *RosGym* reset call with a simulated robot.

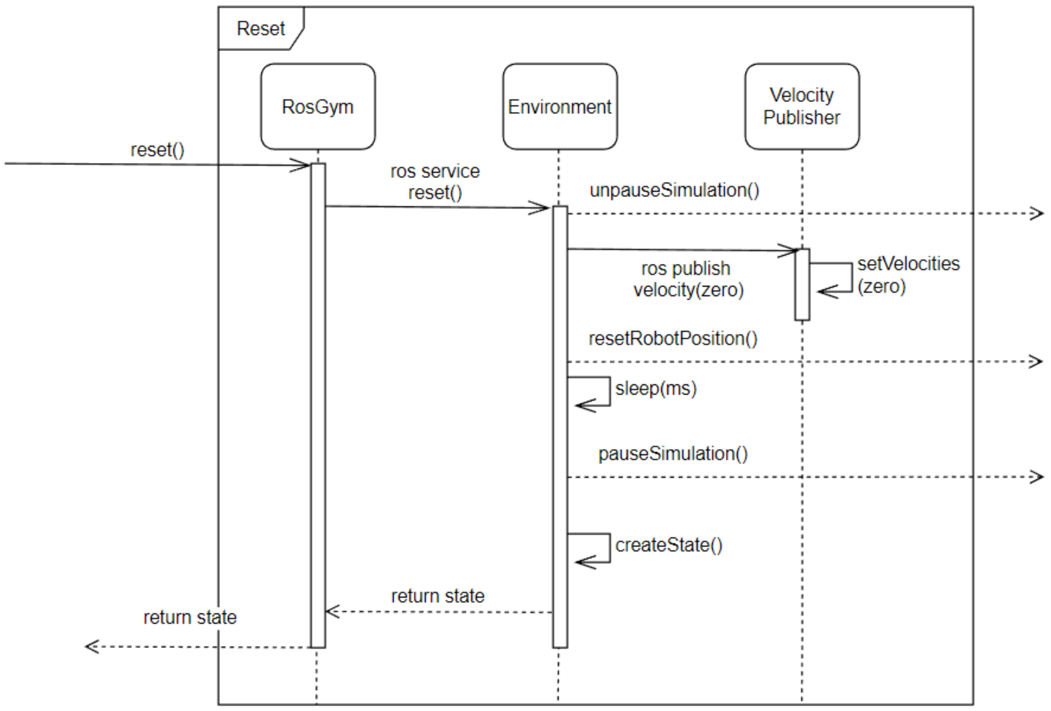


Figure 3.4: Sequence Diagram Reset

The environment publishes a message to unpause the simulation and publishes velocity commands to stop the robot. It then resets the robots position in the simulation to the starting position, followed by a sleep for a set timing, that represents how long an action is executed by the robot. A pause message gets published afterwards to stop the simulation from running until the agent calls another action and the environment creates a state that returns it to the agent.

The step function, shown in figure 3.5, behaves similarly, but takes an action as an argument: An action for the *RosGym* consists of two values with a range from minus

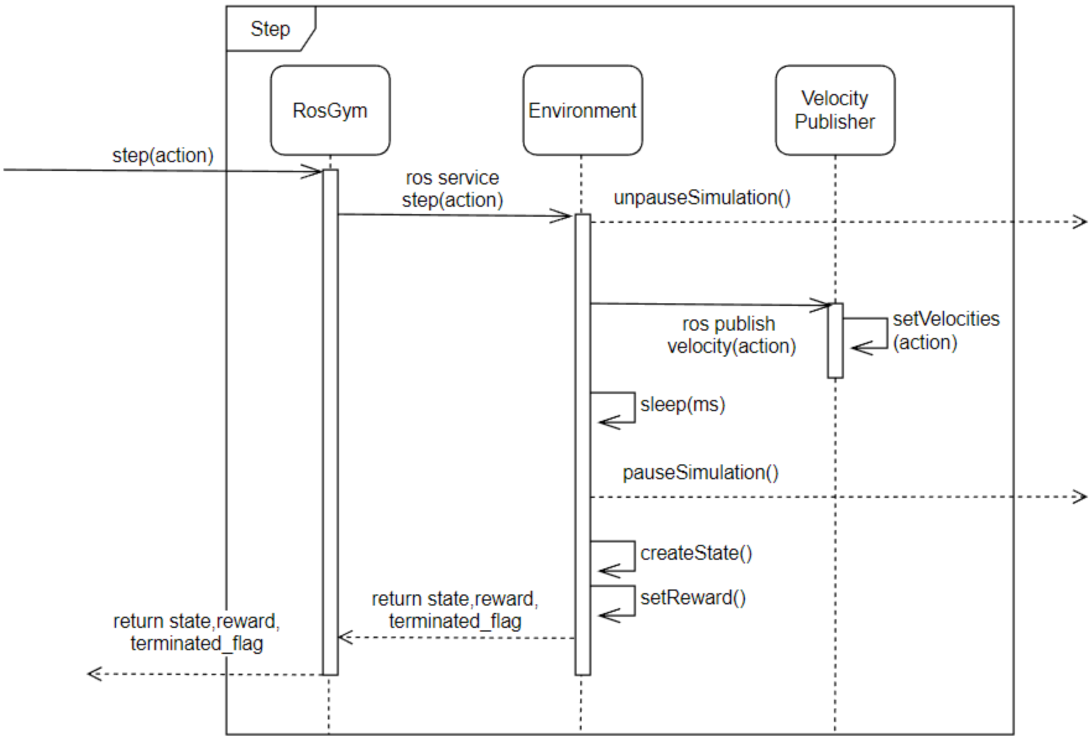


Figure 3.5: Sequence Diagram Step

one to one, where the first value, if positive, represents a multiplier with the maximum linear velocity and a negative first value a multiplier with the minimum linear velocity. The second value represents a multiplier with the maximum rotation, the angular velocity, where a negative value rotates the robot to the right and a positive value to the left. An action containing two ones would make the robot drive forward and to the left with his maximum angular and linear velocity for the set sleep time of the environment after executing an action. The step function returns more than just a following state after taking an action, it contains a reward and a termination signal, created in the *SetReward()* function.

Rewarding

The *Environment* node calculates a reward by looking at the action the agent took based on the state and how it influenced the environment. After executing the action, if the agent is at the goal or too close to a wall, the termination flag is set to true, signaling an end of episode. A positive reward follows for reaching the goal and a negative for terminating on a wall. On a non-terminating state, the a-star path is used for calculating the reward. If the agent rotates the robot towards facing the path up close to him and drives towards the path's direction, the agent gets rewarded. Both conditions have to be fulfilled for the agent to receive a positive reward. The agent gets an additional reward facing towards the path with his front, to incentivize not driving backwards. Getting closer to the goal by shortening the a-star path also gives him an additional reward, to make the agent drive faster.

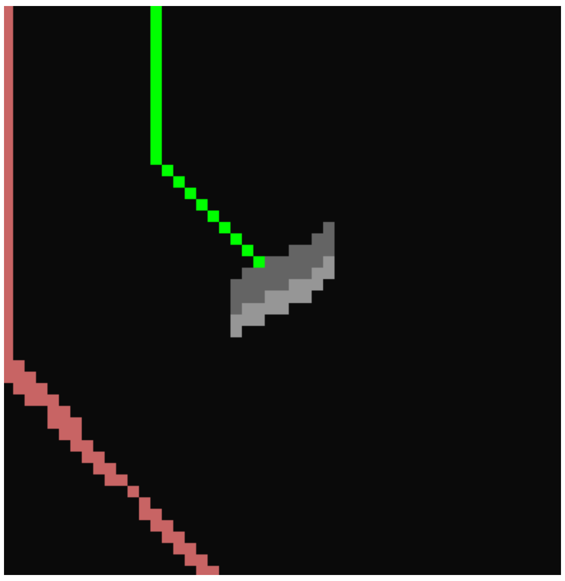


Figure 3.6: Example State Representation

In figure 3.6, a positive rewarding action would be rotating slightly to the left to keep on track with the a-star next to the robot and drive forwards, since he is front facing towards the path. Rotating right or driving backwards would result in a negative reward.

Reinforcement Learning Integration

For running Reinforcement Learning algorithms on our environment, we chose to use the framework *Stable Baselines 3* and its experimental extension *Stable Baselines 3 - Contrib*. This framework provides an extension of the OpenAi Reinforcement Learning algorithms and improves upon them by adding more algorithms and providing a wider range of customization to them. It contains every algorithm that we decide to use in chapter 4 based on OpenAi's Reinforcement Learning algorithm list [26].

We use a *Unity* simulation with the plugin *ZeroRos* to create a stable training environment for the robot. We model our lab room in *Unity* for the robot to learn how to drive and use the Loomo Segway's size for the simulated robot, to have him look identical in the environment states when switched to a real Loomo Segway.

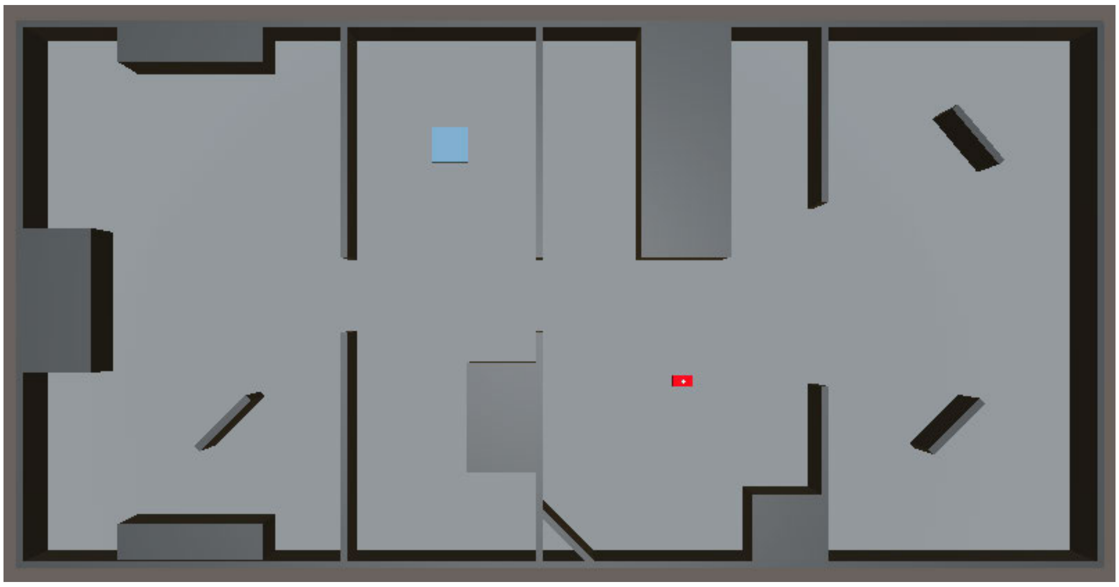
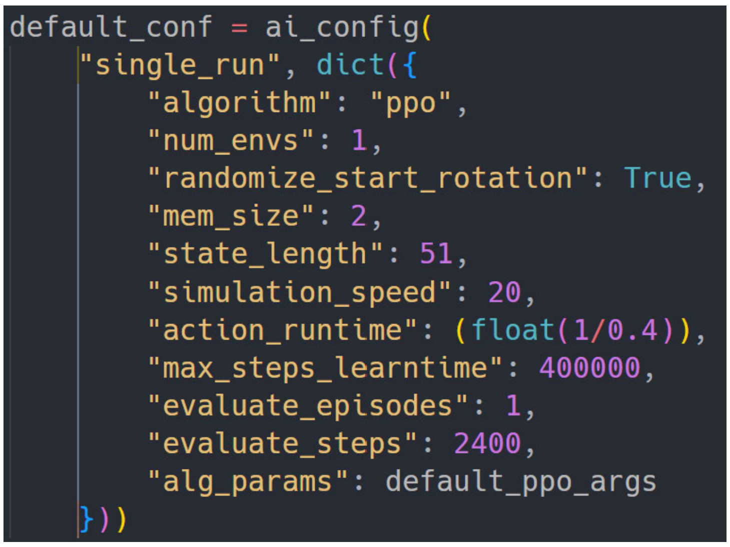


Figure 3.7: Simulated Lab Room

Figure 3.7 shows a top-down view of the modeled lab room. We increased the size of the rooms and door entries and remove chairs and other obstacles to create an easier training environment. We picked multiple starting points for the robot with multiple goal points for each starting point. In this image, the robot is marked as red and the goal point as blue. We extended the basic functionality of the *ZeroRos* plugin by adding a parameter to increase the simulation speed, for faster training wall-time. The sleep functionality in the *Environment* node takes this factor into account, enabling training on a faster simulation and evaluating it on a different real time speed factor. To enable resetting the episode, we added a listener for a teleport command to teleport the robot back to his starting position, with the option to make his starting rotation random. We added a listener for pausing and unpausing the simulation, giving us more control over the basic functionality of the simulation for a better training environment. If the simulation could not pause, and the agent takes a long time after calling step to calculate a new action for the new step, the robot would continue to drive with the last action and make the environment noisy in respect to the time the agent takes before sending a new action. This would provide a big problem, especially on a slower GPU that takes a longer time to update the neural networks.

To train a Reinforcement Learning algorithm on our *RosGym*, we created a scrip that manages multiple configurations for starting the training or evaluating and agent.

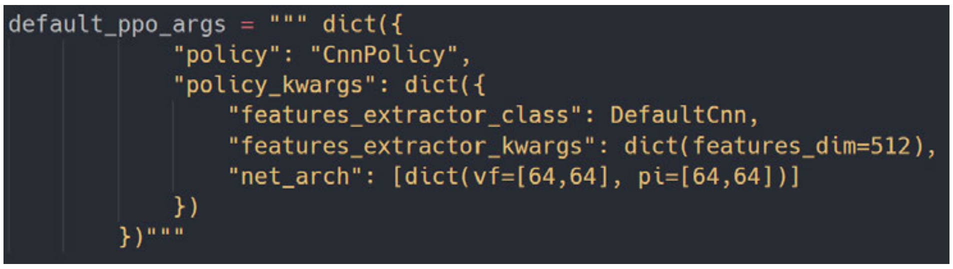


```
default_conf = ai_config(
    "single_run", dict({
        "algorithm": "ppo",
        "num_envs": 1,
        "randomize_start_rotation": True,
        "mem_size": 2,
        "state_length": 51,
        "simulation_speed": 20,
        "action_runtime": (float(1/0.4)),
        "max_steps_learntime": 400000,
        "evaluate_episodes": 1,
        "evaluate_steps": 2400,
        "alg_params": default_ppo_args
    }))
```

Figure 3.8: Example RosGym Configuration

The most important parameters for our *RosGym* are shown in figure 3.8. *algorithm* sets the Reinforcement Learning algorithm, *num_ env* how many simulations are running in parallel, *randomize_ start_ rotation* if we want to randomize the agent's rotation at the start of an episode, *simulation_ speed* how fast the simulation runs, *action_ runtime* how long an action gets executed in Hz, *max_ steps_ learntime* for how many steps the agent trains and *evaluate_ episodes* for how often he gets evaluated with *evaluate_ steps* steps. The *state_ length* parameter sets the length of the state image for the *Environment* node and the *mem_ size* parameter stands for how many images the agent gets as an environment state. Since a single image as a state does not contain the current movement of the robot, we created a stacked frame that gives the agent a sense of movement by having a frame from the past and the current frame as a state. This technique was used in Atari games with agents learning based on the frames of the game, e.g., in learning to play the Atari game *Pong* to see the movement of the ball [20].

