INF
Informatik

Hochschule Reutlingen
Reutlingen University

Hochschule Reutlingen

Reutlingen University

– Digital Business Engineering –

Master–Thesis

Reward Function for Industrial Robot Trajectory Planning with Reinforcement Learning in a Virtual Commissioning Environment

Dustin Walker

Betreuer:	Prof. Dr. ING. Christian Decker
Zweitbetreuer:	Prof. Dr. Alexander Rossmann
Abgabedatum:	30.09.2022



Abstract

To improve the cost effectivenes of industrial automation for medium sample sizes, Reinforcement Learning is utilized to learn the robot control program in a virtual commissioning environment. The goal of this thesis is to design the reward function so that the RL agent learns an optimized trajectory as fast as possible. The reward function incentivizes the agent to solve the task of moving to a target position whilst avoiding obstacles. The reward function is iteratively developed and evaluated based on three performance aspects. The structure of the reward function is motivated by directed experimentation. Reward shaping is utilized and parameters are optimized to achieve the best possible performance. The result is a reward function and accompanying RL system that learns the task after just about 50 000 steps. The experiments and the discussions point out many areas to improve the system.

Kurzfassung

Um die Kosteneffektivität der industriellen Automatisierung bei mittleren Stichprobengrößen, zu verbessern wird Reinforcement Learning eingesetzt, um das Robotersteuerungsprogramm in einer virtuellen Inbetriebnahme-Umgebung zu erlernen. Das Ziel dieser Arbeit ist es, die Belohnungsfunktion so zu gestalten dass der RL-Agent so schnell wie möglich eine optimierte Bewegung erlernt. Die Belohnungsfunktion incentiviert den Agenten die Aufgabe, eine Zielposition anzufahren und dabei Hindernisse zu vermeiden, so schnell wie möglich zu erlernen. Die Belohnungsfunktion wird iterativ entwickelt und anhand von drei Leistungsaspekten bewertet. Die Struktur der Belohnungsfunktion wird durch zielgerichtete experimente motiviert. Reward Shaping wird eingesetzt und die Parameter werden optimiert, um die bestmögliche Leistung zu erreichen. Das Ergebnis ist eine Belohnungsfunktion und ein dazugehöriges RL-System, das die Aufgabe nach nur etwa 50 000 Schritten erlernt. Die Experimente und die Diskussionen zeigen viele verbesserungsmöglichkeiten auf.

Acknowledgement

I would first like to thank my thesis advisor Prof. Dr. ING. Christian Decker at Reutlingen University for his close support of my thesis. In all of our regular meetings, I got impactful insights and guidance towards making great research. I appreciate the valuable technical insights I got from his Ph.D. student Sebastian Kotstein. He served as a great sounding board to improve my methodology for my RL experiments.
I am very grateful that I had the opportunity to write my thesis with the company iT Engineering Software Innovations GmbH. I received all the support and resources I needed to do my research. Especially I want to thank the CEO Wolfram Schäfer.
I could not have undertaken this journey without my mentor Patrick Helbig, who was always by my side guiding me throughout the project. He gave me great insights, helped me debug my code and suggested many improvements after proofreading my thesis.
Many thanks to Florian Jaensch (from ISW) for his research on the project and providing the foundation for my work.
I would like to extend my sincere thanks to my colleagues Lars Emkes and Marvin Häußermann for their support in extending the digital twin in the virtuos software.
Special thanks to Institut für Steuerungstechnik der Werkzeugmaschinen und Fertigungseinrichtungen (ISW) at Stuttgart University and Industrielle Steuerungstechnik GmbH (ISG) for the great cooperation in the research project.

Contents

1 Introduction **1**

2 Foundations **3**
2.1 Markov Decision Process . . . 3
2.2 Reinforcement Learning . . . 3
2.3 Fundamentals of Reward Functions . . . 5
2.3.1 Simple Sparse Reward . . . 5
2.3.2 Reward Shaping and Dense Reward Function . . . 5
2.3.3 Reward Functions for Hierarchical Reinforcement Learning . . . 6

3 Methodology **7**
3.1 The Simulation Environment . . . 7
3.2 Limitations of the Simulation Environment . . . 7
3.3 The Experiments Setup . . . 8
3.4 The Two Training Scenarios . . . 9
3.5 Framework for Evaluating the Performance of a Reward Function . . . 11

4 Move to Position without Obstacle **15**
4.1 Sparse Reward . . . 16
4.2 Simple Negative Dense Reward . . . 16
4.3 Reward Shaping with Linear and Non-Linear Reward . . . 19
4.4 Dynamically Reduced Terminal Reward Over Time . . . 21

5 Move to Position with Obstacle **25**
5.1 Defining the Setup for the New Training Scenario . . . 25
5.2 Reward Function with Penalty for Crashing . . . 27
5.3 Reward Function with Linear Proximity Penalty . . . 28
5.4 Reward Function with Polynomial Proximity Penalty . . . 30
5.5 Analyzing the Intra Episode Performance . . . 32
5.6 Agent Purposefully Makes Itself Blind to Obstacles . . . 34

6 State of the Art for Industrial Robot Programming with RL **37**

7 Findings and Observations **41**
7.1 Findings for the First Scenario . . . 41
7.2 Findings for the Second Scenario . . . 42
7.3 Parameters for the Reward Function . . . 44
7.4 Other Observations . . . 45

8 Conclusion and Future Work **47**
8.1 Conclusion . . . 47
8.2 Future Work . . . 48

Acronyms **53**

Glossary **55**

Selbstständigkeitserklärung **57**

List of Figures

2.1 The agent–environment interaction in a Markov decision process [1, p. 48] 4
2.2 A taxonomy of HRL approaches. by Pateria et al. [2, p. 8] 6

3.1 Reinforcement Learning with a virtual commissioning system 8
3.2 KUKA KR 10 R1100 sixx, as Screenshot from the ISG-virtuos Visualization 10
3.3 Neural Network architecture for the first scenario 10
3.4 Performance metrics for second order feedback control system (Haldia Institute of Technology, [3]) 12
3.5 The three main aspects for evaluating an Reinforcement Learning (RL) algorithm, defined by an example experiment, (rolling average over 200 steps) 13

4.1 Two experiments with sparse reward. The robot reaches the goal only once indicated by a non-zero reward at step 49 500 16
4.2 Experiment with negative reward; 1000 terminal reward 17
4.3 Experiment with negative reward; 1000 terminal reward; total steps in episode . . . 17
4.4 Experiment with negative reward; 0 terminal reward; total steps in episode 18
4.5 Experiment with negative reward; 0 terminal reward, $\frac{1}{10}$ learning rate; total steps in episode 18
4.6 Visualization of different reward functions using reward shaping 19
4.7 Experiment with linear and non-linear reward; (rolling average over 3000 steps) . . . 20
4.8 Experiment with linear and non-linear reward; (rolling average over 3000 steps) . . . 21
4.9 Convergence and stability test with the linear reward function; the agent does not learn to wait in front of the target 22
4.10 Dynamically reduced reward vs. baseline (rolling average over 3000 steps) 23
4.11 Experiment to examine dynamically reduced reward (rolling average over 3000 steps) 24

5.1 New Scenario with wall and floor as an obstacle; new start and target position . . . 26
5.2 Robot with distance sensors 27
5.3 Experiments with -200 penalty for crashing 27
5.4 Distance penalty for one distance sensor visualized 28
5.5 Experiments with linear distance penalty 29
5.6 Linear distance penalty, agent avoids obstacles at all cost 29
5.7 Visualization of linear and polynomial distance penalty 30
5.8 Experiment with a non linear distance penalty with max penalty $= -0.1$ 31
5.9 Comparison between different values for the max distance penalty 31
5.10 The distance between the Tool Center Point (TCP) and the target position for the last 10 000 steps in a training run with a max distance penalty of -0.1 32
5.11 The distance between the TCP and the target position for the last 10 000 steps in a training run with crashing disabled and no distance penalty 33
5.12 The distance between the TCP and the target position for the last 10 000 steps in a training run with crashing disabled, no distance penalty and only 6 inputs to the Q-function 33

5.13 The agent purposefully makes itself blind to obstacles. 34

Chapter 1

Introduction

Traditionally, industrial robots are programmed using online programming, where a technician is teaching in positions on the physical robot with a teach pendant. With the advent of the digitalization of the factory, the industry uses offline programming, where a technician is writing a program for the robot outside the production environment.
The programs written with offline programming can be tested in a virtual commissioning environment. With virtual commissioning, the production environment is simulated in software so that control systems for machine tools and industrial robots can be tested and validated.
Virtual commissioning has improved production productivity due to it eliminating production downtime during programming and testing. But the robots and control systems are still mostly programmed by skilled technicians. This is not a problem if the robot only has to be programmed once a year, but if the product variety increases, the control system has to be reprogrammed for every single variant. Due to this problem, medium sample sizes are not cost-efficient to automate.
The ambitious goal of the government-funded project "KI-Steuerung" [4] is it to utilize a virtual commissioning software as a training environment for a Reinforcement Learning algorithm and learn the control policy for a specific task over night. This thesis is part of the research project. The foundation on which it builds is summarized in the publication [5].
The focus of this thesis is the development of the reward function for the Reinforcement Learning (RL) algorithm. The reward function encodes the specific task and incentivizes the RL agent to behave in such a way that the specified goal is achieved. The scenario considered in this thesis is a single 6 Degrees of Freedom (DoF) robot with a gripper as a hand. The task for the robot to learn is to move to a position whilst avoiding obstacles. The **goal of this thesis** is to design the reward function so that the RL agent learns an optimized trajectory as fast as possible. For the iterative development and evaluation, the developed reward functions are first tested in a simpler scenario without obstacles and then later in the full scenario with obstacles.
First, the foundations for RL and reward functions are explained so that later design choices can be reasoned based on the fundamentals.
In the Methodology chapter, the ISG-virtuos virtual commissioning software, the experiment setup and the two scenarios are presented. Following that, a framework for comparing the performance of an RL agent during a training run is presented.
The main content of this thesis is about developing the reward function in an iterative process of experiments. This content is split into two parts representing two scenarios. In the first scenario, the agent is supposed to move from a start position to a target position. In the second scenario, an obstacle (wall), collision detection and distance sensors are added to the environment. Here the agent is supposed to move from a start position to a target position whilst avoiding obstacles.
To compare the work and design decisions in this, relevant literature is reviewed in chapter6.
Finally, the results and insights are presented in the conclusion and opportunities for future work are suggested.

Chapter 2

Foundations

In this chapter, the foundations for Reinforcement Learning and reward functions most relevant to this thesis are presented. All terms and concepts, necessary to understand the explanations in later chapters, shall be explained in the following. The experiments depicted in this thesis are trained with the DQN algorithm implementation in Stable-Baselines3 [6]. Therefore Q-Learning will be explained in greater detail.

2.1 Markov Decision Process

The Markov decision process (MDP) is a mathematical concept, that provides a framework for simulating decision-making processes. This framework can handle situations where the transitions are partly random and partly determined by the decision-maker. The concept of MDP has been known since at least the 1950s [7]. Nowadays, they are used in many disciplines including economics, automatic control manufacturing and robotics. The MDP is the foundation for all RL algorithms. An MDP is based on a system (environment) with discrete time steps. In each step, a decision-maker (agent) observes the state (S) of the environment and chooses an action (A). Based on the current state and the chosen action the next state is calculated. The calculation of the next state is defined as a transition. Transitions can be non-deterministic, meaning that two transitions (T) with the same input (S, A) may not result in the same new state (S'). After the transition, the Reward (R) is calculated, based on the last state, the chosen action and the resulting state. Based on these symbols, a MDP can be defined as a tuple $\mathcal{M} = \langle S, A, T, R, \gamma \rangle$ where S is a finite set of states, A is a finite set of actions, $\gamma \in (0, 1]$ is the discount factor, $T : S \times A \times S \to [0, 1]$ is the transition probability distribution, and $R : S \times A \times S \to \mathbb{R}$ is the reward function. [8] As explained by Puterman ET al. [9], "The decision maker's objective is to choose a sequence of actions called a policy, that will optimize the performance [cumulative reward] of the system over the decision making horizon. Since the action selected at present affects the future evolution of the system, the decision maker cannot choose his action without taking into account future consequences "
The discount factor γ is used to discount the expected reward in the future. This helps to make the decisions of the agent more explainable and more predictable. The need for a discount factor will be motivated in a later section. The terminology and symbols used above will be used to explain and motivate the design choices in this thesis.

2.2 Reinforcement Learning

Reinforcement Learning is a type of Machine Learning, that is based on MDPs. Sutton ET al. [1, p. 53ff] defines it as follows: "The learner and decision-maker is called the agent. The thing it interacts with, comprising everything outside the agent, is called the environment. These interact continually, the agent selecting actions and the environment responding to those actions and presenting new situations to the agent. The environment also gives rise to rewards, special numerical values that

the agent tries to maximize over time." These relationships are presented in figure 2.1

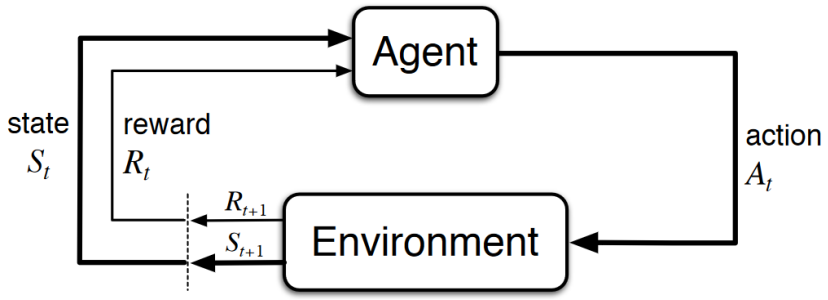


Figure 2.1: The agent–environment interaction in a Markov decision process [1, p. 48]

The environment can be a simulation or it can be a system in the real world with a digital interface. Usually, a simulation is used, as this speeds up the training process. The agent performs actions in the environment and gets signals back from the environment, as described in the MDP.
Reinforcement Learning works with discrete steps. At each step, the agent gets some information from the state of the environment, also called an observation. An observation can, but does not have to, contain all the information captured in the state (S). Based on the observation, the agent chooses an action (A). With the chosen action, the next state (S') in the environment is calculated. Based on the previous state, the chosen action and the resulting state, the reward function calculates the reward $R : S \times A \times S \to \mathbb{R}$. Afterward, the next step is initiated and the cycle starts again with the agent receiving the new observations and choosing an action.
One cycle of these steps is called an episode. After an episode, the environment is reset and a new episode starts. The goal in RL is to train an agent to choose actions in such a way, that it maximizes the cumulated reward over all steps in an episode, called the return. To achieve this, the agent needs to learn a fitting behavior (action) for each situation (state). This behavior is modeled by the agent's policy (π).
RL algorithms are categorized into two kinds of algorithms. There are Off-policy algorithms and On-Policy algorithms. On-Policy algorithms can only learn from experiences gathered using the most recent policy. Off-Policy algorithms can learn from all experiences, no matter the policy used while gathering them [1]. Off-Policy algorithms are more sample efficient because they can do multiple back propagations on the same experience.
A popular kind of Off-Policy algorithm is Q-Learning, also called Value-Learning. This algorithm will be described in the next sections.

Q-Learning

With Q-Learning, the algorithm does not try to learn the best policy directly. The goal is to learn a Value function (Q-function), that can predict the return based on the current state and the chosen action, assuming all future actions are chosen optimally. The optimal policy can then be derived from the Q-function by always choosing the action with the highest expected return.
Tabular Q-learning [10] is one of the first implementations of the Q-learning approach. In tabular Q-learning the Q-function is modeled by a table ($\tilde{q}(s, a)$), that assigns a value (q) to each possible state (s) action (a) pair. During the training, the table gets incrementally populated and updated according to the experiences collected in the environment. Unfortunately, tabular Q-learning is not feasible for solving problems with large state spaces, because the value table has an entry for all unique state-action pairs.
Deep Q-Networks (DQN) proposed by Mnih et al. (2015) [11] is an approach, where the Q-function ($\tilde{q}_\theta(s, a)$) is approximated with a deep neural network with parameters θ. The neural network is trained using mini-batches of experiences (s, a, r, s'), where s' represents the next state. These mini-batches are randomly sampled from the experience replay buffer and used to minimize the

square error between $\tilde{q}_\theta(s,a)$ and the Bellman estimate $r + \gamma\ max_{a'}\tilde{q}_\theta(s,a)$ [7]. To stabilize the learning, the updates are made with respect to a target network with parameters θ' [12].

2.3 Fundamentals of Reward Functions

In this section, the fundamentals for reward functions are explained, so that future design decisions can be motivated based on the fundamentals. At each step, the reward function $R : S \times A \times S' \to \mathbb{R}$ gives the agent a reward, based on the previous state (S), the action(A) the agent chose, and the resulting state (S'). The purpose of the reward function is to give feedback to the agent on how well it is performing. The reward function is supposed to give a greater reward if the agent is in a more desirable area of the state space. The reward function rewards the agent for good actions with a higher reward and penalizes it with a lower reward for bad actions. During the training process, the agent will try to maximize the cumulative sum of rewards in one episode, called the return.
The reward function is one of the trickiest concepts in RL. Rather, than verbally explaining to the agent the goal you want it to achieve, the goal needs to be mathematically encapsulated in the reward function. The reward function needs to be designed in such a way, that maximizing it is equal to achieving the intended goal. Extra care needs to be taken, not to encode an idea of how a task should be solved, as this restricts the agent to solve the problem in a particular way, limiting its capability. To address this challenge, Sutton Et al. [1] give the following advice: "The reward signal is your way of communicating to the agent what you want achieved, not how you want it achieved."

2.3.1 Simple Sparse Reward

A sparse reward is the simplest way of defining the reward function. The agent gets zero reward on every step. Only, when the agent reached a state, that satisfies the defined goal, the agent gets a positive reward. Usually, the goal is a terminal state, meaning the episode ends at this step. For the scenario in this thesis, the agent would only get a reward, if it reaches the target position. Because a sparse reward is so simple, it is easy to ensure that maximizing the return will also satisfy the goal the developer envisions. There is little risk of encoding a bad idea of how the agent is supposed to solve the problem in the reward function. The problem is, that a sparse reward function is significantly harder for the RL agent to learn because it gives only little feedback.
Q learning is mathematically proven to converge to the optimal solution if every state is visited infinite times and the neural network approximating the value function is complex enough [10]. This means using sparse rewards will result in a value function, that eventually converges to a good policy. The problem is, that using sparse rewards is not sample efficient. It may take an impractical amount of time to reach an acceptable result. The problem of sample efficiency with sparse rewards can be tackled using hindsight experience replay [13]. The paper explaining the HER algorithm is presented in chapter 6.

