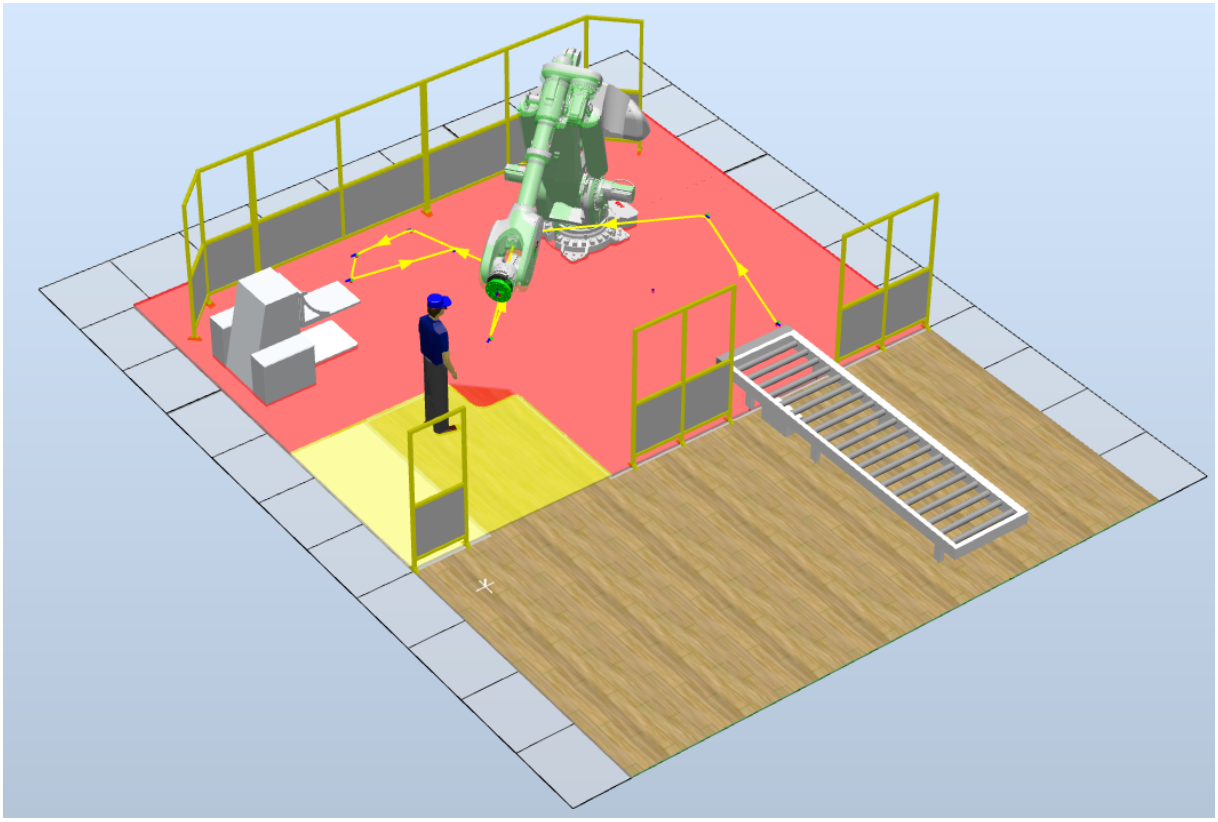


Falsification of Robot Safety Systems using Deep Reinforcement Learning

Automatic Risk Assessment for a more Sustainable Collaborative Environment with Artificial Intelligence

Master's thesis in Systems, Control and Mechatronics

HAMPUS ANDERSSON
DIVYA KARA

DEPARTMENT OF ELECTRICAL ENGINEERING

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2021
www.chalmers.se

Master's thesis 2021

Falsification of Robot Safety Systems using Deep Reinforcement Learning

Automatic Risk Assessment for a more Sustainable Collaborative Environment with Artificial Intelligence

HAMPUS ANDERSSON
DIVYA KARA

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Automation of Systems and Control
Chalmers University of Technology
Gothenburg, Sweden 2021

Falsification of Robot Safety Systems using Deep Reinforcement Learning
HAMPUS ANDERSSON
DIVYA KARA

Co-Supervisor: Constantin Cronrath, Department of Electrical Engineering
Supervisor: Kristofer Bengtsson, Department of Electrical Engineering
Examiner: Bengt Lennartsson, Department of Electrical Engineering

Master's Thesis 2021
Department of of Electrical Engineering
Automation of Systems and Control
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Visualisation of a reconstructed workstation in the simulation program RobotStudio, including an agent, robot, safety zones, AGV, conveyor and fences.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2021

Falsification of Robot Safety Systems using Deep Reinforcement Learning
Automatic Risk Assessment for a more Sustainable Collaborative Environment with Artificial Intelligence
HAMPUS ANDERSSON
DIVYA KARA
Department of Electrical Engineering
Chalmers University of Technology

Abstract

Robot stations today have large safety zones around each station where humans are not allowed to be while the robots are moving. Often these zones are fenced, but they can also be monitored to stop the motions if anyone enters. However, it is common that the total area of a new robot station area must be kept small, or maybe an operator must be able to interact with the station, hence requiring smaller safety zones. In order to reduce these safety zones, integrators need to guarantee that the robot stations design are safe for any type of human behavior by doing a risk assessment and safety evaluation. This will support this risk assessment and safety validation of robot safety zones by developing a function that tries to falsify a station using AI and Deep Reinforcement Learning (DRL) to find defects as unexpected collisions in the environment. The meaning of falsification, in this case, is to find defects in the safety of the environment. Since it is challenging and complex to construct Reinforcement Learning (RL) functions, this work was iteratively developed into several models, where each model was created with increasing complexity and with more dynamics. To begin with, Q-learning was applied to a simple discrete environment, followed by Deep Q-Network (DQN) and finally, a continuous Soft-Actor Critic (SAC) algorithm, all environments simulated in ABB RobotStudio. The algorithm was able to find collisions and dangerous situations in a specific workstation but did not manage to find the expected relation between states in order to use it more generally on other similar workstations without re-tuning the parameters. It was found that the implemented SAC suffers from instability, divergence and was very hyper-parameter sensitive, thus it was difficult to find an optimal solution. For future work, it would be interesting to further investigate the divergence and the difficulties of the deadly triad that occurs, a common problem in reinforcement learning when combining function approximation, bootstrapping and off-policy learning. This can be done by applying the newly developed SAC algorithm called Averaged-SAC which handles the overestimation problem that SAC has and provides a more stable process during training. Furthermore, a new concept that could solve the general problem and the difficulties faced in this thesis is presented and discussed.

Keywords: Artificial Intelligence, Reinforcement learning, Falsification, Q-learning, Deep Q-Network, Soft Actor-Critic, RobotStudio, Collaborative robot

Acknowledgements

Our completion of this thesis would not have been possible without the kind support and help of many individuals. We would like to extend our sincere thanks to all of them.

First and foremost, we would like to give special thanks to ABB for giving us this opportunity to write a thesis on such an interesting topic and for providing us with the necessary work equipment to carry out this thesis. We want to express a deep and sincere gratitude to our supervisor at ABB, Oskar Henriksson, for his constant support, guidance and encouragement. He was always there to demonstrate features in RobotStudio and teach us any technical details needed to help us complete our tasks.

We would like to thank our supervisor Constantin Cronrath, PhD student in the Automation research group at Chalmers, for giving us insight into reinforcement learning and all its problems that it brings. We also thank him for his patience for all the difficult issues with the algorithm that he helped us to go through. This thesis would not have been possible without our supervisor Kristofer Bengtsson, PhD at Chalmers. Thank you for guiding us through this thesis and for always giving necessary suggestions to better this study.

Thank you to Greta Braun and Johan Bengtsson at Gothenburg Technical College (GTC) for including us in the exciting project "Production for Future" and giving us the chance to spread knowledge about sustainability, diversity and technology through the project. We also thank Bengt Lennartsson for being our examiner and Edvin Alfredsson for his feedback on our report. Finally, we would like to acknowledge the great support provided by our family and friends during this journey.

Hampus Andersson & Divya Kara
Gothenburg, June 2021

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms and Terms

ANN	Artificial Neural Networks
CPS	Cyber-Physical System
DDQN	Double Deep-Q Network
DH	Denavit–Hartenberg
DQN	Deep Q-Network
DRL	Deep Reinforcement Learning
IID	Independent and identically distributed
MAE	Mean Absolute Error
MDP	Markov Decision Process
MSE	Mean Squared Error
RAPID	Code language used in RobotStudio
RL	Reinforcement Learning
RobotStudio	Virtual simulation and programming tool that allows realistic simulations of ABB robots
SAC	Soft-Actor Critic
SGD	Stochastic Gradient Decent
TCP	Tool Center Point

1
Introduction

Robot stations are today carefully configured with human interaction in mind. Any kind of hazardous scenarios between humans and robots must be taken into consideration, and adding safety zones or margins in terms of space and speed is required. The safety design phase is thus a very time-consuming part and simplified uncollaborative solutions are often used such as fencing and emergency buttons to save time. This results in a non-humane environment around the robot and it takes unnecessarily large space. ABB has developed a solution for a more collaborative environment called the ABB's SafeMove [1]. This tool creates safe areas where a robot and a human can work together safely, a warning zone that slows down the robot when a human approaches a dangerous area and a stop zone that stops the robots current task to prevent any injuries.

This Master's thesis is in collaboration with ABB and the project Production for Future at Gothenburg's Technical College (GTC). ABB is a leading global technology company with the ambitions of achieving a more productive and sustainable future. Moreover, Production for Future is a project that aims to build a mini-factory and the results from this thesis will be shared with the team, which will be considered when making solutions for more sustainable productions.

1.1 Problem Description

Since robotic arms are considered to be a safety-critical system, it is crucial to find any bugs or faults which fail to retain safety in a robot cell. In the worst case, a collision could lead to that a human is injured by the robot. Hence, the focus of this thesis will be to develop an algorithm that finds unexpected dangerous scenarios by using falsification and Reinforcement Learning (RL). Falsification is a technique that can be used to generate test cases in simulation, which in this case will be dangerous scenarios. The developed algorithm will facilitate for the system integrators when new workstations are developed in the future to make it safer and consequently be able to reduce safety margins that exist now.

1.2 Objective

The purpose of this thesis is to develop a function with RL in ABB RobotStudio, a virtual simulation and programming tool that allows realistic simulations of ABB robots, that should find unexpected dangerous scenarios that can occur in working

stations around ABB robots to establish a safer environment. This is also known as falsifying a Cyber-Physical System (CPS) and is an optimal way of testing where the weaknesses of the system are found. This is done to verify that the system meets its requirements. This shall be utilized to test ABB's SafeMove system to enable further testing of its requirements, to not collide with a human and enable a more collaborative relationship.

1.3 Limitations

This project, however, is subject to several limitations. In order to simplify the problem, the project will be limited to use a general and large uncollaborative industrial robot. The output signals from these robots become easier to analyze due to their rather heavy weight, and with their longer braking distance that it entails, more unpredictable dangerous scenarios can be found.

The agent's behavior is of great essence since it has to explore paths and ways to collide with the robot in a way that is true to a real-life scenario. Therefore, the actions of the agent will be constrained to a model which shall resemble the movement of a human in a two-dimensional space. Moreover, the agent that will be simulated in RobotStudio, will be limited to only move around the robot in the specified area limited by boundaries. The objective is for the collisions to be initiated by the robot rather than the agent walking into the robot directly. In reality, a delay is present between sent instruction and taken action, this delay is omitted but a similar delay is integrated within the created system. The simulated delay is only there to represent the behavior of a robot and does not represent the true delay of the system. Therefore, the solution may differ from a virtual and physical station.

The algorithm is ultimately intended to be used when only the safety feature, SafeMove, is active and where no other system should be able to interfere with the robot. However, the built-in SafeMove system is not applicable for our purpose, as it stops and aborts simulations which is not suitable for our testing method. Hence, a custom version of the system will be used throughout this thesis during the training phase. The customized version works in the same way as the original SafeMove but without interruption in the simulation, which creates a more efficient environment for testing.

Due to confidentiality, the station called "real station" in this thesis is a reproduction of real stations created by looking at images and virtual 3D scanning. Hence, the station does not exactly replicate a true real station.

1.4 Specification of Issue Under Investigation

This thesis aims to answer and discuss the following three main research questions:
Question 1: How can reinforcement learning be used on large industrial robots to

find deficiencies in the collaborative safety measures?

Question 2: How general is the solution, can it solve one specific problem or can it be applied to similar altered workstations?

Question 3: How can the detected dangerous movements alter the workplace and can Artificial intelligence (AI) create a trustworthy work environment?

1.5 Outline of the Thesis

This thesis consists of 11 chapters. This first chapter gives an introduction to the problem that will be carried out in this thesis and the background to it. The second chapter provides the necessary background and concepts in order to understand the methodology of the work. Chapter 3 to 6 describes the falsification of the system on 4 different models and describes how the models were produced. The final model is then applied to a real station in chapter 7. Chapter 8 is dedicated to evaluating the function's flexibility and is followed by chapter 9 where collaborative safety is discussed. In Chapter 10, the results are further discussed and future work has been suggested. In the final section, chapter 11, the most important conclusions of the work are drawn.

2
Background

Industrial robots are commonly utilized to perform tasks with high endurance, speed, and precision, such as lifting heavy objects, performing tasks with repetitive motions and work in dangerous environments. For these tasks the robots have always been constructed with the same safety approach, namely to immediately stop if paths are blocked or if someone might be in danger [2]. This requires several simplified and uncollaborative solutions that take an unnecessarily large space to create a humane environment. ABB's SafeMove system is one solution to this problem, where safe, warning and stop zones are added to the robot station for humans and robots to work together in a safe way. Testing of this system has been done to make sure that the product meets the requirement but further testing can be done in new modern ways. One way is to integrate Deep Reinforcement Learning (DRL) into the system to provoke all possible dangerous scenarios to guarantee full safety for the user, also known as falsifying the system.

2.1 Industrial Robot IRB 8700

The industrial robot IRB 8700 is ABB's largest robot, with 6 degrees of freedom. It has two versions, one called IRB 8700-800 with a payload of 800 kg and a reach of 3.5 m, and another called IRB 8700-550 with a reach of 4.3 m and payload of 550 kg. Commonly, robots that have higher payload capability have a slower speed, but compared to other robots in this size class, the IRB 8700 delivers 25% faster speed. An illustration of the robot IRB 8700 can be seen in Figure 2.1, where the axes are labeled from axis 1 (A) to axis 6 (F). [3]

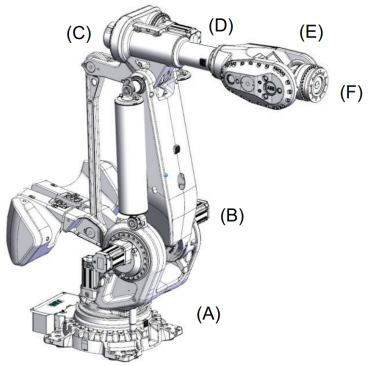


Figure 2.1: Robot IRB 8700, with axis labeled from axis 1 (A) to axis 6 (F). From [3].

2.1.1 Denavit–Hartenberg parameters

The Denavit–Hartenberg (DH) convention is often used to select frame references between consecutive links in robotics applications and mathematically describe the relative position and orientation between the two frames [4]. This enables to essentially express the position and orientation of a joint, relative to another joint. In order to place frames at each joint, there are several constraints on the relationship between the axes. First, the X_i-axis is perpendicular to both Z_{i-1} and Z_i axes. Furthermore, the origin of a joint frame i is always in the intersection of X and Z-axis and the Y-axis is placed according to the right-hand rule.

Each joint i in the robot can be described by four transformation parameters, also called the DH parameters; a_i, d_i, α_i, θ_i [4]. In some applications, the modified DH parameters are used, what differs between the classical and modified DH parameters is the placement of the links coordinate system and also the order of transformation. In the modified DH convention, which will be used in this project, the frame O_i is placed on joint axis i instead of in axis $i+1$ as in the classical , see Figure 2.2.

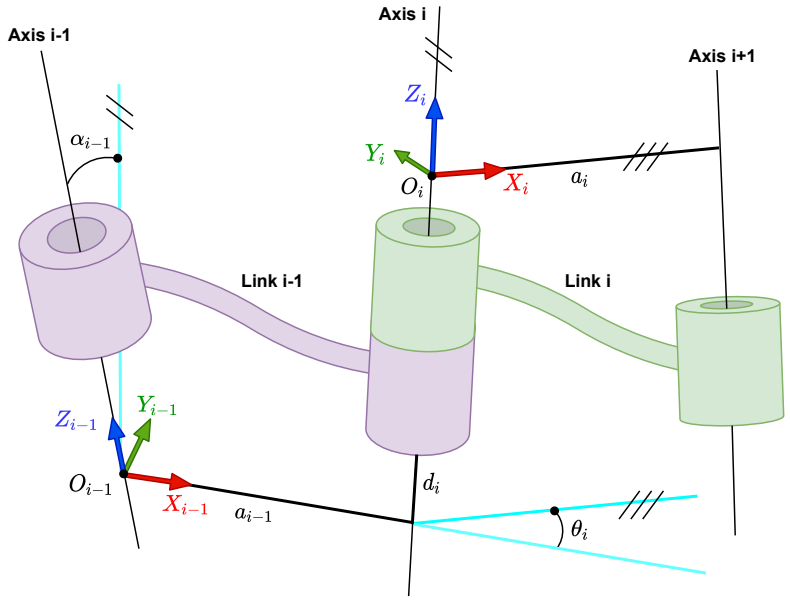


Figure 2.2: Illustration of Denavit–Hartenberg parameters between two links. From [5]. Modified with permission. GNU FDL.

Consider Figure 2.2 and the DH parameters are described as following in each joint:

- a_i : the distance between from Z_i to Z_{i+1} measured along X_i (link length)
- α_i : the angle from Z_i to Z_{i+1} measured about X_i (link twist)
- d_i : the distance from X_{i-1} to X_i measured along Z_i (link offset)
- θ_i : the angle from X_{i-1} to X_i measured about Z_i (joint angle)

Following DH parameters are then used to compute the forward kinematics.

2.1.2 Forward Kinematics

Forward kinematics is a way of calculating the robot's joint positions and orientations given only the joint angles. The start frame, or also called the base frame, is some arbitrary frame at the base of the robot. From this frame, each joint has a positional relation. To express the relation between two frames, Homogeneous transformations can be expressed as

$$p^0 = o_1^0 + R_1^0 p^1, \tag{2.1}$$

where o is the translation and $R(\theta)$ is the rotation between two points , p, in space. This describes the origin frame based on a state further away, but it is more important to gain knowledge from a latter frame based on the base frame, this can be done by multiplying each side with the inverse transformation, which gives

$$p^1 = -R_1^{0T} o_1^0 + R_1^{0T} p^0 = -R_0^1 o_1^0 + R_0^1 p^0. \tag{2.2}$$

This can then be restructured to a more compact notation to simplify calculations, the Homogeneous transformation as

$$\tilde{p} = \begin{bmatrix} p \\ 1 \end{bmatrix} \qquad A_0^1 = \begin{bmatrix} R_1^{0T} & -R_1^{0T} o_1^0 \\ 0^T & 1 \end{bmatrix} \tag{2.3}$$

and the next position is obtained by

$$\tilde{p}^1 = A_0^1 \tilde{p}^0. \tag{2.4}$$

Now any position can be calculated from the base position as follows

$$\tilde{p}^n = A_1^0 A_2^1 ... A_n^{n-1} \tilde{p}^0. \tag{2.5}$$

In robotics, the position is not available, but the joint angles (J_θ) are, which the rotational matrices (R) are dependent upon. With the joint angles, the Homogeneous transformation can be rewritten as

$$T_e^b = \begin{bmatrix} R_e^b(J_\theta) & p_e^b(J_\theta) \\ 0^T & 1 \end{bmatrix}, \tag{2.6}$$

where $p_e^b(J_\theta)$ represents the position vector of the frames origin with respect to the base frame, illustrated in Figure 2.3.

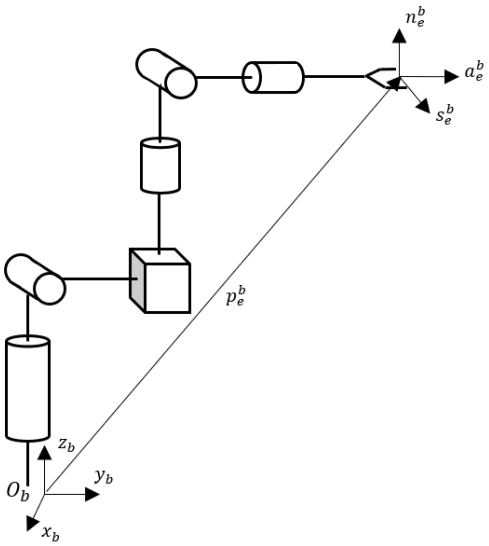


Figure 2.3: Homogeneous transformation illustrated, n_e^b, a_e^b, s_e^b represents the unit vector of the rotational matrix and p_e^b the position vector between current frame and base frame.

The position and an orientation can then be derived from the joint angles as

$$T_n^0(J_\theta) = A_1^0(J_{\theta_1})A_2^1(J_{\theta_2})...A_n^{n-1}(J_{\theta_n}), \tag{2.7}$$

and simplified as

$$T_e^b(J_\theta) = T_0^b T_n^0(J_\theta) T_e^n. \tag{2.8}$$

All angels and lengths that the Homogeneous transformations position vector and rotational matrices are dependent upon are acquired from the DH-parameters. The transformation matrix can be described by the following order of operations

$$T_n^{n-1} = P_{z_{n-1}}(d_n)R_{z_{n-1}}(\theta_n)P_{x_n}(r_n)R_{x_n}(\alpha_n), \tag{2.9}$$

where $P_{x_{n-1}}$ and $R_{x_{n-1}}$ is the translation respective rotation along the x-axis and P_{z_n} and R_{z_n} is the translation respective rotation along the z-axis. The mathematical representation of rotation and translation are stated as in [6].

The forward kinematics for the modified DH-parameters, described in section 2.1.1, are calculated similarly. Because of the difference in placement of the coordinate system the Homogeneous transformation matrix is instead calculated as

$$T_n^{n-1} = R_{x_{n-1}}(\alpha_{n-1})P_{x_{n-1}}(a_{n-1})R_{z_n}(\theta_n)P_{z_n}(d_n), \tag{2.10}$$

where $P_{x_{n-1}}$ and $R_{x_{n-1}}$ is the translation respective rotation along the x-axis and P_{z_n} and R_{z_n} is the translation respective rotation along the z-axis. The mathematical representation of rotation and translation are stated as in [4].

2.2 Falsification of Cyber-Physical System

Cyber-physical systems (CPSs) are systems that combine embedded computing and sensor networks to communicate with the physical environment. Examples of CPSs are smart grids, robotics systems, and medical monitoring. In this setup, the SafeMove which communicates with e.g. the robot and components in the cell are defined as the CPS. Usually, the CPS is developed with a model-based approach and the testing and verification of CPS can be challenging due to the complexity of the software and physical system. Since the the CPS is created manually, it does not have a mathematical model which can be analyzed, the in- and outputs from simulations will be analyzed to generate test cases and data for the agent to enable intelligent testing. These test cases can be used to verify any new solutions, then the falsification process can be redone to find new or other faults. Instead of controlling the robot which can be quite complex, the idea that an arbitrary intelligent agent can expose the system to several situations until a dangerous scenario is achieved will be investigated. Because bringing the dangerous scenario to the subject is easier than taking the subject to a dangerous scenario in the simulation.

Today CPSs are tested with tools and methods such as Breach [7], ”falsify” [8] and FALSTAR [9]. These tools utilize temporal logic to falsify the CPS and in this case, ”falsify” will be the approach to solve the problem. By optimizing the falsification on CPSs, all new scenarios that can occur in a robot cell will be generated.

In comparison to these tools and methods, RL is seen as a candidate to replace temporal logic for falsification [10], [11]. With RL the environment would have to be learned and the goal would be to falsify the system, to make sure that the CPS is exposed to unknown scenarios that it can not control. This is an approach that has been done before in other applications and has been seen to be successful.

Yamagata et al. [11] adopt two state-of-the-art DRL techniques, Asynchronous Advanced Actor-Critic (A3C) and Double Deep-Q Network (DDQN) to solve the falsification problem. The paper covers two parts, first how a problem of falsifying CPS models is transformed into a RL problem. Secondly, the authors evaluate their algorithms to show that the DLR technology is able to reduce the number of simulation runs required to find a falsifying input for CPS models.

Furthermore, Qin et al. [10], used and compared Q-learning and a neural network to train an algorithm to find faults in the system. The authors show in three different autonomous driving cases that the developed algorithm guarantees to find existing faults in the system.

2.3 Reinforcement Learning

RL is one of the three main areas in machine learning where supervised and unsupervised are the other two areas. Typically RL consists of an agent, states, actions, and

rewards where the algorithm learns from its environment through actions and the feedback it gets. The *agent* is the model that should be trained to make decisions. Everything around the agent and what the agent can interact with is considered to be the *environment*. Furthermore, the environment consists of a certain amount of *states* which represent the current state at any point of time, e.g. the current position or speed of the agent or other objects. Based on these states the agent will take one of the predefined *actions* and the environment provides feedback to the agent in terms of *rewards* which indicates how good or bad the taken action was. Figure 2.4 shows a interaction loop between the agent and environment.