The *alg_ params* parameter are parameters for the Reinforcement Learning algorithm.



```
default_ppo_args = """ dict({
            "policy": "CnnPolicy",
            "policy_kwargs": dict({
                "features_extractor_class": DefaultCnn,
                "features_extractor_kwargs": dict(features_dim=512),
                "net_arch": [dict(vf=[64,64], pi=[64,64])]
            })
        })"""
```

Figure 3.9: Example Algorithm Configuration

We used a *CnnPolicy* in figure 3.9, meaning that the agent gets his input from a Convolutional Neural Network, named the *DefaultCnn* in our work, with the output size of the *features_ dim*, in this case 512:

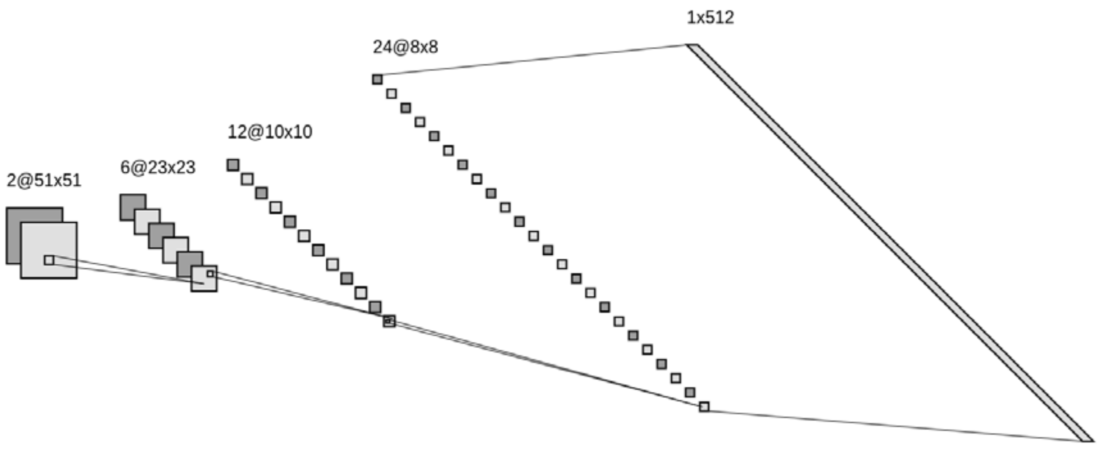


Figure 3.10: DefaultCnn Visualized

We used the CNN shown in figure 3.10 to have a relatively small network structure for a faster training wall-time. The input is two stacked $51 * 51$ images, which are decreased to $23 * 23$ with six filters, a kernel of eight and a stride of two, to decrease the image size. Another convolutional layer with a stride of two and kernel size of five decreases the features to a set of twelve filters, followed by a final convolutional layer with a kernel size of three and 24 filters. The output gets flattened and is the input for a feature representation by a fully connected layer with the *features_ dim* as a size. These features will then proceed to be the input of the critic and policy layers, configured by a parameter named *net_ arch*, where *vf* is the critic and *pi* is the actor network.

3.1.2 Validation

To validate our environment as a suitable learning problem for Reinforcement Learning algorithms in the continuous control task, we first have to test if an agent is capable of learning and achieving higher rewards while training longer, continuously improving on reaching the goal and not terminating on walls. To train an agent for the validation, we use the Proximal Policy Optimization algorithm. More about this algorithm will be presented in chapter 4 chapter, but it is a good starting point for testing an environment, since it is the default Reinforcement Learning algorithm used in OpenAi, due to its stability in learning and robustness to parameters [27]. We use the default algorithm parameters from the framework, our presented default critic and actor layers with the *DefaultCNN* as the neural network structure and run it three times on our example start and goal position, seen in the simulated lab room image before for 400.000 steps. We run three simulations in parallel for the agent to collect data from, meaning that the agent learns from a total number of 1.200.000 training steps.

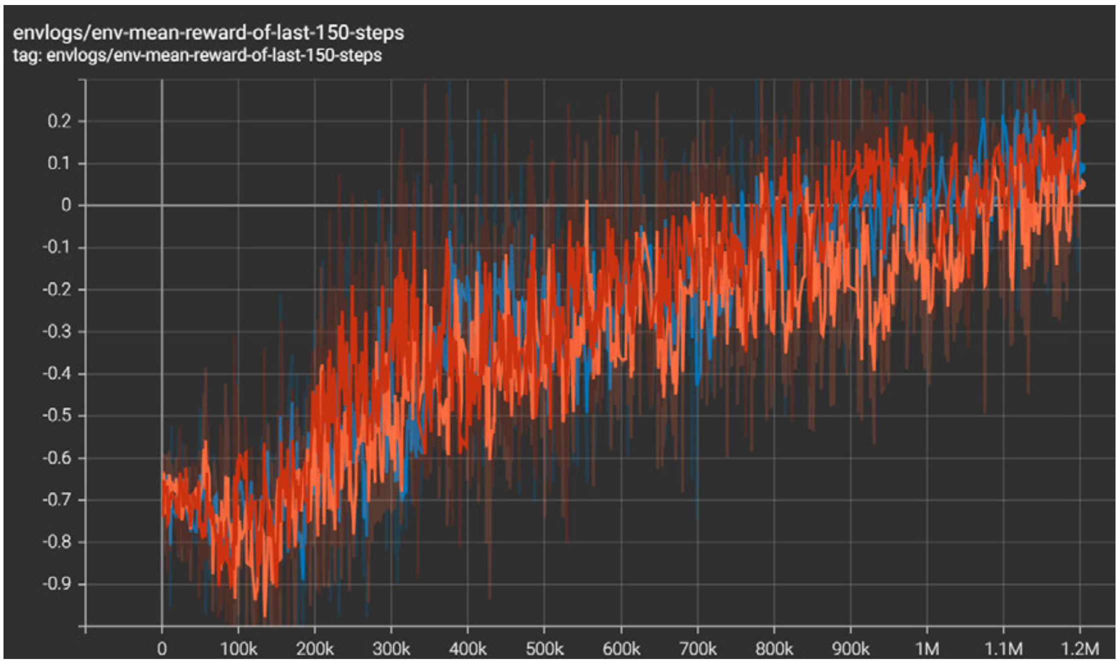


Figure 3.11: Mean Reward of the last 150 steps for the Validation

Figure 3.11 shows the reward logs from multiple runs of the Proximal Policy Optimization algorithm. The X-Axis shows the training step, up to the full 1.200.000 steps, and the Y-Axis the mean reward collected by the agents last 150 steps, to get a smoothed value. The runs themselves, separated by their color are visualized by an exponential moving average filter with a weight of 0.6, to make it less noisy. Each run has a steady improvement in performance and each agent learns a mostly positive behavior, ending in achieving more reward than punishment on an average of 150 steps. This validates the assumption that agents can learn to drive to the goal through our partially observable state representation, and we can use this to run different Reinforcement Learning algorithms on it and analyze their behavior. It also validates our *DefaultCNN* as a suitable convolutional neural network for the image input.

4 Analysis

This chapter is the focus of our work. As mentioned before, we lean on OpenAi's work in choosing Reinforcement Learning algorithms in this work. We characterize our environment presented in chapter 3 and follow up with a categorization from OpenAi's presented Reinforcement Learning algorithms and continue with a filtering process, to find out which we want to further analyze and evaluate through our environments characteristics. Furthermore, we present the concepts of our picked choices and end this chapter with a prediction for each of the choices, how they will perform and why they will perform that way, with a final verdict of which we predict to be a good choice for any similar characterized environment. Our predictions will be answered through the experiments and results in chapter 5 and the discussion in chapter 6.

4.1 Environment Characteristics

In this section we show the environmental characteristics that we have chosen to find suitable Reinforcement Learning algorithms for.

An environments state is the first thing to look at, how big the state space is and how observable it is. Our state space is a $51 * 51$ image with six possible values for each pixel, giving 6 to the power of 2601 different possibilities, which indicates the need for function approximation.

By observability, we mean how much information the state gives. Our state is not fully observable and includes a lot of randomness, since we create an approximation of the robots position and surroundings based on sensor data that contains noise. We leave out the robots internal state as well.

Through these conditions, we don't know or can reasonably model the environment's dynamics, we can't take a good guess how the state changes after taking an action. We don't see the full map and the current movement of the robot.

Looking at the action for the environment next, we have a continuous action, making it a continuous control problem. The action selection also has to be very precise, since both of our two values per action have to be correct to receive a positive reward.

The rewarding is not sparse, meaning we don't just reward on termination, but on each step. We can give the agent for each action a meaningful feedback signal.

Another thing to look at is the sample efficiency, how easy it is to sample from the environment and how fast it is. Training on the real robot would be very sample inefficient. In our case, we have high a sampling efficiency due to the use of our simulation that can be sped up and the possibility to run simulations in parallel.

4.2 Algorithms

Based on our environment characteristics, we conclude which algorithms are in question to further analyze and make predictions about their performance for the continuous control task. First, we give a general categorization based on OpenAi's presented algorithms.

4.2.1 Categorization

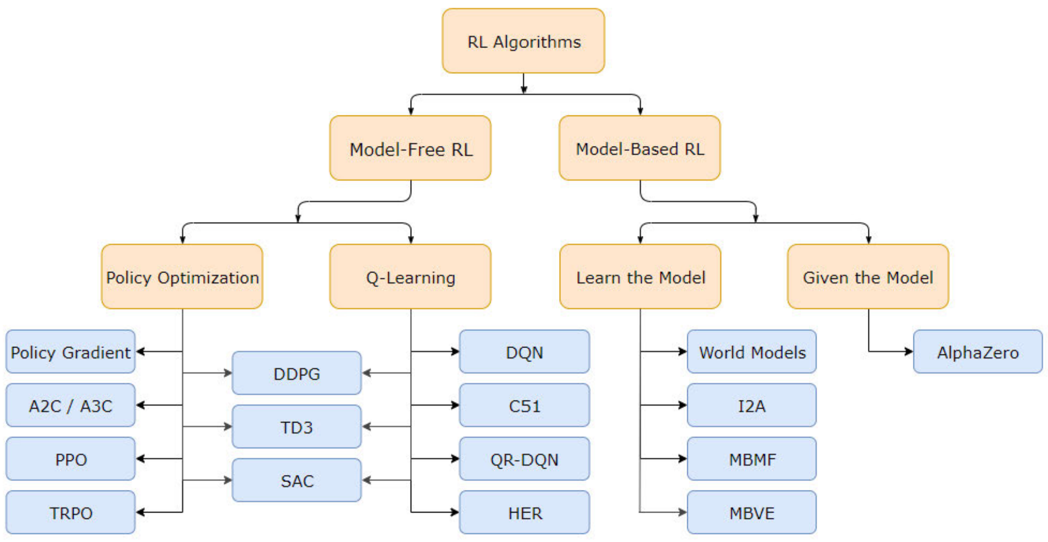


Figure 4.1: OpenAI Reinforcement Learning Algorithms [22]

There are two general categories in Reinforcement Learning algorithms based on the knowledge and predictability of the environment dynamics. The *Model-Based RL* side groups the algorithms into two parts, one where the environment dynamics are known, e.g. *AlphaZero* which was used in chess [30], and one where the environment model is learned and used to train the agent. The *Model-Free RL* algorithms use samples from the environment to train an agent. The *Q-Learning* algorithms learn a value representation of the environment states and actions and act based on these values. The algorithms on the right side under *Q-Learning* are all only learning a state representation, e.g. *DQN*, standing for *Deep-Q-Network*, which is the realization of *Q-Learning* with neural networks as function approximation [20]. The left side under the *Policy Optimization* group represents algorithms where a policy is learned directly that tries to maximize the expected discounted reward, e.g., *A2C*. The algorithms in between *Policy Optimization* and *Q-Learning* are learning a policy that tries to maximize a learned state representation, learned by a critic. *DDPG* for example is an extension of *DPG* for function approximation, providing more stability like *DQN* for *Q-Learning* [19]. The middle ones and the most left ones are all, except the *Policy Gradient* algorithm, actor critic methods, that learn a policy and a value function simultaneously to update through TD errors after an arbitrary number of steps instead of sampling a whole episode. The algorithms on the *Q-Learning* category are all *Off-Policy* algorithms and can update the agent with old samples, while the most left algorithms are *On-Policy*, meaning they have to throw the sampled data from the agent-environment interaction away after updating the policy.

4.2.2 Filtering

We start in the same order as before, from the right side to the left side, based on figure 4.1. The *Model-Based RL* algorithms can not be used in this work. *AlphaZero* expects the environment dynamics, which are not given in our case, and the algorithms that try to learn the environment dynamics can not be used as well, since our state space is not fully observable and has too much randomness, as described in the characteristics of our environment. This leaves us with the *Model-Free RL* algorithms. The *Q-Learning* based algorithms on the right side can also not be used, since we are using continuous actions, leaving us with the *Policy Optimization* algorithms. We will not use the *Policy Gradient* algorithm, since *A2C* gives us an improved version, where the agent can learn after each step instead of after each episode. *PPO* and *TRPO* both prove to be useful extensions of the *A2C* algorithm, making it more stable and sample efficient by making sure that the updated policy will not be too far away from the previous policy, so that an update won't drastically impact the policy's performance after a single update [29] [28]. This leaves us with using *PPO* and *TRPO* for the *On-Policy* case. The algorithms in the middle, *DDPG*, *TD3*, *SAC* all derive from *DPG*. *TD3* and *SAC* are both extensions of *DDPG*, addressing issues in overestimation of actions and stability while learning [12] [13]. We will remove *DDPG* from our list and keep the other two.

We decided to include a newer Reinforcement Learning algorithm named *TQC* to our list. It combines *QR-DQN* and *SAC*, both on the OpenAi presentation and gives us an actor critic version of *QR-DQN* [18]. This gives our work a look at a newer approach, the distributional approach in Reinforcement Learning, where a critic does not directly learn the value of the state, which represents the expected mean reward following the current policy after going into that state, but a distribution of a reward [9]. It enables us to learn multiple possible values with their own probability for a state for example. This learned distribution is called the *value distribution*. It aims to address instability in learning and improve the function approximation, arguing that approximating a distribution of a value of a state has more value than approximating only the mean value of a state.