2.3.2 Reward Shaping and Dense Reward Function

With reward shaping, the RL agent gets intermediate rewards to guide it to the goal. With the industrial robot scenario in this thesis, a dense reward could be a small reward at every step, that is larger, the closer the robot is to the goal. With such a reward function the agent gets more granular feedback from the environment and is therefore able to learn faster with fewer experiences needed.
Reward shaping and a dense reward function improve the sample efficiency drastically, as demonstrated in later experiments. The challenge is to strike a good balance between guiding the agent's exploration towards sensible areas of the state space and not limiting the agent's exploration, prohibiting it to find the best solution. Another problem is that the agent can find a way to achieve

the intermediate rewards without actually getting closer to achieving the actual goal. Reward functions using reward shaping are therefore more susceptible to reward gaming. Tsvarkaleva Et al. [14] define reward gaming as follows: "Reward gaming occurs when the agent's reward function is misspecified and the agent can achieve a high reward by altering or fooling, in some way, its sensors rather than by performing the desired actions."

2.3.3 Reward Functions for Hierarchical Reinforcement Learning

Hierarchical Reinforcement Learning (HRL) is important for solving more complex tasks, which consist of multiple elementary subtasks. Pateria Et al. [2] describes HRL as follows: " Hierarchical Reinforcement Learning (HRL) decomposes a long-horizon reinforcement learning task into a hierarchy of subproblems or subtasks such that a higher-level policy learns to perform the task by choosing optimal subtasks as the higher-level actions. " The survey on HRL by Pateria et al. presents multiple different kinds of HRL as seen in figure 2.2. The survey divides HRL into three dimensions Multiple Task vs Single Task, with subtask discovery vs without Subtask discovery and Multiple Agents vs Single Agent.

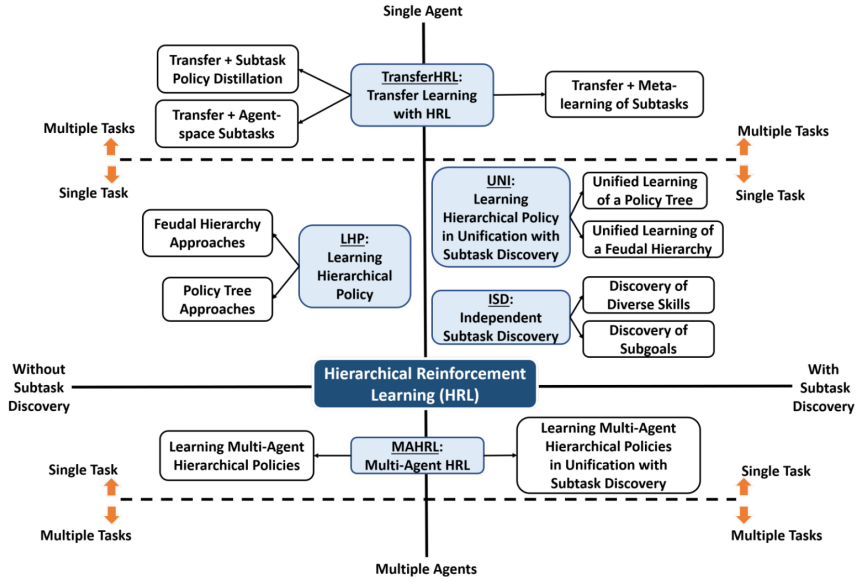


Figure 2.2: A taxonomy of HRL approaches. by Pateria et al. [2, p. 8]

Recently, there are further advancements in the field of reward machines put forth by Icarte et al. [8, 12, 15]. Reward machines are a concept in hierarchical RL, where the Reward function is represented by a state machine. Great progress has been achieved, by using the state as a kind of memory the agent has access to.
HRL will be necessary to tackle future challenges in the research project [4]. Robot control program can consist of multiple subtasks like: "Move to position", "Grip object", "Drop object at position". HRL can be utilized to learn these subtasks effectively and piece them together, to form a complete robot control program. Due to time constraints, the application of HRL techniques will not be discussed further in this thesis.

Chapter 3

Methodology

In this chapter, the methodology and the setup for the experiments are described. First, the virtual commissioning software, and its limitations are presented. Then it is explained, how this software is utilized as the environment and how the RL code interacts with it. Then the two scenarios, used for the experiments in chapter 4 and chapter 5 are presented. Finally, a framework for evaluating the performance of a reward function is introduced.

3.1 The Simulation Environment

The ISG-virtuos virtual commissioning software is used as the virtual environment to simulate the behavior of the industrial robot and its behavior. For every step, it gets the chosen action (A) and then simulates the environment for one timestep to return the new state (S').
The ISG-virtuos virtual commissioning software is a product of one of the three partners in the research project [4]. It is not yet optimized to work well as a simulation environment for RL. Optimizing it to work well as a RL environment is part of the research project.
The ISG-virtuos virtual commissioning software has an advantage over typical RL environments for robotics, it is specifically designed for validating an PLC program against a virtual twin of the production environment. It is in use by many manufacturing companies as their virtual commissioning software for creating digital twins of their production environment and validating the their production setup and PLC programs. The promise is, that if a program behaves correctly in the simulation, it will behave correctly in the real world. The unique selling proposition of ISG-virtuos against other virtual commissioning software is its Real-Time capabilities. It uses the Windows Real-Time system to simulate all inputs and outputs to the programmable logic controller (PLC) so that the PLC does not know if the real hardware, or virtuos is connected to it.
Further information on how the virtual commissioning software is adapted to the use as a RL environment can be read in the puplication "Virtual Commissioning Simulation as OpenAI Gym - A Reinforcement Learning Environment for Control Systems" [5] by Jaensch et al.

3.2 Limitations of the Simulation Environment

In RL, the simulation speed is very important. The most astonishing RL models have been achieved using a very efficient and fast simulation environment, that is parallelized and runs on multiple machines [16–19]. In addition to achieving many simulation steps per second, some papers also train their algorithm for many days and even weeks, to achieve superhuman performance.
At the time of writing, the simulation speed of the environment at hand poses serious limitations to the RL algorithm. The pure simulation of the NC-system is quite efficient. The calculations for determining the robot positions are executed in a separate executable called the Virtuos Solver. The bottleneck in the simulation is the communication between the Python script, the Virtuos (GUI), and the Virtuos Solver. The Virtuos virtual commissioning software could simulate industrial robots

faster than wall clock time, but the communication layer produces bugs. Therefore the simulation is limited to running in wall clock time. Currently, the system can simulate approximately 6800 steps per hour (one step is a small incremental movement of one robot axis). A further limitation is, that the simulation environment is not yet parallelized, to run multiple times on multiple machines. In contrast, OpenAI in their paper Learning Dexterous In-Hand-Manipulation [17] implemented a training environment, with a pool of 384 worker machines, each with 16 CPU cores. This enables them to generate 2 years of simulated experiences per hour [17, p. 7].
Due to the speed limitations of the simulation environment, the complexity of the training scenario needs to be reduced to a minimum.

3.3 The Experiments Setup

The goal of this thesis is to explore different reward functions and find the best one for the described use case by conducting various experiments. In this section, the general setup that is used for the experiments is presented and explained.
The general experiment setup builds on top of the work of Florian Jaensch and specifically the papers "Digital Twins of Manufacturing Systems as a Base for Machine Learning" [20] and "Reinforcement Learning of Material Flow Control Logic using Hardware-in-the-Loop Simulation" [21].
The software components and their interactions are modeled in figure 3.1. There are two component containers: the virtual commissioning software and a Python script.

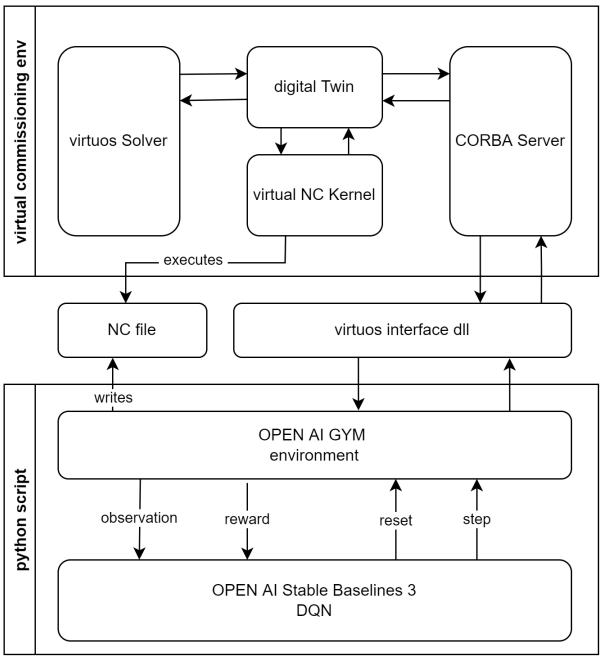


Figure 3.1: Reinforcement Learning with a virtual commissioning system

At the center of the virtual commissioning software, is the digital twin of the production environment. Here, all physical and logical components are modeled digitally. The digital twin models the robots, the material, and the environment around the robots like fences, walls, and tables. The virtual NC Kernel is a simulated robot control unit.
The ISG-virtuos virtual commissioning software also offers a visualization, that allows for monitoring, what the RL agent is doing. The visualization is not necessary for the RL, so it is not explained in further detail.
The responsibility of the Common Object Request Broker Architecture (CORBA) server is to provide an API to access the state information and remotely execute commands in the environment. A dll, invoked by the Python script interacts with the CORBA server. To provide a standardized

interface to interact with the environment, a custom Open AI gym [22] environment is created around the interactions with the virtual commissioning software. The gym environment requires an implementation of the two fundamental methods necessary for RL. The `reset()` method resets the environment and returns the current state (called observation). The `step(action)` method instructs the environment, to perform an action and returns the resulting observation and the corresponding reward. All actions, including the reset, are implemented utilizing an NC file. An NC file is a standardized way to write G-Code for industrial machines. The reset method is implemented as a

```
f"G90 G01 X1={j0} Y1={j1} Z1={j2} A1={j3} B1={j4} C1={j5} F10000\n"
```

command. G90 G01 defines an absolute movement of the robot to the position defined by the six joint angles j0 to j5. The step method is implemented as a

```
f"G91 G01 X1={j0} Y1={j1} Z1={j2} A1={j3} B1={j4} C1={j5} F1000\n"
```

command. G91 G01 defines a relative movement of the robot to the position defined by the six joint angles j0 to j5. At every step in the RL training, the chosen action is translated into a line of NC-Code, which is written in a file and then executed by the NC-Kernel in virtuos.
To train the RL agent, the OPEN AI Stable Baselines 3 (SB3) [6] Python module is leveraged. Amongst the algorithms provided by SB3 the DQN algorithm, explained in section 2.2, is used. DQN is a sample efficient RL algorithm with a discrete action space.
Most of this setup came as a prerequisite to the thesis. As part of this thesis, the setup was modified, so that actions are executed via an NC-file. This required changes in the Python script and the virtuos model.

3.4 The Two Training Scenarios

There are two scenarios. One scenario is utilized in chapter 4, where the robot is trained to move to a specified position. The second is utilized in chapter 5, where the robot is trained to move to a specified position whilst avoiding obstacles. The second scenario is an extension of the first. It has added obstacles, distance sensors, collision detection, and more inputs to the Q-function.
Both scenarios have only one robot. The "KUKA KR 10 R1100 sixx", as presented in figure 3.2 has the technical data listed in table 3.1. It is a common type of industrial robot for small to medium-sized handling and assembling tasks and is therefore a great robot for the proof of concept in this project.

Table 3.1: Technical data for "KUKA KR 10 R1100 sixx" [23]

Maximum reach	1101 mm
Maximum payload	10 kg
Pose repeatability (ISO 9283)	± 0.03 mm
Number of axes	6
Mounting position	Floor
Footprint	320 mm x 320 mm
Weight	approx. 55 kg

In the first scenario, the robot starts the episode in the position illustrated in figure 3.2. The goal of the agent is to reach the red point, which is a fixed point within the reach of the robot. The starting position is the position of the robot arm, where all joints have an angle of 0 . In this state, the Tool Center Point (TCP) has the coordinates $[0.7233, 0.0, 0.9951]$. The TCP is an arbitrary point at the center of the tool (in our case it is a gripper), defined by the manufacturer. The goal has the coordinates $[0.6, 0.6, 0.6]$. The euclidean distance between the starting position and the goal is 0.73 m. To achieve the goal, the agent can move only one of the six axes by 2° incrementally

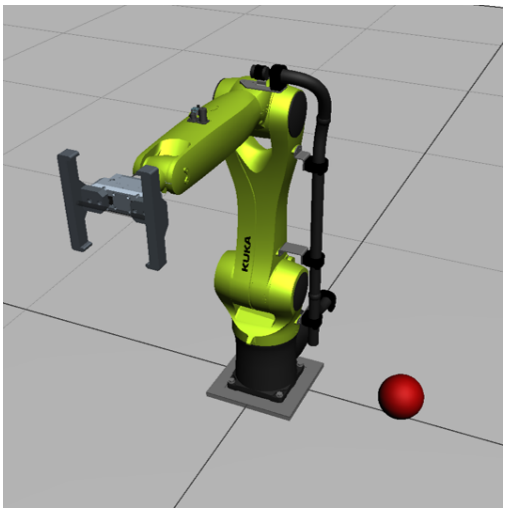


Figure 3.2: KUKA KR 10 R1100 sixx, as Screenshot from the ISG-virtuos Visualization

each step. The goal is considered achieved if the TCP of the robot is within 5 cm of the target position. With this setup, the minimum amount of steps, to reach the goal, is about 30 steps.
The size of the action space of the agent is 13. There are two actions per robot axis. One action is to move an axis in the positive direction and the other is to move it in the negative direction. With six axes, there are twelve actions for movement. The 13th action allows the agent to not move in any direction and stay still for one step.
Usually, in RL the starting position and the target position are varied slightly, so that the neural network does not overfit to a specific scenario, and can generalize better. But for this thesis, only the reward function is investigated, and therefore a stable setup is needed, where we can compare episodes reliably. Randomizing the start and target position would introduce a random variance, that would make it harder to compare the performance of different reward functions.
The neural network (NN) architecture for the value function is a simple fully connected neural network visualized in figure 3.3.

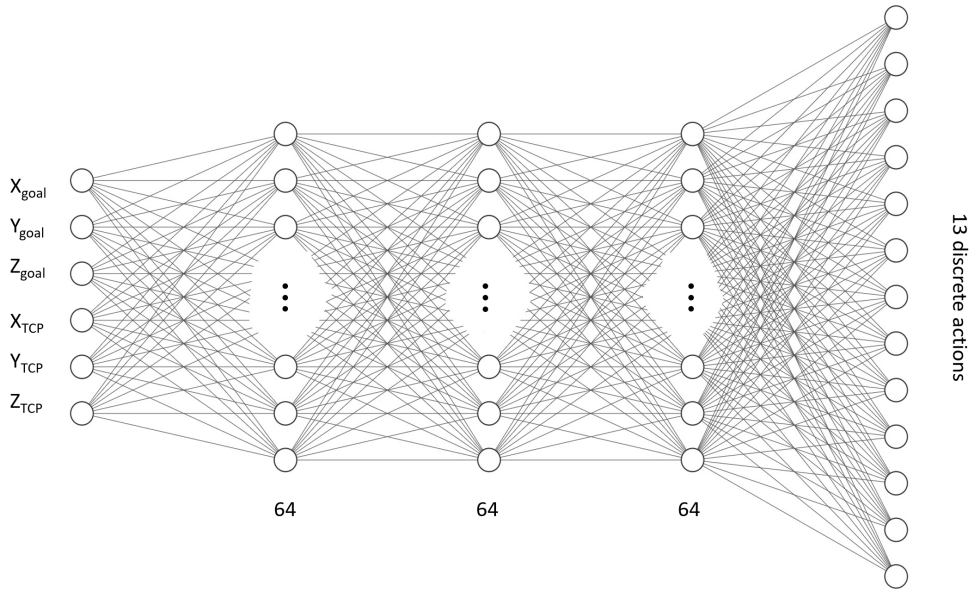


Figure 3.3: Neural Network architecture for the first scenario

As an input, the NN gets the coordinates of the (static) target position and the coordinates of the current TCP position. The NN has three hidden layers with 64 neurons each. The value function

tries to predict the assumed cumulated return for each possible action. The size of the output layer is therefore equal to the size of the action space (13). The intuition is, that the NN will learn to calculate the difference between the target position and the current position and base its value predictions on that.
The parameters used for the Stable Baselines 3 [6] DQN algorithm are as listed in the following table 3.2:

Table 3.2: Parameters for the Stable Baselines 3 [6] DQN algorithm

neural network architecture	[input(6), 64, 64, 64, output(13)]
learning rate	0.001
buffer size	1 000 000
learning starts	1000
batch size	128
tau	1
gamma	0.99
train freq	every 4 steps
gradient steps	-1
target update interval	100
max grad norm	10
exploration fraction	0.1
exploration initial eps	1
exploration final eps	0.05

All experiments are performed using a single PC with an Intel I7 4770K and a GTX 1060 3GB.
The second scenario is an extension of the first. It has added obstacles, distance sensors, collision detection, and more inputs to the Q-function. Further details on how the second scenario differs from the first are explained in section 5.1.