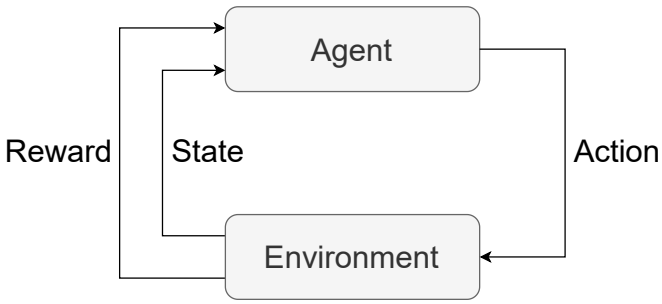


Figure 2.4: Agent-environment interaction loop.

The purpose is to guide the agent towards the goal in the environment, which is described by the states, by maximizing the total reward or expected return. The agent will optimize each action at every state to gain as much reward as possible in order to reach the goal. In this way, the algorithm is pushed towards the goal at each instance. In short, RL is the way of learning the best actions based on reward and punishment, a way of learning by doing, and has many similarities to the humans way of learning [12].

With RL comes challenges, such as designing the reward function, the trade-off between exploration and exploitation and tuning the hyper-parameters. As mentioned RL is based on trial-and-error where the agent learns from exploration and exploitation. Exploration is when the agent decides to gather new information that might lead to a better decision in the future and in real life this is almost impossible to do and is therefore one of the reasons why RL is used in a simulated environment. Exploitation is taking the optimal decisions which return the highest possible outcome from its current knowledge. If the agent exploits too much, it might miss out on other optimal solutions that yet have been discovered, but exploring too much is not optimal either as training time will increase and it might collect unnecessary data. This trade-off is called the explore-exploit dilemma. Compared to other machine learning methods, such as supervised learning which has a fixed data set, RL collects data from its environment by exploring. Since exploring is random, the results can vary from each run which may cause the result for each run to vary. The tuning of hyper-parameters is also a crucial and important task. Rijn and Hutter show in [13] how highly dependent the performance of machine learning algorithms can be on hyper-parameters.

2.3.1 Model-Based and Model-Free Learning

RL is based upon model-based and model-free learning. By having a model the agent has the possibility to predict and think ahead to foresee several paths and the outcome of these. This is a powerful tool that simplifies the policy optimization but the model of the environment is often unknown, hence model-free RL is introduced. Model-free RL is when the agent has no knowledge of the surrounding environment and thus has no model to be constrained to. The agent solely depends on the state values at each instance and will take decisions based on these, it has no idea of what the states represent. The importance of each state will be learned during training and which state-action pair are more important to maximize the reward. In this thesis, model-free learning will be carried out since it does not exist a model of the robot movements and station layout as they change during the simulation and between environments. This makes the agent solely dependent on the states at each time instance to make an optimal decision. What is of great essence within RL is that it must contain the Markov property, thus each state does not depend on previous states and all information needed can be acquired at that state.

2.3.2 Markov Decision Process

Markov Decision Process (MDP) is an mathematical framework to describe an environment in RL and can be represented as a tuple $\mathcal{M}$ as follows

$$\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle. \tag{2.11}$$

Here $\mathcal{S}$ is a set of states, $\mathcal{A}$ a set of actions, $\mathcal{P}$ is the state-transition probability, $\mathcal{R}$ represent the rewards and γ is the discount factor. The objective of the MDP $\mathcal{M}$ is to find a policy π which result in an optimal long term reward. The policy function, defined as

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s], \tag{2.12}$$

describes the behaviour of the agent and more specifically the probability of taking an action A_t from a given state S_t. The policy π in MDPs only depends on the current state and not previous states, this is also called the Markov property [12]. The reward can be varying depending on the objective and can be tuned in many different ways, this is a process of its own.

2.3.3 Reward Function Design

In RL, the agent's decisions are optimized for the cumulative reward, thus the reward determines the agent's preferred action. Hence, the reward function is of great interest since it shapes the goal of the agent. The design of the reward function can be tricky or straightforward depending on the case, e.g. to balance a pole on a cart, a reward can be given for each second the pole is balanced and a penalty when it falls. But more complex environments create more abstract and challenging reward functions. Usually, a reward is not dependent solely on the last action but rather a series of actions which makes it harder to design the reward function. An example is an experiment of the classic game coast runner, where the goal is to win a boat

race [14]. In this example, a large reward is given when winning, but on the way to the goal the player can collect power-ups, which also gives a reward but not as large as winning the race. The main idea is that in the final model the boat will collect as many power-ups as possible and finish the race first. This was not the case, the agent figured out that if it finishes the race it will collect the final reward and the game would be over. Instead, the agent remained on the track, only collecting an infinite amount of power-ups since this would give a larger cumulative reward.

As the example shows, the design of the reward function is of great essence to control the agent in the correct way and can be very challenging to design. The reward function can vary a lot from task to task, but often the idea of RL is to solve a problem that the human can not. Thus, a simple goal is often preferable which sets the goal of the agent loud and clear. In this way, the algorithm can evaluate many methods to fulfill the goal in ways the designer could not. If a too specified reward function is given, the agent will just behave exactly the way the designer wanted, this might be the goal in some cases but often the power of RL is to find new ways and paths that would not be possible otherwise [12]. Furthermore rewards are used to value states, to determine how much the agent can gain in the future from the current state and the bellman equation is one way of estimating this value.

2.3.4 Bellman Equation

To evaluate the value of a state, $V(s)$, one has to foresee the expected sum of rewards. The sum of the cumulative possible rewards for each possible action must be considered to take an appropriate action and the Bellman equation can be used to solve this. The relations between future rewards is of great essence. Each action taken both in a MDP and a RL algorithm affects future rewards, and a decision to optimize this should be taken. A discount factor, γ, is used to implicate the importance of close by or future rewards. The expected reward G_t can be denoted as

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \dots . \tag{2.13}$$

From the agent's perspective, only the immediate reward is available when an action has been taken. But as the agent needs to optimize on future rewards because of the relations the reward function cannot exclude future rewards. With the relation in equation 2.13 the expected return can be simplified as

$$\begin{aligned} G_t &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} \dots) \\ &= R_{t+1} + G_{t+1}. \end{aligned} \tag{2.14}$$

The bellman equation utilizes this relationship to express a connection between the state-value of a state and it's successor states for any arbitrary MDP, as the state-

value is always accessible. This gives

$$\begin{aligned} V(s) &= \mathbb{E}[G_t|S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \sum_a \pi(s,a) \sum_{s'} P_{ss'}(R_{ss'} + \gamma \mathbb{E}[G_{t+1}|S_{t+1} = s']) \\ &= \sum_a \pi(s,a) \sum_{s'} P_{ss'}(R_{ss'} + \gamma V(s')), \end{aligned} \tag{2.15}$$

where $P_{ss'} \in \mathcal{P}$, $s \in \mathcal{S}$, $a \in \mathcal{A}$ and $R_{ss'}$ represents the immediate reward when transitioning from s to s'. The bellman equation calculates the state value and represents the estimated possible reward from each state, by knowing each possible reward the agent can choose the optimal path by choosing the action that maximizes the Value-function [12].

2.3.5 Q-Learning

Q-learning is based upon maximizing the so-called action-value function that is performed off-policy. Off-policy means that it can be updated no matter when the data was collected and how the agent chose to explore the environment at that time instance. The action-value function is a function that maximizes the decision at the current state based on the reward now and where the agent shall end up. Therefore it is not only maximizing at this point but also for future decisions and doing this iteratively creates an optimal action sequence to the reward. What is of great essence is the importance of not settling with the first action sequence to the goal. RL uses exploration as a way to force the algorithm to test new paths all the time which generates new and more reward action sequences to the goal. In the end, each combination of states has an optimal action that would optimize the expected reward. These combinations of states and optimal actions are saved and can be referred to as Q-values. An environment with a small number of states and actions can save these values in a table, this is called tabular Q-learning, see its pseudo-code in Algorithm 1 [12].

Algorithm 1 Tabular Q-learning (Off-policy)

stepsize $\alpha \in (0, 1]$, $\epsilon > 0$
initialize $Q(s, a)$, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, size $\mathcal{S}$ x $\mathcal{A}(s)$
repeat
  Initialize s
  repeat
    Choose a from $Q(s, \cdot)$ using ϵ-greedy
    Take action a, observe r, s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_a Q(s',a) - Q(s,a) \right)$
    $s \leftarrow s'$
  until s is terminal
until no more episodes

The size of the table is fixed to the amount of states times actions available. The Q-value is updated at each iteration just on the specific states, creating a tabular knowledge base of the best actions at each spot. For larger environments with several states and actions, the amount of data becomes too big and therefore DRL is introduced. The table of Q-values is instead approximated by an artificial neural network that is trained as the algorithm explores the world [12].

Previously a work where a robot was taught to adapt its motion with RL when collaborating with humans has been done [15]. In the mentioned work, it was concluded that the tabular Q-learning faster found the global optimum compared to linear- and nonlinear functions. Furthermore, different parallelization methods were implemented and compared in order to speed up the training.

2.4 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) is a combination of Reinforcement Learning and Deep Learning, where Deep Learning (DL) is not considered to be a separate subset of Machine Learning as RL but rather a collection of methods and techniques using Neural Networks to solve ML problems. In comparison to RL, which dynamically learns by trial and error, DRL learns from existing knowledge and applies it to new datasets. DRL is one of the popular types of ML for the reason that it can solve complex decision-making tasks that are real-world problems with human-like intelligence. An example of this is the computer program AlphaGo, developed by DeepMinds, which in 2015 defeated a professional human player in the board game Go [16]. AlphaGo used deep neural networks and tree search to evaluate board positions and actions. It could in that way evaluate the state of the game by predicting possible outcomes, selecting the optimal action for it to win. DRL showed that it does not only solve the task of winning but it showed a whole new way of winning. In the game of Go, the players normally played to secure a win as fast as possible with large margins. AlphaGo instead planned out the game so it could foresee what would happen and that it always won, often with just a few points. The task was to win not win with large margins thus it created a whole new aspect of the game as it played like no one has done before. This section will give an overview of how DRL works and several DRL algorithms will be explained as well.

2.4.1 Artificial Neural Network

The word ”deep” in Deep Reinforcement Learning refers to applying neural networks to estimate the value of states. Artificial Neural Networks (ANN) are several combined nodes in layers, where each node contains a weight and bias. These nodes which represent a linear ”line” can together form nonlinear relations to categorize data. ANNs are incredibly powerful as they can, with the right tuning, categorize data that the human sometimes cannot, and faster.

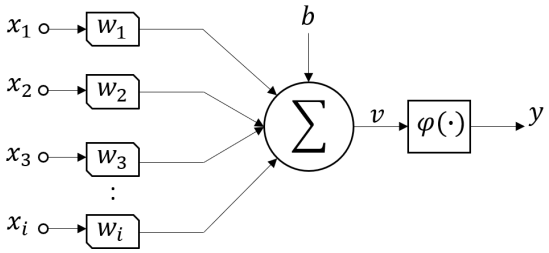


Figure 2.5: Structure of a node from in- to output.

As seen in Figure 2.5, a node receives data from several sources x_i and applies weights w_i to each source, a bias b is also added. The weight can be described as the slope of the line and the bias as the offset in the plane. These are then summed, known as v, and an activation function $\varphi(\cdot)$ is applied to the value gained from the linear transformation which gives the output y. The activation function is used to introduce non-linearity to the output. The node is described mathematically as

$$\begin{aligned} v &= \sum wx + b \\ y &= \varphi(v), \end{aligned} \tag{2.16}$$

where φ represents any arbitrary activation function [17]. A layer in an ANN is defined as one or several nodes that are applied on the same data, each node with its individual weights. By connecting several layers a network can be formed. It takes a certain amount of data as input to the input layer, any number of outputs through the output layer, and each layer between these are known as hidden layers, as seen in Figure 2.6.

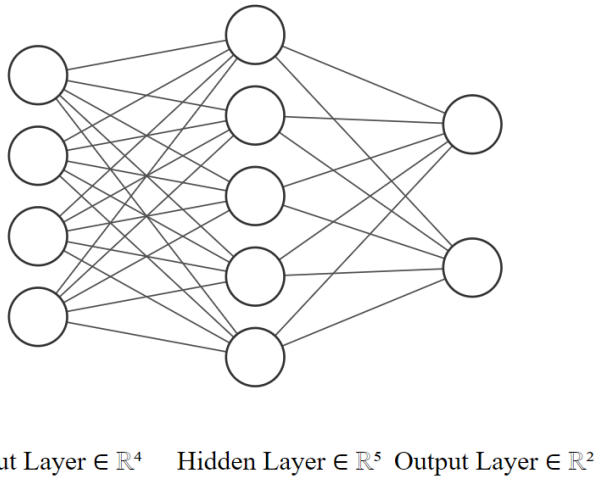


Figure 2.6: Diagram of a Artificial Neural Network with one hidden layer.

The weights of the network can categorize or predict nonlinear data by forming different nonlinear relations. The number of weights in a network is too many to

manually tune it, thus the weights need to be automatically tuned to represent the wanted behavior. Training is used for this, it is a way to use data and optimize the network's weights to adapt and behave as requested. There are two main parts when training a network, a loss function and an optimizer.

2.4.1.1 Loss Function

The loss function is a method of evaluating how well an algorithm models its dataset. A higher loss indicates that the predictions are off, meanwhile a lower loss implies that the model performs better and thus the objective is to minimize the loss function. There are two types of loss functions: classification and regression loss. For this problem regression loss will be used because classification loss is not applicable to this problem. During the thesis the loss function will be different depending on the problem and algorithm. The loss is used in the optimizer to train the network to get better, see the following section.

2.4.1.2 Optimizer

During training, the network needs to be trained to be able to map the input to output. To achieve the desired output, the weights of the network need to be updated and the neurons shall be updated in the direction which reduces the loss function. This is where the optimizer comes in and determines how much and when the parameters should change based on the loss to provide the most accurate result possible. The minimum of the loss function is found by iteratively moving in the direction of the steepest descent. The size of the move is called the learning rate, often referred to as α. There is a trade-off when choosing the size of the learning rate. With a large learning rate, there is a risk of overshooting and missing the minimum. On the other hand, a small learning rate will certainly move towards the minimum but of the cost of time-consuming computations.

There exists several optimization algorithms such as RMSprop [18], Adam [19], Stochastic Gradient Decent (SGD) [20] and Adaptive Gradient (AdaGrad) [21]. A popular optimization algorithm in deep learning is Adam and is said to be a replacement of the SGD for deep learning. The algorithm combines the best properties from the AdaGrad and RMSprop.

2.4.2 Deep Q-Learning

In 2015, a paper called *Human-level control through deep reinforcement learning* was published by Mnih et al. [22] and this was the first time RL and deep learning was successfully combined. Deep Q-Network (DQN) is an algorithm that combines Q-learning and deep neural networks by replacing the Q-table with a neural network. In the paper, a deep Q-network agent was tested on Atari 2600 games and was able to perform on a level of human professional game testers on 49 games. Out of the 49 games the agent achieved more than 75% of the human score in 29 games. The goal was to learn to do the best in environments rather than learning how to play the games. The agent was only given the pixel frames of the games as input since the

authors claim that this is the same input humans and animals have when making decisions in their environments.

Neural networks are usually considered to be very unstable since small changes in weights can make significant changes in the policy of the agent. Considering this the authors presented two key ideas to make the network more stable: *experience replay* and *target network*. The learning now uses two networks, one main network and one target network.

The target network is used as a fixed target, a reference for the main network. This gives a more stable learning and with a fixed target, the learning becomes more supervised. But to not prevent the model from learning the target network is updated to match the main network every n:th step. The pseudo-code for the Deep Q-learning is presented in Algorithm 2 and as seen this is the same pseudo-code as for Q-learning in Algorithm 1 but with an approximated Q-value.

Algorithm 2 Deep Q-learning

Stepsize $\alpha \in (0, 1]$, $\epsilon > 0$
Initialize $Q(s, a), \hat{Q}(s, a)$, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
repeat
  Initialize s
  repeat
    Choose a from $Q(s, \cdot)$ using ϵ-greedy
    Take action a, observe r, s'
    $Q(s, a) \leftarrow Q(s, a) + \alpha \left(r + \gamma \max_a Q(s', a) - \max_b \hat{Q}(s', b) \right)$
    $s \leftarrow s'$
    if step % n = 0 **then**
      $\hat{Q}(\cdot) \leftarrow Q(\cdot)$
    end if
  until target found
until Q converges

2.4.2.1 Experience Replay

The purpose of experience replay is to store a fixed number of past experiences and in [23] it has been shown to improve stability and sample efficiency of training. An experience is defined as

$$e = (s, a, r, s'), \tag{2.17}$$

where s is the state, a is the action, r is the reward and s' is the next state. The algorithm of experience replay can be seen in Algorithm 3. Experiences e are collected and stored in the replay buffer M of size N when navigating through the environment with a ϵ-greedy policy for instance and a minibatch B is sampled from the replay buffer. The replay buffer is typically implemented as first-in, first-out (FIFO) to keep the most recent experiences. The advantage of experience replay is that it allows decorrelating the data so that samples do not have to be used in the

same order that they were stored and can also be used multiple times since the data is not immediately thrown after collection. This gives experiences from different parts of the environment to then update the policy. Thus a gradient step is taken in each network based on all data taken from the replay buffer, it is learning by experience.

Algorithm 3 Experience replay

```
Initiate Replay buffer M of size N
while Training do
  Choose action a based on state s
  Take action a, observe reward r, and next state s'
  Save (s, a, r, s') in memory M
  if size(M) > batch size then
    Sample random minibatch B from M
    for (s_i, a_i, r_i, s'_i) in B do
      Calculate loss J_x
    end for
    for each network x do
      x ← x − λ∇̂ J_x
    end for
  end if
end while
```

2.4.2.2 Target Network

In the Bellman equation the $Q(s, a)$ is given by using the $Q(s', a)$ and since it is only one step between them they can look very similar and be hard for the Neural network to distinguish them. When updating the Q-value in this way the $Q(s', a)$ can indirectly be altered and states nearby as well which leads to instability in learning. For this reason, the weights from the main network are copied into the target network every n-step and are then used as $Q(s', a)$ when updating the $Q(s, a)$ value with the Bellman equation. This will make the learning more stable.

2.4.3 Soft-Actor Critic with Autotuned Temperature

Existing continuous RL algorithms are often on-policy methods, such as Asynchronous Actor-Critic Agents (A3C). On-policy methods are dependent of new samples after each policy update and can not learn from a replay buffer as off-policy methods, thus they are very sample inefficient. But there are some off-policy methods, such as the Q-learning algorithm Deep Deterministic Policy Gradient (DDPG), which is more sample efficient by utilizing experience replay and learns from past samples. The downside with algorithms as DDPG is however that they are very hyper-parameter sensitive and requires a lot of tuning to make them converge. Therefore, Soft-Actor Critic (SAC) was developed by Google in collaboration with the University of California, Berkley, which provides a more sample efficient and robust algorithm compared to other traditional RL algorithms [24]. Experimental

result of SAC has been shown to outperform prior on-policy and off-policy methods in terms of learning speed and robustness.

This off-policy and model-free RL algorithm SAC will not only maximize the expected sum of reward r as standard RL algorithms but also the entropy $\mathcal{H}$ of the policy, weighted by a non-negative weight α, called temperature entropy. The RL problem is therefore stated as

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \sum_{t=0}^{T} \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \rho_\pi} \left[r\left(\mathbf{s}_t, \mathbf{a}_t\right) + \alpha \mathcal{H}\left(\pi\left(\cdot \mid \mathbf{s}_t\right)\right) \right], \tag{2.18}$$

where s_t is the state, a_t is the action and ρ_π the optimal policy. The temperature entropy α controls the trade-off between the reward and entropy. Maximizing the entropy of the policy will result in more exploration in the network and the tuning of hyper-parameters is minimized. In the original paper [24], the temperature α was set to be fixed and had to be manually tuned by the user for each task, which made the algorithm very sensitive. Therefore a new version of SAC was released later in 2019 where the temperature α was automatically adjusted for each task [25].

Compared to the original implementation of SAC which consist of 3 networks; a state value function, a soft Q-function and a policy function [24], the new version with autotuned α includes 5 networks [25]. The five networks are: two soft Q-functions $Q_\theta(s_t, a_t)$, two soft Q-target functions $Q_{\bar{\theta}}(s_t, a_t)$ and a policy function $\pi_\phi(a_t|s_t)$, which are parameterized by θ_1, θ_2, $\bar{\theta}_2$, $\bar{\theta}_2$ and ϕ respectively. According to [25], two soft Q-functions are used because it significantly speed up the training of more complex tasks. The policy is modeled as Gaussian with mean and variance from neural networks and the value functions are modeled as neural networks.
The soft Q-functions are trained to minimize the soft Bellman residual error

$$J_Q(\theta) = \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim \mathcal{D}} \left[\frac{1}{2} \left(Q_\theta\left(\mathbf{s}_t, \mathbf{a}_t\right) - \hat{Q}\left(\mathbf{s}_t, \mathbf{a}_t\right) \right)^2 \right], \tag{2.19}$$

where

$$\hat{Q}\left(\mathbf{s}_t, \mathbf{a}_t\right) = r\left(\mathbf{s}_t, \mathbf{a}_t\right) + \gamma \mathbb{E}_{\mathbf{s}_{t+1} \sim p}\left[V_{\bar{\theta}}\left(\mathbf{s}_{t+1}\right) \right]. \tag{2.20}$$

Equation 2.19 implies that for all state-action pairs (s_t, a_t) in the replay buffer $\mathcal{D}$, the squared error between the Q-function prediction Q_θ and $\hat{Q}$, should be minimized. $\hat{Q}$ can be seen in Equation 2.20 to be the sum of the immediate reward r and discount factor γ multiplied with the expected value of the next state from the target value function $V_{\bar{\theta}}$. The target value function is parameterized from the soft Q-function as

$$V_{\bar{\theta}} = \mathbb{E}_{\mathbf{a}_t \sim \pi_\phi}\left[Q_\theta\left(\mathbf{s}_t, \mathbf{a}_t\right) - \alpha \log \pi\left(\mathbf{s}_t, \mathbf{a}_t\right) \right]. \tag{2.21}$$

The target value is used to stabilize the training. The Q-function works as an estimator for the value of the current state, by knowing the value of the state an appropriate action can be taken and it simplifies the policy learning as it has a reference to see the value of its actions.

The parameters of the Q-functions is updated by the approximation of the derivative as follows

$$\begin{aligned}\hat{\nabla}_\theta J_Q(\theta) =& \nabla_\theta Q_\theta\left(\mathbf{a}_t, \mathbf{s}_t\right)\left(Q_\theta\left(\mathbf{s}_t, \mathbf{a}_t\right) - \right. \\ & \left.\left(r\left(\mathbf{s}_t, \mathbf{a}_t\right) + \gamma\left(Q_{\bar{\theta}}\left(\mathbf{s}_{t+1}, \mathbf{a}_{t+1}\right) - \alpha \log \pi_\phi\left(\mathbf{s}_{t+1}, \mathbf{a}_{t+1}\right)\right)\right)\right).\end{aligned} \tag{2.22}$$

The policy network is trained by minimizing the expected *Kullback-Leibler divergence* [26] which measures the difference of probability distributions. The Kullback-Leibler divergence could in this case be simplified as

$$J_\pi(\phi) = \mathbb{E}_{\mathbf{s}_t \sim \mathcal{D}}\left[\mathbb{E}_{\mathbf{a}_t \sim \pi_\phi}\left[\alpha \log \pi_\phi\left(\mathbf{a}_t | \mathbf{s}_t\right) - Q_\theta\left(\mathbf{s}_t, \mathbf{a}_t\right)\right]\right]. \tag{2.23}$$

To simplify $J_\pi(\phi)$ in equation 2.23 further, the reparameterization trick presented in [24] is used. The trick is very effective on neural networks such as Q_θ and the trick is to reparameterize the policy as

$$\mathbf{a} = f_\phi(\epsilon_t; \mathbf{s_t}), \tag{2.24}$$

where $f_\phi(\cdot)$ is a neural network transformation, ϵ_t is a noise vector from a Gaussian distribution where a mean and covariance are computed. The minimization equation 2.23 can now be rewritten as

$$J_\pi(\phi) = \mathbb{E}_{\mathbf{s}_t \sim \mathcal{D}, \epsilon_t \sim \mathcal{N}}\left[\alpha \log \pi_\phi\left(f_\phi\left(\epsilon_t; \mathbf{s}_t\right) \mid \mathbf{s}_t\right) - Q_0\left(\mathbf{s}_t, f_\phi\left(\epsilon_t; \mathbf{s}_t\right)\right)\right]. \tag{2.25}$$

Since π_ϕ is defined implicitly of $f_\phi(\epsilon_t; \mathbf{s_t})$, the gradient of equation 2.25 could be approximated as

$$\hat{\nabla}_\phi J_\pi(\phi) = \nabla_\phi \alpha \log \pi_\phi\left(\mathbf{a}_t \mid \mathbf{s}_t\right) + \left(\nabla_{\mathbf{a}_t} \alpha \log \pi_\phi\left(\mathbf{a}_t \mid \mathbf{s}_t\right) - \nabla_{\mathbf{a}_t} Q\left(\mathbf{s}_t, \mathbf{a}_t\right)\right) \nabla_\phi f_\phi\left(\epsilon_t; \mathbf{s}_t\right), \tag{2.26}$$

where $\mathbf{a}_t$ is evaluated from $f_\phi(\epsilon_t; \mathbf{s_t})$. By having a varying temperature the exploration could increase in unknown regions and exploit more in others. The automated tuning is done by minimizing

$$J\left(\alpha\right) = \mathbb{E}_{\mathbf{a} \sim \pi_t}\left[-\alpha_t \log \pi_t\left(\mathbf{a}_t | \mathbf{s}_t\right) - \alpha \bar{\mathcal{H}}\right], \tag{2.27}$$

where $\bar{\mathcal{H}}$ represents the target entropy and is set to $-dim(\mathcal{A})$ as recommend in the paper [25]. The benefits of an automatically tuned alpha make it possible to change alpha during run time, increasing exploration at unknown places and exploiting the known places. The pseudo-code of the SAC algorithm with auto-tuned temperature is shown in Algorithm 4.