4.2.3 Concepts

Our final list of choices, based on our environment characteristic, contains *TRPO*, *PPO*, *TD3*, *SAC* and *TQC*. We will present their concepts and continue with predictions based on their concepts matched with our environment characteristics. We are going to look at the *On-Policy* algorithms first and how they handle the shortcomings of this algorithm class.

$$\text{maximize } L(\theta) = E_{\pi_\theta}[\sum_{t=0}^{\infty} \gamma^t R] \tag{4.1}$$

To show the concepts of the following two algorithms, we present a simplified view of the goal of policy gradient algorithms in equation (4.1). The goal is to maximize the the expected discounted long term reward through updating a policy parameter vector θ [34].

TRPO and *PPO* are similar to *A2C* and thus *REINFORCE* and address a major problem in *On-Policy* policy gradient algorithms. The policy used to sample data from is the policy that gets updated. This means, that a big update would completely change the policy and maybe for the worse, since a policy that would collect bad data will also not learn well, since the data is bad. *TRPO*, named *Trust Region Policy Optimization*, updates the policy by the largest step possible while satisfying a KL-Divergence constraint, which describes the distance between two probability distributions. The algorithms equations are shown in equation (4.2).

$$\begin{aligned} \hat{A}_t = \delta_t = r_{t+1} + \gamma V(s_t + 1) - V(s_t) = Q(s_t, a_t) - V(s_t) \\ \rho_{t(\theta)} = \frac{\pi_\theta(\alpha_t|s_t)}{\pi_{\theta_{old}}(\alpha_t|s_t)} \\ L(\theta) = E_t[\rho_t(\theta)\hat{A}_t] \\ \text{maximize } L(\theta) \text{ subject to } D_{KL}(\theta_{old}, \theta) \leq \beta \end{aligned} \tag{4.2}$$

$\hat{A}$ is the advantage function, which represents how much more advantageous it is to take action a in state s, since the Q value is the expected value by taking that action, and the value function V returns the expected value of the whole state. It is approximated by the TD-error advantage, as done in chapter 2. ρ represents the difference between the old policy before taking an update and the policy after updating. The L function represents the performance of the current parameter vector θ, that parameterizes the policy, and the task is to maximize it. Maximizing L means to push the parameter vector θ in the direction of the Advantage-Function, meaning that we want to increase the probability of actions that led to a higher reward then currently expected by the critic. D_{KL} represents the KL-Divergence between the policy before the update and after updating. It searches for a parameter vector θ that maximizes the performance of the policy while respecting the constraint, represented by β. It improves the stability a lot, since keeping the updates based on the distributions close instead of simply taking a step results in a more monotonic improvement while not diverging through updates that reduce the current performance [28]. Solving this constraint problem to make a policy update is a complex task that hurts the scalability and provides problems with using bigger neural networks [29].

PPO, named *Proximal Policy Optimization*, addresses this problem by moving the constraint to the objective of the policy. In this work, we focus on the clipping version of *PPO* that proved to be the most successful [29]. In *PPO*, the policy does not get updated directly with a step size, but gets clipped to update only to a set distance.

$$L_{clip}(\theta) = E_t[min(\rho_t(\theta)\hat{A}_t, clip(\rho_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t)] \tag{4.3}$$

Equation (4.3) shows the function that *PPO* maximizes. The epsilon parameter represents a bound of how far the policy can change. If our advantage of taking an action in a in state s is high, we update our policy towards increasing the possibility of taking that action while being clipped by an upper bound, to only allow a minimal update. Similarly, if the advantage is negative, meaning we want to decrease the possibility of the action, we add a lower bound to not decrease the possibility too far. This results in a less complex algorithm and improves in runtime, while achieving similar performance compared to *TRPO*. This improvement allows us to use multiple updates on a sampled batch of transitions due to respecting the clipping of the policy and fixes the problem with using bigger networks, since we don't need to solve a complex constraint optimization and can use the simple gradient updates [29].

To reiterate, *TRPO* updates a policy by solving a constraint optimization problem, improving the performance of the policy while staying in between a set KL-divergence distance, while *PPO* adds a clipping towards the normal gradient update which emulates the same behavior. These *On-Policy* algorithms explore naturally through the stochasticity of the policy, since actions for a given state are sampled from the policy's probability distribution.

TD3, called *Twin Delayed Deep Deterministic Policy Gradient*, stabilizes *DDPG* and thus *DPG* by adding multiple ideas. In *Q-Learning*, a big problem is the overestimation of a Q value. The Q value includes the maximum of the next Q value, which is a maximum over estimated values. Sutton and Barto present an intuitive example: If we have an MDP where the true Q value of each state is zero, but the estimated values are randomly with some being positive, then the maximum Q value will be positive, introducing a positive bias favoring the Q values that are greater than the others [32, ch. 6.7]. *TD3* uses the same solution as presented by Sutton and Barto in [32, ch. 6.7], called *Double-Q learning*, using two Q functions that are both randomly initialized and the smallest Q value from both Q functions is used as the Q value. Another idea from *TD3* is to update the policy less frequently then the two Q functions. The policy in the *Off-Policy* actor-critic algorithms learns to exploit the current Q function by learning to output the maximum Q value. Updating the Q function more often then the policy results in a more stable learning process, since it does not directly learn to exploit it and lets it update more often. *TD3* makes use of target networks as well, that represent delayed copies of the Q functions and are used in the TD error for calculating the max Q value, to further improve stability. Another approach from *TD3* is to add noise to the action sampled from the policy while updating, as seen in equation (4.4):

$$\delta_t = R_{t+1} + \gamma max_{a=\pi(a|S_t)} Q_{target_{min}}(S_{t+1}, \mu(\pi(a|S_t))) - Q_{min}(S_t, A_t) \tag{4.4}$$

Creating the TD error, we see that the action that maximizes the Q value of the target network in a state S_t is returned by the policy π. The μ function here represents the additional action noise and aims to reduce occurrences of narrow peaks in the value estimations. The *min* under the Q and target Q function represent that both of the double Q and double target Q functions will be evaluated and the minimum resulting Q value for each one used. *TD3* explores like *DPG*, by adding random noise to the action while training the agent [12].

SAC, named *Soft Actor Critic*, takes a new approach to *Off-Policy* learning to address the big issue of exploration by learning a stochastic policy. It changes the Reinforcement Learning task from maximizing the expected reward to maximize the expected reward in addition to keeping an entropy. It learns a policy that maximizes the expected return while keeping a set randomness. The entropy of a policy for a given state is an indicator of how random the current policy is for that state. E.g., a policy that would only favor a single action in a given state would have a low entropy in that state. The agent gets rewarded at each time step by how high the entropy of the policy for the current state is.

$$\begin{gathered} max_{\pi} E[\sum_{t=0}^{\infty} \gamma^t R] \\ max_{\pi} E[\sum_{t=0}^{\infty} \gamma^t (R + \alpha H(\pi(*|S_t))] \end{gathered} \tag{4.5}$$

Equation (4.5) shows a simplified view of the goal of Reinforcement Learning agents. The first equation shows the basic Reinforcement Learning task, to find a policy that maximizes the expected, discounted reward. The second equation shows the task for the *SAC*, to maximize the expected discounted reward in addition to *H*, a function representing the entropy of the policy in a current state. Alpha is a hyperparameter that gives a weight to the entropy. The higher alpha is, the more important the entropy gets and vice-versa. A higher alpha would lead to a higher exploration rate, since the policy outputting actions is learning to be more random. *SAC* incorporates the double *Q-Learning* as done in *TD3*, but explores through the learned stochastic policy, like the *On-Policy* algorithms [13].

The final algorithm we want to look at is *TQC*, named *Truncated Quantile Critics*, and is an extension of *SAC* that also learns a stochastic policy. It addresses the overestimation bias of *Off-Policy* algorithms by learning a value distribution instead of learning the mean value, building on the idea of *QR-DQN*. Since *TQC* combines *QR-DQN* and *SAC*, it helps to look at *QR-DQN* and where it came from and derive *TQC* from that point. One of the first distributional Reinforcement Learning algorithms was *C51*. It learns a probability distribution for a set of rewards [8].

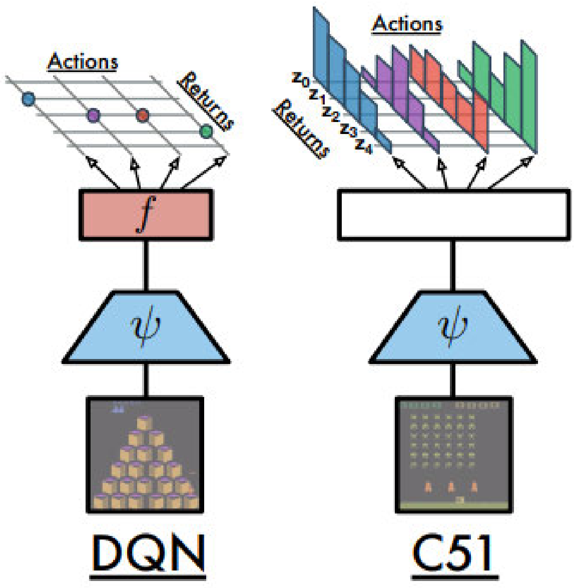


Figure 4.2: DQN and C51 [8]

Figure 4.2 visualizes *DQN* and *C51* and puts them next to each other for comparison. The left side shows the output from a *DQN* Q function approximation. Given an input state, the function approximation returns a Q value for each state. The *C51* function approximation has a set of value expectations named z_i, we will call it an indexed value expectation for simplicity. In this case, there are five indexed value expectations, from z_0 to z_4, where z_0 is the expected maximum return and z_4 the minimum expected return for that action in a given state. Each of the indexed value expectation has a probability attached to it, how likely this return would be for the agent. The Q value is simply the summation of each indexed value expectation times its probability, represented in figure 4.2 by the height of the bar. This approach outperformed *DQN* in almost every tested environment [8]. *QR-DQN* represents the value distribution with fixed probabilities for adjustable expectations of the value instead of having fixed expectations with adjustable probabilities. This approach proved to be even another step better than the *C51* approach and was around 33% more successful on the same tasks [9] and is added in figure 4.3.

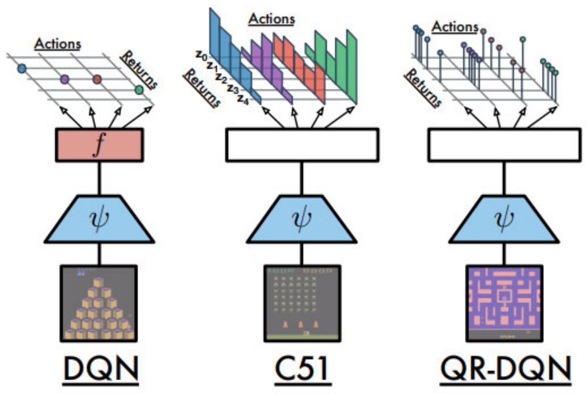


Figure 4.3: DQN, C51 and QR-DQN [8]

On the right sight, we see that *QR-DQN* outputs multiple expected returns, where each return from this learned distribution has the same probability of occurring. Each value is called a quantile, and the Q value is calculated similarly as in *C51*, by the summation of each quantile's discrete return value times the probability, which each quantile shares. An example would be to have five quantiles that each have the probability of twenty percent, so *QR-DQN* would output five values that are expected to happen with a probability of said percentage.

TQC combines this idea of representing a quantile distribution for an action and extends it to the continuous space by combining it with the *SAC* algorithm. Another idea it adds, to address the overestimation bias further, is to drop some quantiles controlled by a hyperparameter, an idea similar to randomly dropping neurons in a neural network to generalize better [18].

4.3 Predictions

Based on our analysis of the concepts, we predict that *PPO* and *TRPO* should both do well, having a stable learning curve slowly increasing the reward collected. Since we are not using a lot of layers in our neural networks, we predict that they both will achieve similar performance. In the *Off-Policy* case, we predict that *TD3* will have the hardest time of all algorithms, since random noise on the exploration will cause problems with our rewarding system, because both acceleration values have to be accurate to receive rewards. Based on the way *SAC* and *TQC* explore, we predict that they will not have problems like *TD3*. We predict that they will both learn faster than the *On-Policy* algorithms, but less stable, and that *TQC* will outperform *SAC* and be as successful as the *On-Policy* algorithms, measured by the reward collected on the evaluation of the agent after training, learning faster than they do while having a less stable learning curve.

Our final prediction and deduction for the reader is to use *PPO* in a sample efficient environment, e.g., learning in a simulation, and to use *TQC* in a sample inefficient environment, e.g., training a robot in the real world.

5 Experiments and Results

In this chapter, we provide a structured approach to answer our predictions made in chapter 4. Reinforcement Learning algorithms have a lot of parameters that require fine-tuning, since they impact the agent's performance substantially and a good set of parameters can be very specific to an environment with its own unique characteristics. Some Reinforcement Learning algorithms are said to require less tuning, like *PPO*, and their default parameters should generally work on a broad variety of environments [27]. We used *PPO* with the default parameters for validating our environment, and they worked well enough, but they still require tuning for each environment to achieve better performance, especially because we want to evaluate them head to head for our continuous control task.

We start our experiments with creating a hyperparameter study through the framework *Optuna* for each Reinforcement Learning algorithm and pick the best four parameter sets for each algorithm to further analyze their behavior, performance and stability. The first section shows our experiment structure, followed by a presentation of the results, to be discussed chapter 6.

5.0.1 Hyperparameter Optimization

To find better hyperparameters, we use *Optuna* with the default optimization strategy *TPE*. Each algorithm will have 60 trials, where each iteration uses three parallel simulations with 400.000 steps, leading to 1.200.000 environment interactions, as done in section 3.1.2.

We use the simulated lab room introduced in chapter 3, where we again use the red spot as the starting position and the blue spot as our goal. The robot starts with a random starting rotation at the start of an episode and each algorithm uses the same Convolutional Neural Network (CNN) structure, named *DefaultCnn*, to create a feature output of the image for the agent and critic networks.

5.0.2 Hyperparameters

Optuna expects an objective function for the hyperparameter optimization that needs a range of possible values for each parameter. The following enumeration in section 5.0.2 shows a description for each parameter with our value choices. These parameters are based on the implementation from OpenAi and the extension through *StableBaselines 3* and *StableBaselines 3 - Contrib*. *StableBaselines 3* provides an optimized parameter list for a lot of different environments and a parameter range list for each of their algorithms [5]. We used the later one for this work and customized it by minimizing a lot of parameters, like the learning rate from a maximum of 1 to a maximum of 0.001, for a more stable parameter search. We decreased the batch sizes to have enough space for our GPU used, the RTX Titan, that has 24 GB of video memory [21], and we run each parameter study with three environments in parallel.