3.5 Framework for Evaluating the Performance of a Reward Function

In this thesis, the performance of reward functions is evaluated under three main aspects:

- time to convergence
- final performance of policy
- stability

The goal is, to maximize all three aspects. Some improvements in the system can improve all three aspects at once, whilst others may increase one, but slightly decrease the other two. The objective of this thesis is to find and understand how changes in the systems affect the three performance aspects.
The three aspects are inspired by performance metrics for feedback control systems. Feedback control systems are evaluated against a step response, where the system starts with an initial offset from the target value, and aims to minimize the difference between actual value and target value as fast as possible. In figure 3.4 such a step response is shown. The system starts at an initial value of 0 and aims to reach a value of 1 as fast and precise as possible.
The settling time T_s is synonymous with the *time to convergence* aspect. The steady-state error e_{ss} is synonymous with the *final performance of policy* aspect. The *stability* aspect is not as easy

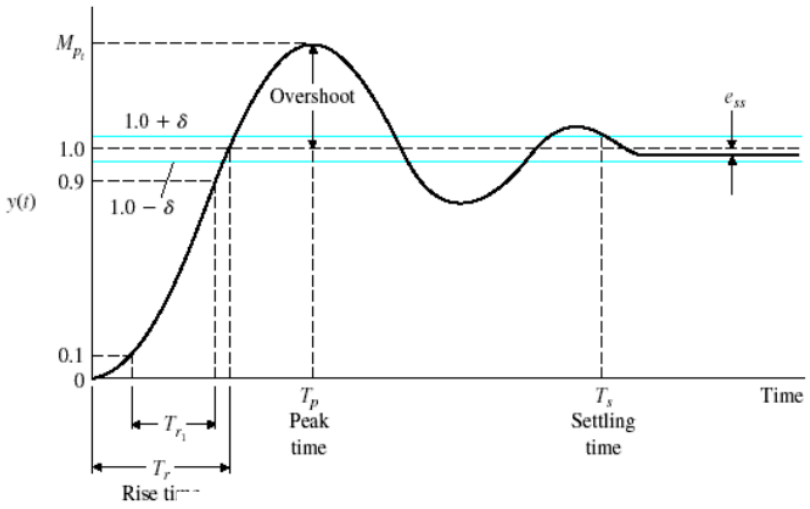


Figure 3.4: Performance metrics for second order feedback control system (Haldia Institute of Technology, [3])

to compare with the equivalent in feedback control systems, as RL algorithms have different mechanisms and different reasons for unstable behavior. During the training, the agent may suddenly diverge and "unlearn" the correct policy.

To further clarify, what the three main aspects for evaluating an RL algorithm mean in practice, and how they can be read from a graph of the training progress, an example scenario is analyzed. A RL training run is a sequence of steps, where the agent collects experiences and learns from them. At the start of a training run, the weights and biases in the neural network are randomly initialized based on a random seed. The training process is divided into episodes, limited to n steps. In the experiments in this thesis the limit is 600 steps. After each episode, the environment is deterministically reset to an initial state. To track the performance during the training, the cumulated episode return is logged after each episode. The intuition is that the higher the reward, the better the policy of the agent.

Comparing the return of the episodes is a common practice in RL research. Yet this practice is not sufficient for the objective of this thesis, as here the reward functions are experimented with. When using different reward functions between experiments, the intuition that a higher reward implies a better policy no longer holds true. A better way to compare two episodes is to evaluate how well the agent has achieved the goal according to another metric. For this thesis, we can compare if and in how many steps the agent has moved the TCP to the target position. This is indicated by the *total steps in the episode*. If the total steps in an episode is less than the maximum steps (600 steps in our case), then we know, that the agent has reached a terminal state, causing the episode to be terminated before the timestep limit of 600 steps. In the experiments in chapter 4 no negative terminal, like *"robot crashed"*, exists, so terminating the episode early means that the agent has reached the target position.

To illustrate, how the three performance metrics can be deduced based on the graph of the total steps for each episode, the example experiment in figure 3.5 is analyzed.

The *time to convergence* aspect is defined by the elbow method, as this method is commonly used in machine learning. Convergence is achieved if the performance of the policy does not improve significantly anymore. To better examine this aspect, the experiment performance should be averaged with a rolling-window average. This makes it easier to visually pinpoint if the average performance of the policy is not improving anymore. If a rolling average is used for a figure, the window size is noted in the figure description.

The *final performance of policy* aspect is defined by how good the policy of the agent is after the learning process has converged. For the use case in this thesis, the performance can be evaluated

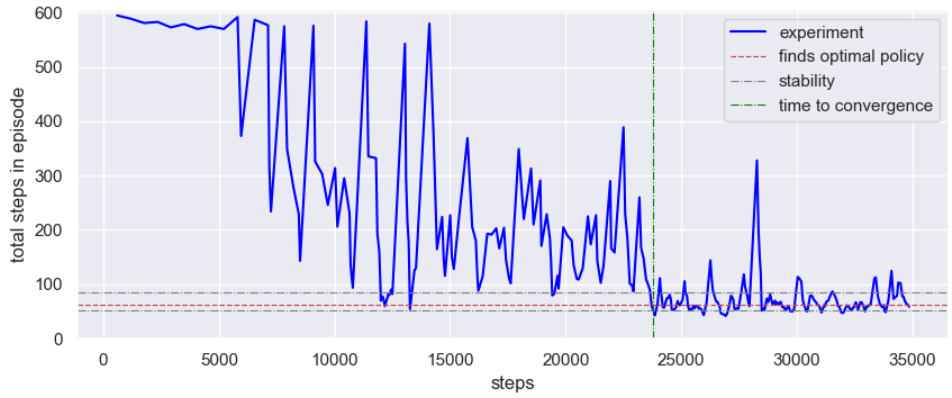


Figure 3.5: The three main aspects for evaluating an RL algorithm, defined by an example experiment, (rolling average over 200 steps)

based on how fast the agent reaches the target position. In particular, the performance is based on how many steps the agent takes to reach the target position in each episode. The fewer steps a policy takes to reach the target position, the better the final performance of the policy. To evaluate the final performance, only the performance at the end of the training process is considered. Determining, when the training process is finished is quite arbitrary. However, the performance should asymptotically converge to its final performance during the training process.
The *stability* can not be judged easily by a single value. Different aspects are contributing to the stability of a RL training. One aspect is the standard deviation σ of the performance after convergence has been achieved. Another aspect of the stability is the resilience to divergence. RL algorithms sometimes diverge away from a sensible policy due to the natural instability and sensibility towards the random seed.



A more rigorous numerical definition performance metrics would be preferable. Nevertheless, for this thesis, the performance aspects are mostly determined by visually interpreting the training plots as illustrated with figure 3.5. This methodology is used because RL algorithms are very sensitive towards the random seed, and multiple training runs need to be performed to achieve a representative image of the training performance. Measurements should not be presented as more precise than the system it measures. A numerical one-dimensional metric as a performance measure would not represent the unprecise and random nature of RL. Therefore visual comparisons for the performance aspects are deemed sufficient for this thesis.

The reward function developed in this thesis should be generalizable for similar scenarios. A reward function, that is designed for a scenario with one obstacle should also perform well in a scenario with three other obstacles. Parts of the reward function, that can not be generalized should be abstracted so that they can be easily parameterized externally.

Chapter 4

Move to Position without Obstacle

In this chapter, the first scenario is examined. The task for the KUKA™robot is to move its TCP to a defined target position. It is examined how different reward functions affect the behavior of the agent and the learning progress. The purpose of these experiments is to find a performant reward function based on the three main aspects for evaluating an RL algorithm as explained in section 3.5 and motivate why this reward function is performing better than others. Firstly a sparse reward function is used as a comparison to show that dense reward is superior. Then different dense reward functions are presented and examined.

Reinforcement learning is very sensitive regarding the random seed, as the random seed not only influences the initial weights of the NN but with epsilon-greedy Q-learning [24] it also partially randomizes the actions of the agent. Both factors determine the experiences the agent gathers and therefore affect the training heavily.

To mitigate this randomness, most experiments in this thesis will be replicated a minimum of three times. It would be preferential, to increase the number of runs per experiment, but as explained in section 3.2 the training using the current simulation environment has significant limitations regarding the training speed (simulated steps per second).

In table 4.1, the experiments performed in this chapter are listed alongside an explanation for why they are performed.

Table 4.1: **Experiments in chapter 4**

Experiment	Figures	Explanation
Sparse Reward	4.1	serves as a comparison to show that dense reward is superior
Negative Dense Reward	4.2; 4.3	shows that a dense reward improves the performance, but this one has stability issues
Negative Dense Reward (0 Positive Terminal Reward)	4.4	tests if the stability can be improved by changing the terminal reward
Negative Dense Reward (0 Positive Terminal Reward; Lower Learning Rate)	4.5	tests if the stability can be improved by lowering the learning rate
Dense PositiveRreward with Reward Shaping	4.7; 4.8	improves stability and introduces reward shaping
Convergence Test	4.9	tests the convergence and stability by training longer
Dynamically Reduced Reward vs. Baseline	4.10; 4.11	tests if a dynamically reduced reward can improve performance

4.1 Sparse Reward

In the first reward function experiment, a sparse reward is used. This experiment is intended to serve as an introduction to how the experiments are carried out and as a baseline for motivating the later experiments.
Figure 4.1 shows two experiments with a sparse reward function, set to run for 162000 steps (one day of computing each). The reward function returns a reward of 1 if the goal is achieved, otherwise, it returns 0. The cumulated return of every episode is plotted, meaning there is one marker for every finished episode. Episodes are terminated after 600 steps, or if the robot reaches the goal. This gives us about 270 episodes for each experiment run. Only in one run in experiment 1 the agent reached the goal. In the 3D Viewer, it can be observed that the agent has a hard time reaching the goal, as it is just performing random actions. Reaching the goal once was not enough for the algorithm to learn a good value function.

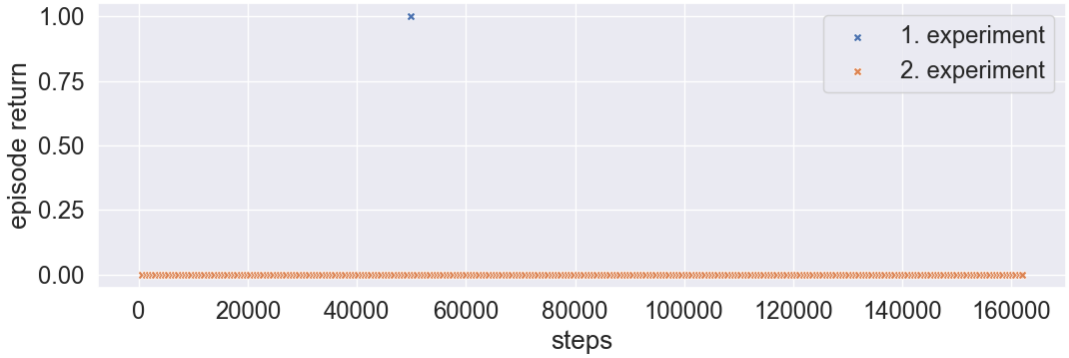


Figure 4.1: Two experiments with sparse reward. The robot reaches the goal only once indicated by a non-zero reward at step 49 500

Q-learning is mathematically proven to converge eventually if the neural network in the Q-function is complex enough. However, it is evident from figure 4.1, that training with a sparse reward function is not efficient. This is why reward shaping with dense reward functions is explored in later sections.

4.2 Simple Negative Dense Reward

In this section, a dense reward function is used to train the agent. The strategy using a dense reward function is to give the agent a small intermediate reward at each step. The reward should be higher if the agent is closer to reaching the terminal goal. In our experiment, we want to give the agent a higher reward, the smaller the euclidean distance between the TCP and the target coordinates is. This behavior is achieved by the reward function in the following formula:

$$\text{distance} = \sqrt{(x_{\text{goal}} - x_{\text{TCP}})^2 + (y_{\text{goal}} - y_{\text{TCP}})^2 + (z_{\text{goal}} - z_{\text{TCP}})^2} \tag{4.1}$$

$$\text{reward} = -\left(\frac{\text{distance}}{\text{maxdistance}}\right) \tag{4.2}$$

With this reward function, the reward is relative to the negative distance. The maximum distance is compensating for different robots with different maximum reaches and is set to 4 meters. The maximum distance compensates for bigger and smaller working spaces. This compensation makes the reward function behave similarly for big and small robots.
In figure 4.2 the training process is visualized for four experiment-runs. The experiments are set to run for 70 000 steps, with each episode limited to 600 steps. After each episode, the total return

of that episode is captured. To see a noticeable difference between episodes where the agent has reached the goal, and episodes where the episode was canceled after 600 steps, a terminal reward of 1000 is given to the agent if it reaches the goal. The positive terminal reward is an additional incentive for the agent to reach the target position.

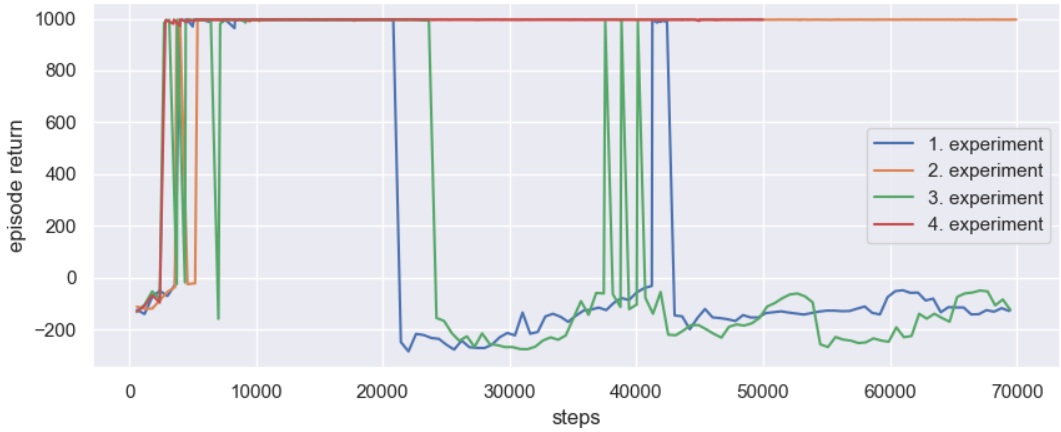


Figure 4.2: Experiment with negative reward; 1000 terminal reward

In the 4th experiment, the agent first reaches the goal after about 2500 steps of training. It looks like in all four experiments a stable q-function has been achieved after 8000 steps of training. But the wild behavior in the 1st and 3rd experiments shows that both agents have unlearned their successful policy and were not able to successfully achieve the task anymore. The two agents could not recover in the 70 000 steps limit.
To further analyze the performance of the training process, in figure 4.3 the *total steps in an episode* are plotted. This gives further insight into how good the agent is performing, as it tells how many steps the agent took, to reach the goal. If the value of the *total steps in an episode* is 600 it means, that the episode was canceled due to reaching the time limit, before the agent reached the goal, or that the agent took exactly 600 steps to reach the goal. The latter case is rather unlikely, so it can be assumed, that the agent did not achieve the goal if *the total steps in an episode* is equal to 600.

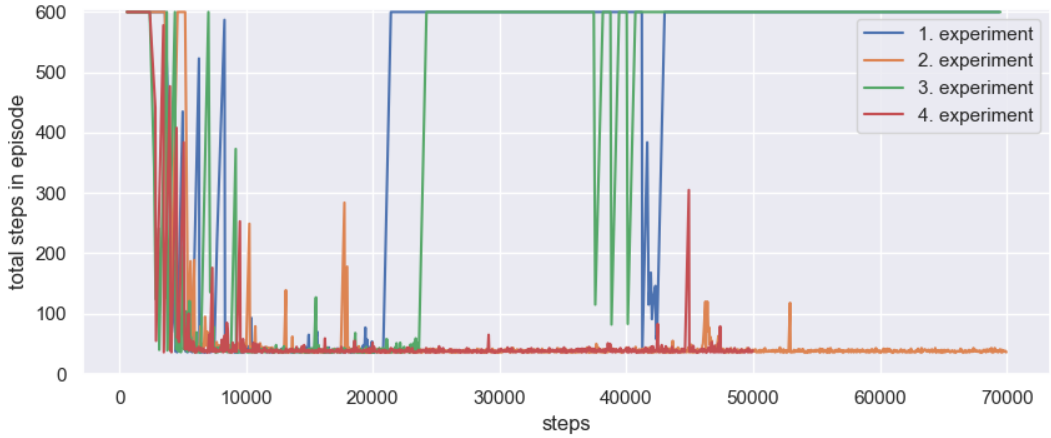


Figure 4.3: Experiment with negative reward; 1000 terminal reward; total steps in episode

In this experiment, the time to convergence is about 11 000 steps. The agents converge to a policy, where they reach the goal after about 38 steps. The stability is a big problem with this setup, as two out of the four experiments completely diverge again.

In the following section a short ablation analysis is performed, to investigate the stability issue. The impact of the terminal reward and the learning rate is analyzed. First, the same experiment as above is performed, but with a terminal reward of 0. The three experiments are shown in figure 4.4.

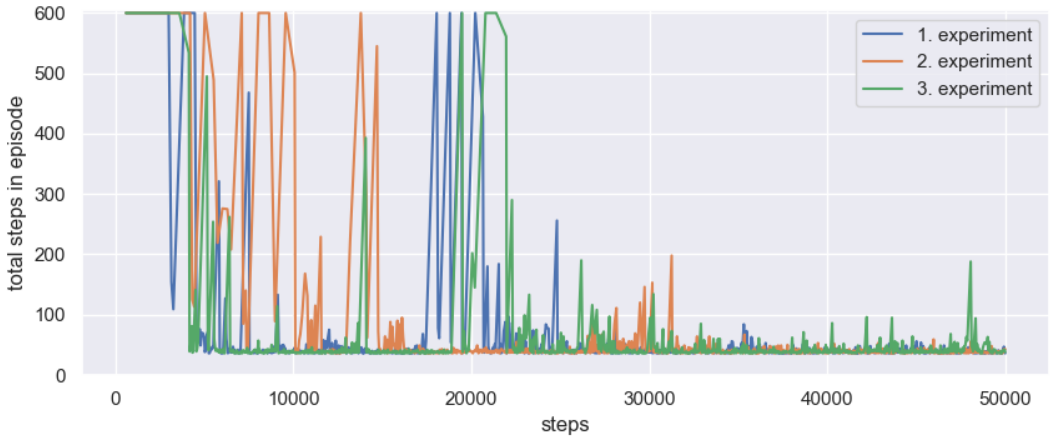


Figure 4.4: Experiment with negative reward; 0 terminal reward; total steps in episode

In this figure, the time to convergence is about 31 000 steps. The agents converge to a policy, where they reach the goal after about 46 steps. The standard deviation after the convergence is slightly worse, but the agents do not diverge, as in the two runs in the previous experiment with a terminal reward of 1000
In the next experiment, the learning rate is reduced by a factor of 10 to $\alpha = 0.0001$. The prediction for this experiment is, that the agent learns slower, but not with a 10th of the speed. Reducing the learning rate should increase stability. The three experiments are shown in figure 4.5.