Algorithm 4 Soft Actor-Critic with Autotuned Temperature

Initialize parameter vectors θ_1, θ_2, ϕ
$\bar{\theta_1} \leftarrow \theta_1, \bar{\theta_2} \leftarrow \theta_2$
$\mathcal{D}$
for each iteration i **do**
  for each environment step **do**
    $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t \mid \mathbf{s}_t)$
    $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1} \mid \mathbf{s}_t, \mathbf{a}_t)$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$
  end for
  for each gradient step **do**
    $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$
    $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
    $\alpha \leftarrow \alpha - \lambda \hat{\nabla}_\alpha J(\alpha)$
    $\bar{\theta_i} \leftarrow \tau\theta_i + (1-\tau)\bar{\theta_i}$ for $i \in \{1, 2\}$
  end for
end for

As the action is taken based on an unbounded gaussian based on a mean, μ, and a standard deviation, σ, approximated from the policy network the possible actions are infinite. To solve these action bounds are enforced on the distribution, the samples are passed through a squashing function *tanh()*. Thus $a = tanh(u)$ where u is the samples from the distribution, $a \in (-1, 1)$ can then be scaled to fit the area of interest. The log-likelihood can be calculated as

$$\log \pi(\mathbf{a}|s) = \log \mu(\mathbf{a}|s) - \sum \log(1 - \tanh^2(u_i)). \tag{2.28}$$

The sum in equation 2.28 has been seen to be numerically unstable and therefore it can be rewritten for stability. From the latest version of Tensorflow [27] the sum can be rewritten as

$$\begin{aligned}
\log(1 - \tanh^2(u_i)) &= \log \operatorname{sech}^2(u_i) && (2.29)\\
&= 2(\log \operatorname{sech}(u_i)) && (2.30)\\
&= 2\log(2e^{-u_i}/(e^{-2u_i} + 1)) && (2.31)\\
&= 2(\log 2 - u_i - \log(e^{-2u_i} + 1)) && (2.32)\\
&= 2(\log 2 - u_i - \operatorname{softplus}(-2u_i)). && (2.33)
\end{aligned}$$

Furthermore, the automated tuning of the temperature parameter α is not always profitable, it can create unsteady training but it encourages exploration more. Thus it can be more profitable to have a fixed α for more robust training [25].

3

Discrete Environment with Q-learning - Model 1

As explained in chapter 2.3, it can be challenging and complex to construct RL solutions. It requires a lot of tweaking and testing until such an algorithm works and therefore this solution will be divided into four models to more easily catch problems and solve them. The first model, this model, will be fairly simple and will be the base for the following models, model 2, 3 and 4 presented later in the thesis. The next models will be built up by adding more dynamics of the problem until an environment that represents a real environment is achieved. Which is a continuous environment where the agent can roam around like a real human in various workstations and where the layout and tasks differ. The fourth model will be close to the final algorithm with some fine-tuning. The final results from each model are also shown as videos in [28].

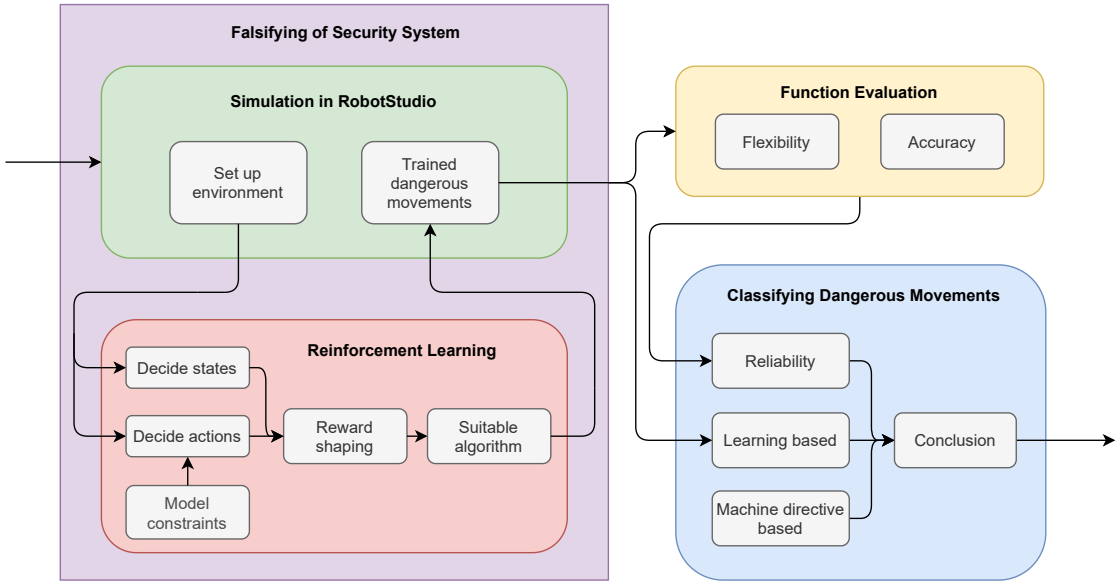


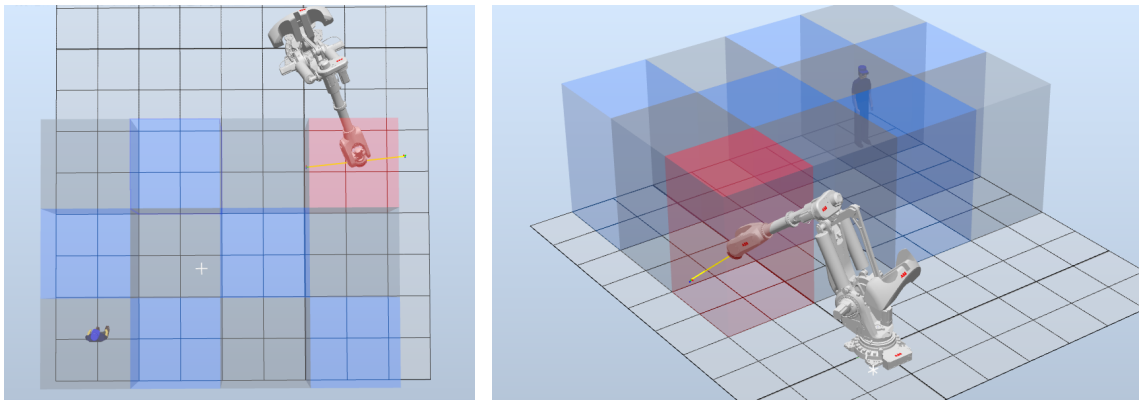
Figure 3.1: Block diagram of the overall method structure where the purple block, *Falsifying of Security System*, will be carried out for each model. The final model will then go through *Function Evaluation* and *Classifying Dangerous Movements with AI*.

Each model can be divided into three main parts; *Falsifying of security system*, *Function Evaluation* and *Classifying Dangerous Movements*, see Figure 3.1. As seen, the

Falsifying of security system chapter consist of the two parts *Simulation in RobotStudio* and *Reinforcement Learning*. The first sub-block Simulation in RobotStudio, is where each model's environment of the workstation is set up. It will also be visualized and simulated in RobotStudio. RobotStudio offers data from the robot and real-time interaction with the environment which can be utilized in several ways. What is of great essence is to find data that can be used to define a dangerous movement and make sure that this is extractable from the simulator. The environment in the simulator can then be used to find weaknesses and dangers.

The RL algorithm is designed by choosing, states, actions, shaping reward function and finding a suitable algorithm in the second sub-block. The algorithm is then applied to the simulated workstation for training to find collisions. The final trained model, model 4, is then evaluated to get a sense of how flexible the algorithm is. Lastly, dangerous movements are evaluated and conclusions are drawn.

The first model, which is the base model, is a simple environment to verify that the concept can be carried out. For the simulation, ABB's largest robot IRB 8700 was chosen since it is more dangerous and between the two available versions mentioned in chapter 2.1, IRB 8700-550 was chosen over IRB 8700-800 because in simulation the robots movement and reach is more important than the payload capacity. The environment is discrete and divided into 12 cells as a 3×4 grid world, see the simulation environment in Figure 3.2.



(a) Top view **(b)** Oblique view

Figure 3.2: The discrete environment of model 1 in RobotStudio shown from two angles. The environment consist of a robot, agent and the cells the agent can move between.

Each cell corresponds to a state in the environment, see the corresponding state in Figure 3.3a. The agent, which is in the form of a human can move between these states by taking one of the four predefined actions; up, down, left or right. In relationship to the environment, the direction of each action is illustrated in Figure 3.3b.

(a) Definition of cell numbers in a grid-world environment of size 3×4.

(b) Definition of action directions relative to the cells.

Figure 3.3: Definition of cells and directions of action in relative to the cells for the discrete environment using Q-learning.

From this, the state space $\mathcal{S}$ and action space $\mathcal{A}$ can be defined as

$$\mathcal{S} = \{S_i\}, \quad \text{where} \quad i = \{0, \dots, 11\}, \tag{3.1}$$
$$\mathcal{A} = \{up,\ down,\ left,\ right\}, \tag{3.2}$$

where S_i represent the current state that the agent is located at each step and where there are $|\mathcal{S}| = 12$ number of states in total. The action space $\mathcal{A}$ is defined by the four predefined actions the agent can take to move between the cells.

Furthermore, the agent will be restricted to move only in the predefined states by remaining in the state when an action that will bring the agent out of the defined environment is taken. In state S_3, the robot will be moving in a predefined path and this will be the cell that the agent strives to be in since that is where the robot can collide into the agent. The robot path is defined by locating two end targets where the robot is moving bi-directionally between, with the same path back and forth. S_8 is defined to be the initial or also called the home position of the environment, where the agent starts and returns to after every collision.

3.1 RobotStudio Station Logic

In RobotStudio a closed loop of the simulation algorithm was implemented consisting of functional blocks in the station logic prompt in RobotStudio that controls the agent and robot. Figure 3.4 depicts a simplified block diagram of the station logic in RobotStudio. The blue block *Controller* seen in the block diagram is responsible for controlling the agent and is implemented in RAPID code. In order to move the agent between the cells in the simulation, so-called LinearMovers were implemented for each action that the agent can take, see the red blocks in the block diagram. For each block, the respective direction was configured, as well as a fixed distance and time duration of a move. The distance was set to move the agent 2 meters in 0.1 seconds in every direction which makes the agent move between the center points of the cells. The controller always made sure to wait until the agent has completed its move before taking the next action. The green block called *Update Position* uses sensors to read the position of the agent and then ensures to update the current agent position in the RAPID code.

Furthermore, the yellow *Home* block in the block diagram was added to position the agent in the home position, S_8. Collisions were detected by setting up a *Collision sensor* between the agent and the robot which flags a signal every time the two of them get in contact, see the purple block. This signal then triggers the home position block which sends the agent to the specified home position.

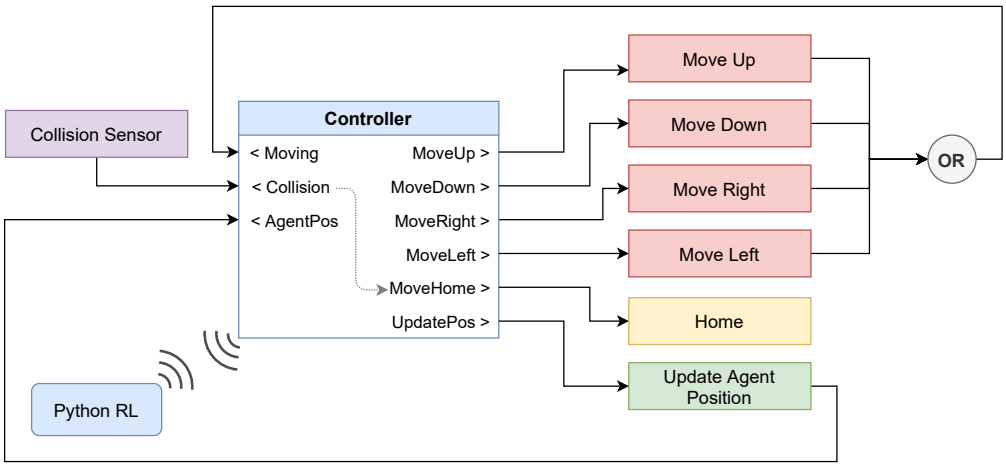


Figure 3.4: Simplified block diagram of the station logic in RobotStudio for model 1.

The controller communicates with the RL algorithm through sockets programmed in RAPID code in Robotstudio. The controller in RobotStudio receives what action the agent should take from the RL function and sends a signal to this direction block to move the agent. In return, the controller sends the current position of the agent to the algorithm in order to calculate the next action. The controller is constrained to only read messages from the RL algorithm when the movement of the agent is complete. More about the RL algorithm in the next section.

3.2 Tabular Q-Learning

The RL algorithm for this model is implemented in Python and during training, it communicates with the simulation through network socket communication. The model will control the agent in the simulation while the robot performs its tasks. For this model, a simple tabular Q-learning was implemented as described in section 2.3.5. A Q-table is first initialized to zero of size $state \times action$ and an action is then chosen by the Epsilon-Greedy exploration strategy. After the action has been taken the Q-table is updated by the Bellman equation as

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_a Q(s',a) - Q(s,a) \right), \tag{3.3}$$

where $Q(s,a)$ is the Q-value for a state s and action a, α is the learning rate, r is the reward, γ is the discount factor and s' is next state. The algorithm starts to explore the values of the state-action pairs thoroughly in the beginning and increasingly starts to exploit more than explore when more knowledge has been gained. The

table is finally updated so that the action that leads to highest reward in each state has the highest value.

3.3 Optimized Q-learning model

By utilizing the tabular Q-learning, the model managed to learn the agent how to move in order to collide with the robot. The agent was given a penalty of -2 when trying to move outside the defined states, an immediate penalty of -1 for each step taken and a reward of $+10$ for a successful collision. After training the Q-learning, the final policy may vary slightly, an example of the final policy obtained from one of the training can be seen in Table 3.1. The green cell represents the initial state S_8 of the environment and the red cell is the terminal state S_3 if the agent collides with the robot.

Table 3.1: One of the computed policies obtained from Q-learning represented by arrows on a grid cell. The green cell represent the start state and the red cell represent the terminal state.

→	→	→	←
↑	→	→	↑
→	→	→	↑

The results from the Q-learning training can be seen in Figure 3.5 which shows the gained reward iterating over 17 epochs. As seen in the plot, the reward converges to 5 which is the highest possible reward, calculated by the collision reward subtracted by the step penalty. From this, it can be concluded that the agent finds the shortest way to a collision.

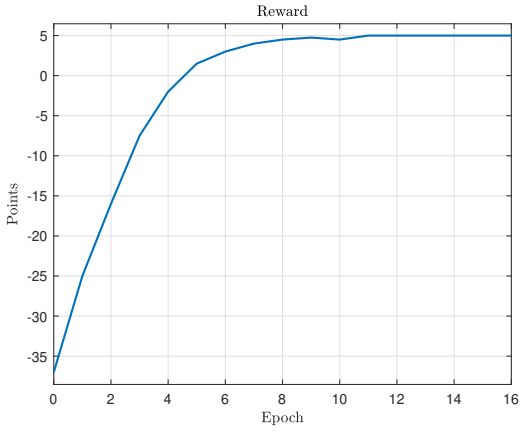


Figure 3.5: Reward gained during Q-learning against number of epochs.

The solution converged quickly in this simple model and took on average 35 seconds to find the optimal policy. The algorithm shows that in a simple environment it can find an optimal path to collide with the robot. However, the problem has been heavily simplified, thus the next step is to make it more complex to approach a more real-life scenario.

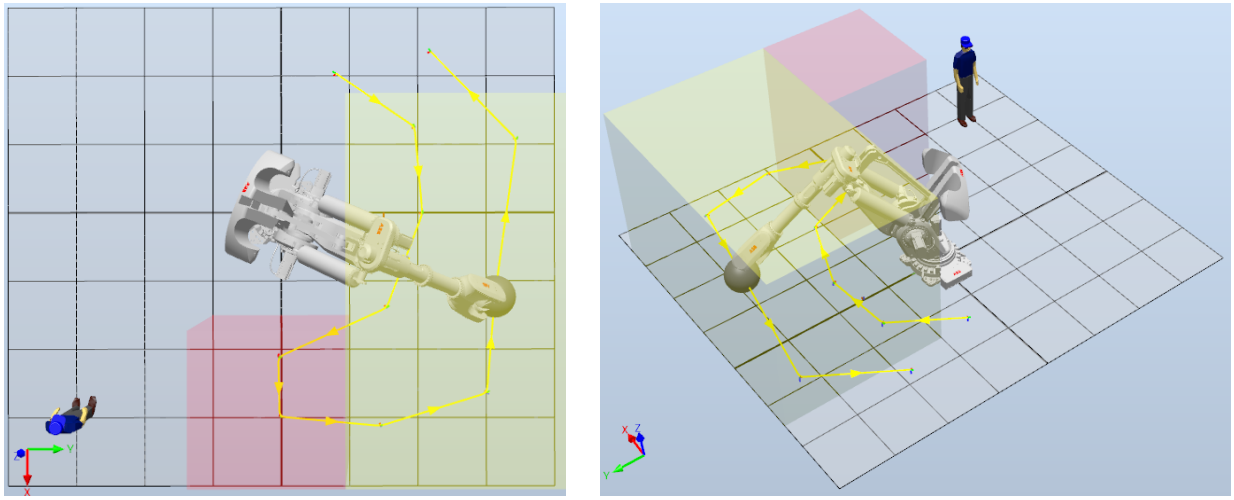
4

Discrete Environment with DQN - Model 2

To increase model complexity, further development of the discrete environment with tabular Q-learning in Chapter 3 was done. First of all, this new model is limited such that the agent should be penalized for taking an action that moves the agent into the robot, the agent needs to be hit by the robot. Furthermore, the robot moves in a larger area, the agent takes smaller steps (0.5 m) and collisions can only occur between the agent and the end-effector of the robot. The end-effector, which is also called Tool Center Point (TCP) now has a sphere fixed to it that is used to sense the collision between the agent and the TCP. Zones that alter the speed of the robot are introduced; a yellow and red zone also referred to as the warning and stop zone respectively. The objective is to heavily reduce the speed of the robot when the agent enters the warning zone and when the agent enters the stop zone, the robot should completely stop and not move until the agent has left the zone.

However, when implementing the speed reduction of the robot in RAPID code when the agent entered the warning zone, a problem was that the speed change was not applied to the robot until it reached the next target in the path. This was improved in the succeeding model. In continuation of this model, the warning zone is treated as a normal unprotected ground where no zone is defined.

To begin with, the environment had a simple path similar to the path in model 1, where the robot moved bidirectionally in one path from the top side of the environment to the bottom part of the environment with added intermediate targets in between. It was noticed that it was difficult for the agent to know in which direction the robot was moving and thus which side of the robot the agent should stand on to get hit. For this particular reason, the path of the robot was modified. First, the path was extended by moving the end target closer to the start target and adding sub-targets to ensure that the robot had a different path on the way back. When the robot then reaches the end target it would not move in reverse the same way back to the start target but instead move directly to the start target with the shortest route. This gives the robot a circular one-directional path to move in, see the path represented as yellow lines with arrows showing the direction in Figure 4.1.



(a) Top view **(b)** Oblique view

Figure 4.1: Environment of model 2 in RobotStudio consisting of an agent, robot, a warning and stop zone that the robot path, represented by the yellow lines, goes through.

Several states are added to the state space $\mathcal{S}$. The new state-space includes agent position (A_x, A_y) relative to the robot, TCP position (R_x, R_y), red and yellow zone flags $(Zone_R, Zone_Y)$ and a collision flag (C) as follows

$$\mathcal{S} = \{A_x, A_y, R_x, R_y, Zone_R, Zone_Y, C\}. \tag{4.1}$$

The four directions used in model 1 are now represented as the length of movement ($\pm$50 cm) in x- and y-axis in the environment. This action-space also has an additional action, namely to stand still $(-)$. The new action-space is now defined as

$$\mathcal{A} = \{x_{-50cm}, x_{+50cm}, y_{-50cm}, y_{+50cm}, -\}. \tag{4.2}$$

The environment is designed with an obvious dangerous zone. This is located above the yellow zone, here an unprotected area that does not limit the robot in any way is added to the model. This is to evaluate that the model can distinguish the zones and find a weak spot outside of them. Keep in mind that the robot's speed is not reduced in the yellow zone and is therefore treated as an unprotected area. The TCP position is obtained by computing the forward kinematics from the robot joints and this is easily done by using built-in functions in RobotStudio, calculated as in section 2.1.2.

4.1 RobotStudio Station Logic

Communication and movement of the robot was in this model reconstructed to a smart component, called *ComsAndControl* seen in Figure 4.2. The smart component works as a server and reads the state parameters from the environment, extracts, converts and sends the states over socket communication to the Python client.

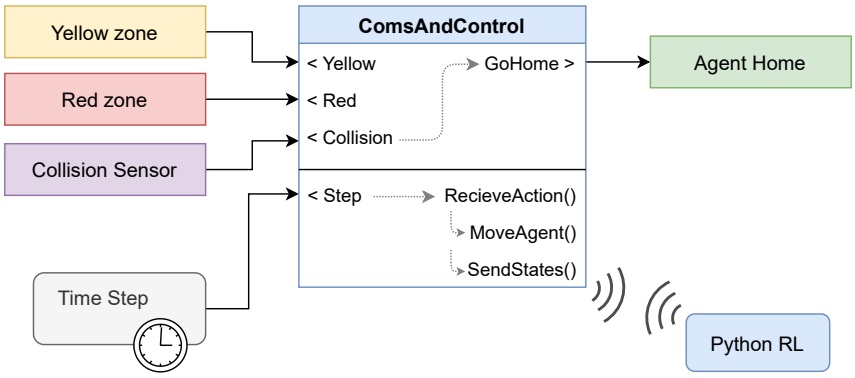


Figure 4.2: Simple station logic diagram over the smart component and Python communication.

Furthermore, the smart component receives actions from the Python client. The agent then performs a linear movement in the given direction of the client, much like the LinearMover in model 1. The component is flexible such that time, distance and object can be changed. This component combines most of the features from model 1, such as the RAPID code, linear-movement and socket communication into one component. Moreover, collision sensors has been added to detect when the agent is in contact with the yellow warning zone or the red stop zone in the environment. and these signals are then sent to *ComsAndControl*.

4.2 Deep Q-Learning Network

The tabular Q-learning method is not applicable to the number of states in this model because it creates too many state-action pairs. To handle the amount of $state \times action$ a neural network is introduced to estimate the Q-value, this is known as Deep Q-Network (DQN).

4.2.1 Defining Neural Network

The network consist of an input layer of size 7 which represent each state, two hidden layers with 32 nodes and an output layer with 5 nodes which represent each action. The network maps the input states $\mathcal{S}$, to (action, $Q(s, a)$) pairs rather than mapping state-action pairs to a q-value as in tabular Q-learning. The weights are initialized randomly at first and after taking an action the weights are then updated using experience replay, described in section 2.4.2.1. In Deep Q-learning the model uses two networks, one is the main network and the other is the target network with the same architecture.

4.2.2 Huber Loss Function

There are several loss functions that can be used, one alternative can be the Mean Squared Error (MSE). The MSE focuses a lot on minimizing larger errors rather

than smaller ones. When using a DQN however this might not be the best since the optimizer would want to radically change the network to minimize the large error. When doing that it also means that the target value will change which leads to that the error might not be reduced as much as when slowly reducing it.

An other alternative is the Mean Absolute Error (MAE), which focus on reducing both small and large error, however larger error does not matter more than the small errors here. Therefore, by combing these two loss functions the Huber loss function is introduced which acts as the MAE when the error is small and MSE when the error is large. This makes the loss function less sensitive to outliers in noisy data. The Huber function is also used in [22] where the DQN model was implemented. The Huber loss function is defined as

$$\begin{aligned} \mathcal{L} &= \frac{1}{B} \sum_{(s,a,r,s') \in B} \mathcal{L}(\delta), \\ \text{where} \qquad \mathcal{L}(\delta) &= \begin{cases} \frac{1}{2}\delta^2 & \text{for } |\delta| \leq 1, \\ |\delta| - \frac{1}{2} & \text{otherwise,} \end{cases} \end{aligned} \tag{4.3}$$

and where B is a mini-batch of transitions from the replay buffer.

4.2.3 Defining Reward Function

To optimize the algorithm, a reward function is defined where a higher score is given for a more dangerous situation. To sum the reward, several conditional variables that can be either 1 or 0 are defined. Its corresponding reward is given when the conditional variable is fulfilled and set to 1.

Two types of collisions can occur in this environment. The first one is that the agent walks into the robot directly and the second one is that the agent gets hit by the robot. Because the first-mentioned event, notated as *Agent Collision*, A_C is discouraged to happen it is given a negative reward of -150. The highest reward of positive $+100$ will be given when the robot initiates the collision with the agent and fulfills the *Robot Collision*, R_C variable. From the previous state and chosen action, the next position of the agent can be approximated by

$$A_{t+1} = A_t + a_t. \tag{4.4}$$

If the new position A_{t+1} of the agent enters within the TCP area it is considered that the agent initiates a collision, which is defined by

$$A_{t+1} \subseteq TCP_p \rightarrow A_C. \tag{4.5}$$

Furthermore, to keep the agent moving inside of the defined working station, it is given a negative reward of -200 when it is trying to take any action which can lead the agent out of the boundaries. This sets the variable A_{OB}, which stands for *Agent out of bounds*.