- Shared by every algorithm:
  - gamma: The discount factor for future rewards.
    [0.9, 0.95, 0.98, 0.99, 0.995, 0.999, 0.9999]
  - learning_rate: The learning rate, defines the step size of moving towards minimizing the loss function for a neural network.
    [0.00001 to 0.005]
  - net_arch: The network architecture of the critic and policy, each entry represents a fully connected layer.
    "small": [64, 64], "medium": [256, 256], "big": [400, 300]
- *TD3*, *SAC*, *TQC*:
  - learning_starts: How many steps are made before starting to train the agent.
    [0, 1000, 5000]
  - batch_size: The number of random transitions that are taken from a history buffer to update the agent every *train_freq* steps.
    [8, 16, 32, 64, 128, 256]
  - buffer_size: The size of the history buffer, the maximum number of saved transitions, since *Off-Policy* algorithms reuse them to update every *train_freq* steps.
    [40 000, 400 000]

  - tau: Weight factor when copying the neural networks weights of the critic to the target network.
    [0.001, 0.005, 0.01, 0.02, 0.05, 0.08]
  - train_freq: Number of steps taken in the environment before updating the policy and critics.
    [1, 8, 16, 32, 64, 128, 256, 512]

- *TD3*:
  - noise_type: Action noise type, where *normal* represents a Gaussian function and *ornstein-uhlenbeck* represents the Ornstein Uhlenbeck process.
    ["ornstein-uhlenbeck", "normal", None]
  - noise_std: The standard deviation of the action noise.
    [0.0 to 1.0]

- 'SAC, *TQC*:
  - ent_coef: The entropy coefficient, how much the entropy weighs for the agent. *auto* here means that it scales up and down based on the certainty of the policy in the current state. If the policy in a current state is uncertain, then the entropy coefficient grows to increase the exploration for the agent and decreases it if the policy of the agent is sure about which action leads to a high expected reward [14].
    ['auto', 0.001, 0.01, 0.1]

- *TQC*:
  - n_quantiles: Number of quantiles per critic. 10 Quantiles would mean that every return value of a quantile has a predicted chance of 10% to appear.
    [5 to 50]
  - top_quantiles_to_drop_per_net: How many top valued quantiles will get dropped to decrease the overestimation bias.
    [0 to n_quantiles-1]

- *PPO*, *TRPO*:
  - batch_size: The minibatch size used to update the neural networks. Each entry in the minibatch represents an environment transition from the last *n_-steps* taken.
    [8, 16, 32, 64, 128, 256]

  - n_steps: How many environment transitions are sampled to update the policy. [16, 32, 64, 128, 256, 512, 1024, 2048]
  - gae_lambda: Factor for weighing each step in to calculate the advantage of an action taken in a state. A *gae_ lambda* factor of 0 would result in the default advantage case, where the advantage is *delta_ t* as in equation (2.7).

$$\begin{aligned} G_t^1 &= R_{t+1} + \gamma V(S_{t+1}) \\ G_t^2 &= R_{t+1} + R_{t+2}\gamma^2 V(S_{t+2}) \end{aligned} \tag{5.1}$$

    In equation (5.1), G_t^1 represents the estimated return for a current step, represented by the next reward sampled and the discounted value of the next state. G_t^2 represents the *n-step* return of the estimated reward with two steps as a return, using the sampled reward of the second step and the value function of the second state added to the current sampled reward. With that, we bootstrap to the second state and not to the next state as done in the normal TD-learning case. *Gae_ lambda* averages over each possible *n-step* reward to take every state value into account and get an averaged state value, where the weight factor grows exponentially for each *n-step* return that gets averaged in. It builds a bridge between the updates after one step, as done in the advantage or *Q-Learning*, and the full return of an episode when updating the agent as in *REINFORCE* while providing a smoother version of the *n-step* TD returns. [0.8, 0.9, 0.92, 0.95, 0.98, 0.99, 1.0]

- *PPO*:
  - clip_range: The ϵ parameter of the optimization goal of *PPO*, equation (4.3). [0.1, 0.2, 0.3, 0.4]
  - n_epochs: How often the policy gets updated using the current sampled transitions. This results in multiple updates for each currently sampled transitions and is allowed due to the policies restriction to move too far away from the old policy.
    [1, 5, 10, 20]
  - max_grad_norm: An additional clipping parameter for the whole gradient update.
    [0.3, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 2, 5]

  - vf_coef: Weighting factor for the critics value of a state while calculating the advantage.
    [0.0 to 1.0]

- *TRPO*:
  - n_critic_updates: Number of critic updates per policy update.
    [5, 10, 20, 25, 30]
  - cg_max_steps: Maximum number of steps for computing the gradient, based on the KL-divergence constraint.
    [5, 10, 20, 25, 30]
  - target_kl: Target KL-Divergence between the policy before updating and after updating.
    [0.1, 0.05, 0.03, 0.02, 0.01, 0.005, 0.001]

5.0.3 Objective Function

Using these parameter ranges and our lab room simulation, we can model the objective function for *Optuna*.

```
Function objective(trial, algorithm, learn_steps, evaluate_steps,
 evaluate_episodes):
    parameters = sample_parameters(trial, algorithm)
    environment = create_environment(RosGym, number_environments=3)
    agent = create_agent(algorithm, parameters, environment)
    agent.learn(learn_steps)
    return agent.evaluate(evaluate_steps, evaluate_episodes)
```

Algorithm 1: Objective Function

Algorithm 1 represents the objective function for *Optuna* to maximize, by evaluating the performance of the current sampled parameters after each trial of the current study. We call the function with our specified Reinforcement Learning algorithm and with our current trial, which keeps the information about how each run performed in past iterations and what parameters to suggest next. We input a set time for the agent to train and evaluate and start to sample parameters for the current trial iteration from *Optuna*. Given our algorithm, the environment and the current parameters for the algorithm, the agent starts learning and returns the reward received after evaluating. This repeating process improves the parameters for the algorithm to find a better set on the long run.

5.0.4 Gathered Data

The objective function returns the score achieved by the agent through driving in the environment. Since a robot in our environment can gather more rewards by driving cleanly to the goal then he gets punished by driving into a wall, say at the very end instead of hitting the goal position, we log how often he gets to a goal or terminates on walls for the results. This does not impact the score for the objective function and will be domain specific knowledge used for the further analysis. We add a position logging to see how the agent explores, based on the Reinforcement Learning algorithm and configuration, by creating a 2D map of his spots visited per time-span while training and in the evaluation for the best trials of each algorithm.

5.1 Results

This section shows the results of the hyperparameter optimization. We then elect the best four parameter sets for each algorithm, by picking the two with the highest value from the objective function and the two with the highest goal to terminating on a wall difference. These parameter sets are tested for their stability, by re-running each set five times, to elect the best set per algorithm. The best sets exploration, stability and learning behavior results are shown afterwards.

5.1.1 Hyperparameter Optimization Results

First, the studies of each trial while running the hyperparameter optimization are shown, followed by the importance of the parameters, how much each parameter impacted the agent's performance,

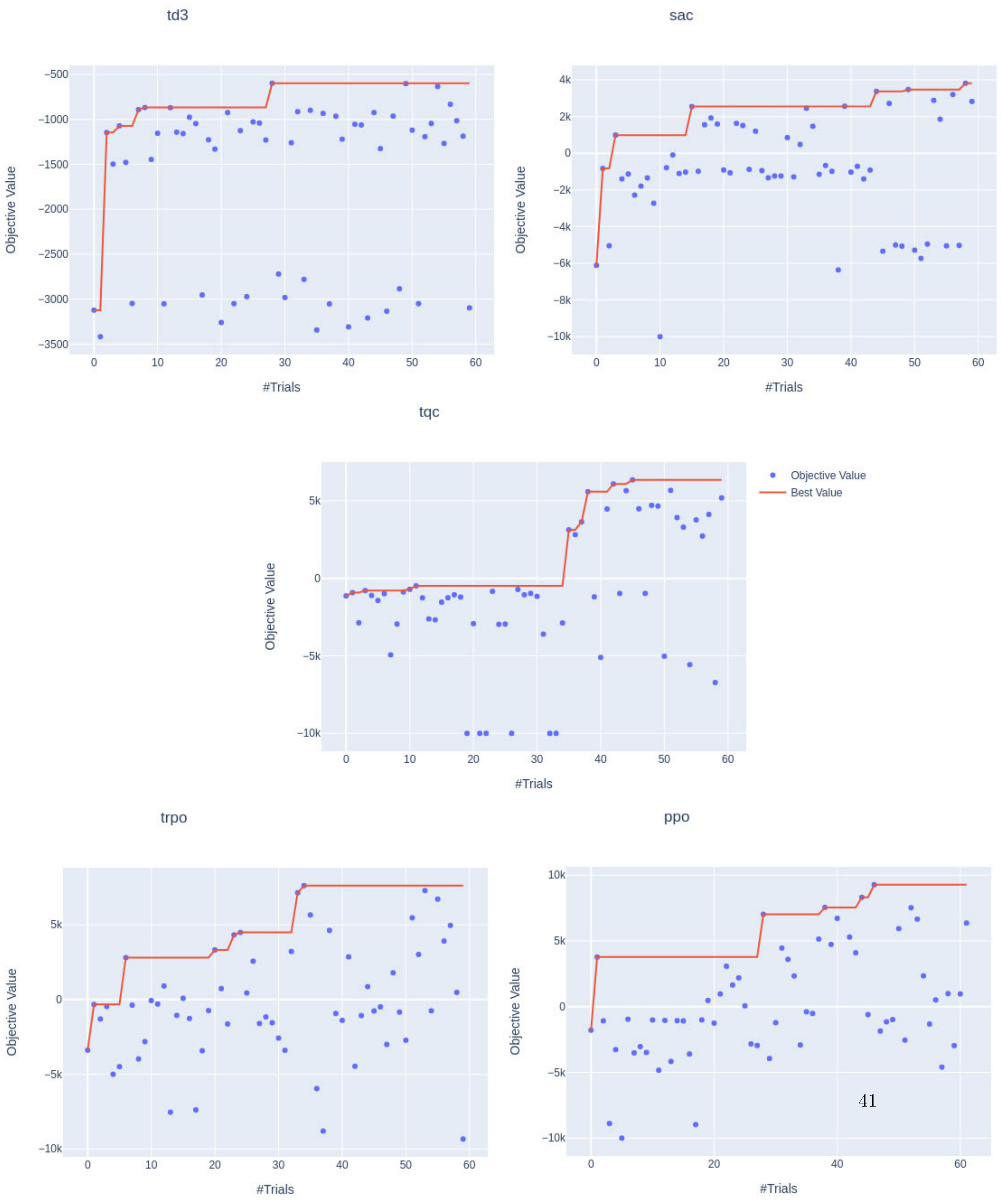


Figure 5.1: Optimization Histories

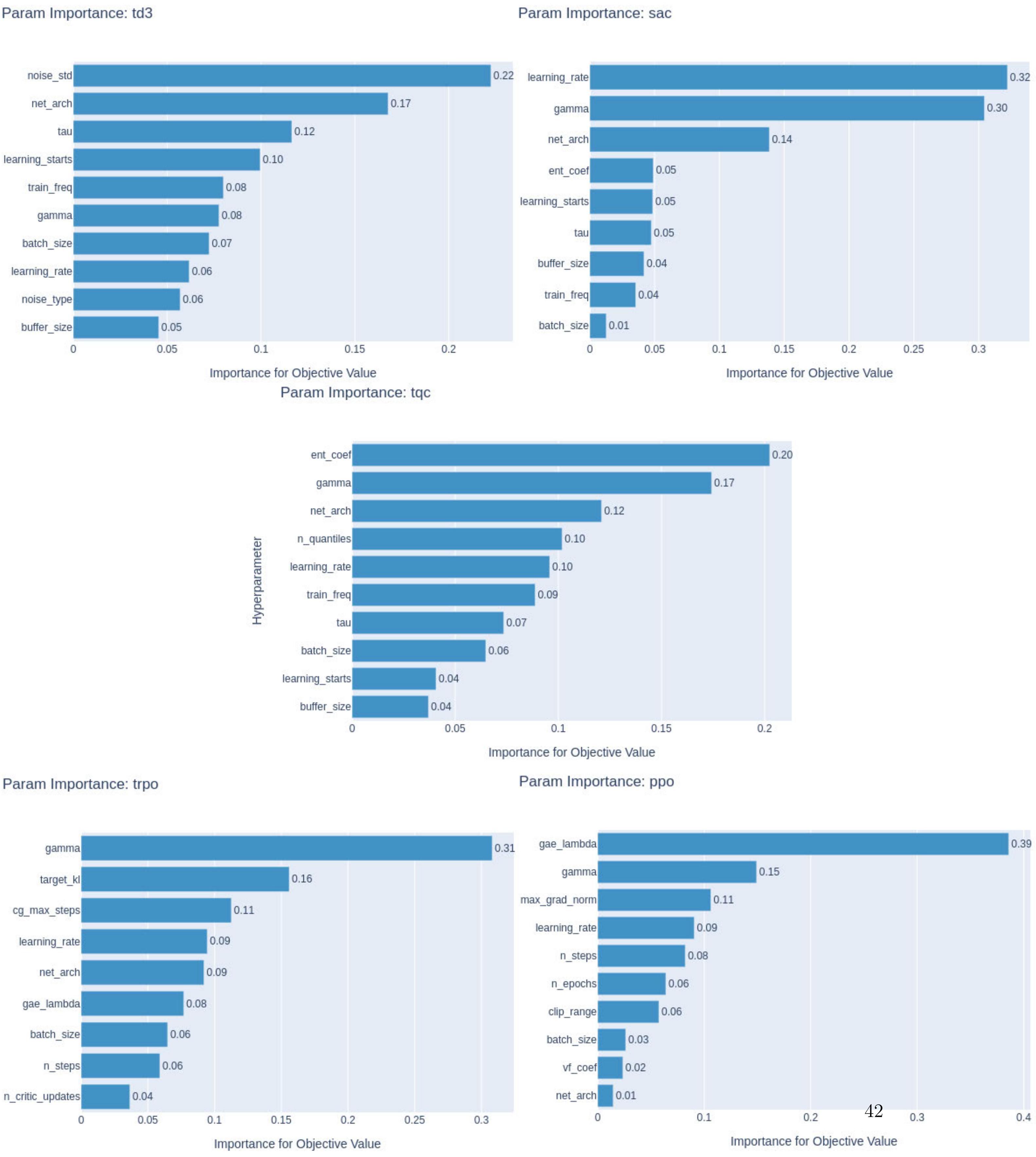


Figure 5.2: Parameter Importance

Figure 5.1 shows the results from the evaluation of the studies. The X-Axis shows the number of the trial, the Y-Axis the returned value from the objective function and the title the algorithm name. The blue dots are representing a trial with the score returned from the agent's evaluation. The red line represents the currently maximum returned value from the objective function. Note that the algorithm *PPO* has a blue dot after the 60 number. The total amount are still 60 trials, but a server-side error stopped a running trial, which was set to disabled but increased the count of later trials by one.