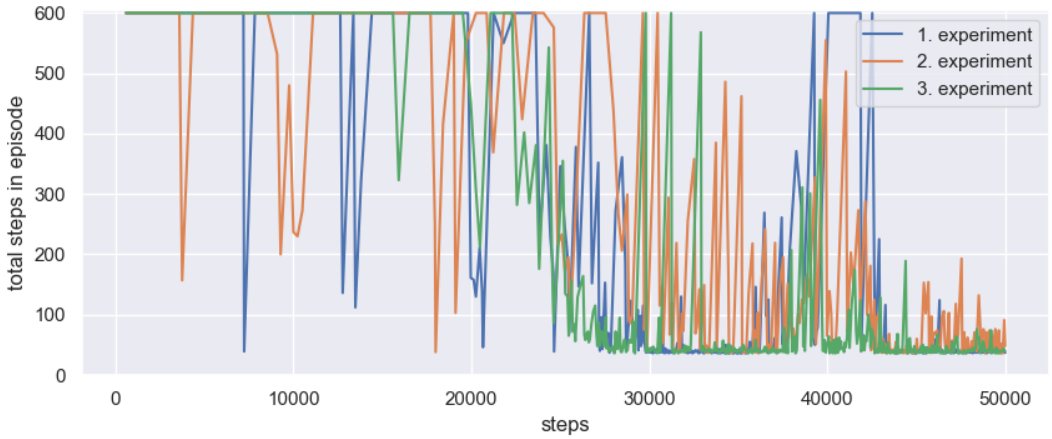


Figure 4.5: Experiment with negative reward; 0 terminal reward, $\frac{1}{10}$ learning rate; total steps in episode

In this figure, the time to convergence is about 50 000 steps. The agent's final performance and the final stability can not be determined, because the maximum simulation limit of 50 000 steps is too low to make a judgement. The simulation was limited to 50 000 steps because of the simulation speed problem explained in section 3.2.
The approach with negative rewards looks promising, as the time to convergence and the final episode performance is good. The stability issue may be fixable with further fine-tuning and longer experiments. There might be an additional issue using negative rewards in the second scenario,

where crashes of the robot are detected and punished with a negative terminal. At this point crashes of the robot into itself or the ground are not detected and just ignored. In practice, the episode should stop after a crash has occurred. If the agent gets a negative reward at every step, the agent wants to end the episode as soon as possible. This could lead to the agent trying to crash on purpose so that the episode gets terminated prematurely.

4.3 Reward Shaping with Linear and Non-Linear Reward

In this section, a new approach to the reward function is explored. For the experiments in this section, the reward at each step is in the range between 0 and +1. This helps to prevent the case, where the agent learns to crash on purpose. Similar as in the previous section, the dense reward should be dependent on the distance between the robot TCP and the target position. The distance is calculated using equation (4.1).
In addition to exploring positive rewards, the effect of non-linear dense reward functions is explored. The following three reward functions are used. Equation (4.3) and equation (4.5) are representing a non-linear reward function, where the reward is in a polynomial relationship with the distance. Equation (4.4) is representing a linear reward function with positive rewards.

$$\text{reward} = 1 - \left(\frac{\text{distance}}{\text{maxdistance}}\right)^{2.5} \tag{4.3}$$

$$\text{reward} = 1 - \left(\frac{\text{distance}}{\text{maxdistance}}\right)^{1} \tag{4.4}$$

$$\text{reward} = 1 - \left(\frac{\text{distance}}{\text{maxdistance}}\right)^{0.4} \tag{4.5}$$

To better visualize the effect of the nonlinearity, the reward functions are visualized in figure 4.6. Special attention should be given to the derivative of the functions, as the derivative at a certain point affects the updates by the gradient descent algorithm to the Value function during the learning process. The bigger the derivative, the bigger the updates calculated by the gradient descent algorithm. The red line, depicting the reward function (4.3) shows a decreasing derivative with smaller distances. The blue line, depicting the reward function (4.5) shows an increasing derivative with smaller distances.

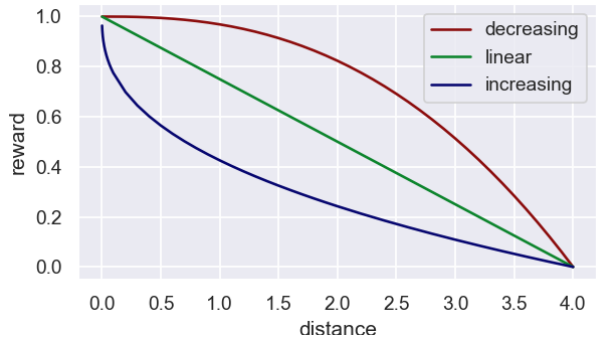


Figure 4.6: Visualization of different reward functions using reward shaping

In the following experiments, the positive reward function and the effect of the non-linear reward shaping are investigated. Each experiment is repeated four times. The timestep limit per episode is 600 steps. The positive terminal reward for reaching the goal is 1000. There is no crashing implemented, so the only terminal state is the positive terminal.
In figure 4.7 the return for each episode is plotted for each experiment run. A rolling window over 3000 steps is used to make the graph more visually interpretable.

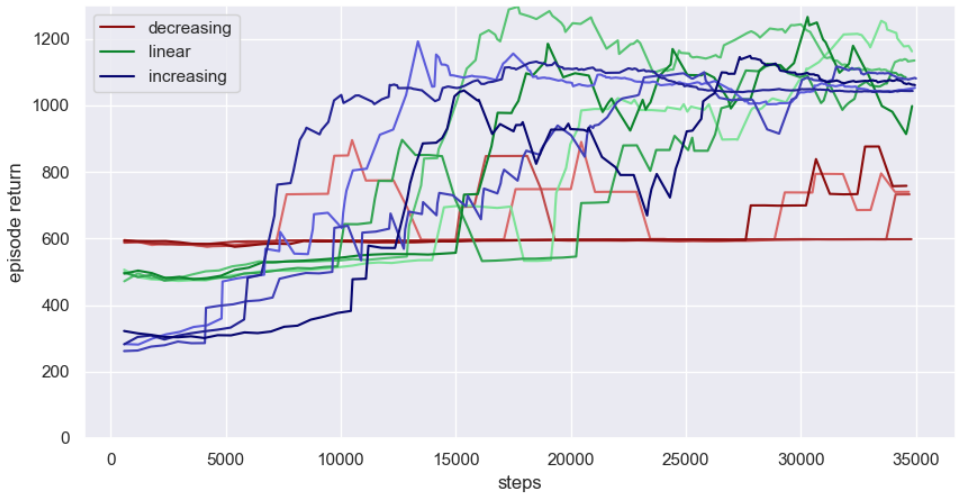


Figure 4.7: Experiment with linear and non-linear reward; (rolling average over 3000 steps)

The experiments colored in red have a return of about 600 at the beginning of the training process. This is because, with the red reward function, the reward per step at 0.73 m distance is approximately 0.98. At the beginning of the training, the robot arm is just flailing around randomly and is not strategically moving towards the goal. This means the return should be approximately return $= 0.98 \cdot 600 = 588$ as seen in the graph. The experiments colored in green have a return of about 480 at the beginning of the training process. This is because its reward per step at 0.73 m distance is approximately 0.8, resulting in a return of approximately return $= 0.8 \cdot 600 = 480$ as seen in the graph. The experiments colored in blue have a return of about 300 at the beginning of the training process. This is because it's reward per step at 0.73 m distance is approximately 0.5, resulting in a return of approximately return $= 0.5 \cdot 600 = 300$ as seen in the graph. The blue experiments have a higher standard deviation at the beginning of the training process compared to the others. This phenomenon is explained by the greater derivative of the blue reward function at and around the 0.73 m mark. With a greater derivative of the reward function, changes in the distance cause greater changes in the reward.

The maximum return, that can be achieved in an episode without reaching the target is 600 because the maximum reward per step is limited to 1. A return higher than 600 means that the agent has reached the positive terminal (reached the target position), terminating the episode with an additional reward of 1000. Based on this, it can be deduced that all but one of the experiment-runs with a decreasing derivative reward function reached the goal a few times during the 35 000 steps training process. Nevertheless, the experiments with the decreasing reward function (depicted in red in figure 4.7) stand out for having the worst performance. The red plot appears to be more blocky than the other two. In the red experiments, the target is reached less often, resulting in longer episodes because the episode is not terminated prematurely. The episode return is only logged after every episode, therefore the red experiments have less frequent data points and therefore appear more blocky.

In the next figure 4.8, the total steps per episode are plotted instead of the episode return. This helps to compare the experiments because the reward functions in this experiment just give out different rewards for the same distance. Therefore, experiment-runs using different reward functions, can not be compared just based on the episode return.

At first glance, it can be analyzed, that the red experiments perform the worst, the green experiments perform better and the blue experiments perform best in all three performance aspects. The blue experiment converges faster, the final policy has better quality and the final stability is better. This can be deduced from the blue plots having a consistently lower total steps count in the final

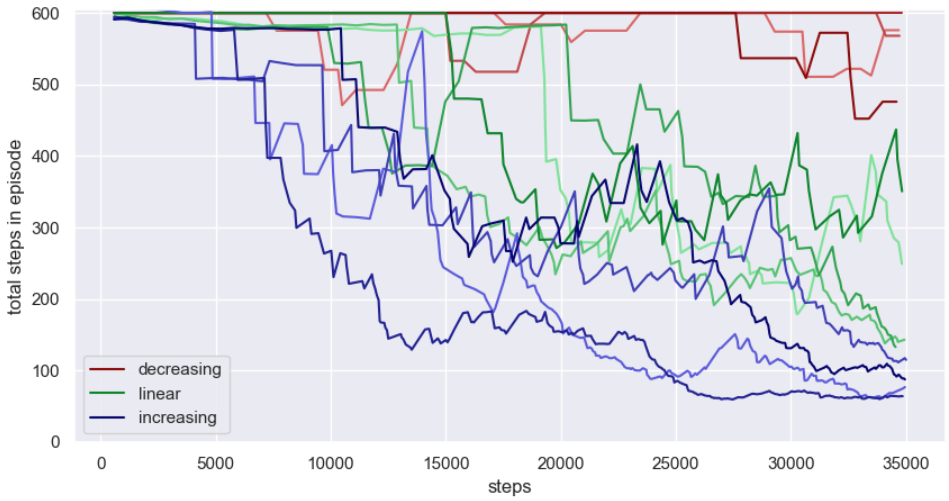


Figure 4.8: Experiment with linear and non-linear reward; (rolling average over 3000 steps)

episodes. This indicated, that the agents using the increasing derivative reward function reach the target faster and more consistently.

The red experiments perform the worst, indicated by them only rarely reaching the target. But why do the red experiments perform worse? This behavior can be explained by the visualized reward function in figure 4.6. The reward for distances smaller than 1 m is almost constant. This gives the agent very little incentive to move closer to the target as it will get approximately the same reward either way. In contrast, the blue reward function has a higher derivative, the closer it gets to the target. This results in a greater incentive for the agent to move closer to the target.

4.4 Dynamically Reduced Terminal Reward Over Time

One question that comes up at this point is if in any of the experiments the agent has learned the optimal policy. The least amount of steps to the goal is about 35 steps.

As explained in section 2.3, it can be difficult to design a dense reward function in such a way, that maximizing its return results in the desired behavior. The desired behavior is for the robot to reach the target position as fast as possible. But the highest possible return would be achieved by quickly moving in front of the target position *without entering it*. If the agent then waits in front of the goal it can accumulate all the positive intermediate rewards without terminating the episode prematurely. Just at the last step, the agent should enter the goal to collect the large terminal reward. Obviously, this behavior is not desirable. To experimentally validate if the agent learns this unwanted behavior, a new experiment over a longer time is conducted. The linear (green) reward function is used to train the agent over 260 000 steps. This experiment is plotted in figure 4.9

As it becomes apparent in the graph, the agent does not learn to wait in front of the goal to maximize the episode return. This can be explained by the properties of the DQN algorithm. With DQN there is a discount factor γ, that discounts expected future rewards. This discount factor increases the stability of the learning process. As an additional benefit, it also prevents the robot from waiting in front of the target position.

In Q-learning, not the policy is learned, but the value function (Q-function) (equation (4.6)) is learned [25]. This function tries to predict the cumulated return for all future rewards in a given state for a chosen action, assuming all following actions will be chosen optimally. r_0 is the reward the agent expects to get for the current step. All expected future returns are discounted by the discount factor $\gamma \in (0, 1]$. Rewards further in the future are discounted more.

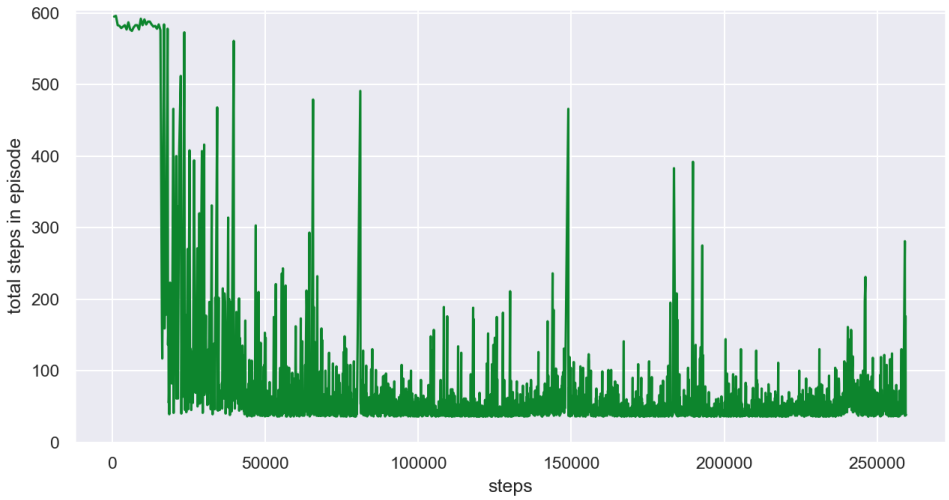


Figure 4.9: Convergence and stability test with the linear reward function; the agent does not learn to wait in front of the target

$$Q(s,a) = \sum_{i=0}^{\infty} \gamma^i r_i \tag{4.6}$$

Based on this knowledge, the value an optimal Q-function should predict for waiting in front of the goal indefinitely can be calculated with equation (4.7). The maximum reward for waiting in front of the goal is $r_i \leq 1.0$ per step and the discount factor used in this experiment is $\gamma = 0.99$.

$$Q^*(s,a) = \sum_{i=0}^{\infty} \gamma^i r_i \leq \sum_{i=0}^{\infty} \gamma^i = \frac{1}{1-\gamma} = \frac{1}{1-0.99} = 100 \tag{4.7}$$

As illustrated by the above formula, the maximum value the Q-function should predict for indefinitely waiting in front of the goal is 100, which is less than the terminal reward for moving to the goal and collecting the full reward. To further elaborate on the proof, the equation (4.8) calculates if the expected reward for waiting for only one step is higher or lower than immediately entering the target.

$$Q^*(s,a) = \text{reward for being close to target} + \text{reward for entering the target after one step} \tag{4.8}$$

$$= 1 + 0.99 \cdot 1000 = 991 < 1000 \tag{4.9}$$

Equation (4.8) shows, that with a positive reward of 1000, an intermediate reward of 1 and a discount factor of 0.99 the agent is not incentivized to wait in front of the target. For this reason, the agent will not learn to wait indefinitely in front of the goal as long as the discounted value for waiting is less than the positive terminal reward. Future experiments could be performed to test if the agent will wait in scenarios with a terminal reward smaller than 100 or scenarios with less discount.



To ensure that the agent does not learn to wait in front of the target, the following condition is required to hold:

$$\text{terminal reward} > \text{max intermediate reward} + \text{discount factor} \cdot \text{terminal reward} \tag{4.10}$$

Even though the proof above shows, that the agent will not learn to wait in front of the goal, the agent may not be sufficiently incentivized to reach the goal as fast as possible. Another problem with the current reward function is that the total episode reward, as shown in the graphs (4.4, 4.2, ...), may be lower for episodes where the target was reached faster. This makes it more difficult (for the researcher) to evaluate the experiments based only on the reward.
To incentivize the agent to reach the target faster, the positive terminal reward is reduced by 1 point for every step in the episode. This behavior is achieved with the equation (4.11).

$$\text{positive terminal reward} = \text{maximum positive terminal reward} - \text{steps count this episode} \quad (4.11)$$

The positive terminal reward of 1000 is reduced by 1 for every step the agent takes. As all episodes are capped at 600 steps, the minimum terminal reward, assuming the agent reaches the goal at the very last step, is $1000 - 600 = 400$.



When utilizing dynamically reduced terminal reward, special care needs to be taken to make sure that the minimum positive terminal reward never gets negative. The equation (4.11) may need to be adjusted when the *positive terminal reward* or the *maximum number of steps in an episode* is modified.

To examine the effect of a dynamically reduced terminal reward, an experiment is conducted, where experiments with a dynamically reduced reward are compared to experiments with a static terminal reward. The expectation is, that the dynamically reduced terminal reward incentivizes the agent to reach the goal faster. Such behavior would result in a better-performing agent.
As the reward function with an increasing derivative (blue reward function) has been the best in all aspects compared in section 4.3, it is used for the experiments in this section.
For one experiment, a fixed terminal reward is implemented and for the other experiments, a dynamically reduced terminal reward as in equation (4.11) is implemented. All other parameters are the same. In figure 4.10 the two experiments with three runs each are visualized. The episode return is plotted with a rolling average over 3000 steps to make the graph more visually interpretable.

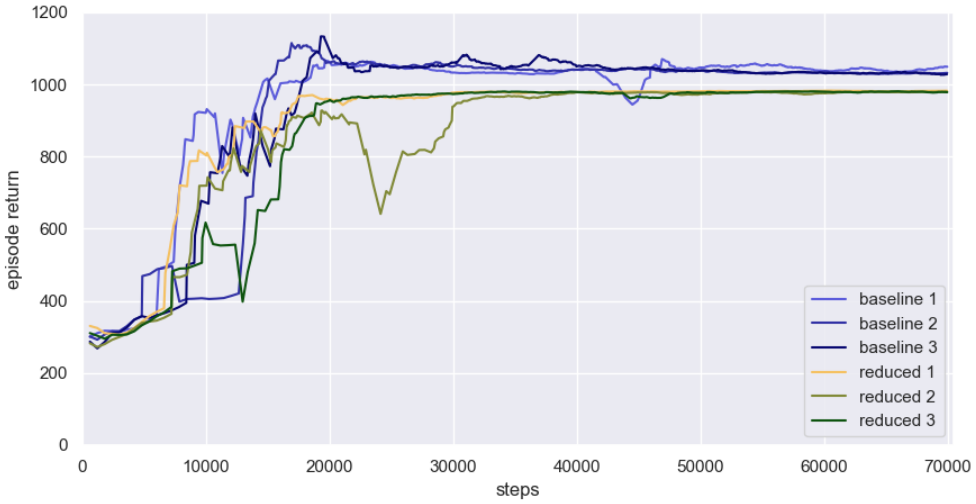


Figure 4.10: Dynamically reduced reward vs. baseline (rolling average over 3000 steps)

The blue lines represent the baseline with a rising derivative reward function and a static terminal reward of 1000. The yellow-green lines are the experiment-runs with a dynamically reduced reward function. The runs are conducted over 70 000 steps so that the final performance of the policy can

be analyzed better. Within the 70 000 steps no catastrophic policy divergence is observed. But it is not guaranteed that such instabilities won't happen with longer or more numerous training runs.
It can be observed that the yellow-green experiment settles at a lower reward. However, this does not mean that its reward function is performing worse. The terminal reward is reduced by 1 point for each step in an episode. With an average of about 50 total steps in an episode, the yellow-green experiment should settle at about 50 points below the baseline experiment.
As the performance can not be judged based on this graph, in figure 4.11, the total steps in an episode is plotted on the y-axis.