In addition to the conditional rewards, there are two rewards obtained for every step. Firstly, the agent obtains an immediate negative reward of -10 for each step it takes. Secondly, the distance in millimeters between the agent and robot, signified by A and R respectively, is calculated by the euclidean distance and the reward is then given by

$$r_d = \frac{10000}{\sqrt{(A_x - R_x)^2 + (A_y - R_y)^2}} \qquad \text{(mm)}, \tag{4.6}$$

where a smaller distance result in a higher reward r_d. This keeps the agent close to the robot so that it can quickly exploit the weaknesses. The final reward function is stated as

$$r_t = r_d + R_C \cdot 100 - A_{OB} \cdot 200 - A_C \cdot 150 - 10. \tag{4.7}$$

4.3 Optimized DQN Model

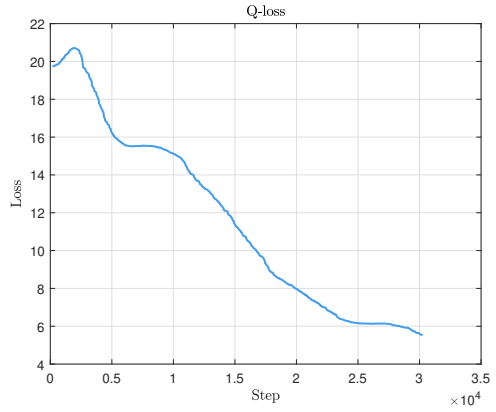
The optimized parameters for this model are presented in Table 4.1 and distinguish what is desired from the model. The ϵ-decay was set very low to force a lot of exploration so that the agent gets a large knowledge base and understands the difference between hitting and getting hit by the robot. The γ was set so that the next 3-4 steps would matter the most because the model tended to stay beside the robot and not engaging when having a larger "horizon". The batch size was set to give relatively fast calculations and a lot of data to not slow down the simulation.

Table 4.1: Optimized parameters for model 2.

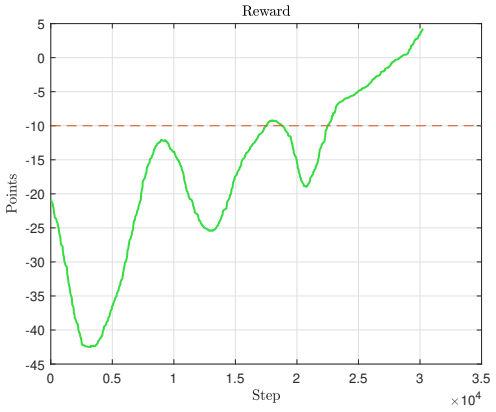
Batch size	γ	Learning rate	$\hat{Q} \leftarrow Q$	ϵ-decay / step
128	0.75	0.001	2000 steps	$5 \cdot 10^{-5}$

After tuning and training the model, the agent was able to learn to find an area where it was getting hit. From the reward function, the agent was also learning that it was not good to try to move out of the cell but instead be close to the robot's TCP because it is a higher probability to get hit there. Therefore, the agent was seen to follow the movement of the robot to then, in a certain area, walk in front of the robot. However, one problem that was encountered by this was that the agent sometimes walked into the red zone when the robot was there as well, which stopped the robot from moving and the agent could stay close to the end effector, collecting rewards for being close to it rather than getting hit.

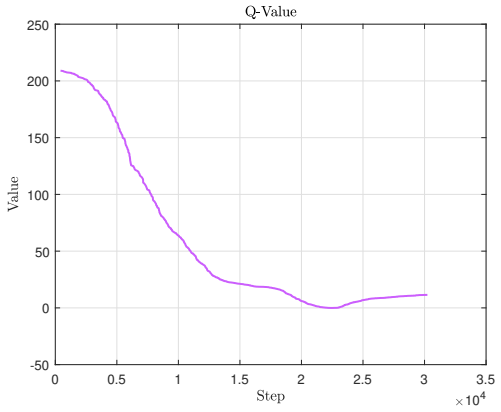
The results of loss, reward, Q-value and distance from training during 30k steps are shown in Figure 4.3. It found an optimal policy after 2 hours and 32 minutes, thus it can be seen that a more complex environment heavily increased the training time.



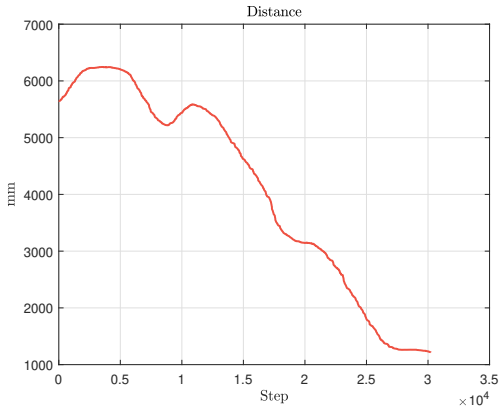
(a) Q-Loss



(b) Reward



(c) Q-value



(d) Distance between TCP and Agent

Figure 4.3: Data during training for 30k time steps of model 2.

The plots have been smoothed in order to see the average value during training. As seen in Figure 4.3a the loss drops to around 6, in Figure 4.3b the reward is dropping in the beginning when exploring and after learning, the reward starts to increase and eventually end up at a positive value of 5. The dashed line represents the penalty for each taken step, going over this threshold means that the model now gains value for each step. The Q-Value in Figure 4.3c starts at over 200 and converges to around 10, this is because it approaches the value of each state which is close to the smoothed reward. Furthermore, it can be seen in Figure 4.3d that the agent learns to be closer to the end-effector of the robot. On average it has a distance of 1 m between each other. In reference, this is the length of one square of the environment floor seen in Figure 4.1.

Figure 4.4 depict the policy of from each state in the environment with the TCP positioned in the red zone. As can be seen the agent has gained so much knowledge that it acknowledges that if the distance is large it should try to approach the TCP to decrease the distance.

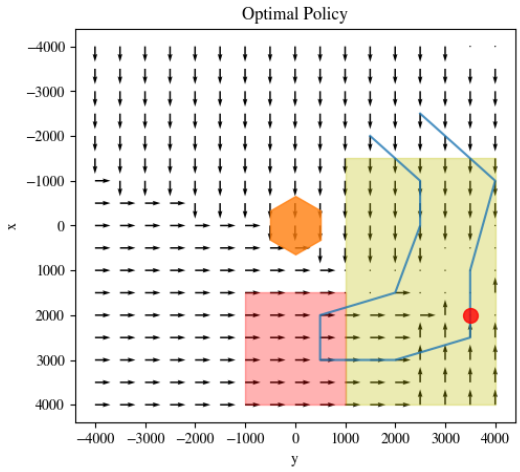


Figure 4.4: Optimal policy from each state in the environment when the TCP, visualized as a red dot is positioned in the red zone.

When the robot's TCP is located outside of the red zone as seen in Figure 4.5, it can be seen that the agent's policies would move it to a dangerous position where eventually the robot will collide with it, which is exactly the desired behavior.

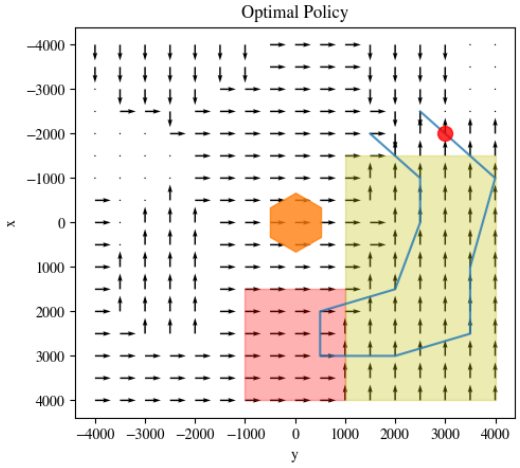


Figure 4.5: Optimal policy from each state in the environment when the TCP, visualized as a red dot is positioned in the yellow zone.

When the robot is located in the red zone as in Figure 4.6, the agent wants to stay in the red zone as well, even tho it stops the robot from moving. This is mainly because the area of the TCP sometimes takes two "spaces", thus it has been learned that if standing in front of or around the end-effector at this spot it will most likely get hit within that space. Furthermore, the agent has also outsmarted the environment, by acknowledging that even if it will not get hit in this space, it would acquire more points by standing close to it than to trying to force a dangerous situation. The agent has learned how to get hit by the robot without moving into it and also how to utilize the environment to its advantage in a way not meant.

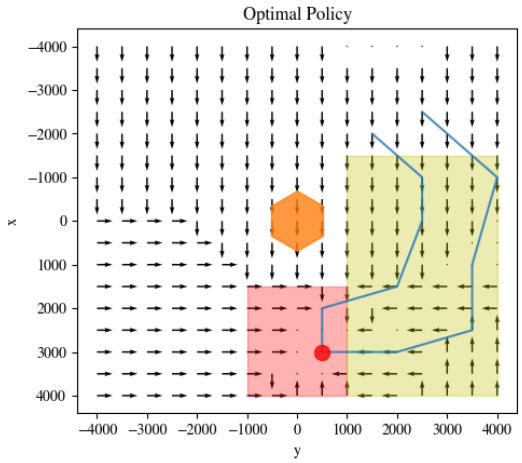


Figure 4.6: Optimal policy from each state in the environment when the TCP, visualized as a red dot is positioned in the safe zone.

During the work with this model, both problems and insights arose. The first problem noticed was that at times, the socket communication between the simulation and the algorithm in Python had a delay which was a clear bottleneck for training this model fast. This is important to improve in future models to accelerate the training.

Other important observations were also made. The importance of setting proper rules for constraining the agent to not walk into the robot was shown to be very important in this model since it was able to find loopholes in it. One reason for this is the quite large sphere fixed to the TCP. This must be further investigated and carefully designed for the next models. Other observations made when tuning the model were that a network with 3 layers seemed to be a sweet spot and worked best. A higher value of the discount factor γ, made the agent consider the long-term reward more, as the agent gained a positive reward depending on the distance at each time instance this made the agent stand still around the TCP. A lower discount factor γ, worked better for this model as it became more focused on the collision reward instead of the large cumulative reward, this can be avoided in the future by instead setting a negative reward based on distance as this would encourage the agent to reach terminal state. For future models an algorithm that explores even more than this DQN in the beginning and learns quicker is preferable.

5

Continuous Environment with SAC - Model 3

As the concept has been established in model 2 the next step is to make the agent more human-like. The action space of the agent should therefore be converted from discrete to continuous to create a more human-like behavior. To do this the agent's actions have been altered to now being direction and speed, which will be a change in degrees and velocity respectively as

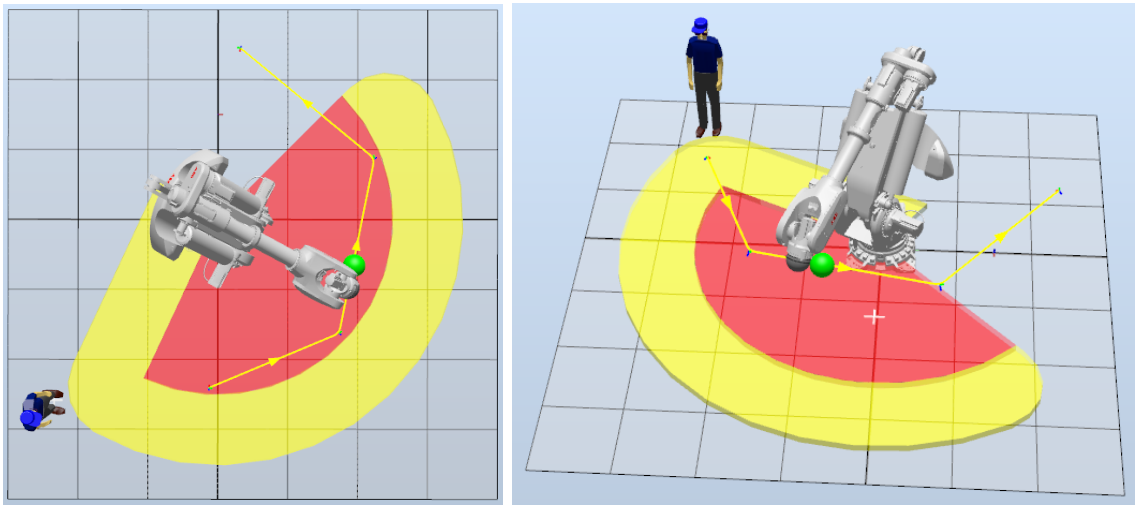
$$\mathcal{A} = \{\Delta A_\theta, \Delta A_\upsilon\} \qquad \text{where } \Delta A_\theta \in [-20°, 20°], \;\; \Delta A_\upsilon \in [-2, 5]. \tag{5.1}$$

Furthermore, the present speed, angle in the plane and TCP speed will be added as states, and the collision state is removed from the states. The red and yellow zones are kept as states as these describe the warning and danger zones in any environment, giving the agent knowledge about the safety system. This gives the new state-space

$$\mathcal{S} = \{A_x, A_y, R_x, R_y, A_\theta, A_\upsilon, R_\upsilon, Zone_R, Zone_Y\}. \tag{5.2}$$

When having a continuous agent the environment can be altered to a more real-life scenario. A red zone is added within the path of the robot, thus if the agent enters this area the robot shall stop immediately. In the outliers of this red zone, a yellow zone is present, this zone reduces the velocity of the robot by 80 % if the agent is positioned. As in previous models, there is one obvious dangerous situation where the robot could be a threat, that is when the path leaves both zones, moving at full speed in a non-secured location. This is to give the new algorithm a clear idea for evaluation.

The path has been changed to not have a specified circular path as in model 2. Instead, the robot has a specified path in one direction and when returning to the starting point again it will take the easiest path in terms of minimum rotation of joints back to its starting point. In simulation, it can be seen that this movement back to its starting point goes on the outside of the specified path shown in Figure 5.1.



(a) Top view **(b)** Oblique view

Figure 5.1: Environment of model 3 in RobotStudio with a yellow slow down zone, a red stop zone and a green sphere predicting the braking position of the TCP.

An extension that has been added to this model is the predicted braking TCP position, visualized as a green sphere in Figure 5.1. In the simulation, the robot stops on command at the current position but in real applications, the robot will continue to move for a distance until it has completely stopped and this distance is usually called the braking distance. The braking distance depends on several variables, of course a larger industrial robot will have a larger braking distance than a smaller one due to its weight. Other parameters such as the robot's speed, inertia, torque, any additional payload, etc. also affect the braking distance. The distance is usually small but can sometimes be as long as several meters [29]. Therefore, it is important to consider the braking distance when the agent is close to the robot's TCP. This also adds more reality to the environment. The distance between the two spheres changes with the speed of the robot, higher speed results in greater distance and contrariwise. However, when the robot has its maximum TCP speed, which is at 3000 mm/s and the distance is at its largest, the distance will still not exceed distances over 250 mm between the spheres in this case. Moreover, the dimensions of the agent are 2000 mm height, 700 mm width and 250 mm depth. This means that in this case, there are minimal chances that the agent could walk in between the spheres without getting hit. If the distance becomes larger in future models, this must be further investigated and taken into consideration to avoid that the agent can walk past this without getting notified that it would be a collision in real life.

In RobotStudio there are two types of braking stops; *category 0* and *category 1*. The former stop, category 0, activates the physical brakes of the robot, which can lead to that the robot deviates from its programmed path while braking. The latter stop, category 1, will electrically engine brake the robot motors, and the last instruction given to the robot will be followed while braking, keeping the robot in its planned path. For the predicted braking TCP position in this model and all future models, category 0 is used.

5.1 RobotStudio Station Logic

The main station logic for model 3 is the same as in model 2. Minor changes has been made to the *ComsAndControl* smart component. Here an input of cell boundary size has been added to easier determine the area the agent should walk in. This is to limit the agent to not walk too far away since no collision will occur far away from the robot. If the agent tries to move out of this specified boundary, its angle is changed by adding 180 degrees to it so that it is facing inward again and can continue roaming around. This leads to that the agent will not get stuck at the boundary borders for a longer time. Furthermore, a port input was added to enable parallelization during training. Apart from this, a smart component called *SignalExtractor* has been implemented which extracts the current TCP position as well as its predicted position after braking. The TCP brake position will at every iteration be updated in order to visualize it in the simulation. The updated station logic where the additional smart component has been added can be seen in Figure 5.2.

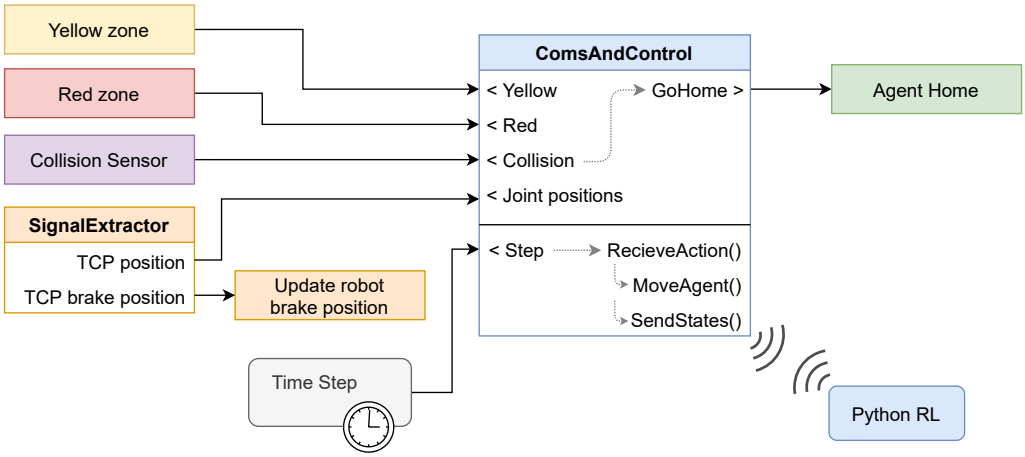


Figure 5.2: Station logic diagram over the smart component and Python communication.

Furthermore, since the agent now is continuous, the distance of every move has been changed to be within 0-50 mm. This is regarded as the speed state as only the length and time of the step is changeable by the agent, thus the difference in step length has been added as action to the agent.

5.2 Soft Actor Critic

Previously discrete actions were taken where the agent was either standing still or moving at a constant speed in predefined directions. To transition from the discrete case to the continuous a new RL algorithm was needed. For this problem where the agent roams around in a almost fully secure environment trying to find any small defects in the security a lot of exploration is needed. Additionally, it is preferable for the agent to learn fast. By studying paper [30] where different model-based RL

algorithms were bench-marked it was clear that Soft-Actor Critic (SAC) algorithm would meet these requirements.

SAC is an algorithm that aggressively explores in the beginning and is one of the top-performing continuous RL algorithms [24]. The algorithm rewards exploring, uses an actor-critic network to judge the actions taken and at the same time integrates experience replay as in DQN, see section 2.4.3 for further details about SAC. An important property in SAC is the temperature hyper-parameter α [25], which controls the trade-off between exploration and exploitation. A high α will result in a lot of exploration and almost a uniform policy that fails to exploit the reward. In contrarily, a low α will learn quickly in the beginning, but then become almost deterministic since it does not explore enough. For this particular reason, α needs to be carefully tuned, it can be done both manually and automatically.

To improve exploration even further the actions are sampled from a uniform random distribution for a fixed number of steps at the beginning of the training. This trick is used in the OpenAI SAC implementation and is used to capture as many random states as possible to create a good learning base [31]. After this, the actor-network is in charge of taking actions as in Algorithm 4. In addition to the original implementation a policy delay a π_{delay} is included. It slows the learning of the policy to stop it from learning to much from a sudden divergence in the Q-value estimation.

5.2.1 Reward Function 1 - Agent Constraint

As explained in section 2.3.3, defining a good reward function is crucial but also challenging. The main objective for the agent in this problem is to collide with the robot, however, this must be done in the right way. A correct collision occurs when the agent is hit by the robot and not when the agent goes straight into the robot. There are no restrictions in the model that limit this from happening, but a large amount of negative reward is given if the agent walks into the robot. This should essentially lead to that the agent learns to get hit by the robot instead of walking into it since this gives much more rewards.

A problem with this is that in previous models the agent collected a small negative immediate reward for every action it takes and if the robot roams around a lot without getting hit it has gained a large number of negative rewards. When the agent then is correctly hit by the robot and gets the collision reward, the total reward could still be negative as the accumulated negative reward is dominant. Therefore, there is a risk that the agent thinks that it is better to quickly and incorrectly walk into the robot to collide which in total gives it a higher reward, as this resets the count. For this reason, the agent only reaches the terminal state when a correct dangerous collision occurs. Previously all collisions were seen as terminal state.

The function for examining if an action is okay or not has been further evaluated,

the previous models simply said that if the agent enters a box of size $n \times n$ centered around the TCP position and goes towards the robot a negative reward is given. This simple solution was possible in the discrete case because the agent walked in four straight directions, but will not work in the continuous since the agent has a continuous movement now. Therefore, a more realistic area around the TCP where collisions can occur and a negative reward is given must be more circular. Figure 5.3 illustrates the parameters used to compute the smallest angle $\Delta\theta$ from the agent angle θ_A.

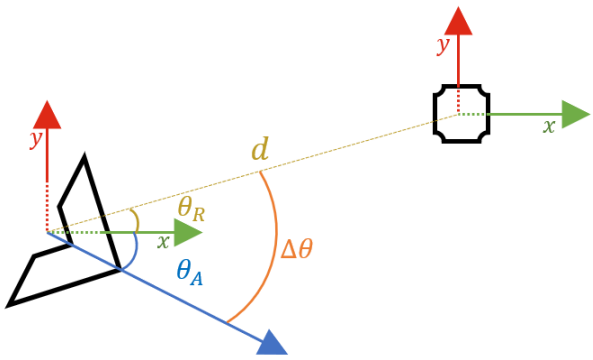


Figure 5.3: Illustration of distance d, agent angle θ_A, relative robot angle θ_R in the simulation and the angle $\Delta\theta$ that shall be computed. Agent is illustrated as the arrow head and the squared box represents the TCP-position.

The delta angle $\Delta\theta$ describes how the robot is positioned relative to the agent and is computed as

$$\Delta\theta = ((\theta_A - \theta_R) + 180) \bmod 360 - 180, \tag{5.3}$$

where θ_A is the angle of the agents action and θ_R is the relative robot angle. Based on $\Delta\theta$ it can be determined if the agent is heading towards the robot in a dangerous situation. Furthermore, the distance d determines if the agent is close enough to be a threat. The reward can then be constructed as described in Algorithm 5 where $L_{\Delta\theta}$ represents the dangerous angle limit, L_d represents the dangerous distance and L_v the speed limit of the agent. These variables describe the agent's dangerous movements, it is desired that the agent acts safely and force the robot to make a dangerous movement, thus these situations are penalized.

Algorithm 5 Reward function model 3 - Agent Constraint

$d = \sqrt{(A_x - R_x)^2 + (A_y - R_y)^2}$
$r = -\frac{d}{750}$
if $A_v + \Delta A_v \notin [v_{min}, v_{max}]$ **then**
  $r = r - 30$
end if
if Collision **then**
  if $\Delta\theta \leq L_{\Delta\theta}$ **and** $d < L_d$ **then**
    $r = r - 250$
  else
    $r = r + 2500$
    $terminal = True$
  end if
end if

5.2.2 Reward Function 2 - TCP speed

The first mentioned reward function implies that the human should not try to engage in a collision by walking towards the robot. This condition can be too restrictive. In a real-life application, the best-case scenario is that the robot stops its movements even when the agent does everything to get hit by it, thus a new reward function is also examined. The new reward function is based on the speed of the TCP and states that the agent is allowed to get hit by the robot in any manner except for when it stands still, as it cannot avoid a collision in this case.

Algorithm 6 Reward function model 3 - TCP speed

$d = \sqrt{(A_x - R_x)^2 + (A_y - R_y)^2}$
$r = -\frac{d}{750}$
if $A_v = 0$ **and** $R_v = 0$ **then**
  $r = r - 100$
end if
if Collision **then**
  if $R_v = 0$ **then**
    $r = r - 250$
  else
    $r = r + 2500$
    $terminal = True$
  end if
end if

This model is less complex and thus puts the system to a harder test. Previously it was assumed that the human did not want to get hit and the human could go close but not engage. In this case, it is encouraged to engage in the robot as it should stop before it gets hit.

5.3 Parallel Deep Reinforcement Learning

As the environments and agent become more complex, the training time increases. To be able to faster evaluate models and speed up training time, a parallel DRL method is implemented. The method utilizes one model to control several agents. The states from each model are gathered and the actor takes an action based on the states given for each environment, the feedback is given and added to the buffer. The main feature of this approach is that the data become more Independent and identically distributed (IID) which is a prerequisite for learning algorithms and has been a weakness within RL [32].

Algorithm 7 Parallelized SAC

Initialize parameter vectors θ_1, θ_2, ϕ
$\bar{\theta_1} \leftarrow \theta_1, \bar{\theta_2} \leftarrow \theta_2$
Initialize $\mathcal{D}$ and n_{env} environments
for each step **do**
  for each environment$\in n_{env}$ **do**
    for each environment step **do**
      $\mathbf{a}_t \sim \pi_\phi(\mathbf{a}_t \mid \mathbf{s}_t)$
      $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1} \mid \mathbf{s}_t, \mathbf{a}_t)$
      $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{s}_t, \mathbf{a}_t, r(\mathbf{s}_t, \mathbf{a}_t), \mathbf{s}_{t+1})\}$
    end for
  end for
  for each gradient step **do**
    $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$
    $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
    $\alpha \leftarrow \alpha - \lambda \hat{\nabla}_\alpha J(\alpha)$
    $\bar{\theta_i} \leftarrow \tau\theta_i + (1 - \tau)\bar{\theta_i}$ for $i \in \{1, 2\}$
  end for
end for

The parallelized SAC algorithm described in Algorithm 7 is known as a centralized learner with asynchronous actors. These actors take actions in parallel and train on a common learner, illustrated in Figure 5.4. As can be seen, the learner is updated after all actors have taken their actions, this should decrease the training time significantly and the data available for the actor at each learning step would be n_{env} times bigger, thus the replay buffer gains a more IID buffer. This makes each learning step faster and more stable.