Figure 5.2 shows the importance of each parameter for each algorithm based on our chosen intervals for the *RosGym*. The title shows the name of the Reinforcement Learning algorithm and the X-Axis the importance in percent for the objective value, for each parameter shown on the Y-Axis. The importance means how much changing it impacted the result of the objective function, how the agent trained and his score of the evaluation.

5.1.2 Elected Trials Results

Based on figure 5.1 and looking at each goal to wall-crash difference, the best trials per study are chosen. Each of these four per algorithm is tested for their stability, by re-running them five times.

Figure 5.3 shows a bar-plot for each algorithm. Each plot contains three bars for a trial with the trials number next to it. These trial numbers represent the run with one of the top four parameter sets from the hyperparameter optimization. The blue bar of a trial shows the average score of five iterations of training and returning the evaluated score, as done in the objective function. The orange bar shows the minimum achieved return of the five iterations and the gray bar the highest return. This is done for each of the four best trial for each algorithm.

Figure 5.4 compares the highest average score achieved from the best trial, which is defined by the highest average score from five iterations of the objective function. The X-Axis shows the Reinforcement Learning algorithms name and the Y-Axis the averaged score.

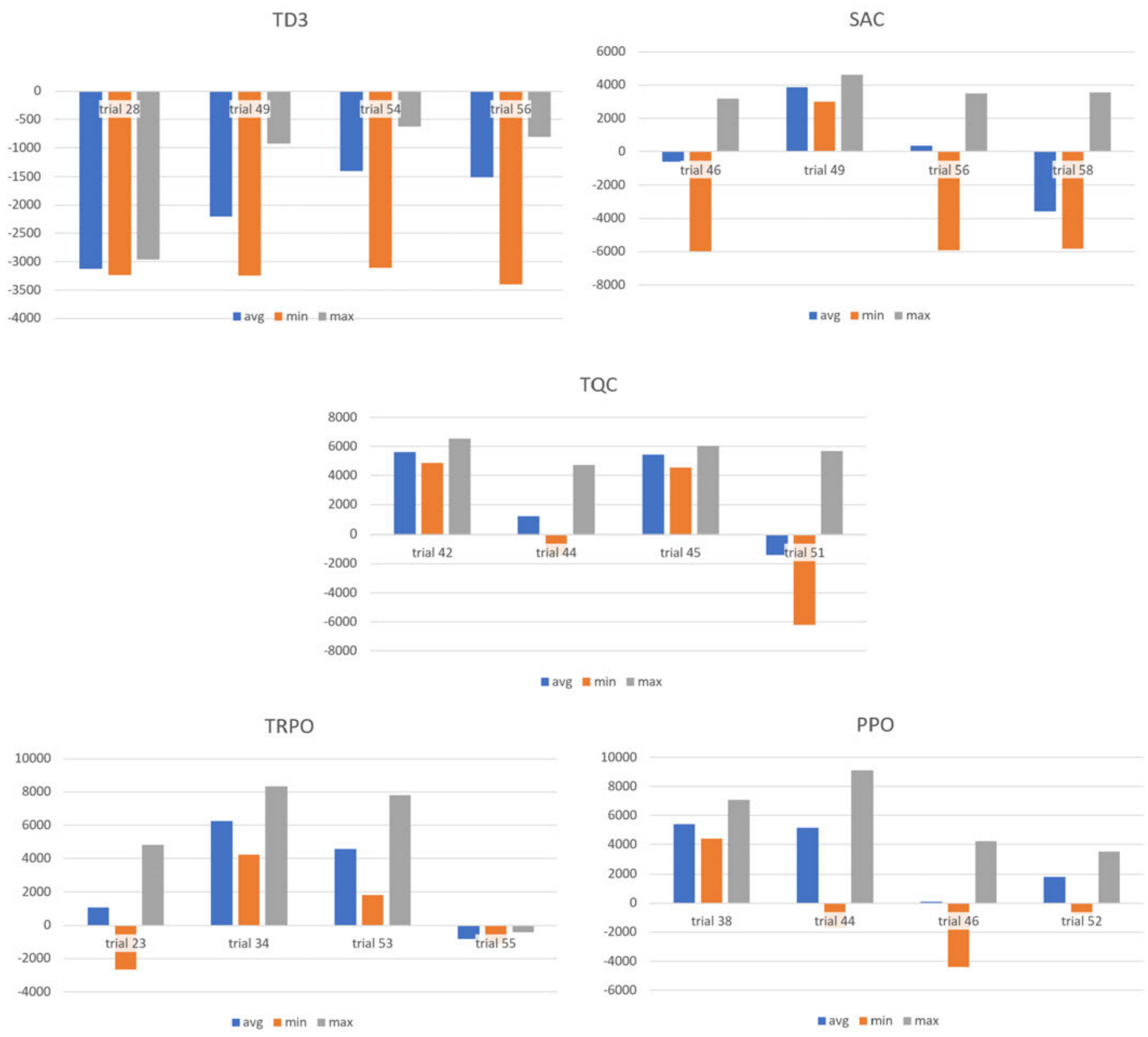


Figure 5.3: Reevaluation Scores

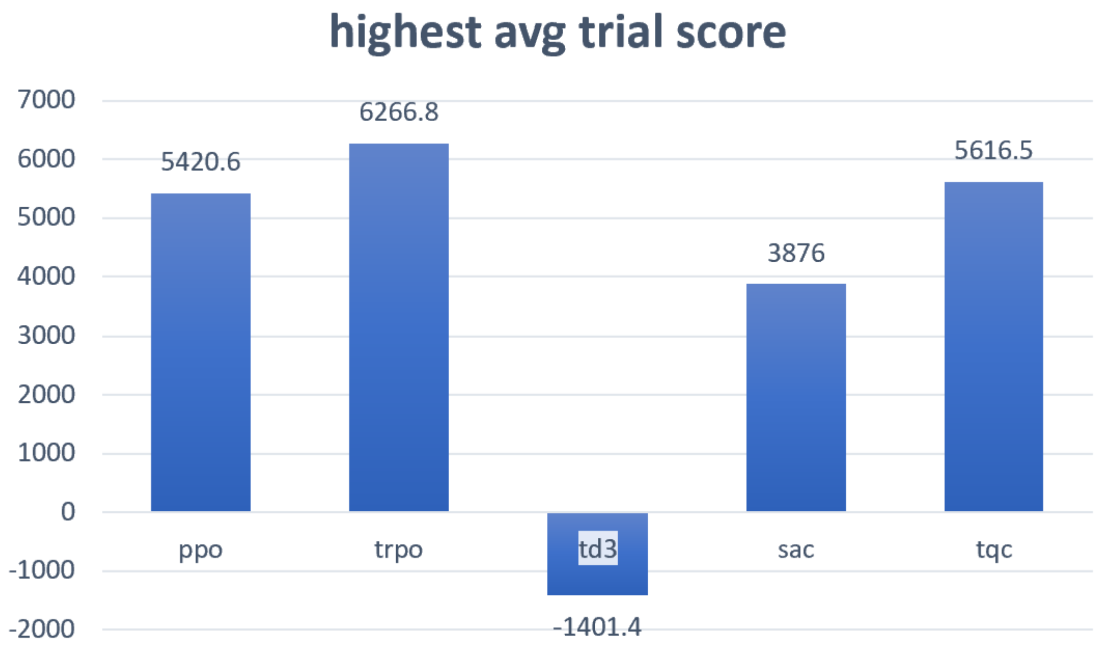


Figure 5.4: The Highest Average Reevaluation Scores

Section 5.1.2 shows the hyperparameters of the best trial per algorithm, and thus the best parameters for each algorithm for our testing environment, based on the respective ranges of parameters and number of trials.

- *TD3*:
  - batch_size: 8
  - buffer_size: 40000
  - gamma: 0.98
  - learning_rate: 0.0000037
  - learning_starts: 0
  - net_arch: [400, 300]
  - noise_std: 0.63978348
  - noise_type: None
  - tau: 0.005
  - train_freq: 32
- *SAC*:
  - batch_size: 64
  - buffer_size: 400000
  - ent_coef: 0.1
  - gamma: 0.95
  - learning_rate: 0.000538
  - learning_starts: 5000
  - net_arch: [256, 256]
  - tau: 0.01
  - train_freq: 64
- *TQC*:
  - batch_size: 128
  - buffer_size: 40000
  - ent_coef: 0.1
  - gamma: 0.9
  - learning_rate: 0.0001153
  - learning_starts: 0
  - n_quantiles: 28
  - net_arch: [400, 300]

  - tau: 0.005
  - top_quantiles_to_drop_per_net: 8
  - train_freq: 1

- *PPO*:
  - batch_size: 8
  - clip_range: 0.3
  - gae_lambda: 0.8
  - gamma: 0.95
  - learning_rate: 0.000047
  - max_grad_norm: 2
  - n_epochs: 1
  - n_steps: 1024
  - net_arch: [400, 300]
  - vf_coef: 0.02424428

- *TRPO*:
  - batch_size: 8
  - cg_max_steps: 5
  - gae_lambda: 0.95
  - gamma: 0.98
  - learning_rate: 0.000072
  - n_critic_updates: 5
  - n_steps: 256
  - net_arch: [400, 300]
  - target_kl: 0.01

5.1.3 Best Trials Training Results

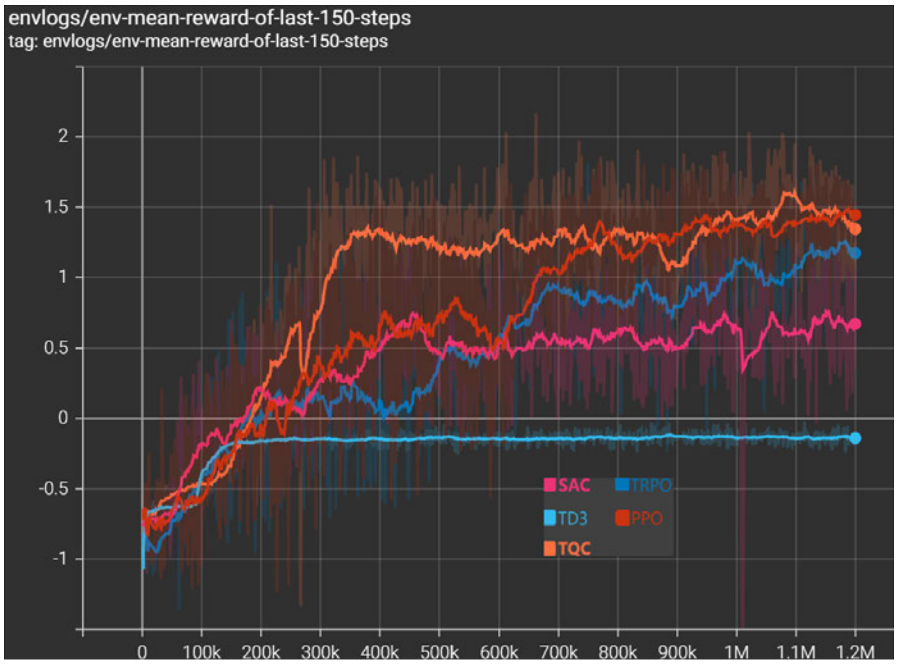


Figure 5.5: Mean Reward Plot

Based on the stability testing, by re-running each promising parameter set five times, we show an in-depth view of each best parameter set's training, starting with the reward received while training in figure 5.5. The X-Axis shows the current step of the environment, and the Y-Axis the mean reward collected over the last 150 steps, as done in chapter 3. The graphs color shows the Reinforcement Learning algorithm that runs with their respective best trials parameters. The graphs are smoothed by an exponential moving average filter with a weight of 0.95.

Figure 5.6 presents the mean episode reward achieved by the algorithms. It calculates the mean for the next point by averaging over each received episode reward. The graphs are again smoothed by an exponential moving average filter with a weight of 0.95. Note that *TD3* is not represented here, meaning that the agent did not drive into a wall, nor reached the goal.

Followed after the mean rewards and mean episode rewards received, we show important loss plots in figure 5.7, once for the critics that try to create a value representation of the states, and once for the actors, that learn a behavior policy based on the critic's representation. The plots show the step size on the X-Axis and critics loss on the Y-Axis.

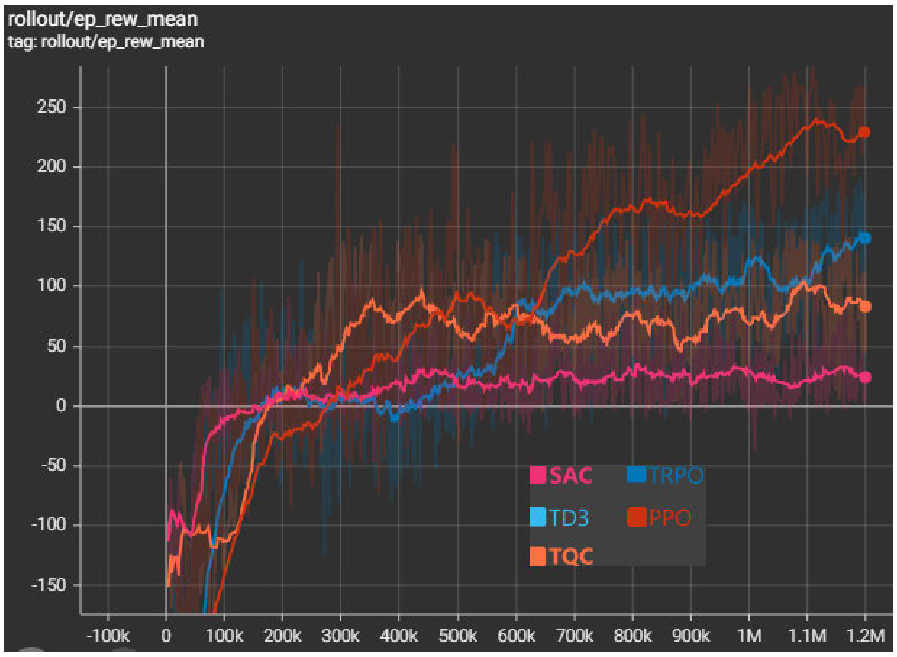


Figure 5.6: Mean Episode Reward Plot

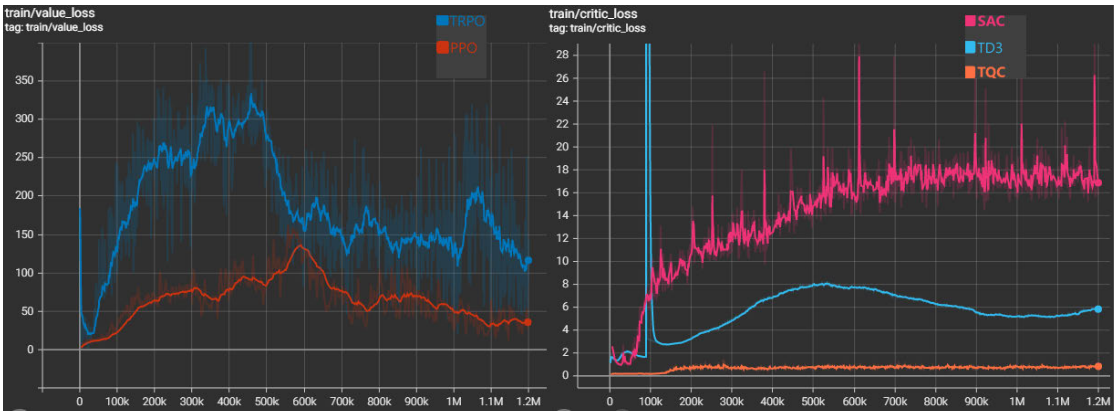


Figure 5.7: Critic Losses

The left graphs are smoothed by an exponential moving average filter with a weight of 0.9 and the right by 0.6. These losses represent the error in the critic while predicting the value of a state, based on the TD-error. If the agent collects the reward that the critic predicted, then this value goes down. E.g., if the agent collects a higher than expected reward after taking an action in a state, this value goes up, followed by a bigger update to the function approximation with the current learning rate, to train the critic to predict it better next time.