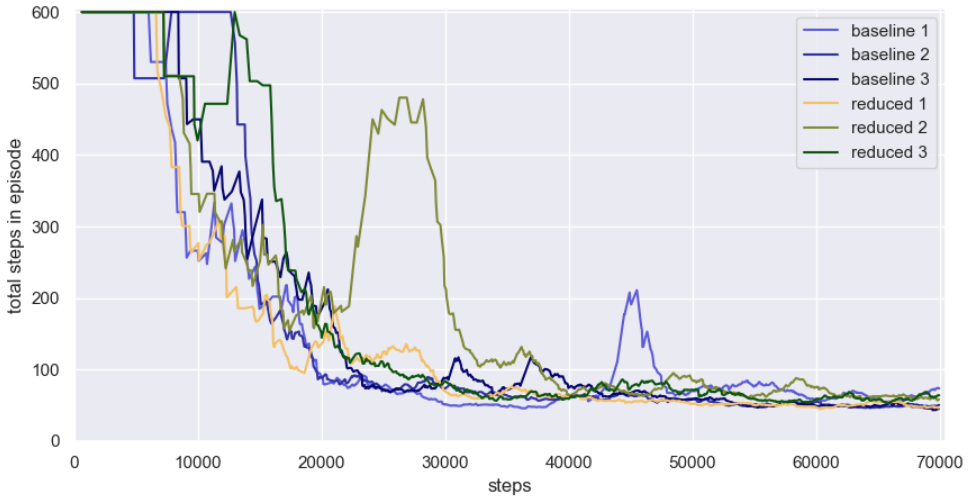


Figure 4.11: Experiment to examine dynamically reduced reward (rolling average over 3000 steps)

All experiment-runs start at 600 total steps, as the episodes are limited to a maximum of 600 steps. A value below 600 means, that the agent has achieved its goal and reached the target position.
Both experiments perform similarly on time to convergence. The performance of the final policy looks similar between the two experiments. To better evaluate this aspect, more and longer training runs would be necessary. Regarding the stability, two outliers are "baseline 1" and "reduced 2". Such unstable behavior is not uncommon for RL. Based on the small sample size no inference can be made on the difference in stability between the two reward functions.
The intuition, that a dynamically reduced terminal reward would incentivize the agent to reach the target position faster can not be confirmed based on the here conducted experiments. Why is that so? As explained in this section, the agent is already incentivized by the discount factor, to reach the target as fast as possible. The reduced terminal reward does not give the agent additional information and the feedback it provides is limited. The feedback provided by the reduced terminal reward is only once per episode and therefore not as helpful as a denser reward.
Some research [1] recommends, that the reward function should only consider information, that the Q-function also has access to. If the reward is influenced by information, the Q-function does not have access to, the Q-function might get confused and more uncertain about its predictions. Unknown and not derivable information is like randomness to the Q-function. Such randomness might hinder the learning process. It should be noted here that such negative effects of the dynamically reduced terminal reward couldn't be detected in the experiment.



> The author of this thesis does not recommend utilizing a dynamically reduced terminal reward to incentivize the agent to achieve the goal faster. The reasons presented should also apply to other use cases of RL not involving industrial robot trajectory planning.

Chapter 5

Move to Position with Obstacle

In the previous section, the goal for the agent was to move to a defined target position. The new goal in this scenario is to move to a defined target position whilst avoiding obstacles.
To correctly model this scenario, some changes in the environment need to be made. These changes are explained in section 5.1. Following that, a reward function is iteratively developed and analyzed. In table 5.1, all experiments performed in this chapter are listed alongside an explanation for why they are performed.

Table 5.1: **Experiments in chapter 5**

Experiment	Figures	Explanation
−200 Penalty for crashes	5.3	if the robot crashes, the episode ends and the agent gets a penalty
Linear Proximity Penalty	5.5	the agent gets a penalty proportional to how close the robot is to obstacles
Polynomial Proximity Penalty	5.8	introduces a polynomial proximity penalty
Comparison of Different Values for Maximum Distance Penalty	5.9	experiment what is the best value for the max distance penalty
Intra Episode Performance (max distance penalty = -0.1; crashing enabled)	5.10	analyses the performance in the last 10 000 steps in a training run
Intra Episode Performance (max distance penalty = 0; crashing disabled)	5.11	analyses the performance impact of crashes being enabled
Intra Episode Performance (max distance penalty = 0; crashing disabled; no distance sensors)	5.12	analyses the performance impact for more inputs to the Q-function

5.1 Defining the Setup for the New Training Scenario

In the scenario used in the previous section, the robot could not crash, because in the simulation environment crashes were not detected. The robot could move through itself and through the ground with no problem. For the scenario in this chapter, a system in the environment is implemented to detect if the robot has crashed into itself or other objects in the environment.
To test the agent's ability to learn to avoid obstacles, a floor and a wall are placed in the environment. The agent is supposed to avoid touching these obstacles. To detect crashes, (light blue) boxes are positioned to enclose the outer dimensions of the industrial robot. A geometrical intersection between any of the boxes and the environment is detected as a crash. When a crash is detected, the episode needs to be reset, as there is no value in trying to teach the agent what to do after a crash has occurred.

As illustrated in figure 5.1 the wall is positioned in front of the robot between the start and the target position. In the following experiments, we want the robot to move over the wall to the target. For this reason, the start and the target are positioned on opposite sides of the wall. Now the agent needs to move the robot up and over the wall to reach the target position. This is a hard task to learn because the agent needs to move upwards to avoid the wall, which increases the distance to the target. The agent needs to sacrifice reward in the present for a greater reward in the future.

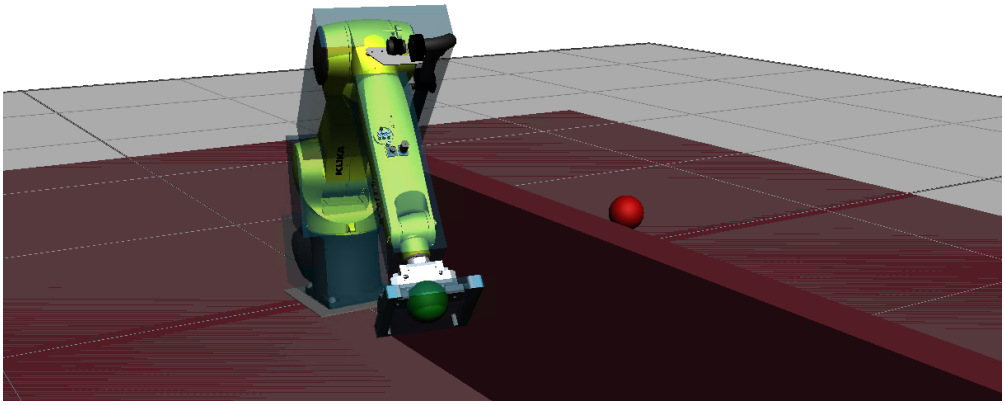


Figure 5.1: New Scenario with wall and floor as an obstacle; new start and target position

With the current configuration of the neural network, the agent will have a tough time finding a sensible policy. This statement can be explained by the following analogy.

Imagine you are trying to walk through a labyrinth in complete darkness, without utilizing your hands to feel for the walls. Every time you bump into a wall, you get reset to the beginning of the labyrinth. Such a task would be very hard to solve even for a human. To successfully walk the Labyrinth a perception of the environment is needed. The described task would be easier if you could see or feel the distance to nearby objects by holding out your hands in front of you.

It could be argued that the agent should be able to solve the simple task without any extra sensors, and that is true. The problem is that the agent won't learn to avoid obstacles, it will just learn to move up a few times at the beginning of the episode. This is not a sensible policy.

The agent needs extra sensors, to perceive its surroundings. As illustrated in figure 5.2 extra sensors are added to the model of the robot. Each sensor is a line that originates from the surface of the robot. A sensor outputs the distance to the nearest object in its direction. The output of the sensors is bound between 0 m and 2 m. The upper limit of 2 m helps the neural network in the Q-function to learn more efficiently. The reasoning for that is similar to why inputs to a neural network are normalized in machine learning [26]. In total, 25 Sensors are distributed around the robot. The number of sensors is kept low at this stage of the project, but the total number of sensors may increase in the future.

The output values of the 25 sensors are fed as inputs into the Q-function (neural network). The intuition is that the Q-function will learn to associate particular sensors with the direction they are facing. This works well as long as the sensors are always looking in the same direction, but when the robot moves its joints the directions of the sensors will change.

The Q-function needs to know the angles of all joints to appropriately associate sensors with a direction. Therefore, the Q-function (neural network) also gets the 6 joint angles as an input. The joint angles are supplied in their radial notation, bound between $-\pi$ and $+\pi$.

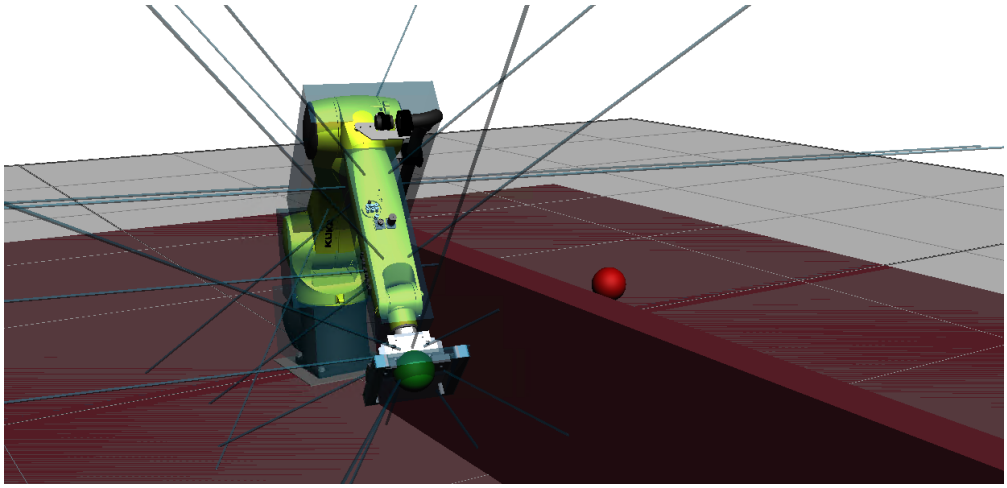


Figure 5.2: Robot with distance sensors

5.2 Reward Function with Penalty for Crashing

With the changes made in the previous section, an experiment is to be conducted to find out if the reward function of the previous section is able to perform well in the new scenario. As it got motivated in the previous section, the episode needs to be reset if a crash is detected. To discourage the agent from crashing, it gets a penalty of -200 for crashing as a negative terminal reward. The value of this penalty is chosen so that episodes where the agent crashes end up with a negative return for the episode. For the learning process, the specific value of the penalty is only of secondary importance, because the actual penalty of a crash is that the episode is terminated. When the episode is over, the agent does not have the chance to collect the positive terminal reward. In figure 5.3 an experiment with 3 runs is visualized. The episode return ranges from a return of about -200 when the robot has crashed immediately to a reward of over 1000 when the robot reached the target position.

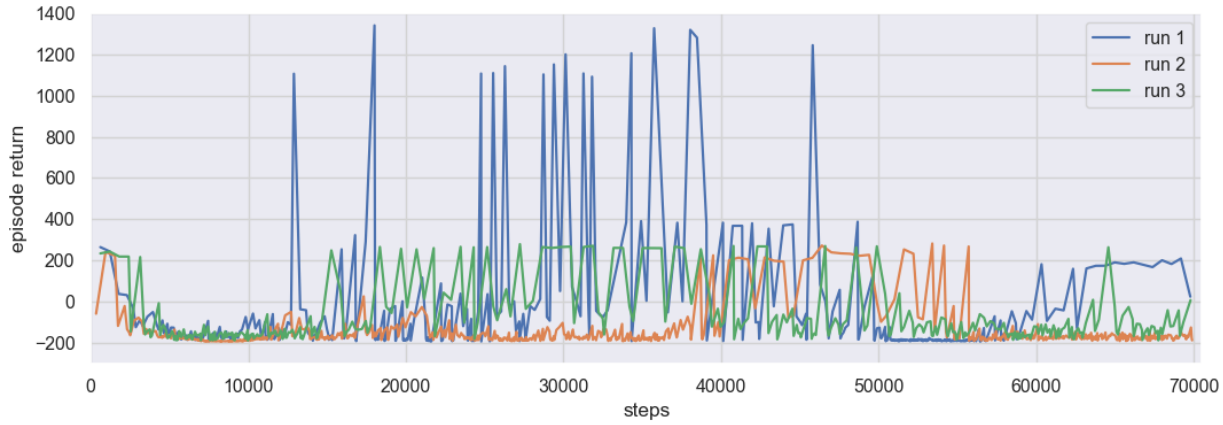


Figure 5.3: Experiments with -200 penalty for crashing

Only run 1 was able to move over the wall and reach the target position a few times. This is indicated by the spikes over 1000. At all points where the episode is close to -200, the robot crashed in just a few steps. At all points where the reward is positive, the agent was able to avoid the obstacles. All three runs start with a random policy, so at the start of the training process, they sometimes crash and sometimes they reach the 600 step limit without reaching the target position. At about 4000 steps, all three agents have learned that moving to the right (towards the target) gives more reward. But if the agents only move to the right, they hit the wall and get a terminal

reward of -200. This behavior is indicated by the averaged return going downwards in the range of 3000 to 13 000 Here, the agents behave similarly in all 3 runs. They consistently crash into the wall. At about 15 000 steps, run 3 learns to avoid crashes sometimes. At about 39 000 steps, run 2 learns to avoid crashes sometimes. Run 1 was able to reach the target position a few times, as indicated by the spikes over 1000 reward. But at 50 000 steps run 1 develops a policy that makes it crash consistently into the wall, giving it a consistent reward of about -200. The performance is not good, as none of the agents were able to learn a reasonable policy during the 70000 steps.

5.3 Reward Function with Linear Proximity Penalty

The question now arises why the agent is not able to avoid crashing into walls. As explained in section 5.1, the agent has 25 distance sensors to detect obstacles. The Q-function gets the distance information from all distance sensors. It can be observed that there are scenarios, where the Q-function for example gets the information that one sensor is very close to an obstacle, but the agent still moves in this direction, because the agent does not yet know what sensor corresponds with which direction. Subsequently, the robot crashes into the wall.
The agent has either not yet learned not to move in a direction where an obstacle is already close, or it has not yet connected a particular sensor with the corresponding direction and meaning. But why was the agent not able to learn this? A possible answer is that the feedback of the reward function for crashes is too sparse. The agent only gets feedback by the reward function after it has already crashed into a wall. This is sparse feedback because feedback towards avoiding obstacles is only given once per episode (when a crash happens).
The reward function should give more frequent feedback towards avoiding obstacles. The goal is to incentivize the agent not to move in a direction where an obstacle is close by. The agent should be penalized for being in close proximity to obstacles. This is achieved by incorporating a distance penalty into the reward function. The distance penalty for an individual sensor is defined by the function (5.1).

$$\text{penalty}_i = \frac{\text{distance sensor max value} - \text{distance}_i}{\text{distance sensor max value}} \cdot \text{max penalty} \tag{5.1}$$

$$\text{penalty sum} = \sum_{i=0}^{N} \text{penalty}_i \tag{5.2}$$

The distance sensor max value is 2 m. For distances that are more than that, the sensor just returns 2 m. Function (5.1) returns a bigger penalty the smaller the distance is. The maximum penalty per sensor is -0.1 for now. The penalties for all $N = 25$ sensors are summed up as the penalty sum. The penalty sum is calculated after every step and added to the total reward of the step.
The distance penalty for one sensor is visualized in figure 5.4.

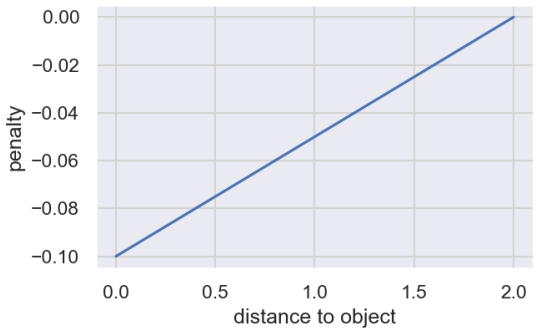


Figure 5.4: Distance penalty for one distance sensor visualized

If the distance sensor shows a distance of 2 m, there is no penalty. If the distance sensor shows a distance of 1 m, the penalty is $0.5 \cdot (-0.1)$. And if the distance sensor shows a distance of 0 m, the penalty is $1 \cdot (-0.1) = \text{max penalty}$. In figure 5.5 the described reward function is utilized to train the agent over 64 000 steps.

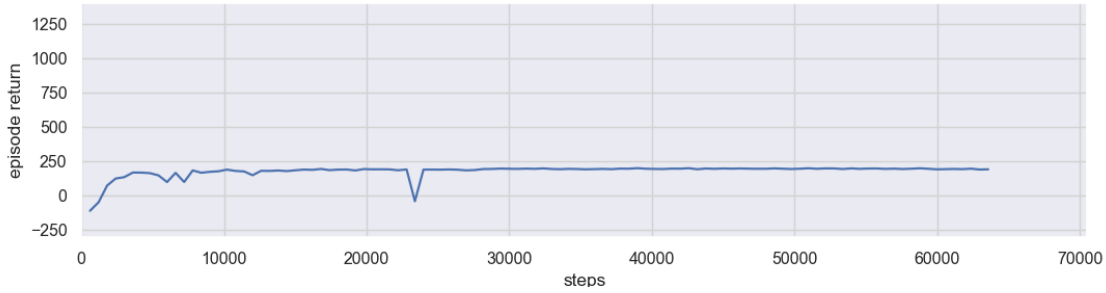


Figure 5.5: Experiments with linear distance penalty

In the first episode, the agent moves randomly. It does not hit the wall or the ground. As the robot is close to obstacles, the cumulated reward in the first episode is negative. Then the agent learns very quickly to avoid obstacles. The only time, the agent crashes in the experiment is at step 23 000 This is indicated by the small downwards spike. The agent is not able to reach the target position. This is indicated by the fact that the episode return is never close to 1000 , which is the terminal reward for reaching the goal.
To investigate, why the agent was not able to achieve the goal, a screenshot of the 3D viewer in the environment is presented in figure 5.6. The screenshot is from a typical episode during the training. Observing the agent in the 3D viewer, it became apparent that the agent prioritized moving far away from obstacles over moving towards the target position. In the screenshot, the agent positioned itself in such a way, none of its sensor beams touch any obstacle. In such a position the distance penalty sum is zero. In the screenshot, the robot leans in the direction of the target position. The agent tries to be as close to the target position as possible without any of its sensor beams touching obstacles.