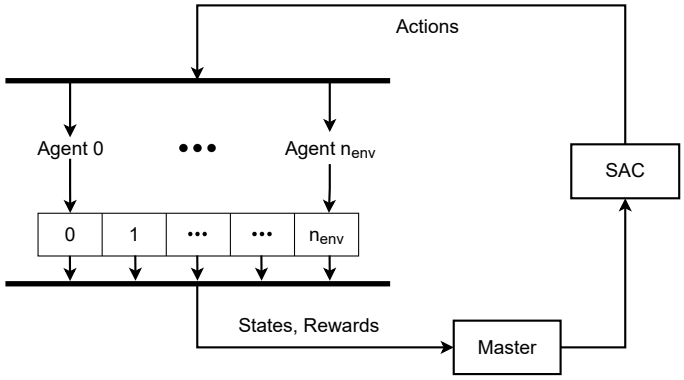


Figure 5.4: Parallel reinforcement learning method using SAC.

5.4 Optimized Simplified SAC Model

An early problem during training was the lack of good collisions in the replay buffer. In the continuous case, the amount of data is much larger and therefore the good collisions, which only saved one data point of the collision before the simulator was reset, can easily get lost. Thus, the reset after each collision was removed.

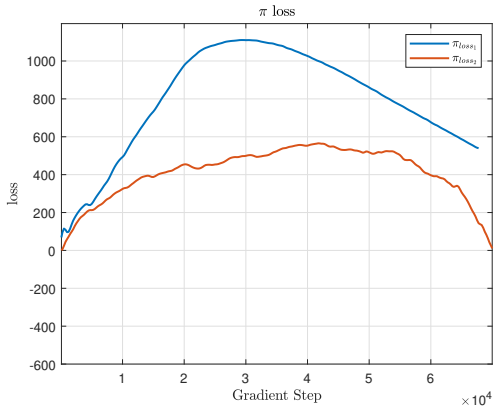
As seen in Algorithm 5 and 6 the terminal state is only given on correct collisions. The negative cumulative reward could else be prevented by walking towards a bad collision and reach the terminal state, thus encouraging bad collisions.

The addition of a distance reward is of great essence for the speed of learning, if a reward was only given when a collision would have happened the training would take a very long time, as the data points of good rewards are so rare. The distance reward makes the agent try to keep close to the robot thus, in the end, it will be easier to find the weak spots in the environment. Two environments were evaluated with the same parameters, seen in Table 5.1, but with the different reward functions.

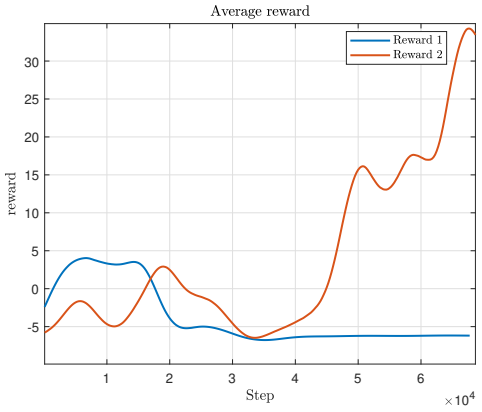
Table 5.1: Optimized parameters for model 3.

π_{lr}	α_{lr}	Q_{lr}	α	γ	Random steps	Epoch size	π_{delay}	Δ_{steps}	bs
3e-4	3e-4	3e-4	0.2	0.99	16000	4000	2	2	256

A large number of random start steps was chosen as it creates a good buffer for the agent and could give the actor plenty of data to understand the behavior of the environment. The policy delay, π_{delay}, stops the actor from learning too fast but still updates the alpha and critic to the desired behavior and the two gradient steps per iteration reduces the training time. In order to choose which reward function to use, a comparison between the two reward functions during training was done and the results are shown in Figure 5.5.

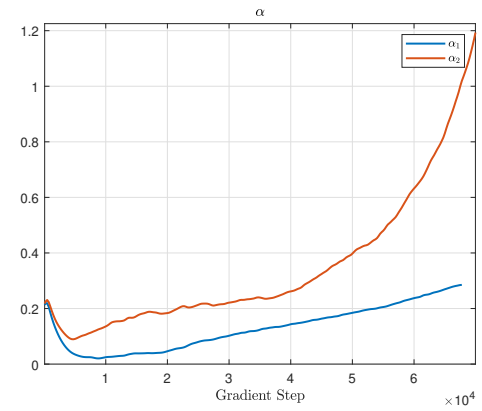


(a) π-loss comparison



(b) Average reward comparison

(c) Q-loss comparison



(d) Temperature α comparison

Figure 5.5: Results from the comparison of reward functions 1 and 2 during training of model 3.

As seen in Figure 5.5b reward function 1 converges to a very low reward after the initial random steps. The agent stands still in the environment and this is because of the complexity of the reward function. After converging to this solution the replay buffer was filled with states from the start position and nothing else, thus it stood still and trained on the same states repeatedly. This problem occurred more often during training for reward function 1. Figure 5.5d shows that α is not increasing as much for reward function 1 as for reward function 2. This could explain why the agent with reward function 1 stands more still in the environment, since it does not weight exploration as much. However, with reward function 2, Figure 5.5b clearly shows that the reward is increasing and the model finds an optimal solution that maximizes the average reward. In Figure 5.5a, the π-loss for reward function 2 is lower than for reward function 1, which clarifies that reward function 2 is more certain. It can be seen in Figure 5.5c that the Q-loss early starts to converge, after six thousand steps the α and π-loss also started to diverge and the agent did eventually forget the optimal solution, which resulted in that the agent stood still in the environment. The optimal policy was found after 1 hour and 15 minutes, after this the solution began to diverge. The problem of the sudden divergence during training will be further investigated in the following model, model 4.

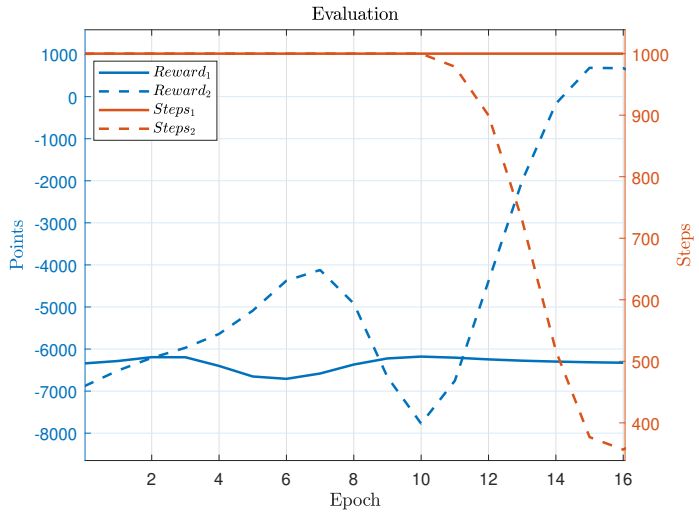


Figure 5.6: Evaluation of reward function 1, represented as solid lines and reward function 2, illustrated as dashed lines.

In Figure 5.6 it can be seen that reward function 2 performs the best of the two reward functions. From the fact that reward function 2 was slower and less prone to error, it was chosen as the reward function to continue development on. It is not as complex as reward function 1, but solves the task and truly sets the system to a test. What is of further importance is that neither of the two functions networks converged during training, thus the SAC algorithm needs further development to be able to give a robust solution.

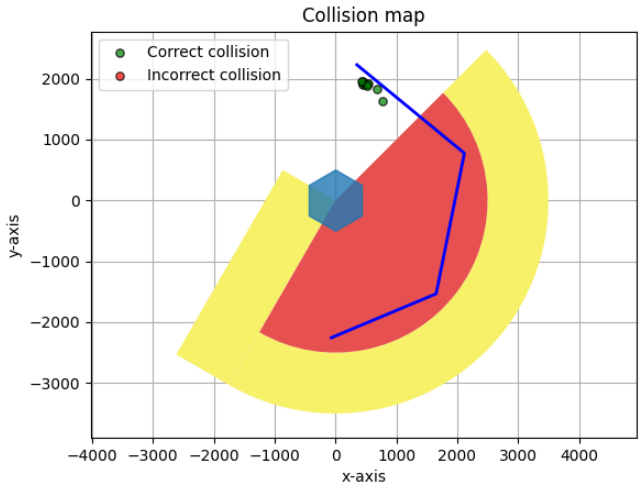


Figure 5.7: 10 collisions mapped over the workstation, marked as green and red dots respectively. The workstation consist of the robot depicted as a blue hexagon in the middle as well as the red and yellow zones.

After training the model with the chosen reward function the collisions were tracked

to analyze if the collisions were clustered in any certain area or more specific in the expected unprotected area outside the red and yellow zones. As can be seen in Figure 5.7, all correct collisions are clustered in the upper unprotected zone. This means that the agent has gained knowledge over where and how collisions should take place to maximize the reward.

6

Integrating Forward Kinematics - Model 4

To make the model applicable on collisions between the agent and all joints of the robots the state-space is further extended and the action-space remains unchanged from previous model as

$$\mathcal{S} = \{A_x, A_y, J_1, J_2, J_3, J_4, J_5, J_6, R_v, A_\theta, A_v, Zone_R, Zone_Y\} \tag{6.1}$$
$$\mathcal{A} = \{\Delta A_\theta, \Delta A_v\} \qquad \text{where } \Delta A_\theta \in [-20°, 20°],\ \Delta A_v \in [-2, 5]. \tag{6.2}$$

The state-space $\mathcal{S}$ now contains all joint values of the robot and with this further knowledge the agent can find the position of each joint. This makes the collision area much larger and more applicable in a real-life scenario. The break position of each joint is also available and will be utilized in the reward function.

The first step is to verify the concept, that the agent can understand and find collisions with forward kinematics calculations. This will be in the same environment as the previous model but the agent will be able to collide with all joints. After verification, further development is done to both the algorithm and reward function to create a stable and trustworthy solution, tested in a more secured environment.

6.1 RobotStudio Station Logic

The main station logic continues to be the same as in the previous model with some modifications in the smart components, see the overall block diagram of the station logic in Figure 6.1. The difference is that collision with all joints is now considered rather than just the TCP to make it more realistic. The *SignalExtractor* now extracts all joint angles, each of their braking angles and finally the robot's DH parameters. For this reason, the TCP position and its braking position are removed, since these can be obtained from the joint values. The *ComsAndControl* component takes the joint values as input and is forwarded to the RL algorithm. The braking joint angles are used to visualize the robot's braking position during simulation using a JointMover component. The integration of DH parameters enables any robot to be tested with the current algorithm.

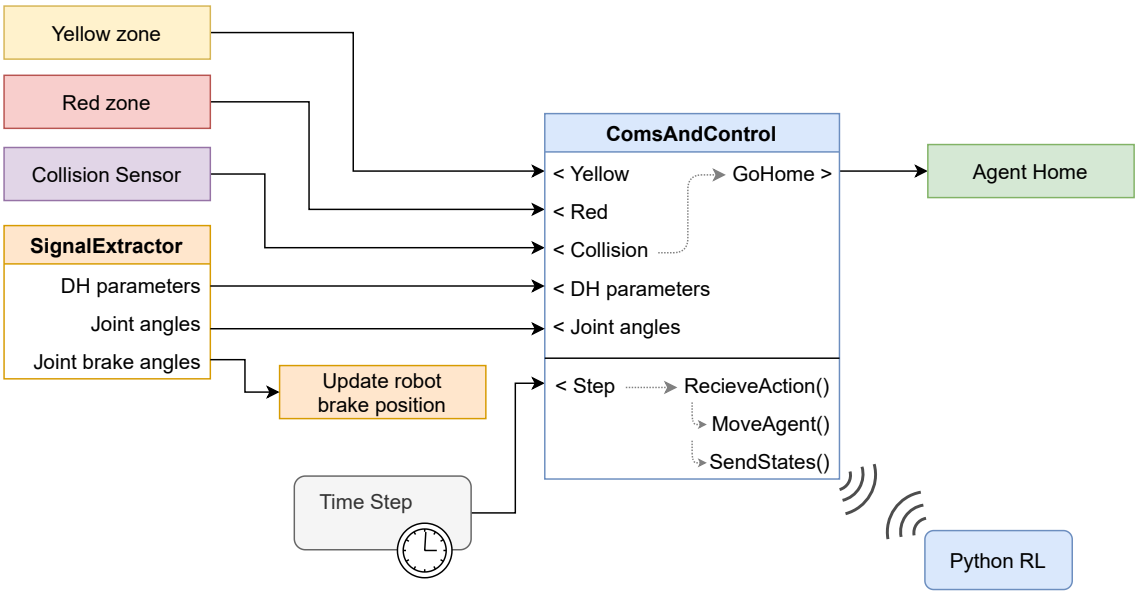
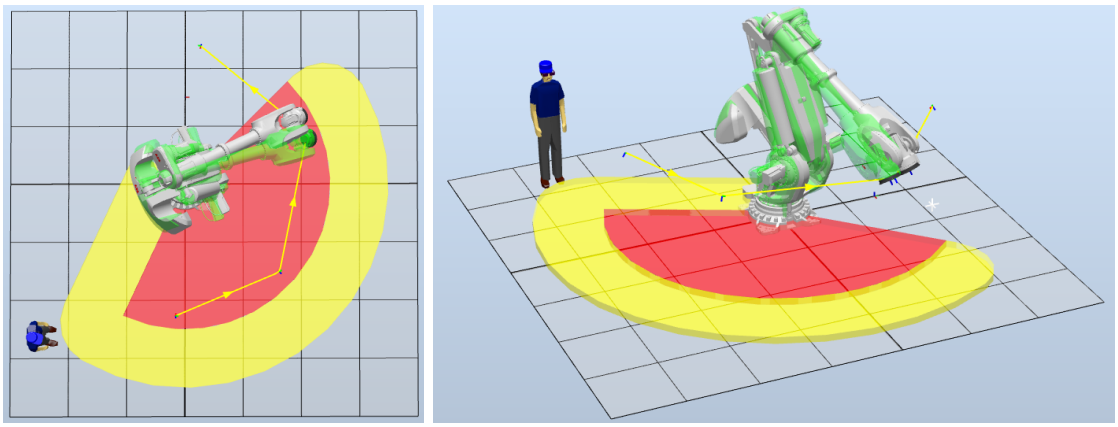


Figure 6.1: Station Logic diagram over smart components and Python communication for model 4. Notice that the *SignalExtractor* extract joint angles instead of TCP position.

6.2 Concept Verification

As mentioned, the algorithm will first be verified on the same environment used in model 3, see section 5. However, now collision can instead occur with the whole robot and therefore the green predicted braking TCP sphere is now replaced with the whole robot's predicted braking position, see Figure 6.2. This is done to evaluate that the algorithm still works with the new states before further developing it.



(a) Top view **(b)** Oblique view

Figure 6.2: Simple environment of model 4 in RobotStudio with a yellow slow down zone, a red stop zone and a green robot predicting the break position.

6.2.1 Soft Actor Critic

After the results in model 3, section 5, SAC is the algorithm that will be used to solve the extended problem as well. Model 3 in section 5 failed during training so further improvements were done to the algorithm. The sudden changes in the behavior of the agent were mostly dependent of overestimated Q-values, this could be a consequence of the large rewards and penalties for good and bad collisions. The Q-function, also known as the critic, over-trained on these high values, thus lost the knowledge in the process as all states became high valued. To prevent this, the reward for good and bad collisions were lowered and reward standardization was implemented during training as

$$r_s = \frac{r - \bar{r}}{\sqrt{\frac{\sum (r_i - \bar{r})^2}{n}}}, \qquad \text{where} \qquad \bar{r} = \frac{1}{n} \sum r_i, \tag{6.3}$$

and where r_s is the standardized reward. Thereby each reward batch that is trained, is standardized, a mean around zero and within a fixed interval. The standardization makes it easier when calculating the loss to decide which states the gradient should approach and which it should avoid and it limits our reward within a fixed interval. The downside is that it moves the mean of the reward which can affect the agent's will to live, as a negative reward can become positive and vice versa, thus changing its will to find a terminal state. Furthermore, a smaller buffer was used as the policy tends to diverge from the optimal path when too much data was available as the critic could not successfully estimate the values of the states. By lowering the number of states the critic more frequently experience the same states which make it converge faster. The drawback of this is that a smaller replay buffer only saves the latest actions and states, which would converge with time. As they converge the replay buffer eventually only contains the optimal solution. At this point, the Q-function overfits these values and might diverge. To solve this, 5% of the buffer was allocated to random states which would prevent it from overfitting as fast, visualized in Figure 6.3.

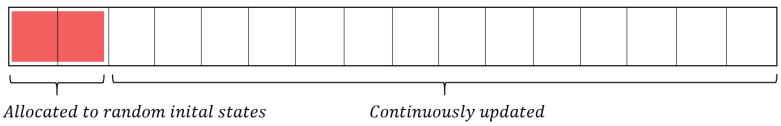


Figure 6.3: Replay buffer memory allocation.

6.2.2 Defining Reward Function

Previously the collision prevention has been quite straightforward as it has been in a 2-dimensional plane between positions. Now the problem is more extensive as the danger needs to be approximated from the angles of the robot. At the beginning of each simulation, the Denavit–Hartenberg (DH) parameters are extracted and forwarded from RobotStudio to the algorithm in Python. With this, a mathematical model of the robot can be created and the transition matrices, T, from the base frame

to the individual joints are available. Through forward kinematics the position of each joint can then be calculated as described in section 2.1.2. The reward function is created to reward the distance between the agent and each joint. It is scaled to generate a higher reward for being close to the joints further out on the robot since these joints tend to have a higher velocity and consequently are more dangerous. But if the robot has a slow speed the penalty will decrease because then it is considered to be less dangerous. The reward function can be seen in Algorithm 8, where T_i is the transformation matrix between joint i and base frame, and J_{p_i} is the joint position of joint i.

Algorithm 8 Reward function for simplified model 4

$J_{p_i} = T_i(J_{\theta_{\{0:i\}}})$ for $i \in \{0, 5\}$
$d_i = i\sqrt{(A_x - J_{p_{ix}})^2 + (A_y - J_{p_{iy}})^2}$ for $i \in \{0, 5\}$
$r = -\frac{\frac{1}{n}\sum d_i}{750}$
if Collision **then**
  if $R_v = 0$ **then**
    $r = r - 40$
  else
    $r = r + 40$
    $terminal = True$
  end if
end if

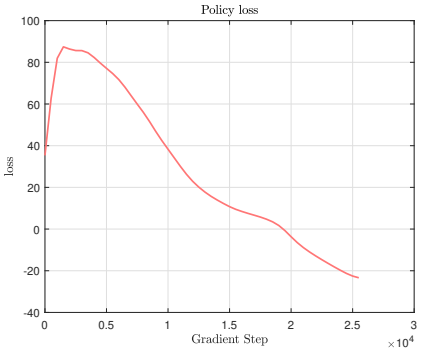
6.2.3 Optimized SAC Concept Model

Table 6.1: Parameters for optimized SAC concept model.

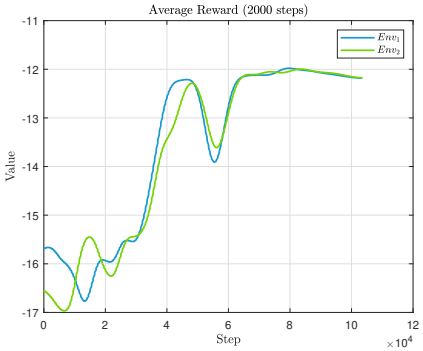
Parameter	Value
α	0.2
γ	0.99
τ	0.005
α_{lr}	$3 \cdot 10^{-4}$
π_{lr}	$3 \cdot 10^{-4}$
Q_{lr}	$3 \cdot 10^{-4}$
n_{env}	2
batch size	512
random steps	0 steps
e_{steps}	4 steps
Replay buffer	50000
Allocated initial buffer	2500
Epoch size	4000 steps
Hidden layer size	256

In the environment that the concept was verified on, seen in Figure 6.2, the new model solved the task at hand very fast. The difference between this and the previous

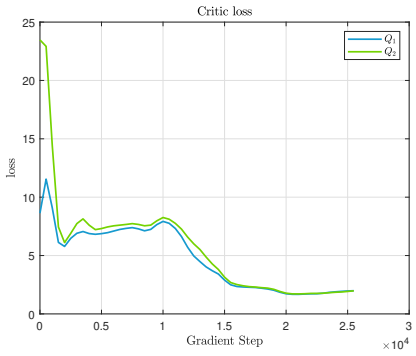
model is that the agent has the possibility to collide with the whole robot. In this case, the base of the robot and the TCP both pose a danger as they are not always located in a safe zone. The quest in hand was thus easy and the policy had converged quickly. A relatively small buffer was used, a part of this was allocated to initial states to prevent it from overfitting during training. To make sure that enough data was available during training the policy was only updated each fourth step and a larger batch was used to smooth the training. The values used for all parameters are presented in Table 6.1. The enhancements with a smaller buffer, allocated initial memory, delayed learning and reward standardization gave better results compared to previous ones in section 5. All networks converged to a solution, as seen in the plots in Figure 6.4.



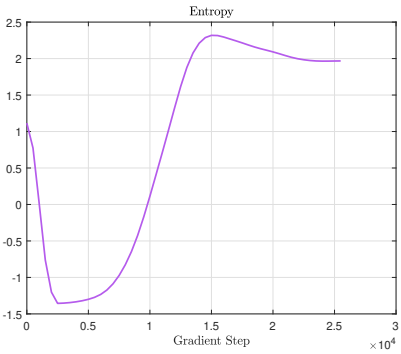
(a) π-loss



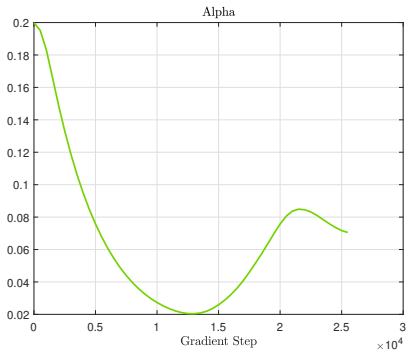
(b) Average reward



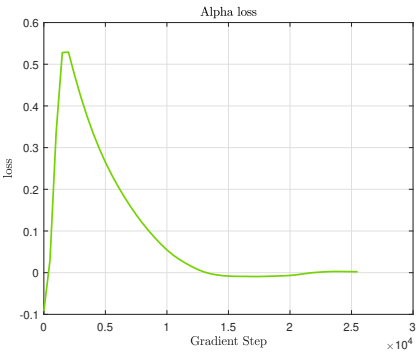
(c) Q-loss



(d) Entropy $\log \pi$



(e) Temperature α



(f) α-loss

Figure 6.4: Data obtained during training of model 4.

As seen in the Temperature α plot, Figure 6.4e, the exploration is not so heavily weighted as it has a relatively low value. It approaches zero as it finds a good solution quickly and after approximately $1.2 \cdot 10^4$ steps the alpha value increases again and encourages more exploring. It does not increase too much until it quickly turns and settles at a constant value. This behavior of finding and testing one solution and not being satisfied with it, to then after a while exploring for a new solution, is exactly what is desired from an algorithm like this in this situation. A better solution was not found, which can be seen in the average reward, Figure 6.4b. The fact that no other solution was seen as better after some exploration the Q-loss, Figure 6.4c, quickly converges and the policy π returns to the previous better solution. This can be seen as a small bulk in Figure 6.4a at $1.75 \cdot 10^4$ gradient steps. Results from the Entropy $\log \pi$ and α-loss can be seen to converge in Figure 6.4d and 6.4f respectively.

During training an evaluation of the current policy, π, was done after each epoch. In the evaluation stage, only the mean of the policy action is taken to remove any randomness, the results can be seen in Figure 6.5.

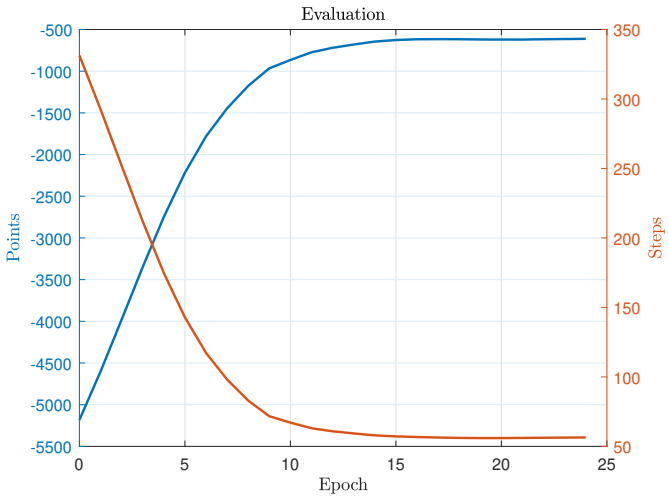
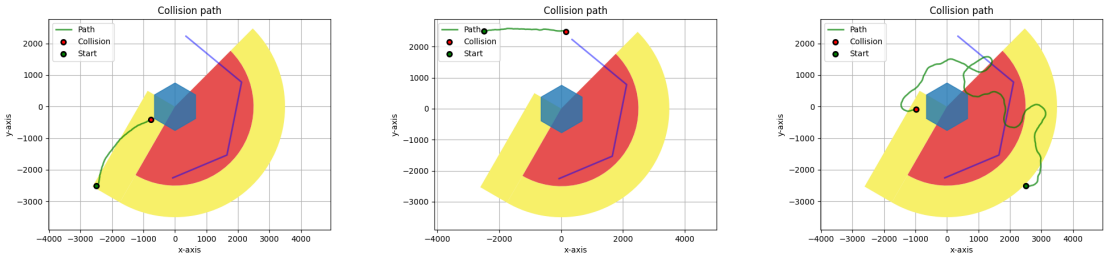


Figure 6.5: Evaluation during training, reward and steps to collision.