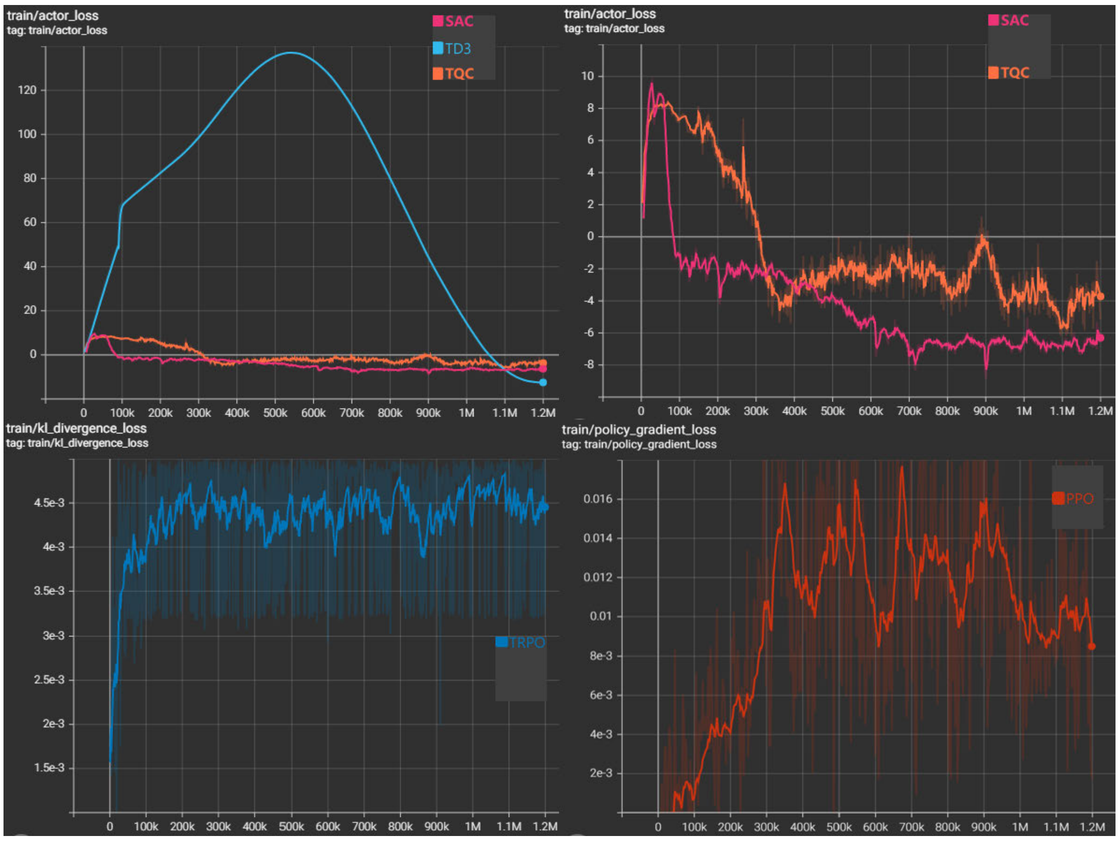


Figure 5.8: Actor Losses

The X-Axis of each plot in figure 5.8 shows the current step in the environment and the Y-Axis the loss. These plots show the losses of the agent's actor, where the top left plot shows the *Off-Policy* actor losses. The top right shows the same, but removes *TD3* to show a closer up view between *SAC* and *TQC*. The bottom left plot shows the KL-Divergence loss instead of a direct actor loss, which represents the mean of the KL-Divergence on updating, since it updates the policy by maximizing the policies objective through a constraint optimization search. The bottom right plot shows the policy gradient loss of *PPO*, representing how much the policy is changing on updates. This directly correlates with the clipping of the policy. This loss going down indicates that the policy takes actions that are expected from the critics state representation and does not need to update much, since the probabilities of taking actions in a state are already close to the advantage calculated from the critic. The top plots are smoothed the same way as the critic loss plots, the *Off-Policy* ones with a weight of 0.6 and the *On-Policy* plots by 0.9.

5.1.4 Exploration

To finalize the results of the best parameter set's evaluation, we present how they explored the lab room while training and evaluating. Figure 5.9 shows a rotated and mirrored

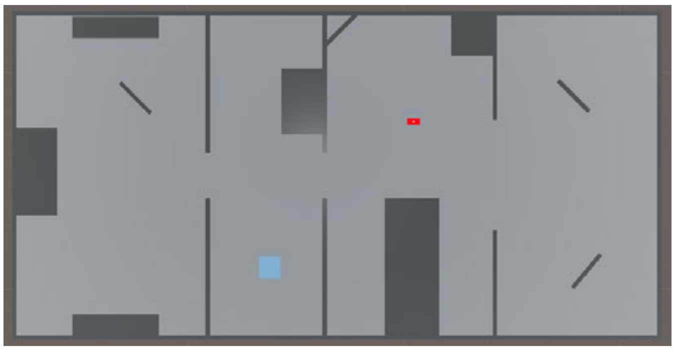


Figure 5.9: Lab Room 2D View

view of the training environment, with the red object being the robot and the blue field the goal.

The following plots, figure 5.10, figure 5.11, figure 5.12, figure 5.13 and figure 5.14, are a 2D view of the agents current position. The title of the plot shows the algorithm name and in which step they were taken with a sample size of 240.000 steps, to show the difference in exploration in later training stages. The bottom right plot shows the positions of the agent while evaluating. A red dot represents the agents starting position, a darker blue dot the agent's goal point. The cyan dots are the agents position, meaning that he was at that spot in between the time frame.