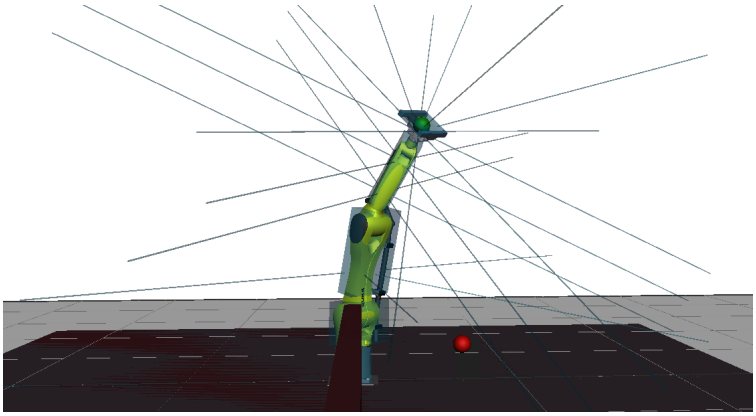


Figure 5.6: Linear distance penalty, agent avoids obstacles at all cost

This is a great example of an agent that optimizes the reward specified by the researcher but does not quite do what was intended. Theoretically, ignoring the short-term penalty to reach the target position and collecting the 1000 terminal reward would result in a greater reward than moving away and waiting. The agent is not able to learn this behavior, because it is not incentivized to explore the correct state space. This is the classic RL exploration problem. The reward function with a linear distance penalty is hindering the agent from efficiently exploring the environment.
A possible adjustment to solve this problem could be to reduce the max penalty and strike a better balance between the distance penalty and the reward for moving closer to the target position. But

this is not a good approach. The reason for that and a better solution are explored in the next section.

5.4 Reward Function with Polynomial Proximity Penalty

Let's envision a robot in an industrial production environment with a few obstacles, like a cage, a conveyor belt, a large box with raw materials and another robot. Now let's envision this robot moving some material from the box to the conveyor belt. Now envision yourself operating the robot with a controller in your hand. You are moving the robot around whilst avoiding obstacles. What obstacles are you paying the most attention to? If you are in the middle of the environment and all walls are 2 m away from the robot, there is no need to put any focus on them. If an object is 2 m away, you are not concerned about moving a bit closer to it, as it is still far away. If it is 1 m away, there is still no need for concern. If an obstacle is 0.5 m away, maybe you start getting aware of it. If an object is 0.25 m (approx. the height of DIN A4 paper) away, you start focusing on it and pay close attention not to get much closer.
This information needs to be represented by the reward function for the distance penalty. In the state-reward space, the agent wants to move in a direction, where the derivative of the reward is the highest because, in this direction, the cumulated reward grows the quickest. With a linear distance penalty, the derivative is constant at all distances. The agent puts as much focus on moving away from an obstacle that is already a meter away, as it puts focus on moving away from an object that is just 1 cm away. To incentivize the agent in a way that is similar to the thought experiment above, the derivative needs to be varied depending on the distance.
On far away objects, there should be a negligible derivative, so that the agent puts negligible focus on moving away from an obstacle that is already far away. On near objects, the derivative should be high, so that the agent puts more focus on moving away from that object. This behavior is modeled by the following reward function.

$$\text{penalty}_i = \left(\frac{\text{distance sensor max value} - \text{distance}_i}{\text{distance sensor max value}}\right)^{10} \cdot \text{max penalty} \tag{5.3}$$

$$\text{penalty sum} = \sum_{i=0}^{N} \text{penalty}_i \tag{5.4}$$

In figure 5.7 the new polynomial reward function is plotted alongside the linear one. The figure illustrates how the derivative requirements are met with this reward function.

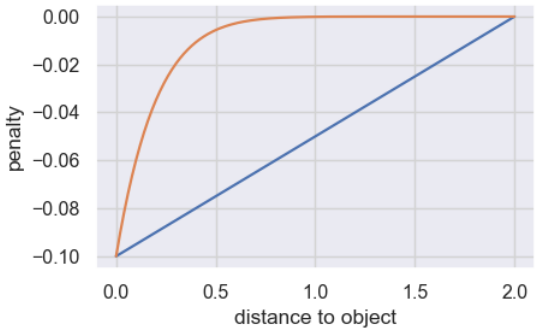


Figure 5.7: Visualization of linear and polynomial distance penalty

Modifying the exponent in the reward function (5.3) will change the derivative distribution.



How tight the agent will move around obstacles can be influenced by the exponent in the reward function for the distance penalty (5.3).

The described reward function for the distance penalty (5.3) is combined with the increasing derivative reward function (4.5), a terminal reward of 1000 for reaching the target and a negative terminal penalty of -200. In figure 5.8 three experiment-runs are visualized.
The experiment-runs start similarly to the experiment with the linear distance penalty. It appears that the likeliness of crashes increases after 10 000 steps of training. at around 17 620 steps, run 3 has an episode where it reaches the goal. The percentage of successful episodes increases over the duration of the training. It appears that at the end of the training also the percentage of episodes with crashes decreases.

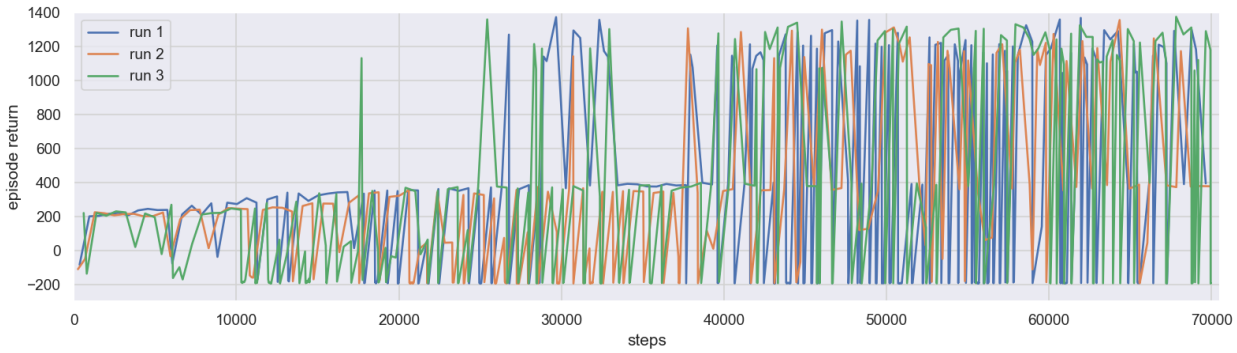


Figure 5.8: Experiment with a non linear distance penalty with max penalty = −0.1

This experiment proves that a non-linear distance penalty is far superior to the linear version.
To find out what a good value for the max distance penalty is, different values for it are tested. An educated guess for sensible values is $-0.1, -0.04, -0.01$. These three values are tested with three experiment-runs each.

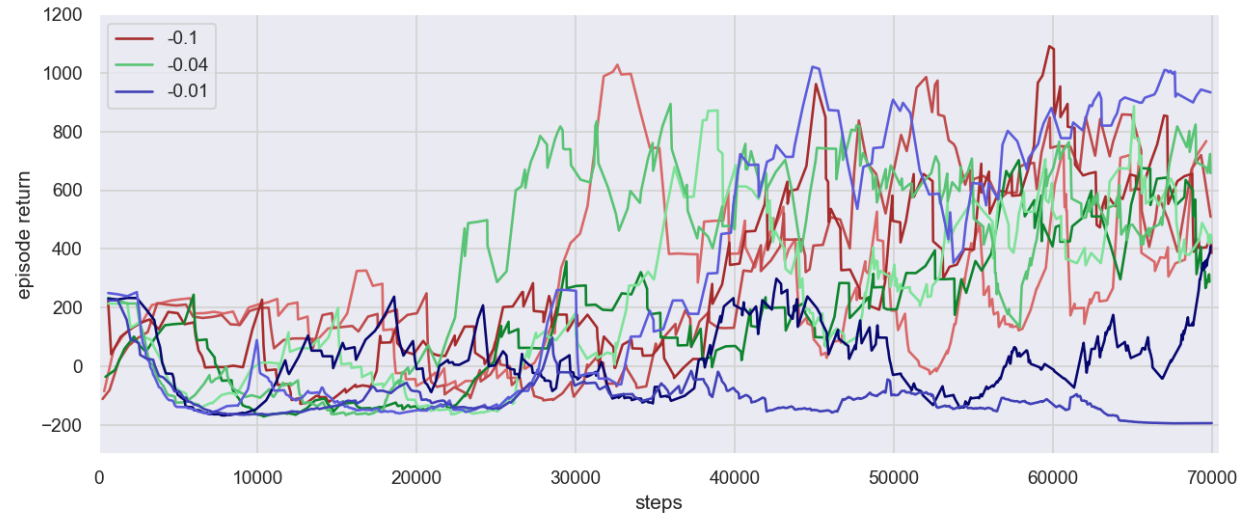


Figure 5.9: Comparison between different values for the max distance penalty

The most striking result is the performance of the blue plots, representing the max distance penalty = −0.01, because it has the best as well as the worst performance. Stability and consistency are crucial for more complex tasks in the future. Thus, a max distance penalty of −0.01 or higher can not be recommended. The experiments with a max distance penalty of −0.1 (red) and −0.04 (green) perform similarly. Based on the limited training time, no accurate judgement on the best value for the max distance penalty can be made.
Because the dynamically reduced positive terminal reward has been rejected in section 4.4, episodes

where the agent takes longer to reach the target still report a larger episode return than episodes where the target was reached in fewer steps.



It can be assumed that agents with a higher max distance penalty make greater detours around obstacles as it has a greater incentive to move away from obstacles. Therefore the smallest but still stable max distance penalty should be chosen for the best performance. This hypothesis can not be further analyzed in this thesis due to time constraints.

5.5 Analyzing the Intra Episode Performance

In figure 5.9 it becomes apparent that the performance of the agent in the scenario with a wall is substantially worse than in the first scenario without obstacles. This section discusses what might be the cause for such a difference in performance.
To get a better insight into the behavior of the agent during the episodes, in figure 5.10 the distance between the TCP and the target is plotted for the last 10 000 steps in one of the training runs with a max distance penalty of −0.1. The episode resets are indicated with markers, where the color shows the cause for the episode reset. At the green markers, the agent has reached the goal. At the yellow markers, the step limit of 600 steps per episode was reached. At the red markers, the agent crashed into an obstacle.

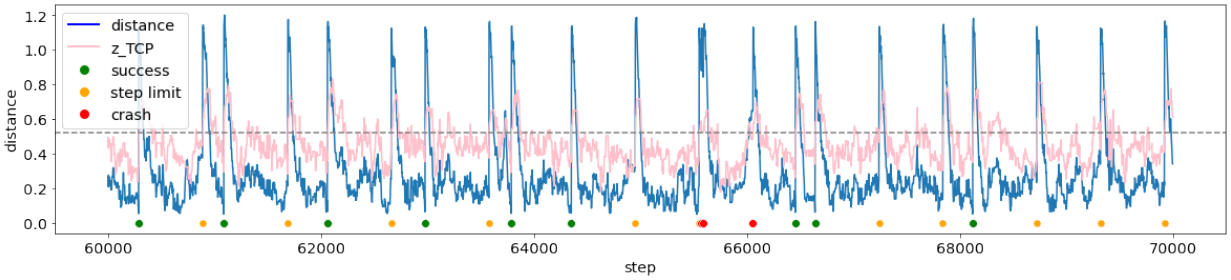


Figure 5.10: The distance between the TCP and the target position for the last 10 000 steps in a training run with a max distance penalty of −0.1

The blue line plots the distance between the TCP and the target position. The pink line plots the z-coordinate (i.e. the height) of the TCP. The gray dashed line shows the height of the TCP at the start position.
After a reset, the TCP is set to a position on the opposite side of the wall from the target at a distance of about 1.12 m. The pink line indicates that after a reset, the agent first moves upwards to get over the wall. After the agent has traversed over the wall, it moves downwards again, to reach the target.
Based on the spike downwards of the blue line after a reset, it can be deduced that the agent is efficient at moving over the wall close to the target. The agent seems to have some difficulties once it gets close to the target. At around 0.2 m distance to the target, the agent fails to find its way for the last few centimeters.
In the last centimeters of the episode, the scenario is pretty much the same as in the previous chapter 4, so a similar performance should be expected. To figure out why this performance gap exists, a few experiments are performed. Firstly, an experiment with the second scenario (as described in section 5.1) with crashing disabled and without a distance penalty is conducted. The Q-function still gets all the inputs defined in section 5.1. In figure 5.11 only the last 10000 steps of the above described experiment are plotted.
In this plot, there are no red markers indicating crashes, as crashes have been disabled for this experiment. In all but three experiments the agent reached the goal within the 600 step limit. This is indicated by the three yellow markers. As indicated by the pink line, the agent is not moving

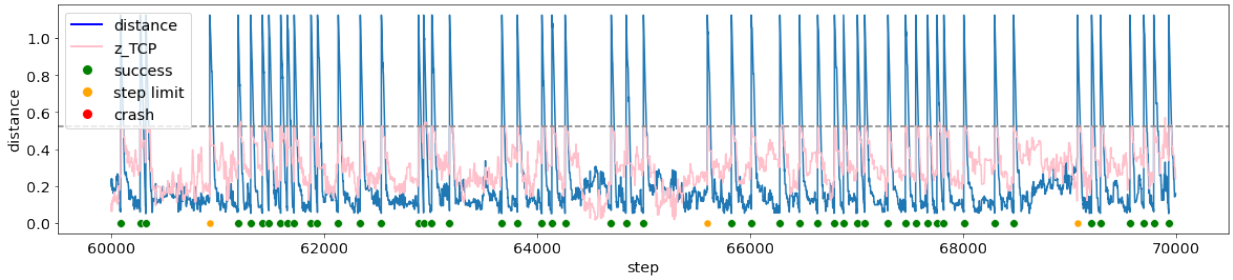


Figure 5.11: The distance between the TCP and the target position for the last 10 000 steps in a training run with crashing disabled and no distance penalty

upwards at the beginning of an episode, because it does not have to avoid hitting the wall. For the experiment under consideration, the performance of the agent in the last 10 000 steps of the training is significantly better than in the experiment with crashing enabled and the distance penalty. This performance discrepancy may be explained by the shorter path to the target and the fact that the agent does not have to learn to avoid hitting walls.

When compared to the experiments in the previous chapter, the performance is still worse than expected. The agent is still struggling at the last few centimeters towards the target.

Now the only difference between the experiment in section 4 is the new start and target position and that the distances of the distance sensors are an input to the Q-function. To figure out if all the new inputs are confusing for the Q-function, an experiment with the new start and target position, but only with the target position and the current TCP position as input is conducted. This experiment is plotted in figure 5.12.

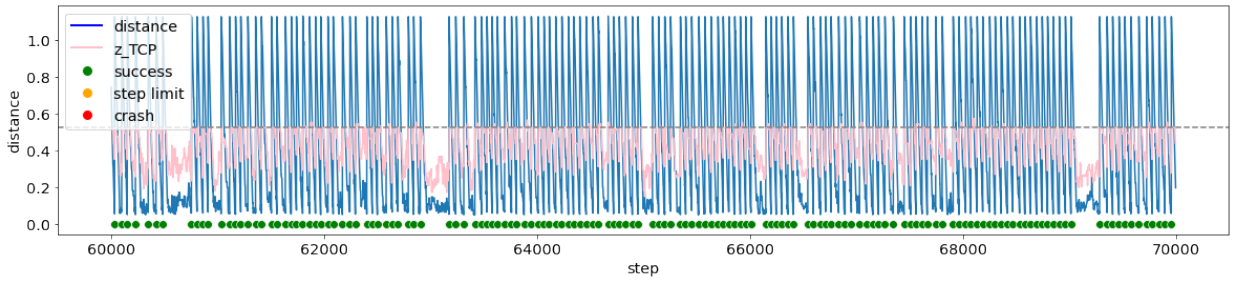


Figure 5.12: The distance between the TCP and the target position for the last 10 000 steps in a training run with crashing disabled, no distance penalty and only 6 inputs to the Q-function

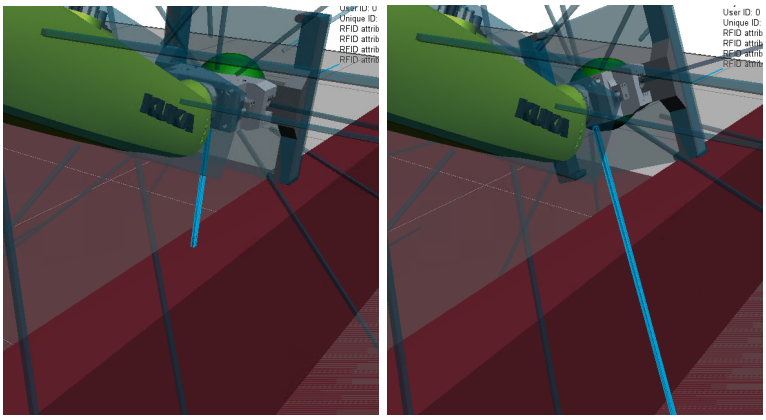
In this experiment, the final performance is very good. The agent reaches the target position in every episode within the 600 step limit. In most of the episodes, the agent moves straight into the target position without much struggle at the last centimeters. Only a few times, the agent takes a bit longer to reach the target. This may be explained by the 5% exploration rate.

The experiments in this section are not conclusive enough to give a good answer, on where the problem at the last centimeters in the trajectory comes from. However, based on the last three experiments it can be suggested, that increasing the number of inputs to the Q-function slows down the learning progress. To figure out if the agent with more inputs to the Q-function will eventually reach the same performance as the agent with fewer inputs, a longer experiment would need to be conducted.