After completing training, the workstation is evaluated by placing the agent in an arbitrary position in the environment and letting it find a collision. Figure 6.6 depicts three different cases where the agent's path to collision is illustrated by the green line, the green dot represents the start position and the red dot illustrate where the collision occurs. As seen in Figure 6.6a the agent walks straight to a collision with the robot base. It has found that the robot's counterweight in the back protrudes and will be moving if it approaches it from the safe or yellow zone. This is the shortest way to collision from its starting position since the path of the robot is going through the red zone in the bottom half of the station. The same yields for the second case, Figure 6.6b, where it goes straight to the unprotected area, however if the robot would not have moved in that area at that time, the robot might have found another position for collision. Lastly, from the third plot in Figure 6.6c it can be seen that it seems to try to collide in the unprotected area at first. But either the robot was not close to it at the time or it might have missed the collision.

Therefore, it continues to seek for collision around the robots counterweight as in the first case.



(a) Original start position in bottom left.

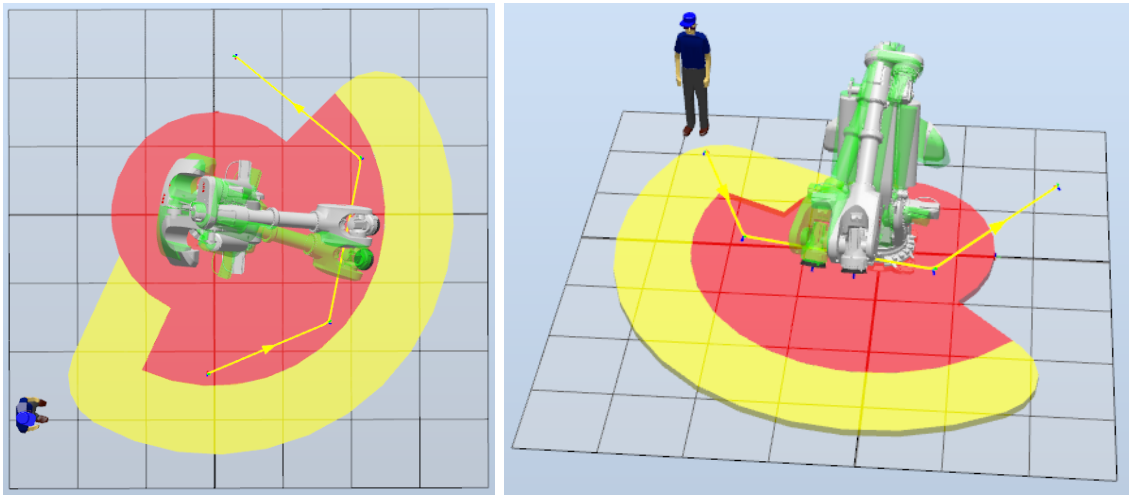
(b) Start position in the upper left corner.

(c) Start position in the bottom right half.

Figure 6.6: Path to collision during evaluation on model 4, illustrated by a green line, from three different positions, shown as green dots.

6.3 Concept Development

The algorithm quickly found a good solution in the previous example and the concept could be verified but as the back piece of the robot constantly was in an unprotected area the task at hand might have been too easy. Therefore, the station was modified by placing a red zone around the base of the robot, thus the agent can no longer easily walk straight into it and gain rewards. This developed environment will be more suitable for the task at hand, where there is less collision space than the previous one, as seen in Figure 6.7.



(a) Top view

(b) Oblique view

Figure 6.7: A more secured environment for model 4 with additional red zone added around the base of the robot.

6.3.1 Redefining Reward Function

One problem with the SAC algorithm that was noticed was that the algorithm has problems with negative rewards and cannot get much information from this due to equation 2.23. As the Q-value shall represent the cumulative value of the state it should become negative if all rewards are negative. The policy loss is calculated based on a positive Q-value and a negative value gives an increasing policy loss, thus it will never converge to a solution as it takes a gradient step in the wrong direction. To compensate for this the reward function was forced to be made purely positive.

Due to the modification in the environment, the number of terminal states becomes rarer, and will therefore take less space in the replay buffer than before. To compensate for this, the positive reward function must be altered to a more continuous manner. The distance d is calculated as the weighted average and a scaling that promotes speed R_v is added to the previous reward function. Mathematically, this can be expressed as

$$r = \frac{1}{2}\tanh\left(\frac{R_v^{1.75}}{d^3 + 0.1}\right) \tag{6.4}$$

and is visualized as a 3D graph in Figure 6.8. The figure illustrates that a small distance is not profitable if the robot is standing still and that the agent should seek to be close to the robot when the robot has high speed. This function has been developed from observations of the environment and has been arbitrarily chosen to get the desired structure.

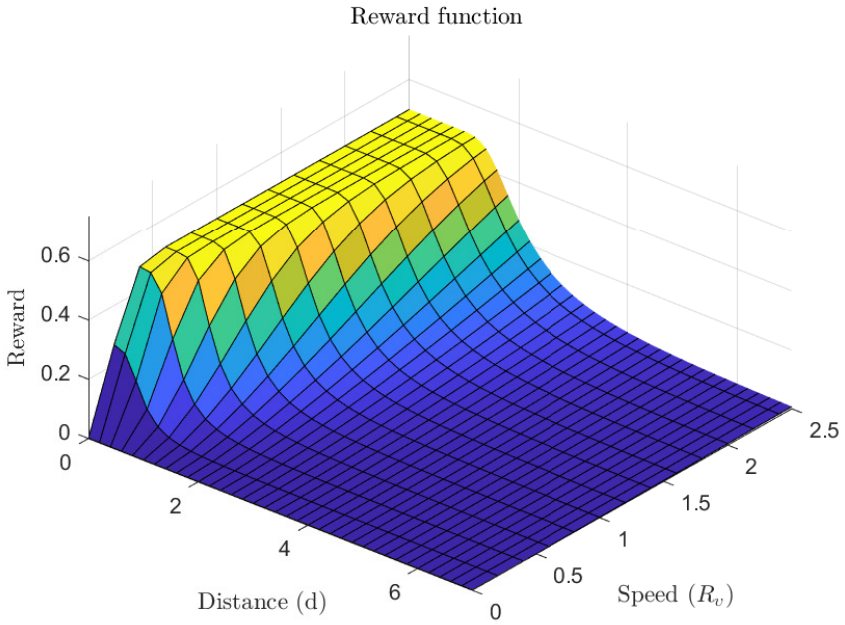


Figure 6.8: 3D visualization of reward function presented in equation 6.4.

Furthermore, the collision reward is calculated to prohibit the agent from avoiding

collisions and collect a large cumulative reward. Therefore, the continuous reward is constrained within the area $[0, 0.5]$, as it simplifies calculations and keeps the reward within a reasonable scale. Given that the reward discount $\gamma = 0.995$ and the maximum reward is $r_t = 0.5$ at each step, the maximum cumulative reward of the visualized reward in Figure 6.8 can be calculated as

$$\begin{aligned} &r_t\gamma^0 + r_{t+1}\gamma^1 + ... = r, \qquad \text{where} \quad r_t = r_{t+n}, \\ &r_t(1 + \gamma + \gamma^2 + ...\gamma^n) = r, \\ &0.5(1 + 0.995 + 0.995^2 + ... + 0.995^n) \approx 99.99. \end{aligned} \tag{6.5}$$

From equation 6.5 it can be verified that a collision reward of 100 will always be the most rewarding decision. In this way, the disadvantage of positive rewards, when the agent can gain more reward by collecting positive rewards than going to the terminal state is removed and the agent will still try to reach the terminal state to gain the largest cumulative reward. As seen in Figure 6.8, the agent will try to optimize this function, which if possible will move it towards a dangerous position where the distance to the robot is small and the robot has a high TCP speed. But the large reward for collisions also makes it possible to find collisions where the situation is not as dangerous but still a threat. And as the reward function now has a clear limit, $r \in [0, 100.5]$, the reward standardization can be removed, as it was made to bound the previously boundless reward. The final reward function for the environment is presented in Algorithm 9, where the distance d is calculated as in the previous reward function.

Algorithm 9 Reward function for modified workstation of model 4.

$J_{p_i} = T_i(J_{\theta_{\{0:i\}}})$ for $i \in \{0, 5\}$
$d_i = i^{1.25}\sqrt{(A_x - J_{p_{ix}})^2 + (A_y - J_{p_{iy}})^2}$ for $i \in \{0, 5\}$
$d = \frac{\sum d_i}{\sum i^{1.25}} 10^{-3}$ for $i \in \{0, 5\}$
$r = \frac{1}{2}\tanh(\frac{R_v^{1.75}}{d^3+0.1})$
if Collision and $R_v > 0$ **then**
  $r = r + 100$
  $terminal = True$
end if

6.3.2 Optimized Forward Kinematics SAC Model

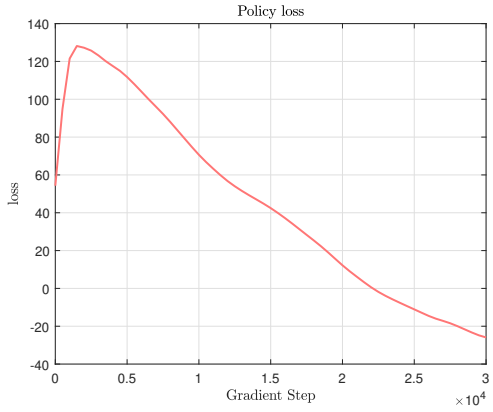
The modified workstation was retrained to evaluate if it could find collisions without easily colliding with the robot's counterweight as in the simplified workstation. Because of the change in the reward function, the algorithm had to be tuned again. The new hyper-parameters used for this case are presented in Table 6.2 and the plots from the training are shown in Figure 6.9.

Table 6.2: Parameters for optimized forward kinematics SAC model.

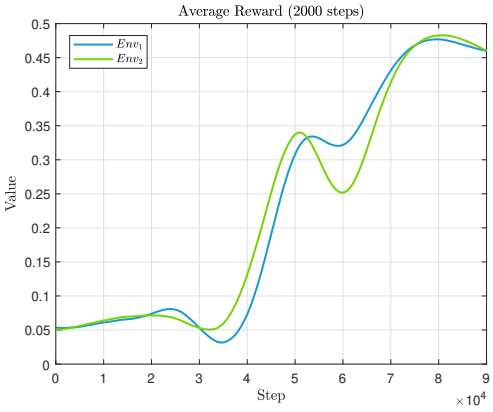
Parameter	Value
α	0.75
γ	0.995
τ	0.005
α_{lr}	$3 \cdot 10^{-4}$
π_{lr}	$3 \cdot 10^{-4}$
Q_{lr}	$3 \cdot 10^{-4}$
n_{env}	2
batch size	1024
random steps	512 steps
e_{steps}	3 steps
Replay buffer	50000
Allocated initial buffer	2500
Epoch size	3000 steps
Hidden layer size	256

Since the reward function was changed from negative to positive values for this model, it can be noticed that the rewards seen in Figure 6.9b are all positive. Furthermore, the reward is seen to gradually increase and eventually both environments converges towards a reward of around $r = 0.45$. Due to a very large collision reward, the Q-loss can be seen to have a quite wavy shape while decreasing and finally converge before it reaches a loss of 2, see Figure 6.9c.

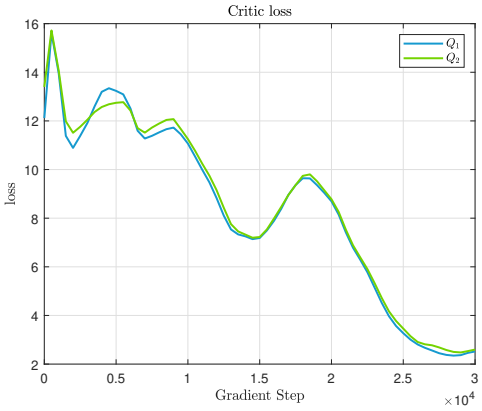
As seen in Figure 6.9e the algorithm starts to explore with a $\alpha = 0.75$ and then it decreases towards 0 but eventually converges at 0.05, thus encouraging exploitation. As in the simplified workstation the π-loss in Figure 6.9a can also be seen with a small "bulk" on the graph at around $1.5 \cdot 10^4$ gradient steps. This is because it thinks it has found an optimal solution because it gains a lot of rewards, seen as the first incremental increase in the reward plot, and therefore it starts to bulk since it wants to converge. But after around $2 \cdot 10^4$ gradient steps the reward starts to increase again and the π-loss starts to flatten and converge again. Lastly, the Entropy log π can be seen to converge towards 2, in Figure 6.9d, as expected as this represents our target entropy and α-loss converge towards 0 in Figure 6.9f as it becomes certain of the exploration/exploitation ratio.



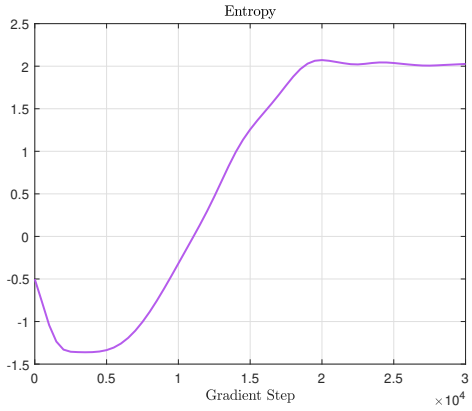
(a) π-loss



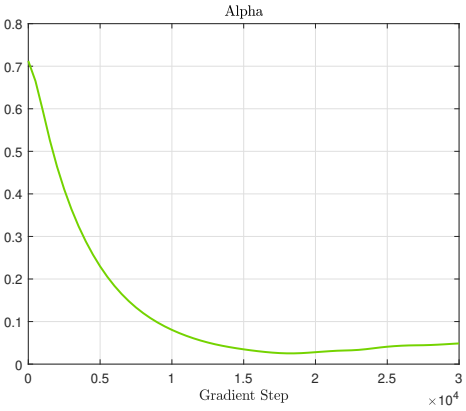
(b) Average reward



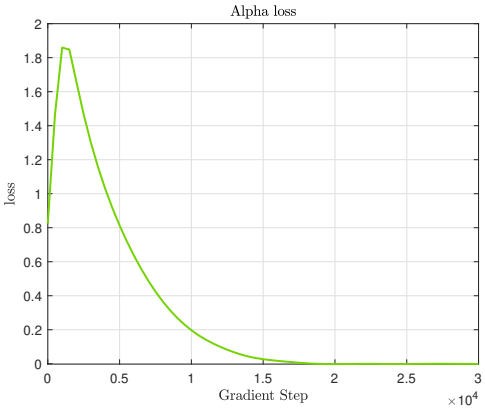
(c) Q-loss



(d) Entropy $\log \pi$



(e) Temperature α



(f) α-loss

Figure 6.9: Data obtained during training from modified workstation of model 4.

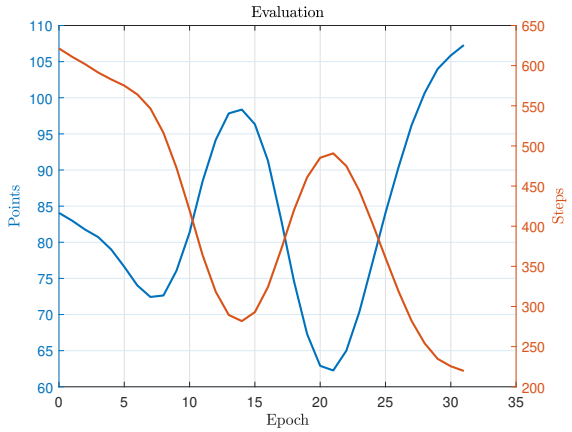


Figure 6.10: Reward and steps to collision from evaluation during training.

The results from evaluation during training can be seen in Figure 6.10, where the model seems to miss collisions between epoch 12 and 20, but then recovers and the reward increases and the steps to collision is decreased towards the end. Despite a more complex model, the solution converged and found a solution relatively fast after 1 hour and 20 minutes in this clear dangerous scenario.

In addition to the evaluation during training in Figure 6.10 an evaluation of the agent's path to a collision was done and the result is shown in Figure 6.11. Here the color of the path indicated the speed of the agent, where 0.05 represents the agent's full speed and 0 when it stands still. Regardless of where the robot is when the agent starts to walk, the agent goes straight to the unprotected area in full speed through the red zone and then it waits for the robot to come to that area and hit the agent. This behavior is exactly what was expected and desired, it does not simply go to the unprotected counterweight but instead the unprotected area where it collides with the TCP. Comparing this behavior to the simplified workstation, this is more stable since it knows exactly where to walk and stands still when positioned in the right position to get hit instead of walking in circles as in the simplified model.

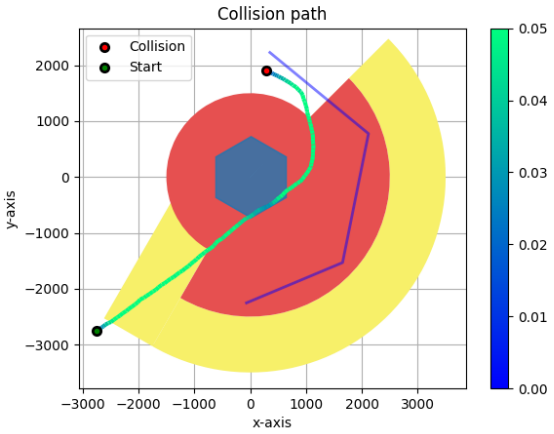
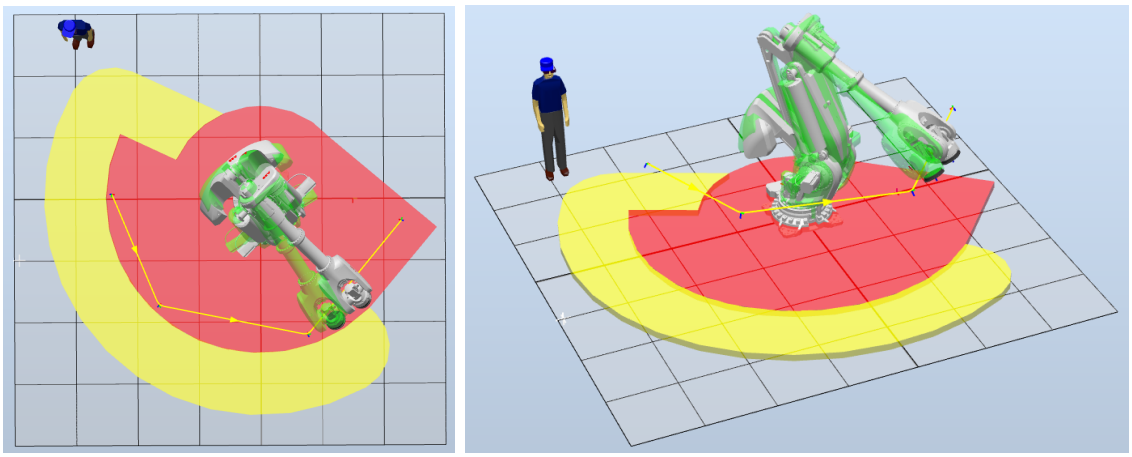


Figure 6.11: Path to collision with the agents speed indicated by color.

6.4 Secured Environment Test

As the previous solution managed to find collisions in a semi secured environment, the environment is further changed to evaluate if the agent can find collisions in a fully secured environment where the obvious weak spot that the previous solution found is removed. This is done by extending the red zone to cover the whole robot path, see Figure 6.12. The modified model is retrained again with the same reward function but with some tuned hyper-parameters. The buffer size was increased to 100000 and as this environment would have a very small amount of collisions they would most likely get lost in the buffer, to solve this each state where a collision occurred was added to the buffer 50 times, so that during training the collisions would get more acknowledgment.

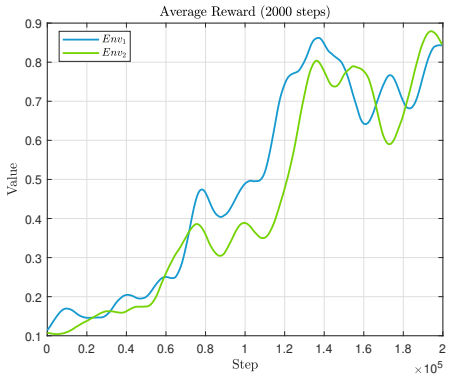


(a) Top view **(b)** Oblique view

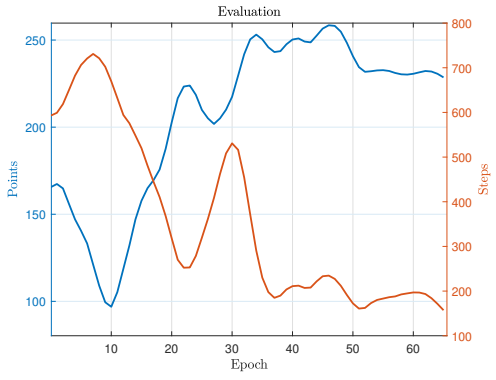
Figure 6.12: Fully secured environment for model 4 with additional red zone added to the whole robot path.

The results from the training and evaluation can be seen in Figure 6.13. As seen in Figure 6.13a the average reward can be seen to increase over time for both environments and also during evaluation in Figure 6.13b. The number of steps before collisions are found during evaluation can be seen to decrease which indicates that the agent finds collision faster over time. Worth mentioning is that this model can be seen to take twice as many epochs to converge to a good solution, thus the harder the collision is the longer training time is needed. It took the algorithm 3 hours to converge to a solution in this more complex case, so complexity of the model heavily affects the training time.

The solution has as previous models been evaluated by arbitrarily positioning the agent in random positions to visualize the path to a collision. This evaluation has been done from four different positions and can be seen in Figure 6.14.

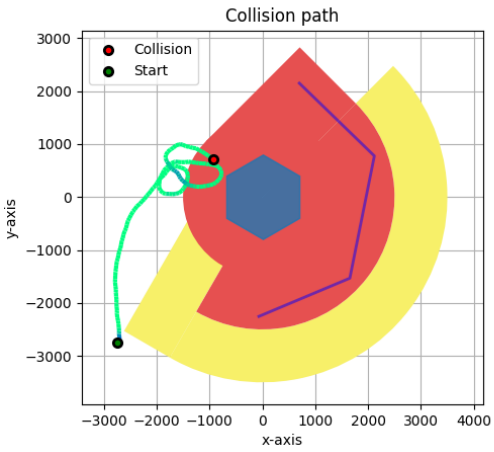


(a) Average reward from two environments during training.

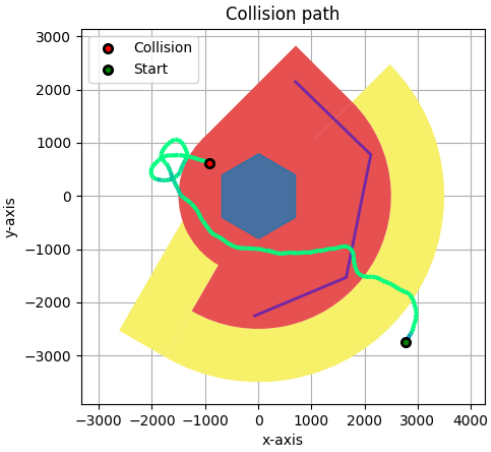


(b) Total reward and steps to collision from evaluation.

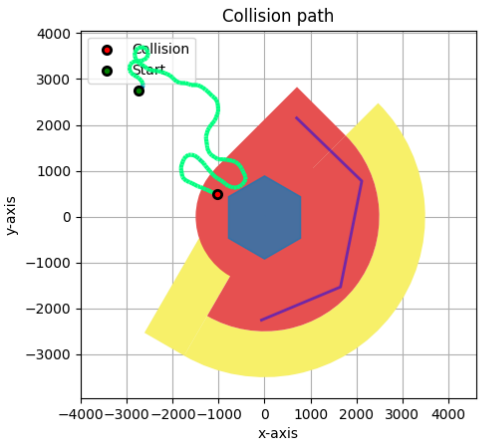
Figure 6.13: Results from training and evaluation of the modified model with a fully secured path.