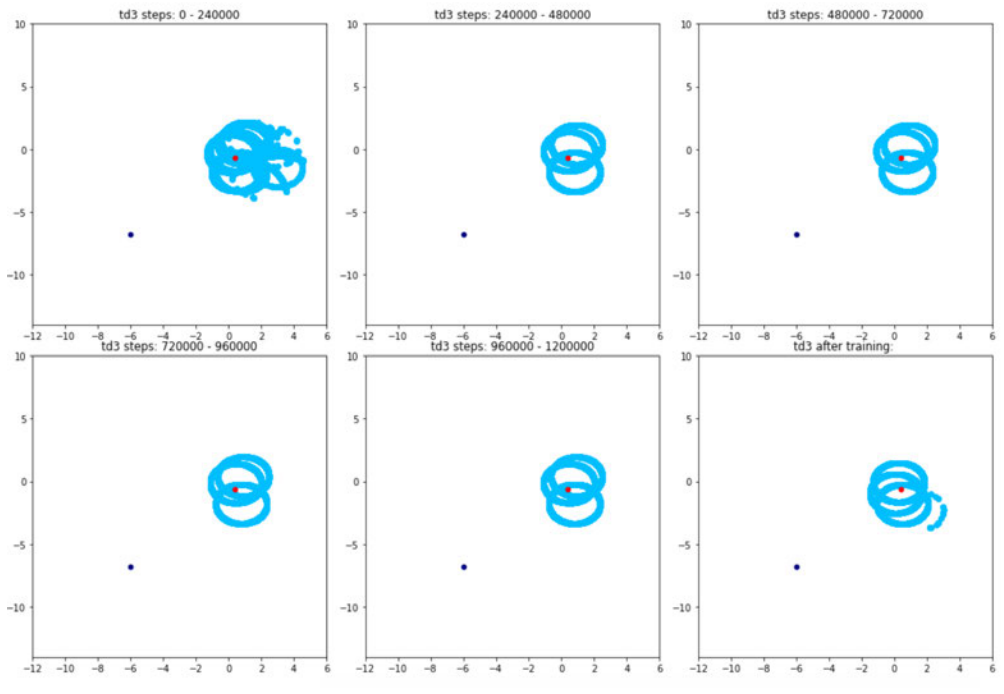


Figure 5.10: TD3 Exploration Footprint

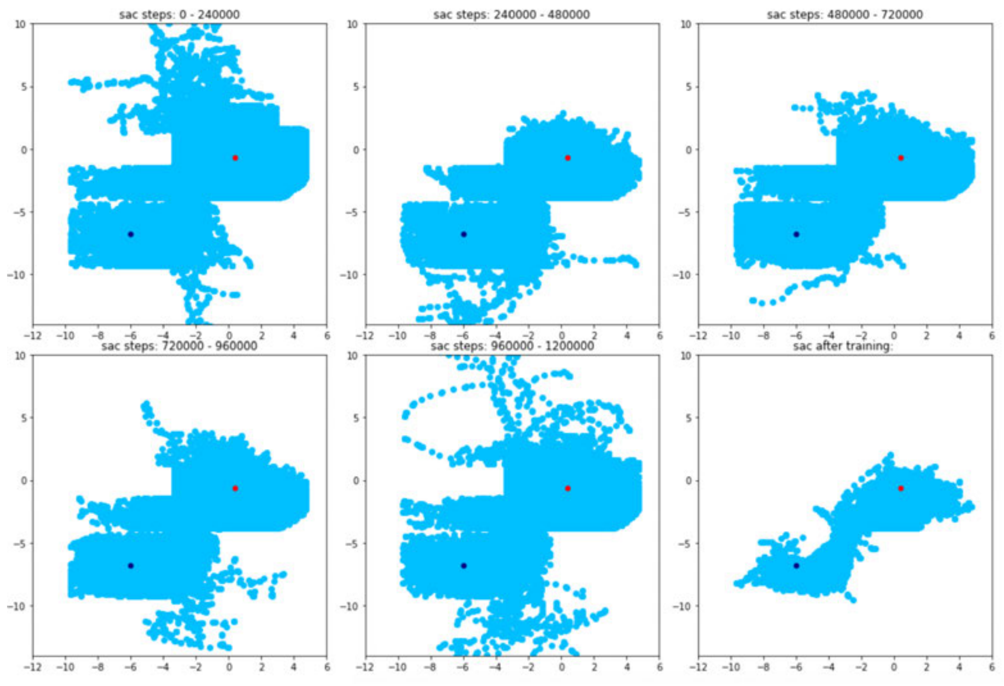


Figure 5.11: SAC Exploration Footprint

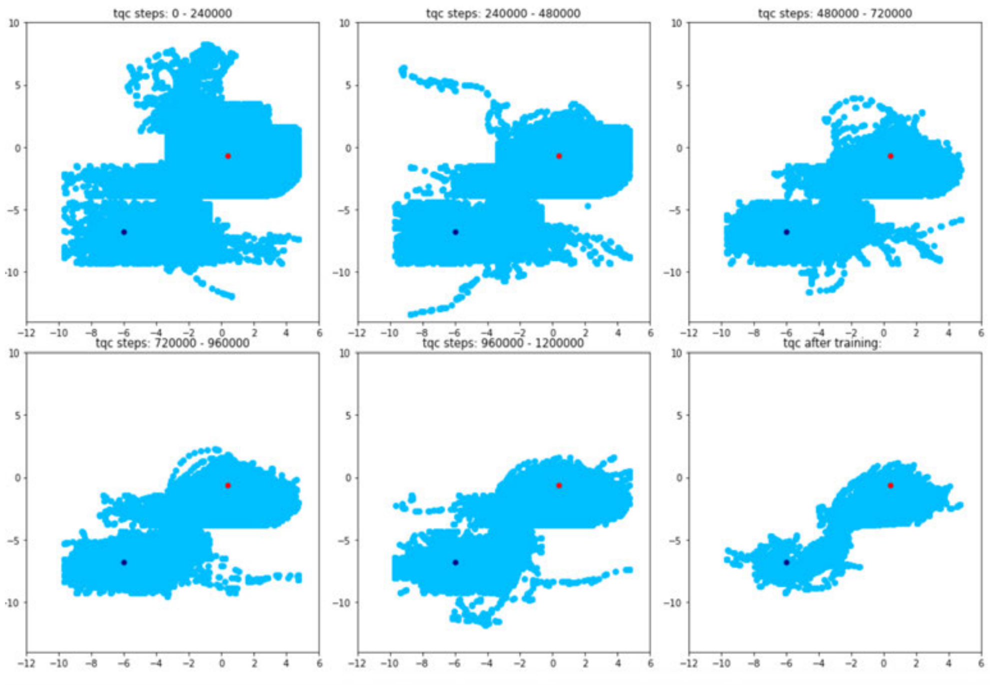


Figure 5.12: TQC Exploration Footprint

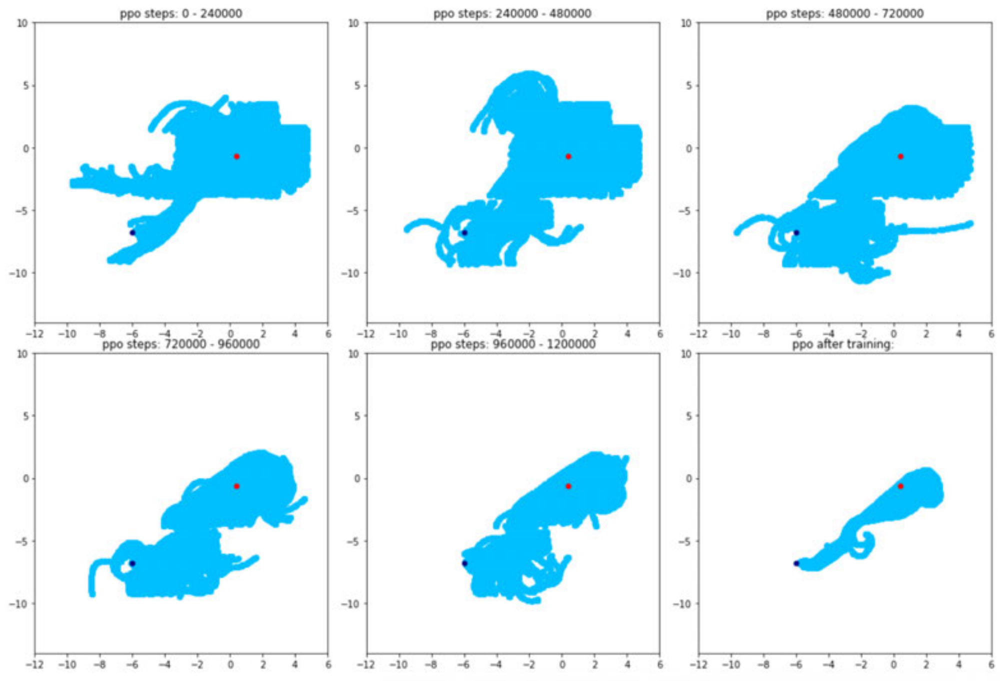


Figure 5.13: PPO Exploration Footprint

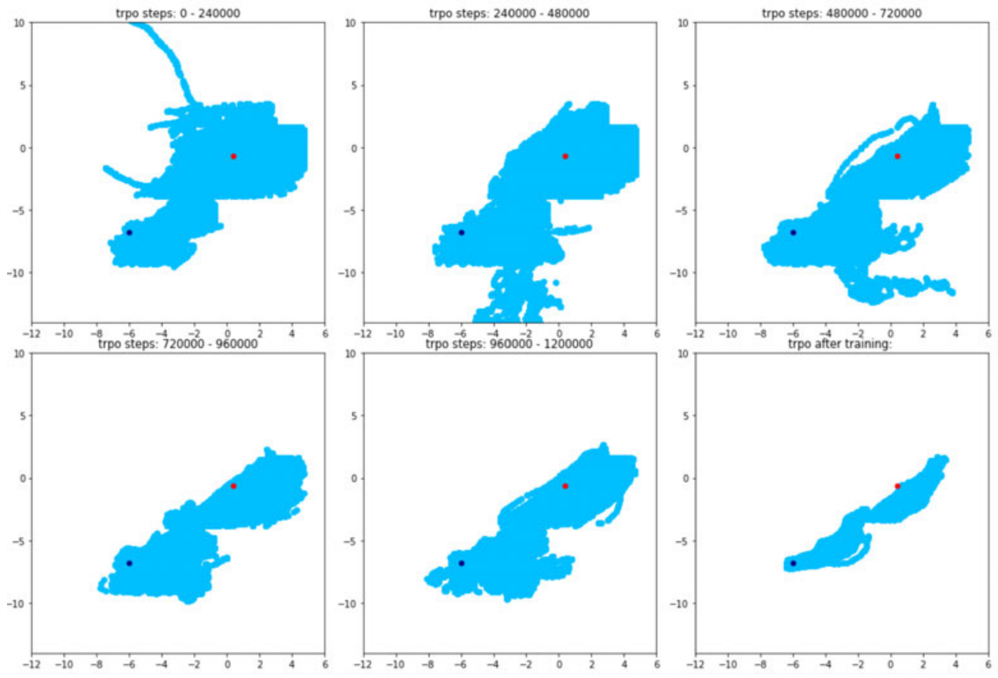


Figure 5.14: TRPO Exploration Footprint

6 Discussion

In this section, we discuss the results of the experiments from the previous section. These results are the foundation of analyzing the behavior of each algorithm and are used to answer our predictions made in chapter 4. We discuss our presented structure and answer which algorithm we predict to be good and suitable for a continuous control environment.

6.1 Hyperparameter Optimization Evaluation

The experiments start with a structured way to find a good set of hyperparameters for each algorithm in a given time, in our case 60 trials, where each trial trains an agent for 400.000 steps with three parallel simulations.

6.1.1 Optimization History Evaluation

We start with evaluating our hyperparameter optimization. In the optimization history plots in figure 5.1 we evaluate the general pruning of *Optuna*, if the framework can find a good set of hyperparameters with our given parameter ranges. Each study finds better functioning trials over time. In the *TD3* study, no trial finds a set which results in achieving a positive score, while the rest do, meaning that the algorithm is not suitable for our environment, at least given our parameter ranges. The other studies find a set with a positive score and improve over time as well, with *TQC* finding a good trial only after 30 trials. It indicates that this algorithm had a harder time finding a good set and parameters had to be finely tuned, but after finding a good set, many more trials resulted in a positive score, proving that *Optuna* works. *PPO* and *TRPO* both have a lot of trials with good scores and are generally more broadly scattered around, while the *Off-Policy* algorithm have more tight groups of trials with similar performances. They either find a good set that works, or a set that doesn't and the performance stays relatively the same. The *On-Policy* algorithms on the other hand can be tuned more fine-grained and get lots of different trial scores. It shows that they are less hard to tune and why *PPO* performed good as an environment evaluation, many parameter sets just work.

6.1.2 Parameter Importance Evaluation

The following figure 5.2 shows the parameter importance for each study, to see which parameter for which algorithm impacted the result of the objective function the most.

Starting with *TD3*, the biggest impact for the agent was made through the *noise_-std* parameter, controlling the noise, the *net_arch* parameter, controlling the size of the agents policy and value function networks and *tau*, the weight factor for the target network updates. The noise in particular has the highest impact of 22% on the agent's performance, since it controls the exploration of the agent. What we did not expect was the low importance for the *noise_type* parameter, which contains the option to use no noise at all. It seems that this algorithm failed to learn a good policy with either a noise active or no noise active at all, meaning that the exploration through added noise to the action was not enough to find a good policy. Since our environment rewards based on the correctness of both actions, an added noise results in a high probability that the reward will be negative. Furthermore, the calculation of the TD-error in *TD3* uses the noise for calculating the max Q value, which results in a high probability of a wrong max Q value. This kind of exploration is thus not very suitable for our environment.

SAC had a strong importance for the *learning_rate*, that controls the neural network update step size and *gamma*, that makes the agent short- or long-sighted, both around 30%. The first one makes sense, since it controls how much the agent updates, but looking at the other algorithms, no other seems to give that much weight to it. It hints that the pruner tried a wider range of the learning rate and the results had a big difference in performance on *SAC*. The importance of other parameters is thus perceptually less important in this study, in particular *ent_coef*, controlling the stochasticity and thus the exploration of the agent, but is still the fourth most important parameter. *gamma* also had a lot of impact, since it controls how fast the agent tries to reach the goal. This parameter has to be tuned specifically for each environment, represented by its high parameter importance in each study that had successful trials.

TQC gave the most weight to the *ent_coef* parameter, since it has to explore a lot to learn the distribution of rewards for the states. The *n_quantiles* parameter, controlling how many quantiles for the reward distribution are learned per action in a state, is also important with 10%. *TQC* seems to be harder to tune, given that the importance for the parameters are all closer together and that the optimization took a lot of trials to find a suitable set.

TRPO gave next to *gamma* with 31% a lot of weight to the *target_kl* parameter having 16% and the *cg_max_steps* parameter having 11%. These are the two parameters controlling the learning of the agent through the KL-Divergence constraint optimization and thus impact the agent's performance a lot.

PPO gave a lot of weight to *gae_lambda* with 39%, considerably impacting the agent's performance through changes of said parameter. *gae_lambda* weighs in how much the multistep returns are weighed in while calculating the advantage of the agent in a current state. This advantage is directly bound to the agent's maximization task with the clipping added while updating, resulting in a big impact of the agent's performance. Interestingly, the *net_arch* parameter had little to no impact for this algorithm, proving further why it is a good algorithm to test environments or to have a good working reinforcement learning algorithm. It does not depend on a very specific neural network size and in our case worked with every of our chosen sizes.

6.2 Elected Trials Evaluation

Picking the four best trials for each study and reevaluating them multiple times, we see that each algorithm but *TD3* found a suitable set for our environment that had a stable performance, always getting a positive score, seen in figure 5.3. This was very important, since a lot of run to run variance can impact the performance, seen by how many trials had a very good run but a bad one, e.g., trial 44 from *PPO*. The best run in said trial had the highest score achieved by any algorithm, but contained a run with a negative score and generally had a lower average score, making this parameter set too noisy. In figure 5.4 we can see that *PPO*, *TRPO*, *TQC* and *SAC* had a trial with only positive scores of the multiple re-evaluations. *PPO*, *TRPO* and *TQC* had a similarly high average score on their best trial set, with *TRPO* having the highest. This parameter set from *TRPO* outperformed every other algorithm, proving our assumption of *PPO* and *TQC* being the best ones for our environment as false. It seems that directly optimizing the KL-Divergence, without the approximation made through the clipping as done in *PPO*, results on average in finding a better policy.

Looking at the parameter sets values that achieved the best average score in section 5.1.2, we can see that *gamma* was on every algorithm a rather lower value of or under 0.98, indicating that our environment was a shortsighted problem, where he received the most reward through getting fast to the goal. This happens through the rewarding system that rewards fast movement and the big reward on reaching the goal. Every good running algorithm used the higher neural network size of 400 and 300 neurons for this environment, hinting at a rather complex approximation problem. Another similarity is that a high entropy of 0.1 resulted in a better policy for *SAC* and *TQC*, showing the importance of exploration in our environment. The *Off-Policy* algorithms learned the best with a smaller batch size, since the current transitions are used directly to update the agent, while *SAC* and *TQC* used a higher batch size to have a bigger pool of transitions from the memory buffer. *TD3*, as seen in figure 5.2, did not learn well with either a noise active or without and the best run used no noise while training.

6.2.1 Best Trials Training Evaluation

Following with an in depth view of the agent's performance on the best trials in figure 5.5, we see how *TQC* learned a good policy the fastest and kept it stable. It did not improve much afterwards but found it very fast, due to the sample efficiency from *Off-Policy* learning. *SAC* on the other hand did not find a good policy very fast, but in a reasonable time, and like *TQC*, stagnated around an average value of the last 150 steps. *TD3* completely failed to learn a policy and stagnated on a value quickly, not getting an average score above zero. The *On-Policy* algorithms had a different behavior, they did not stabilize on a set value but continued to get an improved score, further shown in the followed image of the episodic mean rewards, figure 5.6. *PPO* continuously improves, getting a higher reward per episode after time and receiving the highest mean reward while training the agent, resulting in an agent's policy that takes a longer time driving to the goal, but collecting a lot of reward on the way. *TRPO* seems to stagnate for some time, but after around 400.000 steps, getting to a higher plateau. The general reward collected over the last 150 steps increases for *TRPO*, as in *PPO*, continuously over time. In general, we have a behavior similar to what we saw in the optimization history. The *Off-Policy* algorithms stagnate on a value, while the *Off-Policy* algorithms have a smoother learning curve and thus cover more different scores. In the policy learning theorem, the policy always improves after each update and the new policy is kept close to the old policy in the cases of *TRPO* and *PPO*. This behavior can be seen in the stable and small improvements over time. What is interesting to look at is how fast *TQC* found a very good policy. If the sample efficiency of our environment would be low, e.g., we train on a real robot, then this algorithm would be, as predicted, a very good choice. In just 350.000 steps, the algorithm performed on average from the last 150 steps as good as *PPO* on around 780.000 steps, being more than twice as fast. Looking at the trend, *PPO* and *TRPO*, with their continuously improving scores, are proving to be a better choice for longer training sessions, especially seeing how they improve the episodic reward over time.

Looking at the losses for the agents critics in figure 5.7, we see that each of the algorithms performed well in learning to predict the value of a state. Each algorithm stagnates after around 800.000 to 900.000 steps. The agents actors behave similarly, stagnating on a value at around 400.000 steps, seen in figure 5.8. *TD3* is an outlier, since the actor loss gets really high and the policy is not finding a lot of rewards. In general, each good learning algorithm has a stable loss curve for their respective actors and critics, meaning that they have a stable learning experience, as represented before by the scores achieved.

6.2.2 Exploration Evaluation

To finalize the behavior analysis of the algorithms, the exploration results are evaluated.

TD3 learned a policy that just drives in a circle, avoiding walls to not get punished. Seeing that the agent directly learned to drive like that, after around 240.000 steps, means that he exploits the current value representation and does not find a way out of the current behavior. This is generally a failure of exploration and shows that, as predicted, the way of exploring with random noise is not suitable in our environment.

SAC explored a lot, with the *ent_coef* being on the highest value of 0.1 from our respective ranges. The exploration seems to narrow down after some time, explained by the agent learning a good value representation, but increases at the end of training. This happens due to the sudden spike seen in the figure 5.5 in around 1.01 million steps area, where the agent had a negative spike in performance, increasing the exploration again. After training, where the policy is set to deterministic in the *Off-Policy* case while evaluating, the agent had a narrowed down movement and learned a good policy. He was still close to walls and not narrow enough to achieve higher scores, but still learned to get to the goal reasonably well.

TQC, with the same *ent_coef* as the best trial in *SAC*, explored a lot on every stage of training and learned a similarly narrow path to the goal. The higher score that it achieved must mean that the agent drove faster and more often to the goal, with a generally better movement then *SAC*, but due to the learned entropy with a similar noisy and big footprint.

The *On-Policy* algorithms explore directly through the stochasticity of their policy and their exploration views are very similar. We can see the improvement of the policy directly on each time window, by the narrowed down footprint over time, and on the evaluation, the policy finds a very narrow footprint in reaching the goal for both algorithms. This, however, highlights the problem of the *On-Policy* algorithms at the same time. If an agent's policy changes for the worse, it could get stuck in a bad policy, in our case on a narrow path, that only worsens the policy with bad collected data that does not contain enough information to improve the policy. A problem stated in section 4.2.3. On the other hand, on a good trained agent, we get the behavior we prefer, a narrow path to the goal with a smaller footprint.

The *Off-Policy* algorithms, especially due to the entropy coefficient, keep a bigger exploration footprint while training and evaluating.

6.3 Verdict

Starting with our optimization, we can see that *Optuna* improved the hyperparameters over time with multiple trials. We found a good parameter set with our more limited range of hyperparameters and showed that they are suitable in finding a good set to solve our task. Our way of reevaluating the best trials shows how much a run to run variance can impact the results and why this part of our work was important to further analyze the algorithms performances and behavior. The authors of the algorithms papers claimed a lot of contradictory results, where this big run to run variance could have impacted their results by a lot. We see that *PPO* for example had the best single run in the re-evaluation of trial 44, while the rest of runs in that trial were not as close as good. Putting them in perspective with our own controlled environment and giving each algorithm the same amount of training time to find a suitable set, with averaging over multiple runs, we build an even playing field that gave us a more clear perspective on their performance and behavior.

Looking at our predictions made in section 4.3, we see that they were mostly confirmed by our results. *TD3* did not learn a good policy in our environment, where the other *Off-Policy* algorithms did. *TQC* was a very good upgrade of *SAC* through the introduction of distributional reinforcement learning, as claimed by the authors of *TQC* [18], and achieved a better performance. They both learned very sample efficient, especially *TQC*, that learned very fast and achieved an average score as high as the *Off-Policy* algorithms twice as fast while training. Their performance while training was not as smooth as the ones from the *Off-Policy* algorithms and stagnated. On the *Off-Policy* side, *PPO* and *TRPO* learned a policy as good as *TQC* and had a stable, continuously improving performance. We predicted a similar score from both of the *Off-Policy* algorithms, but surprisingly, *TRPO* outperformed *PPO* and achieved the highest average score in the evaluation.