Now, it might seem that decreasing the number of sensors could help speed up the learning process. But, in the next section, a counterargument is proposed, on why the number of sensors should be increased.

5.6 Agent Purposefully Makes Itself Blind to Obstacles

Watching the agent in the 3D viewer, an interesting behavior can be observed. To move towards the target position, the robot first needs to move up and over the wall. To achieve an optimal trajectory, the robot should move narrowly over the wall. At the position where the gripper (i.e. TCP) is just over the wall, as shown in figure 5.13 (a), the distance sensor pointing downwards shows a very close distance. The agent is incentivized to not have sensors that show a small distance to obstacles. In this situation, it is expected that the agent moves up to increase the distance, or moves left to move away from the wall and simultaneously move towards the target. However, it can be observed that the agent sometimes tilts its gripper to the left or right, as shown in figure 5.13 (b) instead of moving away from the obstacle.



(a) agent sees wall (b) agent moved axis 6

Figure 5.13: The agent purposefully makes itself blind to obstacles.

Tilting the gripper causes the distance sensor to miss the wall and instead detect the floor, which is further away. This behavior is unexpected because the agent purposefully makes itself blind to obstacles. The reason for the behavior is simple. The agent gets less distance penalty if its sensors do not see the obstacles, because then the reward function also does not receive the information that an obstacle is close. With this setup of the reward function, the agent gets rewarded for blinding itself.

What problems does this cause? Firstly, the agent is wasting steps if it is tilting its gripper to avoid obstacles. This results in an inefficient and suboptimal trajectory. Secondly, the agent has a greater risk of crashing when it purposefully blinds itself to obstacles. The NN used in the Q-function does not have any memory from the state in the previous step. If the distance sensors do not see the wall, the agent has no chance of knowing that there is an obstacle. The Q-function could be equipped with a long short-term memory (LSTM) to enhance its temporal perception.



To avoid the problem of willful blindness, the number of distance sensors should be increased substantially. Another option is to calculate the distances to obstacles in a different way. A distance sensor with a cone-shaped field of view could be utilized. Or a physics engine could be used to calculate the distance between the complex shapes of the robot and obstacles in multiple directions. Another solution would be to give the Q-function the distance sensors as input, but calculate the distance penalty based on the smallest distance of the robot surface to an obstacle. For this approach, a physics engine would be necessary to calculate distances between complex surfaces.



Adding more distance sensors to the robot may improve the final performance of the agent and may help mitigate the willful blindness problem. But, adding more distance sensors to the agent may also slow down the learning process. A sensible compromise between these two options has to be made.

Chapter 6

State of the Art for Industrial Robot Programming with RL

In this chapter, relevant literature about and around industrial robot programming with RL is presented. The purpose of this section is to highlight and explain related literature. The review shows where related literature is similar, and where the differences are. The reviewed literature consists of a paper about other simulation environments for RL, a paper summarizing RL in robotics, three papers implementing RL for robotics and reward functions for robotics RL and three papers that propose methods for alleviating the problem of manually developing a reward function.
In the paper **A Systematic Comparison of Simulation Software for Robotic Arm Manipulation using ROS2** [27], the authors compare the most popular simulation software for robotic arm manipulation. The environments are compared in two scenarios: a pick and place task and a throwing task. Based on these tasks, the accuracy of the physics engine, the simulation speed, and the resource usage are compared. A new approach is the utilization of Robot Operating System (ROS) version 2. The paper concludes with: "From our review and experimental results, we found that current robot simulation software could not be used to develop a digital twin. This is because the simulators considered in this paper cannot maintain a repeatable simulated scene over time. We hypothesize that a continuous feedback mechanism is needed between the simulation and reality similar to [28] in order to maintain an accurate representation of the real environment." [27, p. 7].

The paper **Review of Deep Reinforcement Learning for Robot Manipulation** [29] is a literature review that focuses on different RL algorithms. The authors review a total of 42 papers. The paper finds that the key problems limiting the progress in robotic RL are *sample inefficiency*, *Exploration and Exploitation*, *Generalization and Reproducibility* and the *sim2real gap*. The authors say: "Sample inefficiency is one of the main reasons that seriously limit the applications of RL in robot manipulation". The paper states "The current winner is model-based "shallow" algorithms such as Probabilistic Inference for Learning COntrol (PILCO)" [29, p. 3]. For the problem of Reproducibility, the authors address that "we would possibly need [to] agree on proper experimental methodologies and proper evaluation methods and metrics to address the problem" [29, p. 3]. The problem that there are no formally agreed upon evaluation methods and metrics also reflects in this thesis.
The **Optimal Reward Framework** proposed by Singh Et al. [30,31] discusses the reward function as an interplay of an extrinsic reward for the externally provided goal and an intrinsic reward for exploration, manipulation, play, etc.
The paper **Deep Reinforcement Learning-Based Safe Interaction for Industrial Human-Robot Collaboration using Intrinsic Reward Function** [32] builds on top of the Optimal Reward Framework proposed by Singh Et al. [30, 31] to train a collaborative robot (Cobot). They want to "reduce [the] accumulated risk [of crashes] and assure task completion time" [32, p. 4]. The input to the agent is an 18-dimensional vector, consisting of the robot state, the target coordinates, and the current obstacle position. Similar to my thesis, the goal in the paper is to move from

an initial position to a target position whilst avoiding an obstacle. "The action that the agent takes is the velocity for each joint" [32, p. 10]. The authors use the Deep Deterministic Policy Gradient (DDPG) algorithm to train their agent. Furthermore, they "apply [their] DPG-based reward function optimizing approach to the DDPG algorithm and propose the intrinsic reward DDPG (IRDDPG) algorithm to learn the safe control policy" [32, p. 14]. In short, they separate the reward function into an intrinsic and an external part where the intrinsic reward function is trained alongside the policy and evaluated based on the external reward function. This helps them to fine-tune their (manually designed) reward function. In [32, figure 7] the performance of their approach is plotted. The agent reaches convergence after about 6000 episodes ($\lessapprox$ 900 000 steps). This is about 10x more steps than in the experiment described in figure 5.7.

The paper **Deep Reinforcement Learning With Optimized Reward Functions for Robotic Trajectory Planning** [33] is similar to this thesis. The authors train a robotic arm with RL to move from a starting position to a target position whilst avoiding obstacles. In contrast to the reward function described in my thesis, here the target position also defines a TCP orientation. The authors propose two new reward functions. The "Azimuth reward function includes the position reward function and the orientation reward function. The position reward function is built by Gaussian distribution and Triplet Loss function; orientation reward function is modeled by Coulomb's law" [33, p. 2]. Secondly, they "propose another reward function at subtask-level to provide global guidance for the agency" [33, p. 2]. The obstacle avoidance reward function is based on the smallest distance between the TCP and the obstacle and shaped with the gaussian function (in section 5.3 in my thesis, a polynomial is utilized for shaping). In contrast to the reward function designed in chapter 5, this reward function only knows the distance of obstacles to the TCP and can therefore not penalize movements that risk a crash with another part of the robot, like the lower arm of the robot. To evaluate the performance, they (similar to my thesis) use "three indicators, including convergence rate, mean value and standard deviation of convergence. Convergence rate and mean value are used to verify the learning efficiency, and the standard deviation is for robustness." [33, p. 6]. They use the Distributed Proximal Policy Optimization (DPPO), Deep Deterministic Policy Gradient (DDPG) and Asynchronous Advantage Actor-Critic (A3C) methods for training. The best performing method is A3C, with the Azimuth reward function, which converges after only about 9 episodes [33, table 2]. This is a better performance than achieved in my thesis. This result would need to be further investigated because 9 episodes are very little time for an agent to learn a robust policy.

In the paper **Continuous control actions learning and adaptation for robotic manipulation through reinforcement learning** [34] a robotic arm (Franka Emika Panda) with a gripper is trained to position its gripper over a cube, grasp it, and lift it off the table. This paper is reviewed because it shows how a similar goal as in this thesis can be solved with a different simulation environment and a different action space. The authors use the Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC) algorithms. The state-space is a 46-dimensional vector containing "joint positions, velocities, gripper joint position, end-effector pose and velocity, [the] cube's position and a relative position vector of cube and grip site" [34, p. 6]. The action space is 8-dimensional and continuous. It contains the velocities for the 7 axes and the gripper position. The dense reward function is split into the gripping part and the gripping and lifting part. The reward function for the approaching is a weighted sum of three functions. The first is the negative euclidean distance between TCP and the cube, shaped with the tanh() function. The second penalizes high velocities of the robot if it is within 8 cm of the cube. The third rewards the agent for an open gripper. The gripping and lifting part of the reward function can be read in the paper [34] on page 5. Some extra reward functions are added to the sum for special objectives. To manage redundancies, a penalty is "added to the reward function in the case that joint positions come close to the joint limits" [34, p. 5]. A smoothing reward is added to the smooth control signals, to avoid jerky behavior and limit joint accelerations. To avoid collisions, a (sparse) penalty is given, if the robot collides with obstacles. The authors noted that the training efficiency increased if the special objectives were trained in series and not all at once. The authors use the MuJoCo

simulation environment. “The simulation has been roughly sped-up 2000 x faster than real-time in order to train faster (requiring approximately 4 and a half hours of wall-clock time)” [34, p. 7]. In the experiment with the PPO algorithm, the agent learned a successful behavior after 5 million timesteps.
The next three papers explore how RL for robotics can be done without manually designing a dense reward function.
In the book section **Inverse Reinforcement Learning** [35], instead of learning a policy using a reward function, inverse reinforcement learning (inverse RL) considers the problem of extracting a reward function from an observed (nearly) optimal behavior of an expert (human) acting in an environment. The reward function can be approximated by a simple feed-forward neural network.
In the paper **Deep Reinforcement Learning from Human Preferences** [36], the authors switched out the reward function for a reward predictor, based on a neural network. During the training process, the reward predictor is trained alongside the policy. To train the reward predictor, two short videos of the robot performing in the simulation are shown to humans. The human selects the clip that shows better behavior. Based on this human feedback, the reward predictor is trained, which in turn is used to train the policy. The method is evaluated and tested in multiple (simple) tasks in the MuJoCo simulation environment. The authors conclude: “With 700 labels we are able to nearly match reinforcement learning [with a manually designed reward function] on all of these tasks. Training with learned reward functions tends to be less stable and higher variance, while having a comparable mean performance.” [36, p. 7].
In the paper **Hindsight Experience Replay** [13], the authors propose a new technique for deep RL. With Hindsight Experience Replay (HER), a sparse reward function is utilized. As discussed earlier and illustrated in the experiment in section 4.1, the agent struggles to learn anything, because every episode is a failure. What HER can do in this scenario is take an episode, where the agent did not reach the target position, and then change the target position retrospectively to the position the agent is at in the end. The agent gets a positive reward in the last step for achieving the goal. The DQN algorithm has a replay buffer, where all experiences are saved as a chain (trajectory) of (state action reward) tuples. For the HER technique to work, the target position needs to be encoded in the state and be an input to the Q-function, as it is in the scenario of this thesis. With the HER technique, one episode of 600 steps gathered in the simulation where the agent did not reach the target position can be modified by changing the three target coordinates in the state in each of the 600 tuples in the replay buffer to the coordinates reached in the last step. The reward in the last tuple is set to one. The Q function can then be trained on this synthetic experience. This procedure can be done with all other positions (coordinates) that are visited during the episode. With this technique, up to 600 episodes worth of experience can be generated from one episode of simulation. This drastically increases the sample efficiency.

Chapter 7

Findings and Observations

The goal of this thesis is to figure out how the reward function impacts the performance of the RL agent and how the reward function should be designed to maximize the performance. In the methodology chapter, the performance was broken down into the three aspects: time to convergence, final performance of policy and stability. The reward function is developed in two scenarios, one without and one with obstacles.

7.1 Findings for the First Scenario

For the first scenario (without obstacles) four alternatives for the reward function are examined. The **sparse reward function** performs exceptionally bad. The agent is not able to learn because a sparse reward provides too little feedback. The reward function with a **negative dense reward** performs well in the time to convergence and final policy performance aspect, but the stability is bad because the policy randomly diverges. This stability issue is analyzed with the two following experiments. In the experiment in figure 4.4, the positive terminal reward is set to zero. This decreased the final policy performance and has a worse time to convergence, but showed no diverging policies in the three experiment-runs. In the experiment in figure 4.5, the learning rate α is decreased by a factor of 10. This measure drastically reduced the overall performance. Unfortunately, no concrete conclusions regarding the effects on stability can be drawn, because the training could not be simulated for long enough. Both, reducing the terminal reward and reducing the learning rate showed potential in reducing the issue of the policy randomly diverging, but the experiments could not be performed long enough to be sure. Further research would be necessary to confirm this.
Because a random divergence of the policy has the potential of ruining an entire training process, such instability needs to be avoided. Especially in production, where the scenario would be more complex and take more time to train the cost of a failed training process is high. For further experiments, the reward function is designed in such a way that the average reward in a single step is positive.
In section 4.3, +1 is added to the previous reward function to create a **positive dense reward** function. Even though the only difference to the previous is an added constant, the positive dense reward function has a worse overall performance. But no diverging policies were observed during the training with the positive dense reward function. An explanation for this behavior could not be provided in this thesis.
Furthermore, the effect of **reward shaping using an exponent** is explored. The experiment in figure 4.8 demonstrates that reward shaping on the distance reward function has a significant impact on all three performance aspects. With an exponent of 0.4, the time to convergence, the final policy performance and the stability is improved over the baseline without reward shaping. The exponent for the distance reward should be in the range between 0 and 1. A value of 0.4 worked well in the scenario of this paper. In the paper [32], the authors use the Gaussian distribution to shape the reward function.

The experiment in figure 4.9 tests if the agent would learn to wait in front of the target. The experiment shows no signs of such behavior. Later it has been shown that the discount factor is the reason. Further, it has been shown that equation (4.10) needs to hold, to ensure that the agent does not learn to wait in front of the goal.
In section 4.4, an attempt was made to further incentivize the agent to reach the goal faster. With a **dynamically reduced reward**, the terminal reward gets lower the longer the agent takes to reach the goal. In figure 4.11 no significant improvements in any of the three performance aspects can be detected.
The complete reward function developed for the first scenario without obstacles is presented as Python code below. This reward function combines all findings from chapter 4. This function should provide the best performance for the first scenario. All parameters are marked with the upper-case camel-case notation. How to choose the parameters is explained in section 7.3.

Listing 7.1: reward function for: move to target position

```
def reward_Function(TCP_coordinates, steps_count_this_episode):
    x_TCP, y_TCP, z_TCP = TCP_coordinates
    # calculate the Euclidean distance to the target coordinates
    distance = np.sqrt((X_TARGET - x_TCP) ** 2 + (Y_TARGET  - y_TCP) ** 2 + (
    Z_TARGET  - z_TCP) ** 2 )
    # if the step limit is reached, the episode is done
    if steps_count_this_episode >= STEP_LIMIT:
        done = True
    # calculate the reward
    reward = 1 - (distance / POSITION_REWARD_MAX_DISTANCE) ** 0.4
    # if the TCP is in close proximity to the target, the agent gets the positive
    terminal reward and the episode is done
    if distance <= GOAL_ACHIEVED_AT_DIST_IN_M:
        reward = POSITIVE_TERMINAL_REWARD
        done = True
    return reward, done

```

7.2 Findings for the Second Scenario

In chapter 5, the agent learns to move to a position whilst avoiding obstacles. The first finding, explained and motivated in section 5.1, is that the agent needs a **perception of the environment** to train efficiently. The virtual commissioning software only provides limited functionality. The solution is 25 distance sensors, which provide the distance to the nearest object. The maximum distance a sensor outputs should be limited to some value with the same magnitude as the working space of the robot. To enable the agent to learn which distance sensor corresponds to which direction, the joint angles need to be added to the input of the Q-function. This implementation is dissimilar to the works in [32, 33]. The authors of the paper [32] give the position and orientation of the obstacle directly to the Q-function. The problem with this approach is that it is very specific to a scenario. If a second obstacle is added, the agent has to be trained from scratch. In contrast, the distance sensor approach can be utilized in many scenarios with multiple static and dynamic obstacles.
In the first experiment, the same reward function as in the previous chapter is used, except, a **penalty for crashing** is added. This experiment shows that the sparse feedback from the crash penalty is not enough for the agent to learn efficiently.
To provide denser feedback to the agent, a **linear proximity penalty** is added to the reward function. The proximity penalty is realized as a sum over all 25 sensors, where the penalty increases for smaller distances. The result of the experiment is that a linear distance penalty is difficult for the agent to learn because now the agent has more incentive to move away from an obstacle than it has to move towards the target position.

To solve this problem, in the next experiment a **polynomial proximity penalty** is utilized. The proximity penalty is shaped with an exponent with the value of 10. With this reward shaping, obstacles that are further away are not as important as obstacles closer to the robot. With this reward function, the agent was able to learn to consistently move over the wall and towards the target position.

In the next experiment, the best value for the **maximum distance penalty** is experimentally determined. The maximum distance penalty is the maximum penalty per sensor. In the experiment, visualized in figure 5.9, three values are selected [-0.1, -0.04, -0.01]. Of these values, the value of -0.01 has performed the worst. Both, -0.1 and -0.04 have performed similarly well, and seem to be a good value for the maximum distance penalty. However, this value would need to be adjusted if the total number of distance sensors is changed.

In section 5.5,the **performance in an episode** is analyzed. The visualization in figure 5.10 gives insights into the performance of the agent. Even after a relatively short training process, the agent is quite good at avoiding the obstacle and moving over the wall. The graph shows that the agent struggles at the last few centimeters towards the target. The experiments performed in the section suggest that more inputs to the Q-function slow down the learning process. Further research is needed to figure out why the agent struggles in the last few centimeters, and how the performance in the last steps can be improved.

In the last section of the chapter, an interesting observation is discussed. **The agent blinds itself to obstacles**. When observing the agent in the 3D visualization of the virtual commissioning software, it can be noticed that the agent sometimes tilts its gripper, so that its distance sensors no longer see the obstacle, instead of actually moving away from the obstacle. This behavior increased the risk of crashes and makes the trajectory unnecessarily longer. The problem could be counteracted by increasing the number of sensors or using sensors with a field of view instead of just a one-dimensional ray. Another solution could be to give the Q-function the values from the distance sensors but calculate the distance penalty based on the smallest distance from the robot surface to an object. With such a setup the agent can't cheat the inputs to the reward function by tilting its sensors away.