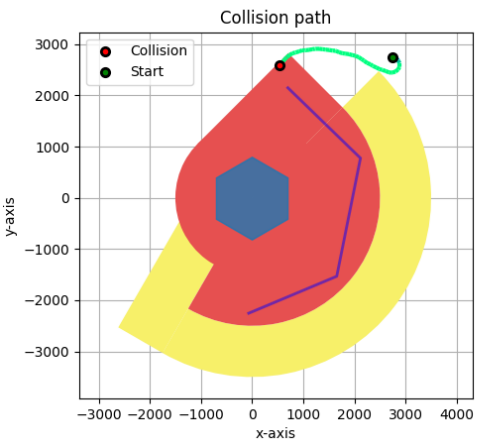
(a) Path 1



(b) Path 2



(c) Path 3



(d) Path 4

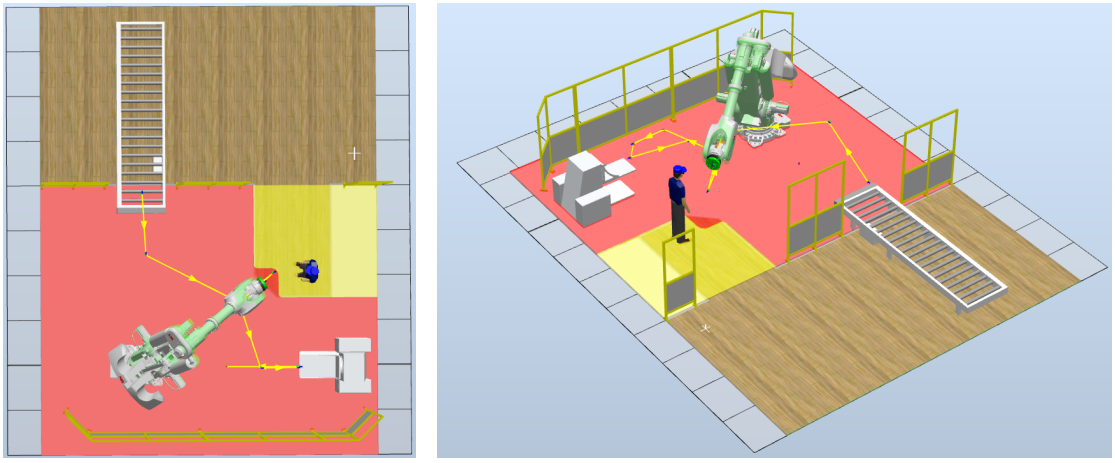
Figure 6.14: Agent's path to collision from four different start positions.

In this concept, there are no obvious weak spots that the agent can get hit on. The agent instead learns to utilize the delay in the system to collide with the robot, as the only criteria is that the robot shall be moving when a collision happens. As seen in Figure 6.14, the agent finds certain locations where it has time to move into the red zone and get hit before the robot has fully stopped. Important to notice is that the collision often occurs with the predicted break position, which is normally not visible in the simulator. Furthermore, it can be seen that in Figure 6.14a, 6.14b and 6.14c the agent finds the same location to collide with the robot's base but in Figure 6.14d the agent is able to collide with the TCP instead. This shows that the agent has found multiple deficiencies in the environment which were not clear to see at first glance.

7

SAC Applied on Real Station

The final environment that the SAC algorithm will be tested on, is a representation of a real work environment. PADME, *Process Automation for Discrete Manufacturing Excellence*, is a project driven by ABB partnered with RISE, MdH och Level21 [33]. The purpose of the project has been to investigate and show how the process industry's highly digitized and proven systems can be used in the manufacturing industry. During the project, a testbed was carried out at ABB's robot manufacturing in Västerås, where process automation was implemented in one of the robot factory's most complex manufacturing cells. In the cell, people work side by side with six robots, eight automatically controlled trucks (AGVs) and other equipment. One of these stations has been the inspiration for the reconstruction of a real station in this thesis. The reconstructed simulation environment in Figure 7.1 can be seen to include one of the robots and AGVs, as well as one of the operators working with the robot in the cell. The robot has the task of retrieving a part from a conveyor, bring it to the operator who can walk on a defined zone to work with the part, and then finally leave the part on the AGV.



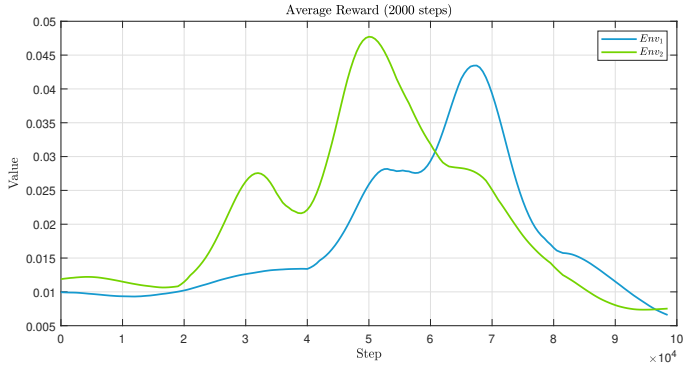
(a) Top View **(b)** Oblique view

Figure 7.1: Reconstruction of real station including AGV, conveyor and fences.

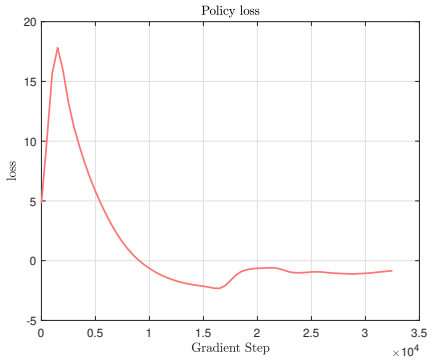
In the physical and simulated station, the agent is only allowed to walk on the wooden floor shown in Figure 7.1, note that the protruded wooden floor is covered by the yellow warning zone. In addition, the physical station is equipped with led lights on the fences that indicate when it is safe for the agent to walk out on the protruding wooden floor and work with the robot. However, since the worst-case

wants to be found, the led light fences are removed and the agent in the simulation can walk on the protruding floor whenever. As seen in the simulated environment, the overlapping area for where the robot and agent can collide is very small. In addition, the robot has a fairly long path outside the wooden floor which minimizes the time the robot spends moving over the wooden floor. To find a collision in this environment where the chance of collision is rare and also very time-limited, will be rather hard and the agent requires good timing to find collisions. For this test, the same reward function that was used last in model 4 in section 6.4 will be used.

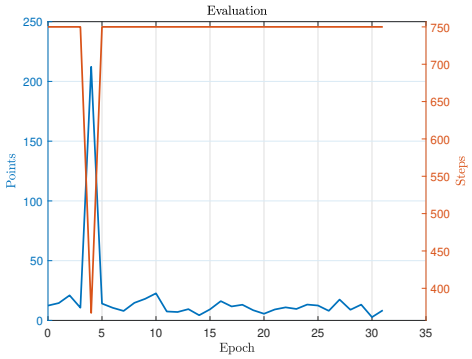
The results from training on the real station with the same parameters as in model 4 are shown in Figure 7.2 It can be seen that the agent is able to find collisions at times, but not frequently enough because of the rare occurrences as previously explained.



(a) Average reward on two environments.



(b) Policy-loss



(c) Evaluation during training.

Figure 7.2: Agent's path to collision from four different start positions.

The reward in Figure 7.2a is seen to initially increase but then settles at almost zeros as the policy loss in Figure 7.2b converges. It can also be seen from the evaluation in Figure 7.2c that the agent converges to a solution where no collision is present. It can be argued that the environment is safe but as collisions were seen during the evaluation and in the peaks of the reward this is not the case. There exist delay collisions but because of the speed of the robot and the size of the workplace, the agent could not learn to find the collisions. Mainly because of the distribution of data, as the valuable data occupies a too small space in the replay buffer.

8
Function Evaluation

The trained models can be seen to work in specific environments with specific set of parameters. An interesting aspect of the developed solutions is now to see how flexible they are by changing several parameters in the environment. The chosen parameters that will be alternated to evaluate the flexibility of the solutions are; the zones, Agent's start position, the robot path and the robot speed. This chapter will describe and show results from each of the different parameters and evaluate if the solution is able to adapt to the new altered environment. The model used will be the optimal solution in section 6.3.2 as this was a model that learned without having to specifically add extra collision data to it, thus it should be less prone to have overfitted on the collision data. The environment that will be altered is that shown in Figure 6.7, as this is an environment that can be altered in several ways.

8.1 Zones

To test how general the trained algorithm is, a red zone is positioned at the unprotected area where the agent collides the most, see Figure 8.1. Consequently, the agent can not collide as it has before and will have to try to change its action according to the changed environment.

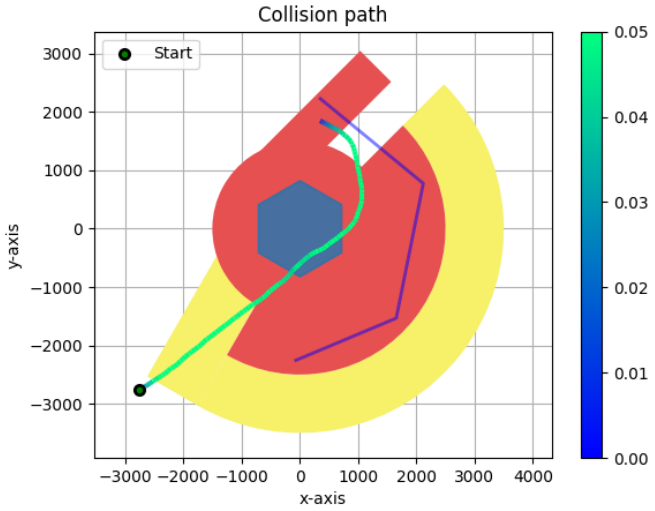
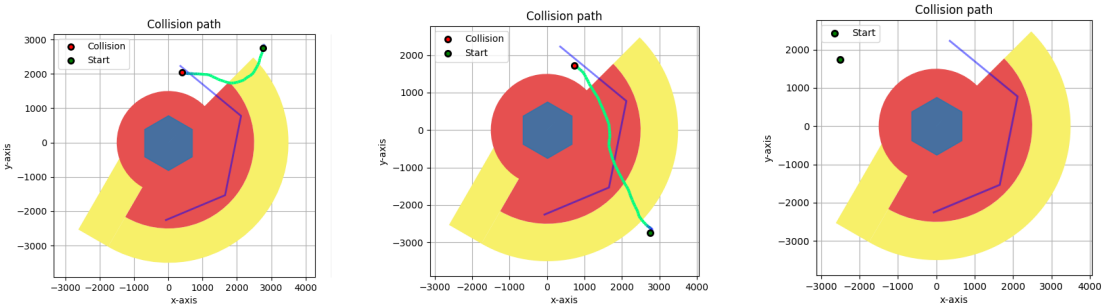


Figure 8.1: Evaluation of the modified workstation where a red zone has been added.

As seen in Figure 8.1 the agent takes the same actions as previously and does not take any initiative to move away from the new red zone which now is a safe place. Thus the agent is not as flexible as wanted, it has not found a connection between zones and the possibility of collisions.

8.2 Start Position

During training, the agent had the same starting position in all models. It is therefore interesting to evaluate if a model has explored enough of the environment in order for the agent to reach the terminal state from other start positions. For this evaluation, the solution has been applied to three different start positions and the results are illustrated in Figure 8.2.



(a) Start position in the upper right half.

(b) Start position in the bottom right corner.

(c) Start position in top left corner.

Figure 8.2: Path to collision during evaluation on model 4, illustrated by a green line, from three different positions, shown as green dots.

The agent did successfully collide from the right top and bottom position shown in Figure 8.2a and 8.2b. But when placed in the upper left corner the agent stood still and could not find a way to collision, see Figure 8.2c. This is most likely because the agent has not been to this area and explored it enough.

8.3 Robot Path

For evaluating how well the solution can adapt to a new path, a new path has been set which goes further away from the robot base than previously in the warning zone. The new path can be seen as a blue line in Figure 8.3. In the figure the agent, as in previous evaluations, follows the same learned path and does not consider the robot's changed route. As a result, it does not successfully collide with the robot, instead, it places itself at its predicted location and stays there waiting for the robot.

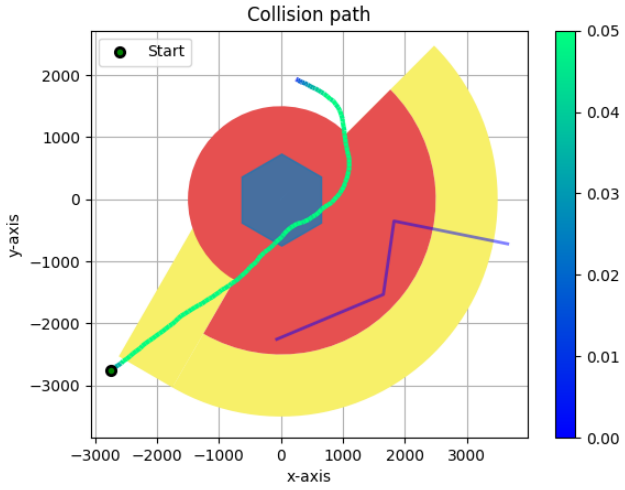


Figure 8.3: Evaluation of the modified workstation where the robot path has been altered.

8.4 Robot Speed

Since the robot has been set to move in maximum speed for all stations, this evaluation will be done by lowering the speed of the robot to 25% of its maximum speed.

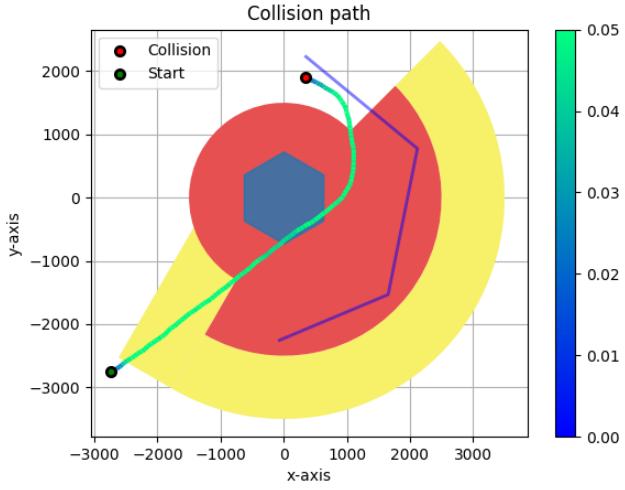
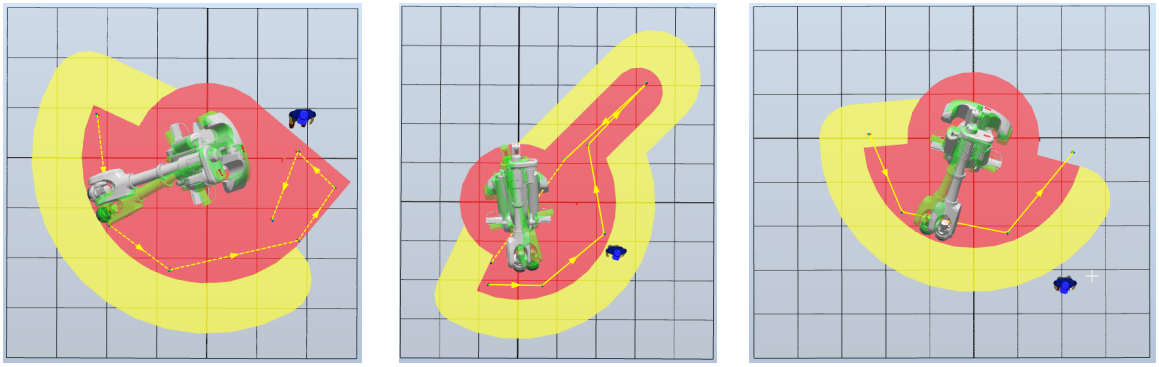


Figure 8.4: Evaluation of the modified workstation where the robot speed has been reduced.

The result of this was no different from when it went at full speed, see Figure 8.4. The agent still successfully collides with the robot although the speed of the robot was changed. This is as expected since the agent clearly has shown to follow a fixed path more than the robot's behavior. It has most likely become overfitted on the available data.

8.5 Flexibility Enhancements

As the solution has been shown to not learn the relation between robot and agent, it is interesting to evaluate if the agent would find the relation when training on different environments. The three environments the agent is trained on now can be seen in Figure 8.5.

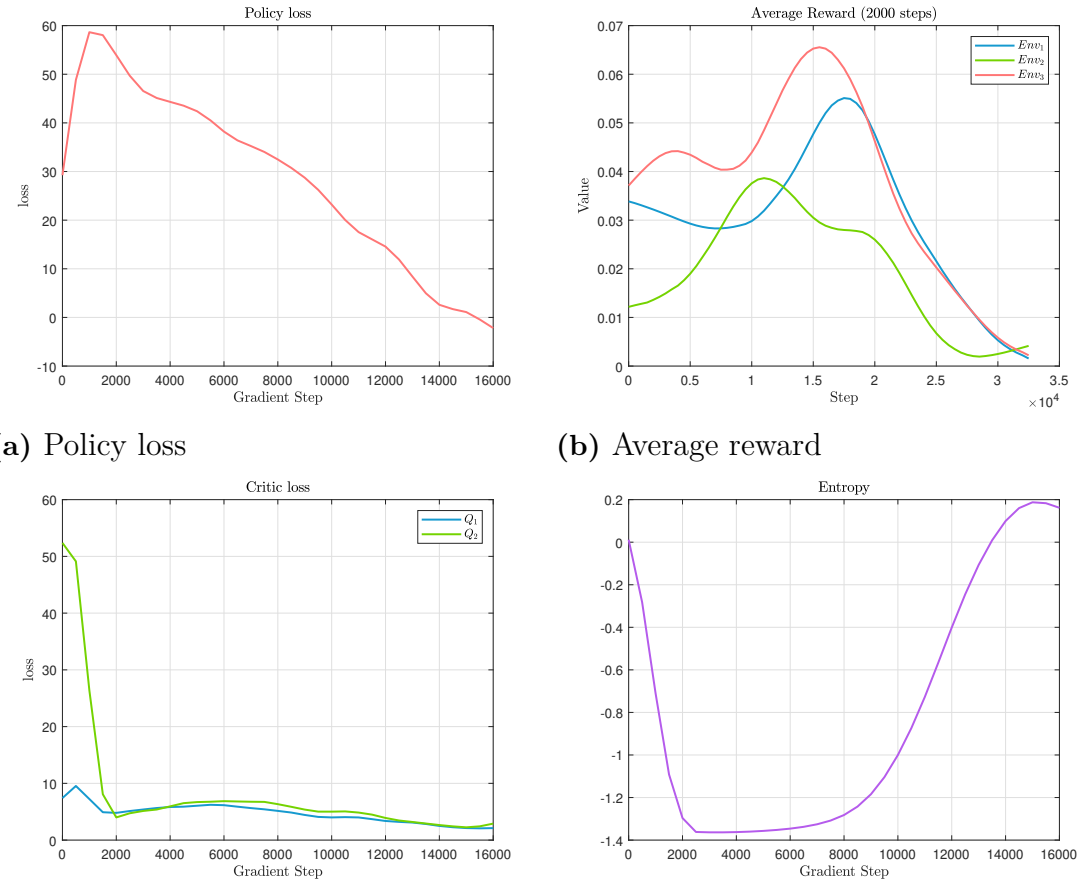


(a) Environment 1 **(b)** Environment 2 **(c)** Environment 3

Figure 8.5: Three different environments used to encourage a more general behaviour.

The three environments seen in Figure 8.5 all have different zones, paths and clear dangerous situations. The agent is trained on these environments in parallel to create a more flexible solution, if possible. The results from training are shown in Figure 8.6.

It can be seen that the average reward increased and then decreases in Figure 8.6b. At the same time the policy becomes more secure and the entropy higher in Figure 8.6a respectively 8.6d, thus it becomes more certain and exploits rather than explore. The initial exploration finds certain positions with rather high rewards, the states often differ between the different environments but some positions give the same states but different rewards. The fact that the same states can give different rewards confuses the agent, this is mainly because of the position states. In the end, it satisfies with a solution that is the best position for all environments, a position that always returns a reward larger than zero in each environment. It chooses to stick with one position where it always gains some points instead of roaming around to high reward situations where the reward differs between the environments. Thus the results show that this model lacks flexibility. It solves specific cases but can not find the relation between the robot position, agent position and zones. This is not surprising. To find a collision, the agent must find a certain location based on the available states. As the states are position and joint angles, it finds a relation between these, thus it finds a relation in a fixed environment where a position is always a certain distance from the robot and the zones. If the environment would change, this relation is lost. The specific distance, dangerous situations that have been found are lost as well and the knowledge it has does not matter anymore. To solve this a new model must be created, a model with states not so dependent on



(a) Policy loss

(b) Average reward

(c) Critic loss

(d) Entropy

Figure 8.6: Results from training on three different environments which are used to encourage a more general behaviour.

position in the workplace but a model dependent on the relation between the agent and robot position relative to the red zone delimiter. The relation to the red zone delimiter creates a global relation because if the agent can not find a collision close or outside of it, it most certainly is impossible to find a collision. And the red zone is always present, thus it should always be possible to navigate base on it, thus it should be able to navigate in any environment as long as the red zone is present.

9

Collaborative Safety

In a workstation where a human works closely with a robot, it is important that the human is and feels safe, both physically but also psychologically. This chapter will discuss factors that keep and makes the human feel safe when working in a robot cell.

9.1 Physical

Stations today utilize different types of safety techniques to reduce the risk in workstations, such as physical fences, lasers and sensors that are for instance pressure sensitive to locate the human's position relative to the robot [34]. The workstation design should follow the International safety standards by the International Organization for Standardization (ISO) which provides guidance for risk assessment to identify hazards, evaluate risks and finding solutions to mitigate risks. Dangerous movements are considered to be dangerous when a human is injured or is close to being injured. There are different types of accidents that can occur with robots; collision accidents, trapping accidents, mechanical part accidents and other accidents. This project focuses on collision accidents and obvious factors that affect physically dangerous movements in collisions are the robot's speed, reach, weight and size. It is for example a higher risk to get injured around a large, heavy robot with a long reach and high speed than the opposite. Although other factors such as the tools or payload of a robot also affect how dangerous the robot is. A smaller robot that has a low speed can quickly become dangerous if it for instance holds a sharp object that can injure humans. Furthermore, a robot with a higher payload will lead to a longer braking distance, which in turn increases the risk of a human being injured during braking.

An easy and common, but space-consuming, solution used today to keep humans safe around industrial robots is to set up physical barriers in form of fences around the robot. When there are no humans in the cell, the robot can work at full speed without the risk of harming anyone. However, usually in these cases, it is not expected that the human should be in the cell and when the robot is working. Therefore, when an human enters the cell, the robot immediately stops or reduces its speed very much. This does not allow any sustainable collaboration between the robot and human in such environments as it either stops working or reduces productivity. To create space for a collaborative environment, the robot can be equipped with speed safety functions that change its speed according to the humans position. Although

a large, low-speed robot around humans does not rule out that humans are not at risk of injury. Also, it is hard to predict brake distance and therefore account for it when programming/designing the safety system. [2]

In the workstations developed in this thesis, the stations have safe, warning and stop zones that prepare the robot to quickly stop when the human is approaching with less braking distance which makes it safer. The station is assumed to be safe, but really it has weak spots and does not follow the ISO. In the weak spot of the simplified workstation, the robot's TCP moves at full speed and hits the human, which in reality will cause a lot of damage to the human. However, when training the algorithm for collision with the whole robot it was noticed that the large counterweight on the back of the robot IRB8700 is a risk of hurting the human. Even though the counterweight does not have the same speed and torque as the TCP when hitting the human, it is still an area where unexpected collisions can occur that needs to be considered.

Although a station can be considered to be safe, the station would most likely not fulfill the ISO rules and regulations, because the current ISO is not updated according to the collaborative requirements and it is hard to set constraints that ensure complete safety when collaborating with large robots like these. This makes it hard to develop collaborative environments that satisfy the ISO.

9.2 Psychological

Today there exists almost no machine-directive that approves collaborative robot cells, mainly because of the physical dangers mentioned earlier but also because of the psychological dangers. The technology to create a collaborative cell exists but more than the technology, trust must exist. The main applicable safety measures are either done within the robot control system or the robot cell to make the hazardous unavailable. If one says that the robot had optimal control and an optimal cell, that would meet all standards, would one be able to work with the robot collaboratively in a safe manner? Physically yes, but mentally no, working with large industrial robots is mentally challenging and several factors have to be considered to make the collaboration healthy. The most important one is trust, if one can not trust the robot the collaboration is not healthy.

In "The Role of Trust in Human-Robot Interaction" [35] there are five system properties that affect the collaborative trust from the system side, *System reliability*, *System faults*, *System predictability*, *System intelligibility and transparency* and *Level of Automation*. In *System reliability* and *System faults* the robot in question does not behave as expected, such as wrongfully estimating a position or start to move when not supposed to, these errors are severe and the more they happen the more trust is lost from the user. *System predictability* represents the uncertainty of the system, when the robot behaves unpredictably the trust is lost fast, this includes faults. It is also stated that if a certain fault is common and thus very predictable the trust is gained again, if the error is expected the user does not worry as much about it.

System intelligibility and transparency is the way of explaining one's actions and knowledge to the user, a very important aspect. The more knowledge about the robot's actions and the reasoning behind them the more trust the robot is given, this also relates to human interaction as if one knows the reasoning behind one's actions the person is more trusted if the reasoning is logical to the other. *Level of Automation* states that the more autonomous a system is the complexion becomes very opaque to the other end which lessens the understanding of actions, this in its turns affects the *System intelligibility and transparency*, thus the level of automation may be a problem but it is a solvable problem through *System intelligibility and transparency*.

With this it can be seen that this solution is a tool that should guarantee safe zones where the robot should not be able to hit the operator, the tool is based on *millimeter* precision which is possible with today's Lidar systems and correctly estimated break positions, which are both trustworthy. The reliability of the system is in jeopardy as the evaluation is done in a simulator, the simulator tends to differ from the real-life application. Furthermore, the implemented model has been shown to give different results during training, some more stable than others and this instability during training makes the function unreliable. The system is seen as trustworthy within *System faults* as this is based on the gathered data but the *System reliability* is not trustworthy because of the instability during training and the real robot cell versus simulator difference.

As the dangerous zones are based on reinforcement learning the agent needs to explore as much as possible, if not given enough time to explore, the agent might miss some of these zones. Furthermore, the function is very hyper-parameter sensitive and hence it can not always guarantee a correct result as the parameters must be tuned to gain an optimal solution for that specific environment. Therefore, the system is not seen as trustworthy within *System predictability* as it is too dependent on if used in the correct way and the cell integrators knowledge. As the agent clearly shows the path to the dangerous situation the *System intelligibility and transparency* is very good, as the user itself can determine if this is a danger or not. But on site the robot cell might need to visually show where the human is estimated to be in comparison to the safe zones, thus visualizing if the robot will start or not. The *Level of Automation* in this case is not that high as it mostly is a function that verifies the cell designer's layout of the safety system, thus it is a two-way verification system to guarantee safety.