Our final verdict is to use *TQC* for a sample inefficient continuous control task. It was hard to tune for our environment, but given our approach, one should be able to find a good parameter configuration to use it for a continuous control task, learning a good policy fast. If the environment is sample efficient, one should use *PPO* or *TRPO*. Given how important the network architecture is for *TRPO* in respect to *PPO* and how easy *PPO*'s hyperparameters are to tune, *PPO* proves to be the easier algorithm to use and to get running for an environment. Looking at our results however, *TRPO* outperformed every algorithm on average and should be considered and tested on a sample efficient environment first. They learned slower than *TQC*, that even outperformed *PPO* on average, but the agents had a stable and continuously improving learning curve. Especially the presented footprint showed us how they learned a policy that narrowed down the path to the goal after each time-window for our continuous control problem and that precise movement was more important than learning a policy fast.

7 Outlook

We finalize our work by providing a short summary, mention related work and discuss future expansion.

7.1 Summary

Summing up the thesis, we created an environment to analyze continuous control algorithms using *ROS* and *Unity*, where a robot has to drive to a goal while avoiding walls, and validated it with OpenAI's default Reinforcement Learning algorithm *PPO*. We highlighted important characteristics of our environment, most notably that it is a model-free continuous control task that needs precise actions from an agent to get rewards, has a high sample efficiency and a large state space. Based on that, we concluded to analyze model-free actor critic algorithms, *On-Policy* and *Off-Policy* ones. For the *Off-Policy* algorithms, we picked *TD3*, an algorithm that provides a robust implementation of *Q-Learning* for continuous control, *SAC*, that adds an entropy term to make the learned policy stochastic, and *TQC*, that extends *SAC* by predicting the distribution of possible Q-Values instead of a flat value. Our *On-Policy* actor-critic algorithms chosen were *TRPO*, that uses a KL-divergence to not update the policy too much, and *PPO*, that uses a clipping parameter to approximate the same behavior in a computational faster and less complex way. Given their concepts, we predicted that *TRPO* and *PPO* will have the most stable learning behavior and perform the best, but take more time to learn a good policy then the *Off-Policy* algorithms. We predicted that *SAC* and *TQC* will perform similarly to *TRPO* and *PPO* through the added entropy, but have a less stable learning curve, while *TQC* itself outperforms *SAC* through the addition of learning a distribution. We predicted *TD3* to perform the worst due to the lack of stochasticity in the learned policy and how it explores. Our final prediction was to use *PPO* in sample efficient continuous control environments and *TQC* in sample inefficient ones. To answer these predictions and further analyze the behavior of the algorithms, we used the hyperparameter optimization framework *Optuna*. With the best parameter sets found in our given time and parameter ranges, we reevaluated them with multiple runs, to reduce the run to run variance, and picked the ones with the highest average score. We showed an in depth view of the best runs performance from the picked trials of through the loss functions of the actor and critics and plots from the rewards collected while training. To give perspective about their exploration behavior, we added a 2D view of the training environment, containing their positions from training sections and the evaluation as points. We discussed the hyperparameter optimization results first, that *Optuna* found better parameters for each algorithm over time, proving it as a good way to search for functioning parameters. Our results showed the importance of the parameters from each algorithm as well, showing us that the discount factor played the biggest role in achieving a good performance, especially for the *On-Policy* algorithms. The *On-Policy* algorithms were the easiest to tune and continued to achieve better performance over the training duration, while the *Off-Policy* algorithms were harder to tune, learned faster but had a stagnation in their performance. Looking at the exploration footprints, *SAC* and *TQC* had a noisy exploration through the entropy term and their evaluation footprint was still relatively big in perspective to the *On-Policy* algorithms. The *On-Policy* algorithms had a footprint that narrowed down per training step and a very narrow path to the goal at the evaluation. As predicted, *TD3* did not manage to perform good on our environment and learned a policy that just drove in circles avoiding walls and never reached the goal. *TQC* learned a good policy twice as fast as the *On-Policy* algorithms and outperformed *SAC*, making it the best choice for a sample inefficient environment. *PPO* performed as good as *TQC* after training and was the easiest to tune. *TRPO*, in contrary to our predictions, outperformed every algorithm tested on average. Even though *TQC* learned a good policy very fast, the smaller footprint was an important factor for our environment, since the robot learned a more robust path to the goal, and makes the *On-Policy* algorithms the better choice for our task and is our choice for sample efficient environments.

7.2 Related Work

[10] presents a benchmark suite containing multiple continuous control tasks. They benchmark multiple algorithms, containing *TRPO*, *REINFORCE* and *DDPG*.

[16] addresses the problem of reproducibility in Reinforcement Learning benchmarks in continuous control tasks. They address the problem of tuning hyperparameters, show their significance, talk about noisy results of benchmarks, environment stochasticity and general variance in Reinforcement Learning algorithms.

[15] investigates the challenges in reproducing results from the algorithms papers and highlights the problems in experimental reporting differences and algorithm performance variances, which makes it hard to value extensions of algorithms. They suggest guidelines to make future deep Reinforcement Learning algorithm reports more reproducible and comparable.

[11] is a case study about algorithm performance variances through implementation differences. They mention how *PPO* runs better through code-level optimizations that are missing in *TRPO* and is the reason why it performs better in the paper publication of *PPO*.

7.3 Future Work

This work can be extended in multiple ways.

More algorithms can be added to analyze and compare them to the ones presented here.

One could run the hyperparameter optimization more exhaustively to find out if the results change and test new parameter ranges.

We only had time for testing the algorithms on the modeled lab room with the set goal and starting point, but many more start and goal points are already programmed in, as well as two more maps. One could use them to analyze how generalized the algorithms can learn if trained on multiple stages and tasks.

Adding dynamic objects into the environment would be interesting too, it makes it more stochastic and thus more difficult for the algorithms to learn a good policy.

A very interesting extension would be to further analyze the algorithms by running them on the real Loomo Segway and compare how the simulated training translates to the real world. It is already possible to do so and tested, but not exhaustively enough, especially in an analytic context, and could be a big point of interest for future expansion.

Another interesting point would be to run the concepts presented here, with picking the right algorithm and optimizing the parameters, on other environments, to see how this approach translates to other tasks and how the results look like there.

Finally, what is missing in a lot of testing environments is a way of visualizing the exploration as done in this work, so adding metrics to visualize them and to test how similar the exploration behavior is would also be a good extension.

Bibliography

[1] *Loomo Segway.* – URL https://shop.segway.com/de_de/segway-loomo-robot.html. – Accessed: 2022-07-18

[2] Achiam, Josh: *DDPG.* 2018. – URL https://spinningup.openai.com/en/latest/algorithms/ddpg.html. – Accessed: 2022-07-18

[3] Achiam, Josh: *Spinning Up Open Ai.* 2018. – URL https://spinningup.openai.com/en/latest/. – Accessed: 2022-07-18

[4] Akiba, Takuya ; Sano, Shotaro ; Yanase, Toshihiko ; Ohta, Takeru ; Koyama, Masanori: *Optuna: A Next-generation Hyperparameter Optimization Framework.* 2019

[5] Baselines3, Stable: *SB3 Zoo.* – URL https://github.com/DLR-RM/rl-baselines3-zoo. – Accessed: 2022-07-19

[6] Bergstra, James ; Bardenet, Rémi ; Bengio, Yoshua ; Kégl, Balázs ; Shawe-Taylor, J. (Hrsg.) ; Zemel, R. (Hrsg.) ; Bartlett, P. (Hrsg.) ; Pereira, F. (Hrsg.) ; Weinberger, K.Q. (Hrsg.): *Algorithms for Hyper-Parameter Optimization.* 2011. – URL https://proceedings.neurips.cc/paper/2011/file/86e8f7ab32cfd12577bc2619bc635690-Paper.pdf

[7] Bischl, Bernd ; Binder, Martin ; Lang, Michel ; Pielok, Tobias ; Richter, Jakob ; Coors, Stefan ; Thomas, Janek ; Ullmann, Theresa ; Becker, Marc ; Boulesteix, Anne-Laure ; Deng, Difan ; Lindauer, Marius: *Hyperparameter Optimization: Foundations, Algorithms, Best Practices and Open Challenges.* 2021

[8] Dabney, Will ; Ostrovski, Georg ; Silver, David ; Munos, Rémi: *Implicit Quantile Networks for Distributional Reinforcement Learning.* 2018

[9] Dabney, Will ; Rowland, Mark ; Bellemare, Marc G. ; Munos, Rémi: *Distributional Reinforcement Learning with Quantile Regression.* 2017

[10] Duan, Yan ; Chen, Xi ; Houthooft, Rein ; Schulman, John ; Abbeel, Pieter: *Benchmarking Deep Reinforcement Learning for Continuous Control*. 2016. – URL https://arxiv.org/abs/1604.06778

[11] Engstrom, Logan ; Ilyas, Andrew ; Santurkar, Shibani ; Tsipras, Dimitris ; Janoos, Firdaus ; Rudolph, Larry ; Madry, Aleksander: *Implementation Matters in Deep RL: A Case Study on PPO and TRPO*. 2020. – URL https://openreview.net/forum?id=r1etN1rtPB

[12] Fujimoto, Scott ; Hoof, Herke van ; Meger, David: *Addressing Function Approximation Error in Actor-Critic Methods*. 2018. – URL https://arxiv.org/abs/1802.09477

[13] Haarnoja, Tuomas ; Zhou, Aurick ; Abbeel, Pieter ; Levine, Sergey: *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*. 2018. – URL https://arxiv.org/abs/1801.01290

[14] Haarnoja, Tuomas ; Zhou, Aurick ; Hartikainen, Kristian ; Tucker, George ; Ha, Sehoon ; Tan, Jie ; Kumar, Vikash ; Zhu, Henry ; Gupta, Abhishek ; Abbeel, Pieter ; Levine, Sergey: *Soft Actor-Critic Algorithms and Applications*. 2018. – URL https://arxiv.org/abs/1812.05905

[15] Henderson, Peter ; Islam, Riashat ; Bachman, Philip ; Pineau, Joelle ; Precup, Doina ; Meger, David: *Deep Reinforcement Learning that Matters*. 2017. – URL https://arxiv.org/abs/1709.06560

[16] Islam, Riashat ; Henderson, Peter ; Gomrokchi, Maziar ; Precup, Doina: *Reproducibility of Benchmarked Deep Reinforcement Learning Tasks for Continuous Control*. 2017. – URL https://arxiv.org/abs/1708.04133

[17] Juliani, Arthur ; Berges, Vincent-Pierre ; Teng, Ervin ; Cohen, Andrew ; Harper, Jonathan ; Elion, Chris ; Goy, Chris ; Gao, Yuan ; Henry, Hunter ; Mattar, Marwan ; Lange, Danny: *Unity: A General Platform for Intelligent Agents*. 2018. – URL https://arxiv.org/abs/1809.02627

[18] Kuznetsov, Arsenii ; Shvechikov, Pavel ; Grishin, Alexander ; Vetrov, Dmitry: *Controlling Overestimation Bias with Truncated Mixture of Continuous Distributional Quantile Critics*. 2020

[19] Lillicrap, Timothy P. ; Hunt, Jonathan J. ; Pritzel, Alexander ; Heess, Nicolas ; Erez, Tom ; Tassa, Yuval ; Silver, David ; Wierstra, Daan: *Continuous control with deep reinforcement learning.* 2019

[20] Mnih, Volodymyr ; Kavukcuoglu, Koray ; Silver, David ; Graves, Alex ; Antonoglou, Ioannis ; Wierstra, Daan ; Riedmiller, Martin: *Playing Atari with Deep Reinforcement Learning.* 2013. – URL https://arxiv.org/abs/1312.5602

[21] NVIDIA: *RTX Titan.* – URL https://www.nvidia.com/de-at/titan/titan-rtx/. – Accessed: 2022-07-19

[22] OpenAI: *OpenAi SpinningUp RlIntroduction.* – URL https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html. – Accessed: 2022-07-19

[23] OpenRobotics: *Gmapping.* – URL http://wiki.ros.org/gmapping. – Accessed: 2022-07-18

[24] OpenRobotics: *ROS wiki.* – URL https://wiki.ros.org/. – Accessed: 2022-07-18

[25] OpenRobotics: *Why Ros.* – URL https://www.ros.org/blog/why-ros/. – Accessed: 2022-07-18

[26] Raffin, Antonin ; Hill, Ashley ; Gleave, Adam ; Kanervisto, Anssi ; Ernestus, Maximilian ; Dormann, Noah: *Stable-Baselines3: Reliable Reinforcement Learning Implementations.* 2021. – URL http://jmlr.org/papers/v22/20-1364.html

[27] Schulman, John ; Klimov, Oleg ; Wolski, Filip ; Dhariwal, Prafulla ; Radford, Alec: *openai-baselines-ppo.* – URL https://openai.com/blog/openai-baselines-ppo/. – Accessed: 2022-07-19

[28] Schulman, John ; Levine, Sergey ; Moritz, Philipp ; Jordan, Michael I. ; Abbeel, Pieter: *Trust Region Policy Optimization.* 2017

[29] Schulman, John ; Wolski, Filip ; Dhariwal, Prafulla ; Radford, Alec ; Klimov, Oleg: *Proximal Policy Optimization Algorithms.* 2017

[30] Silver, David ; Hubert, Thomas ; Schrittwieser, Julian ; Antonoglou, Ioannis ; Lai, Matthew ; Guez, Arthur ; Lanctot, Marc ; Sifre, Laurent ; Kumaran, Dharshan ; Graepel, Thore ; Lillicrap, Timothy ; Simonyan, Karen ; Hassabis, Demis: *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm.* 2017

[31] Silver, David ; Lever, Guy ; Heess, Nicolas ; Degris, Thomas ; Wierstra, Daan ; Riedmiller, Martin: *Deterministic Policy Gradient Algorithms.* 2014

[32] Sutton, Richard S. ; Barto, Andrew G.: *Reinforcement learning: An introduction.* MIT press, 2018. – URL http://incompleteideas.net/book/the-book.html

[33] team fsstudio: *ZeroSimROSUnity.* – URL https://github.com/fsstudio-team/ZeroSimROSUnity. – Accessed: 2022-07-18

[34] Zhang, Junzi ; Kim, Jongho ; O'Donoghue, Brendan ; Boyd, Stephen: *Sample Efficient Reinforcement Learning with REINFORCE.* 2020. – URL https://arxiv.org/abs/2010.11364

Erklärung zur selbstständigen Bearbeitung

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

		[redacted]
Ort	Datum	Unterschrift im Original