The complete reward function developed for the second scenario with obstacles is presented in the following Python code. This reward function combines all findings from chapter 4 and chapter 5. This function should provide the best performance for the second scenario with obstacles. All parameters are marked with the upper-case camel-case notation. How to choose the parameters is explained in section 7.3.

Listing 7.2: reward function for: move to target position whilst avoiding obstacles

```
def reward_Function(TCP_coordinates, steps_count_this_episode, distance_sensors, crashed):
    x_TCP, y_TCP, z_TCP = TCP_coordinates
    # calculate the Euclidean distance to the target coordinates
    distance = np.sqrt((X_TARGET-x_TCP)**2 + (Y_TARGET-y_TCP)**2 + (Z_TARGET-z_TCP)**2 )
    # caclulate the reward
    reward = 1 - (distance / POSITION_REWARD_MAX_DISTANCE)**0.4
    # calculate distance penalty
    distance_penalty = 0
    for sensor in distance_sensors:
        distance_penalty += ((DISTANCE_SENSORS_MAX_VALUE - sensor.distance) / DISTANCE_SENSORS_MAX_VALUE)**10 * (MAX_DIST_PENALTY)
    reward = reward + distance_penalty
    # if the robot has crashed, the episode is done, and the agent gets a penalty for crashing
    if crashed:
        reward = NEGATIVE_TERMINAL_REWARD
        done = True
    # if the step limit is reached, the episode is done
    if steps_count_this_episode >= STEP_LIMIT:
```

```
18          done = True
19      # if the TCP is in close proximity to the target, the agent gets the positive
        terminal reward and the episode is done
20      if distance <= GOAL_ACHIEVED_AT_DIST_IN_M:
21          reward = POSITIVE_TERMINAL_REWARD
22          done = True
23      return reward, done
24
```

7.3 Parameters for the Reward Function

In the previous section, the overall structure of the reward function is explained and presented. In table 7.1, all parameters, used in the reward function are explained and the value, which was used or is recommended for the scenario in this thesis is presented. All these parameters influence the performance of the RL system.

Table 7.1: **Parameters for the reward function**

Parameter	Value	Explantation
position reward max distance	2 m	This is used to ensure, that the position reward is in the range of 0 to 1. It can be set to 2× maximum reach of the robot.
goal achieved at distance	0.05 m	The target coordinates can not be reached exactly. This parameter defines the distance at which the target position is considered achieved.
positive terminal reward	1000	The reward for reaching the target position.
negative terminal reward	−200	The penalty for crashing.
step limit	600	The maximum number of steps in an episode. After that, the episode is reset.
distance sensors max value	2 m	Maximum distance the distance sensor outputs, The value should be in the same magnitude as the maximum robot workspace.
maximum distance penalty	$-0,04$	This value balances the incentive of the agent to move towards the goal against the incentive to move away from obstacles. The value should be adjusted if the number of sensors is changed.

Besides the parameters used inside the reward function, other parameters were found to have an important influence on the performance as well. These parameters are explained in the following paragraphs.

The **number of distance sensors** and their distribution over the surface of the robot are important for the performance of the robot. The distance sensors enable the agent to perceive its environment so that it can avoid obstacles. The number of sensors is a compromise between better perception and less efficient learning. The number of sensors should be increased to more than 25, to improve the perception of the agent. But as explained in section 5.5, the learning process may take longer, if the Q-function has more inputs to learn from. If the number of sensors is changed, the maximum distance penalty parameter would need adjustment.

In section 4.4 the impact of the **discount factor** was discussed. The discount factor is a measure in the DQN algorithm to improve the stability in the training process. A side effect of the discount factor is that it incentivizes the agent to reach the target position faster. As illustrated in equation (4.10), the discount factor needs to be balanced with the terminal reward and the maximum reward per step, so that the agent does not want to wait in front of the target.

A stable value for the **learning rate** in this scenario is between 0.001. Lower or higher value for the learning rate could be utilized to increase stability or increase the learning speed.
The action used in the scenario of this thesis is to move one joint by 2°. This action was chosen to increase the speed at which the agent can learn the task. Such a granular action would probably not be precise enough to perform meaningful tasks in the real world

7.4 Other Observations

In comparison with similar research, the speed at which the agent learned to achieve the task is outstanding. This can be attributed to the (uncommon) discrete action space, which enables a sample efficient algorithm. It should be clarified here, that the speed measured in simulation steps (i.e. the sample efficiency) is very good. However, the speed measured in wall clock time is not good, because the simulation environment (ISG virtuos) is only able to simulate in real-world time. One goal of the project [4] is to improve the simulation speed of ISG virtuos.
There are key differences between the system proposed in this project and other research. With most RL research for robot trajectory planning, the goal is to have a dynamic agent controlling the robot in real-time. An agent could be trained to pick up randomly placed objects from a box and place them on a conveyor belt. The agent is equipped with a sensor, like a depth camera or a stereo camera setup to perceive its environment. The benefit of such a setup is, that the agent can dynamically adapt its trajectory on the fly. In contrast, the goal of this project is to train the agent in the simulation environment and record its best trajectory for solving the specific task in the specific scenario. This trajectory is logged as individual NC commands. These NC commands can then run on the robot in the real world. The benefit of this approach is that the robot does not have to be programmed by an expert, and it learns in the simulation environment how to solve the task as efficiently as possible.
The action space of the RL agent is limited to the capabilities of the NC file interface. In the experiments in this thesis, the actions were incremental relative movements of the robot joints. Such movements are very inefficient and slow because the robot can only move one joint at a time and not move the joints simultaneously. To have a chance of creating optimized trajectories, the action space needs to be changed to a multi-discrete or continuous action space. Furthermore, the inverse kinematic capabilities of the NC-kernel should be leveraged to enable actions as a movement in the coordinate system and not based on individual joint angles. One challenge with a continuous action space is, that algorithms with a continuous action space are typically less sample efficient.

Chapter 8

Conclusion and Future Work

In this chapter, the thesis is concluded and opportunities for future work are presented.

8.1 Conclusion

The goal of this thesis was to design the reward function so that the RL agent learns an optimized trajectory as fast as possible. The two requirements of an optimized trajectory and a fast learning process can be extrapolated to the three performance aspects defined in section 3.5 time to convergence, final performance of policy, and stability. In this thesis, the reward function is incrementally developed and evaluated to maximize these three performance aspects. The reward function developed in this thesis is able to achieve the specified task of moving from a start position to a target position whilst avoiding obstacles. In the thesis, I was able to improve the performance of the reward function drastically with a collection of improvements. If the developed reward function is optimal for the here-defined performance aspects can not be determined. The final reward function is presented at the end of section 7.2.
In chapter 4, the reward function for the scenario without obstacles is developed. The sparse reward function gives too little feedback for the agent to learn efficiently. The first insight is that a dense reward function is necessary to learn. The second reward function has a dense negative reward proportional to the distance of the TCP to the target position. This is learning fast and the learned trajectory is good, but it is unstable due to random divergences of the policy during the training process. The reason for the instability was not found. With the dense positive reward function, the time to convergence and the final performance of the policy is a bit worse, but no major instability was noticed. Because random divergences can be detrimental to the learning process, the dense positive reward function is deemed superior to the dense negative reward function. Utilizing reward shaping with an exponent between 0 and 1, the performance of the dense positive reward function can be improved significantly.
In section 4.4 it is experimentally and mathematically shown that the discount factor γ incentivizes the agent to reach the target in as few steps as possible. In an experiment, it is shown, that when further incentivizing the agent to be as fast as possible, the performance of the agent does not improve noticeably.
At the beginning of chapter 5 it is motivated why the agent needs extra sensors to learn the task of avoiding obstacles. As a proof of concept, 25 distance sensors are strategically distributed over the surface of the robot so that the agent can percieve its surroundings.
In the first experiment of chapter 5 it is demonstrated, that only a penalty for crashes is too sparse of a reward for the agent to learn efficiently. A dense proximity penalty is implemented in the reward function to give dense feedback for the obstacle avoidance behavior. The second experiment demonstrates that it is difficult to correctly balance the proximity penalty with the reward for moving towards the target. As a solution to that problem, the proximity penalty is shaped with an exponent with the value of 10. This makes the agent give more attention to the

closest obstacles. In the fourth experiment, it is experimentally shown that a good value for the parameter *max distance penalty* is in the range from -0.1 to -0.04.
In section 5.5 the performance within episodes is analysed. Unfortunately, the experiments were not conclusive on why the agent struggles at the last few centimeters. In section 5.6 an interesting phenomenon is presented, where the agent tricks the reward function, by making itself blind to near obstacles. Options for solving that problem are listed.

8.2 Future Work

A standout feature of the thesis is the usage of a simulation environment that has not been used for RL before in such a way. A comparison between virtuos and other simulation environments similar to the comparison in [27] has the chance of bringing up weak points that can be worked on, but also show standout benefits of virtuos. An interesting research question is to investigate if virtual commissioning can be utilized to bridge the sim to real gap in RL simulation.
Performing the experiments with the virtuos environment, it became apparent, that the simulation speed of the environment needs to be improved if it is to be used for RL training. Furthermore, the simulation environment needs to be parallelized, so that training data can be collected faster.
The action space used in the experiments in this thesis is quite limiting and it is clear, that an optimal trajectory can not be achieved with incremental movements of only one joint at a time. The action space needs to be continuous. The question is how such actions can be realized with the NC-file interface. Another possibility to improve the trajectory could be some kind of post-processing on the learned trajectory logged in the NC-file. Paths could be smoothed and actions could be combined and parallelized.
With a new action space, new algorithms would need to be tested. DDPG, PPO, A3C, SAC and others are promising algorithms that support a continuous action space. The HER algorithms could be utilized to improve the sample efficiency drastically.

Bibliography

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning: an introduction*. Adaptive computation and machine learning series, Cambridge, Massachusetts: The MIT Press, second edition ed., 2018.

[2] S. Pateria, B. Subagdja, A.-h. Tan, and C. Quek, "Hierarchical Reinforcement Learning: A Comprehensive Survey," *ACM Computing Surveys*, vol. 54, pp. 109:1–109:35, June 2021.

[3] A. BHATTACHARYYA, "Lecture Note, The Performance of Feedback Control Systems."

[4] B. für Bildung und Forschung, "KI-Steuerung – Erforschung einer automatisierten Steuerungsprogrammierung mittels Maschinellem Lernen."

[5] F. Jaensch, L. Klingel, and A. Verl, "Virtual Commissioning Simulation as OpenAI Gym - A Reinforcement Learning Environment for Control Systems :accepted," *Third International Conference on Artificial Intelligence for Industries (Ai4i)*, 2022.

[6] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-Baselines3: Reliable Reinforcement Learning Implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021.

[7] R. Bellman, "A Markovian decision process," *Journal of mathematics and mechanics*, pp. 679–684, 1957. Publisher: JSTOR.

[8] R. T. Icarte, T. Q. Klassen, R. Valenzano, and S. A. McIlraith, "Teaching Multiple Tasks to an RL Agent using LTL," p. 10, 2018.

[9] M. L. Puterman, *Markov decision processes*, vol. 2 of *Stochastic Models*. Elsevier, Jan. 1990.

[10] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, pp. 279–292, May 1992.

[11] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, 2015.

[12] R. T. Icarte, T. Q. Klassen, R. Valenzano, and S. A. McIlraith, "Reward Machines: Exploiting Reward Function Structure in Reinforcement Learning," *Journal of Artificial Intelligence Research*, vol. 73, pp. 173–208, Jan. 2022. arXiv: 2010.03950.

[13] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba, "Hindsight Experience Replay," *arXiv:1707.01495 [cs]*, Feb. 2018. arXiv: 1707.01495.

[14] M. Tsvarkaleva and L. A. Dennis, "No Free Lunch: Overcoming Reward Gaming in AI Safety Gridworlds," in *Computer Safety, Reliability, and Security. SAFECOMP 2021 Workshops* (I. Habli, M. Sujan, S. Gerasimou, E. Schoitsch, and F. Bitsch, eds.), Lecture Notes in Computer Science, (Cham), pp. 226–238, Springer International Publishing, 2021.

[15] R. T. Icarte, T. Klassen, R. Valenzano, and S. McIlraith, "Using reward machines for high-level task specification and decomposition in reinforcement learning," in *International Conference on Machine Learning*, pp. 2107–2116, PMLR, 2018.

[16] OpenAI, C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. d. O. Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang, "Dota 2 with Large Scale Deep Reinforcement Learning," *arXiv:1912.06680 [cs, stat]*, Dec. 2019. arXiv: 1912.06680.

[17] OpenAI, M. Andrychowicz, B. Baker, M. Chociej, R. Jozefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, J. Schneider, S. Sidor, J. Tobin, P. Welinder, L. Weng, and W. Zaremba, "Learning Dexterous In-Hand Manipulation," *arXiv:1808.00177 [cs, stat]*, Jan. 2019. arXiv: 1808.00177.

[18] B. Chen, S. Song, H. Lipson, and C. Vondrick, "Visual Hide and Seek," *arXiv:1910.07882 [cs]*, Oct. 2019. arXiv: 1910.07882.

[19] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play," *Science*, vol. 362, pp. 1140–1144, Dec. 2018. Publisher: American Association for the Advancement of Science.

[20] F. Jaensch, A. Csiszar, C. Scheifele, and A. Verl, "Digital Twins of Manufacturing Systems as a Base for Machine Learning," in *2018 25th International Conference on Mechatronics and Machine Vision in Practice (M2VIP)*, pp. 1–6, Nov. 2018.

[21] F. Jaensch, A. Csiszar, A. Kienzlen, and A. Verl, "Reinforcement Learning of Material Flow Control Logic Using Hardware-in-the-Loop Simulation," in *2018 First International Conference on Artificial Intelligence for Industries (AI4I)*, pp. 77–80, Sept. 2018.

[22] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," Tech. Rep. arXiv:1606.01540, arXiv, June 2016. arXiv:1606.01540 [cs] type: article.

[23] "KUKA KR 10 R1100 sixx," Feb. 2021.

[24] B. C. Stadie, S. Levine, and P. Abbeel, "Incentivizing Exploration In Reinforcement Learning With Deep Predictive Models," Nov. 2015. arXiv:1507.00814 [cs, stat].

[25] V. Dewanto and M. Gallagher, "Examining average and discounted reward optimality criteria in reinforcement learning," July 2021. Number: arXiv:2107.01348 arXiv:2107.01348 [cs, eess].

[26] J. Sola and J. Sevilla, "Importance of input data normalization for the application of neural networks to complex industrial problems," *IEEE Transactions on Nuclear Science*, vol. 44, pp. 1464–1468, June 1997. Conference Name: IEEE Transactions on Nuclear Science.

[27] F. P. Audonnet, A. Hamilton, and G. Aragon-Camarasa, "A Systematic Comparison of Simulation Software for Robotic Arm Manipulation using ROS2," Apr. 2022. arXiv:2204.06433 [cs].

[28] P. Chang and T. Padir, "Sim2Real2Sim: Bridging the Gap Between Simulation and Real-World in Flexible Object Manipulation," Feb. 2020. arXiv:2002.02538 [cs].

[29] H. Nguyen and H. La, "Review of Deep Reinforcement Learning for Robot Manipulation," in *2019 Third IEEE International Conference on Robotic Computing (IRC)*, pp. 590–595, Feb. 2019.

[30] S. Singh, R. Lewis, and A. Barto, "Where Do Rewards Come From?," Jan. 2009.

[31] S. Singh, R. L. Lewis, A. G. Barto, and J. Sorg, "Intrinsically Motivated Reinforcement Learning: An Evolutionary Perspective," *IEEE Transactions on Autonomous Mental Development*, vol. 2, pp. 70–82, June 2010. Conference Name: IEEE Transactions on Autonomous Mental Development.

[32] Q. Liu, Z. Liu, B. Xiong, W. Xu, and Y. Liu, "Deep reinforcement learning-based safe interaction for industrial human-robot collaboration using intrinsic reward function," *Advanced Engineering Informatics*, vol. 49, p. 101360, Aug. 2021.

[33] J. Xie, Z. Shao, Y. Li, Y. Guan, and J. Tan, "Deep Reinforcement Learning With Optimized Reward Functions for Robotic Trajectory Planning," *IEEE Access*, vol. 7, pp. 105669–105679, 2019. Conference Name: IEEE Access.

[34] A. A. Shahid, D. Piga, F. Braghin, and L. Roveda, "Continuous control actions learning and adaptation for robotic manipulation through reinforcement learning," *Autonomous Robots*, vol. 46, pp. 483–498, Mar. 2022.

[35] P. Abbeel and A. Y. Ng, "Inverse Reinforcement Learning," in *Encyclopedia of Machine Learning* (C. Sammut and G. I. Webb, eds.), pp. 554–558, Boston, MA: Springer US, 2010.

[36] P. Christiano, J. Leike, T. B. Brown, M. Martic, S. Legg, and D. Amodei, "Deep reinforcement learning from human preferences," July 2017. arXiv:1706.03741 [cs, stat].

Acronyms

CORBA Common Object Request Broker Architecture.

DoF Degrees of Freedom.

DQN Deep Q-Networks.

HRL Hierarchical Reinforcement Learning.

LSTM long short-term memory.

MDP Markov decision process.

NN neural network.

PLC programmable logic controller.

RL Reinforcement Learning.

ROS Robot Operating System.

SB3 Stable Baselines 3.

TCP Tool Center Point.

Glossary

dll Dynamic-link library (DLL) is Microsoft's implementation of the shared library concept in the Microsoft Windows and OS2 operating systems. .

euclidean distance In mathematics, the Euclidean distance between two points in Euclidean space is the length of a line segment between the two points. It can be calculated from the Cartesian coordinates of the points using the Pythagorean theorem, therefore occasionally being called the Pythagorean distance..

experience replay buffer the collection of experiences, gathered in the environment; used to train the neural network (value-function) in DQN.

Real-Time a Real-Time System must guarantee a maximum time until a scheduled task is completed.

reward machine reward machines are a concept in hierarchical RL, where the Reward function is represented by a state machine. Great progress has been achieved, by using the state as a kind of memory for the agent..

target network a trick employed in the double DQN algorithm to stabilize the learning..

virtual commissioning virtual commissioning is a technique, where the physical production environment is modeled in a 3D simulation, so that new hardware and new software can be tested without halting the production line.

Selbstständigkeitserklärung

Ich versichere, dass ich diese Thesis/Arbeit ohne fremde Hilfe selbständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie alle wörtlichen oder sinngemäß übernommenen Stellen in der Arbeit gekennzeichnet habe. Die Arbeit wurde noch keiner Kommission zur Prüfung vorgelegt und verletzt in keiner Weise Rechte Dritter.

....

Ort, Datum Dustin Walker