Furthermore, there are several other applications that need to be done after the work of our function, the most important is the *System intelligibility and transparency*, as the more the operator understands the actions and behavior of the robot the more trust is given, by knowing the behavior it is in turn both more *predictable* and *reliable*. Thus trust is the key to a healthy collaborative relationship.

10
Discussion

This chapter will discuss the results from the developed function, evaluation and also important key factors that can be further developed in the future.

10.1 Environment

The replica of the real station is not a fair representation of the real world station and the simplified environments are very simple and do not take into account all aspects of a station. First of all, the simple environment has an unrealistic design, where it has a very obvious spot where it is exposed to hazards. This design was done on purpose to make sure that the algorithm could find the obvious danger. Secondly, when determining if the robot is moving, the velocity of the TCP was used since logically if one joint is moving it is likely that that TCP also moves and has a speed. However, it is still physically possible that the TCP has a speed of zero and the other joint are moving but is omitted in these models since those types of movements are not used. In the replica of the real station, the path is roughly set by watching videos of a real station and an arbitrary speed was set, thus details may be left out.

10.2 Discrete Models

In the first two discrete models, where Q-learning and DQN were used respectively, the environment was fairly simple and the task was not very complex. As expected in these simple models, the agent managed to solve its task and find the obvious collision spots rather fast. However, due to limited time, the slow yellow zone in model 2 was not active. Therefore, the robot moved at full speed when the agent entered the slow yellow zone. This may be one reason why it was easier for the agent to get hit in this environment since the robot had quite high speed when the agent approached it.

Furthermore, in the early stages of development for model 2, the agent was constrained to not take a calculated step if it would end up at a position where the robot was located at. Because this would not be a realistic behavior to walk straight into the robot and therefore the possibility for the agent to do so was removed. As a consequence, the function approximation never reached or experienced these states and could not approximate the value of them. In some cases, these states were very highly valued because the agent had not experienced any penalties when interacting

with these states before. This lead to that the agent tried to reach these states instead of the goal defined by the reward function and these unreachable states became more valuable. For this reason, the agent should never be constrained to avoid bad behavior or states, it must be exposed to both. If a certain behavior wants to be avoided, it must be stated in the reward function and not in any way that prevents the agent from learning. By experience, it will eventually learn to avoid these states by itself.

10.3 Reward Design

In each of the presented models, the reward function has been designed with the intention of finding a relationship that makes the agent find the dangerous situation rather than a certain position. Usually, RL applications are case-specific, which means that the reward function and hyper-parameters only work in the intended model. It has been proven to be very difficult to create a reward function that can be applicable in many different environments and in real life since the model often learns case-specific scenarios [12]. Without exception, this has been a challenge in this project as well. Despite attempts to create a general reward function, the results from section 8 show that the model has been trained case-specific anyway. The reason for this is mainly because the data it has been trained on has been so case-specific and not necessarily an error in the reward function itself. The problems faced throughout this thesis have forced the model to work with less amount of data in a certain environment and with positive rewards. The positive reward forced the distance reward to be less important thus making it harder for the agent to extract knowledge from it, as the collision reward had to be larger than the cumulative distance reward.

When developing the concept in model 4, it was chosen to use positive rewards instead of negatives and a problem with this is that the agent does not mind taking longer time than necessary to reach the terminal state. This is because the positive rewards result in a larger cumulative reward when taking a long time as it gains a small positive reward for every step. In contrast, the use of negative rewards will make the agent want to find the terminal state as quickly as possible to avoid collecting more negative rewards and thereby maximizing the cumulative reward. Hence, negative rewards would be preferable to use but unfortunately, it did not work in this case because of problems with the policy loss.

As for now, the reward function tries to maximize the reward based on distance and collision, it will therefore not always try to engage to the first obvious collision and might delay the collision to collect the distance reward. This means that certain clear collision sites might be ignored as it rather collects the distance reward and engages to a collision when it no longer can collect the high distance reward. In this way, it only converges to one or two collision sites in the environment and not more. Other collisions can not be found unless solving these cases and retraining the model. Hopefully, this could have been solved with negative rewards as it would always try to find the closest collision to stop the negative cumulative reward.

10.4 SAC and Negative Rewards

As mentioned in section 6.3 it was noticed that negative rewards became a problem for the algorithm. What is interesting is that negative rewards were used in the SAC reward functions in both Algorithm 6 and Algorithm 8. However, they were not completely negative, they had some very high scaled rewards for collision compared to the distance penalty. As the negative reward did nothing for the agent to find a solution, the agent instead tried to approach the single state where it got a positive reward, a collision. What was noticed during the training of these versions was its instability. They often failed or diverged, this was because of the negative rewards and the size of gamma. As gamma was set to 0.99, the actor did not see far enough to estimate a positive Q-value and as it became negative, the policy diverged. Despite this, the agent did successfully find a solution as seen in section 6.2. This was because the collision was close enough to the starting position for the agent to have the collision in sight so that the Q-value could be estimated at a positive value. This problem was not solved until in model 4, section 6.3. Here a fully negative implementation was first introduced based on speed and distance which did not work. The error may have been found to be in equation 2.23, when α converges towards zero as

$$J_\pi(\phi) = \mathbb{E}_{\mathbf{s}_t \sim \mathcal{D}}\Big[\mathbb{E}_{\mathbf{a}_t \sim \pi_\phi}\Big[\underbrace{\alpha \log \pi_\phi\left(\mathbf{a}_t|\mathbf{s}_t\right)}_{\to 0} - Q_\theta\left(\mathbf{s}_t, \mathbf{a}_t\right)\Big]\Big], \tag{10.1}$$

which it usually does as it becomes more confident in its policy, see for instance Figure 6.9e. As α goes to zero the policy loss becomes solely dependent on Q_θ and as Q_θ is negative the policy loss would take a step in the wrong direction, as the negative value is subtracted, and this is what causes divergence when using negative rewards. Thus, a solution for this would be to try another policy-loss function between the entropy/exploration and the estimated Q-value. Or a solution as in this thesis where it is guaranteed that the terminal state has the largest reward, thus it would like to approach it anyway, as it is more profitable than just hanging around gathering an accumulative reward. It could be that this implementation of the SAC algorithm is faulty and thus this error occurs. But in the original SAC paper, all benchmarks were done on environments with rewards in the range $[0, 1]$, thus only positive and could not be compared with [25]. If the negative reward had been solved the reward itself would instead be purely negative, to try to force it to a terminal state, a collision. The collision reward could then be removed and the remaining reward, visualized in Figure 6.8, shifted to the span $[-1, 0]$.

10.5 Replay Buffer

In model 3 and model 4, divergence was experienced during training after approaching the optimal solution. The first suspicion was that the reason for divergence was due to lack of good collisions in the replay buffer. Because during training, these states became a smaller part of the buffer for each step and eventually it diverged to another solution, as the Q-value was wrongfully estimated. There were two ideas

to fix this problem, a prioritized replay buffer that prioritizes the less known values or to simply remove the environment reset after a detected collision. The latter one was implemented and tested, by removing the environment reset several collision states were added to the replay buffer instead of just a single one in the event of a collision. This solution initially gave positive results as it gained a lot of knowledge about where the good states were, but it came with its disadvantages. Because the environment was not reset after each collision, the agent tried to find a path from one collision to another instead of returning the agent to its start position and finding a path from there. Consequently, the agent essentially learned to stand still in a collision because it was the most rewarding action at the time. This led to that the path to a collision from the start position was trained away and forgotten, and instead the behavior of standing in a collision was learned. By forgetting the path to a collision, it stood still at the start position and from this point, it diverged again. By focusing too much on finding collisions, all knowledge about how the agent got to the situation were lost. The ultimate goal should therefore not always be the highest priority as the path to the goal can get lost along the way.

For model 4 in section 6, a smaller replay buffer was used as the solution to stop divergence, as it should make it easier for the critic to converge to a solution. The smaller replay buffer made the actor almost converge to an optimal solution, but when it almost had converged the replay buffer contained almost solely states of the same path to the collision. At this point *catastrophic forgetting* occurred, *catastrophic forgetting* is when the network overfits to states and actions only seen in the optimal policy. But at some point, the agent will explore states where its predictions are way off. Because Q-functions also use their own predictions to bootstrap new Q-values, seen in equation 2.22, it can start a runaway feedback process where the agent starts to choose a sub-optimal path through the environment. Eventually, it diverges from the optimal path, this is one part of the deadly triad. A solution to this was to fill 5% of the replay buffer with states from the beginning, thus a portion of different states are always present during training to keep it from overfitting on the optimal policy. This solution did delay the divergence but it did not remove it, to fully remove this problem the allocated space with initial random states could be randomized as the training goes, thus the network would not be able to overfit at these states too. Having a constantly changing random part of previous states could help the training but where these states would come from and how to determine which are different from the optimal policy to keep the divergence away is a problem on its own.

10.6 The Deadly Triad

The sudden divergence after finding the optimal solution was still an inevitable problem until the end of the project and the reason for this may be due to the *Deadly Triad* in RL. The Deadly Triad is known for causing divergence during learning when combining function approximation, bootstrapping and off-policy, which has been theoretically proven [36]. By using bootstrapping, the Q-value can be estimated by using the estimation of the next state-action pair. The off-policy method

enables to update the policy by using the best Q-value of the next state. As the environments in models 3 and 4 are continuous and the number of states is too many to create a value function for each state or state-action pair, function approximation is used to compute the Q-values. The problem with this when the Q-value is updated with bootstrapping from two similar states s, s' and similar actions a, a' the neural network can get confused in a way that essentially implies that the Q-values for the two states are approximately the same, $Q(s, a) \approx Q(s', a')$. This eventually leads to divergence and is commonly known as *catastrophic forgetting*.

The authors in [36] investigate the reason why deadly triad occurs. A large experiment was made on the Atari game where the following parameters were changed: bootstrap target, the number of steps before bootstrapping, level of prioritization and network size. From this experiment, several conclusions were made. First, it was observed that no unbounded divergence where values become NaN occurred. However, soft-divergence occurred where values got unrealistic high. Secondly, it was concluded that that bootstrapping on separate networks and correction of overestimation bias will lead to less divergence. It will get harder for the function to diverge if using longer multi-steps. Furthermore, the size of a network can affect divergence since larger networks exhibit more instabilities than smaller networks. However, this can be balanced in other ways, such as introducing a more stable update rule. Lastly, it will more easily diverge with stronger prioritization of updates.

10.7 Function Flexibility

In section 8 the chosen model was evaluated in several ways and the solution was shown to not be very flexible. It was not able to handle change in paths or zones and has therefore clearly not found a relationship between the available states. The solution simply found a dangerous position instead of learning a dangerous behavior. Furthermore, the function is very hyper-parameter sensitive between different layouts. As the policy is trained on one single layout it is expected that it finds the solution for that layout only, thus it is assumed that the solution is overfitted on the available data. And as the replay buffer is very low it will after a while only experience the optimal states, thus these will be heavily weighted and no other actions at these states would be considered. As tested in section 8.5, the actor could, during training, instead be exposed to several different environments, which the general reward function would work better on. This would force the actor to find connections between the states and not just positions. This would encourage a more clever agent, an agent that takes decisions based on knowledge, tries to find positions where the robot moves fast and close to the agent and where the agent might be in harm's way.

To find dangerous positions/situations with the current states the agent had to relate to a certain position in the current environment in one way or another. Thus, the algorithm is constrained from any possible general knowledge by the fact that it has to locate specific dangerous situations in that workplace. To solve the general

problem, more general states that apply to the zones are needed. Then, the current implemented SAC algorithm must be altered to better handle large replay buffers. So that the agent could train on many different combinations of layouts, states and collisions without diverging. It is not guaranteed that this algorithm could handle this but it has the potential to.

10.8 Collaborative Safety

As mentioned in chapter 9, *Collaborative Safety* the safety shall be considered both physically and psychologically for humans. When simulating the environments in this thesis a lot of focus was put to where the agent could be hit relative to the robot's TCP. However, when using the whole robot to collide with, it was noticed that not only the TCP hurt the agent, but also the counterweight at the back of the robot was an unexpected place to collide with the agent that should be considered when designing robot cells. Therefore, it is important to focus on the whole robot rather than only the TCP when doing a risk assessment of a station. As previously discussed, the developed function needs further work before it can be useful, since it is very unstable and unreliable now. After a relation between states has been found, the function has become more stable and can be applied to a more general station to find unexpected dangerous spots, possible safety margins can be reduced in workstations. Other manual tests that are done today could then be eliminated in the future. Furthermore, a general solution would be very time-saving as the training would not be needed as often or in the optimal case only once.

Even if a station would be physically safe to work in for an agent, with smaller safety margins and higher speed of the robot, it should not be forgotten to also keep it safe in a psychological way for the agent. It is important that the agent can trust the robot that it works with for healthy collaboration. Today, a lot of restrictions and rules are set which would make it hard for the agent to trust the robot if safety margins would be reduced since the rules do not even allow it. Thus, the question occurs, how can a person trust it if the rules do not? This solution is based on a simulation and hence the simulation versus real-life behavior must be considered. To trust something only tested in simulations does not create great trust, but there are possibilities to fix this.

At site, the function could visualize the most dangerous paths from the current position. Giving a clear image that it knows where the human is, the worst-case scenario and that there is no danger there. The case where it does not find any solution at all is the more complex one, as how to visualize the danger is rather hard. One way of doing this is by creating a heatmap of the Q-values as the Q-values represents the values of each state, and the value is dependent of the danger. If no collision is present it will converge to a solution where the most valuable states are the ones close to it and from our reward function. One can estimate how far away these are based on the value of the state. A heatmap that shows the Q-values in reference to the distance and TCP speed in the reward function can show how *dangerous* the paths and location with the optimal actions are. If they all have a low

score we can see that there are no locations with a clear danger, which can easily be visualized on a heatmap of the environment. This becomes a clear tool that can show the user where this algorithm thinks the most dangerous paths and locations are.

But this tool is not trustworthy enough to be solely dependent on at this time, further testing and manual verification are needed when designing robot cells. This is a tool to help the designer find faults caused by human error, creating a safer work environment, a more sustainable work environment and should be a general aid in the design phase. The values of each state does not at this time truly show the dangers, but can hopefully do in the future.

10.9 Future work

There are a few improvements that can be made to this solution that was not done in this thesis due to lack of time. One improvement would be to weigh the speed of each joint when calculating the distance between the agent and robot, as suggested in Algorithm 10. This would make the agent aware of which joint has the most movements and highest velocity and thereby have a higher probability to collide with. Currently, the TCP is heavily weighted in the current solution because of the simple movement in the environments.

Algorithm 10 Reward suggestion

$J_{p_i} = T_i(J_{\theta_{\{0:i\}}}) \quad$ for $i \in \{0, 5\}$
$d_i = J_{v_i}\sqrt{(A_x - J_{p_{ix}})^2 + (A_y - J_{p_{iy}})^2} \quad$ for $i \in \{0, 5\}$
$d = \frac{\sum d_i}{\sum J_{v_i}} 10^{-3}$ for $i \in \{0, 5\}$
$r = \frac{1}{2}\tanh(\frac{R_v^{1.75}}{d^3+0.1})$
if Collision and $R_v > 0$ **then**
  $r = r + 100$
  $terminal = True$
end if

One idea that was raised late in the project was that the solution's flexibility might depend on the position values included in the states. Instead of using the position, one could use lengths from the agent to each joint of the robot and its angles. But this in turn would still relate to a specific position relative to the robot and thus the robot would train to find these positions. Nevertheless, this leads to the introduction of a new concept that is more generalized and would better suit the general case that could be implemented in the future.

Instead of using positions and angles relative to the origin/robot-base, euclidean distances to the intersection of the path between robot/agent and the red/yellow zone can be used, seen as the crossing of the purple dashed line and the red zone delimiter in Figure 10.1. Thus, instead of the relation between robot and agent, a

relation between the length from robot to zone delimiter and agent to zone delimiter could be found. The same reward function can almost be used but instead of the agent trying to always find positions close to the robot, it will try to find positions close to one another in relation to a distance from the zone delimiter, which is the place where the most dangerous positions are.

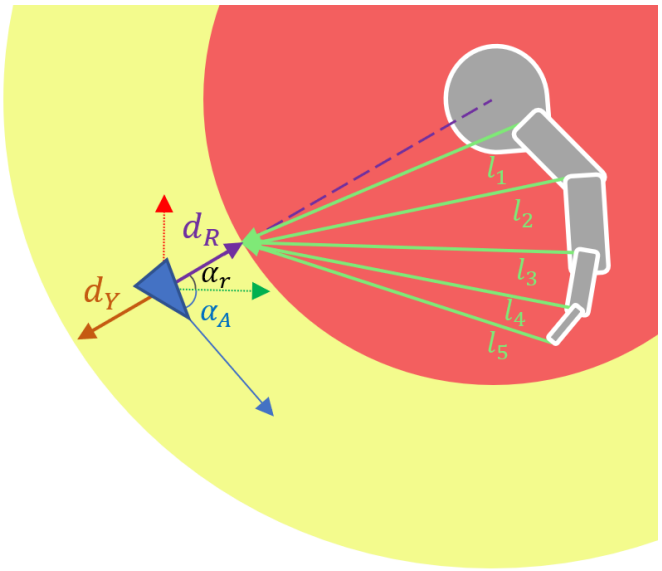


Figure 10.1: States visualized for a future concept model. The blue arrow represents the agent and the grey joints illustrates the robot with five degrees of freedom.

The state space for this concept would then contain the length to the red zone limit in angle with the robot base, d_R, and the length to the yellow zone limit in angle with the robot base, d_Y. Furthermore, the agents angle in the plane, α_A, is included to enable the agent to connect actions with rotations in the plane. The angle to the red zone delimiter, α_r, to find further connections between α_A and actions taken, a landmark in the plane is also included. Finally, the state space contains lengths between each axis and the red zone delimiter, l_x and joint values J_x as in model 4. As a results, the state space is now defined as

$$\mathcal{S} = \begin{Bmatrix} d_Y & d_R & \alpha_A & \alpha_R & v_A & v_R \\ J_1 & J_2 & J_3 & J_4 & J_5 & J_6 \\ l_1 & l_2 & l_3 & l_4 & l_5 & l_6 \end{Bmatrix} \tag{10.2}$$

and the action space remains the same as in model 4 as

$$\mathcal{A} = \{\Delta A_\theta, \Delta A_v\} \qquad \text{where } \Delta A_\theta \in [-20°, 20°],\ \Delta A_v \in [-2, 5]. \tag{10.3}$$

It can be noticed that the zone states used in model 4 are no longer present in the state space, instead, the zone delimiter lengths, d_R and d_Y contains this information. A negative value represents a presence within the zone. The idea is that if this model is trained on several different environments or even on the same environment, it should in theory be able to find a connection between position, length towards safety zones and lengths from robot to safety zones which are factors that are not so

specific for environments. The dangerous situation becomes defined by the agents and robots behavior around the stop delimiter and not purely around fixed positions in the plane. The delimiters are changing in each environment while the positions are fixed, thus there is no hard-coded solution and it becomes a relation between states instead. The drawback is that the delimiter position must be found, thus a smart component in RobotStudio is needed to extract all available data about the red and yellow delimiters, which is much work as it is not available as of today.

During the process of this thesis, a new version of SAC, called Averaged-SAC, was published by F. Ding et al. [37]. As earlier mentioned, SAC suffers from poor stability and insecurity during training. According to the authors, the problem of divergence in SAC comes from that the soft Q-value has an overestimation problem due to the maximization operation used to get the Q-value close to the optimization target as quickly as possible. The proposed algorithm solves the overestimation problem by taking the average of K number of previously known Q-value estimates to calculate the soft Q-value as

$$\hat{Q}\left(\mathbf{s}_t, \mathbf{a}_t\right) = r\left(\mathbf{s}_t, \mathbf{a}_t\right) + \gamma \mathbb{E}_{\mathbf{s}_{t+1} \sim p}\left[\frac{1}{K} \sum_{K=1}^{K} V_{\bar{\theta}}\left(\mathbf{s}_{t+1}\right)\right]. \tag{10.4}$$

This reduces the overestimation error and provides a more stable training process. The conducted experimental tests in the paper show that increasing the number of previously learning states K results in better performance up to a value when the performance starts to decrease. Overall, this method has been shown to improve performance on SAC and could be an extension to add to the solution developed in this thesis in the future to improve its performance.

10.10 Advantages

Most of the dangerous situations found through this thesis have been situations where the robot or its break position left the stop zone. Thus, the task faced could be simplified by just examining the break position at each possible time step and see when it leaves the red stop zone. This would give all the dangerous situations when the robot leaves the area. But with our solution, one can include the safety delay of the security system, which makes it possible for our agent to walk into the red zone and get hit. This was often seen during training, that the agent found positions where it got hit inside the red zone but eventually these disappeared as the agent found more predictable collisions. Hence, this solution creates a more human-like testing, not only seeing if the robot moves outside the safe zone but to truly test both positioning, delays and overall the state of the security system. This enables better testing but is very dependent on good simulations, as RobotStudio delivers.

The new future concept discussed earlier that is based on lengths from the stop delimiter, could have further taken advantage of these delays, as it would have the knowledge of the minimum distance needed between the axis and stop delimiter for a collision to happen before the robot has stopped.

11
Conclusion

The objective of this thesis was to falsify a robots safety system to find unexpected collisions in industrial environments by developing a RL function. The purpose was to provide integrators an aid tool to facilitate when designing robot installations, specifically collaborative ones. The tool could then be used to verify the safety system, reduce safety margins and speed up the robot to more effectively use the factory premises and increase productivity in factories without sacrificing human safety. This was done by simulating simple workstation environments in RobotStudio, implementing RL algorithms that to begin with were simple and discrete and finally lead up to a continuous SAC algorithm.

The function developed in this thesis using RL has been shown to be able to find both hidden and obvious collisions in simplified environments. Currently, it can not find the relation between states and can therefore mainly be used on a specific problem that it has been trained on. In order to use it on other similar workstations, the hyper-parameters in the algorithm requires to be re-tuned for each task and environment. This was mainly due to the state's relation to a collision, the agent only had data available which forced it to determine collisions mainly on positions, thus it could not relate to changes in the environment. Even if it does not find a collision, the Q-values can be used to visualise where the algorithm finds the most dangerous paths or locations. This could be used to find potential dangers that are not yet collisions or visualize that there are no dangers at all. Other factors as reward shaping, the deadly triad and lack of solutions to the SAC algorithm were also reasons why the function was not able to find the relation and be a more generic solution.

From the thesis, it was concluded that due to restrictions and laws it can be hard to remove current safety margins in practice to enable a more collaborative environment. Although, theoretically it is possible but is also about the operator trusting the robot. As for now, the algorithm is not stable enough to produce a trustworthy result as it often diverges during training and forgets important data. If it finds a solution it usually only finds the one or two most valuable collision sites and not all of them. To find all of them, the collisions sites must be secured and the algorithm applied again to find new or other collisions, which would become a very time consuming process. Therefore, it needs further work to be used on real environments as a general tool.

Many tests and experiments were left out in this thesis due to lack of time, and

training DRL algorithms is very time-consuming. It could take hours before a decision of how good or bad the algorithm is can be made. The algorithm still suffers from divergence which seems to be a common problem in RL and it would be interesting to further investigate this in future work. Furthermore, a new general concept has been suggested which could solve many of the difficulties faced through this development process. The concept is inspired by the experience gained throughout the testing and evaluation of this current solution where states are based on a global relation and the relation to the stop zone delimiter is used rather than only the robot agent relation.

Bibliography

[1] ABB Robotics, *Application manual: Functional safety and SafeMove2*, 2021. DOI: 3HAC052610-001.

[2] E. Jernheden and D. Berndtson, "Construction and commissioning of an industrial robot system according to the machinery directive," Department of Electrical Engineering, Chalmers University of Technology, Gothenburg, Tech. Rep., 2017.

[3] ABB Robotics, *Product specification IRB 8700*, 2020. DOI: 3HAC052852-001. [Online]. Available: https://search.abb.com/library/Download.aspx?DocumentID=3HAC052852-001&LanguageCode=en&DocumentPartId=&Action=Launch.

[4] J. J. Craig, *Introduction to Robotics: Mechanics and Control*, 3rd Edition. Pearson Prentice Hall, 2009, ISBN: 9780201543612.

[5] Ollydbg, *File:DHParameter.png*, 2010. [Online]. Available: https://commons.wikimedia.org/wiki/File:DHParameter.png (visited on 06/10/2021).

[6] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo, *Robotics*, ser. Advanced Textbooks in Control and Signal Processing. London: Springer London, 2009, p. 632, ISBN: 978-1-84628-641-4. DOI: 10.1007/978-1-84628-642-1. [Online]. Available: http://link.springer.com/10.1007/978-1-84628-642-1.

[7] A. Donzé, "Breach, A Toolbox for Verification and Parameter Synthesis of Hybrid Systems," in, 2010, pp. 167–170. DOI: 978-1-4673-6090-6/14/. [Online]. Available: http://link.springer.com/10.1007/978-3-642-14295-6_17.

[8] J. Eddeland, S. Miremadi, M. Fabian, and K. Åkesson, "Objective functions for falsification of signal temporal logic properties in cyber-physical systems," in *IEEE International Conference on Automation Science and Engineering*, 2017, ISBN: 9781509067800. DOI: 10.1109/COASE.2017.8256285.

[9] V. Raman, A. Donzé, M. Maasoumy, R. M. Murray, A. Sangiovanni-Vincentelli, and S. A. Seshia, "Model predictive control with signal temporal logic specifications," in *53rd IEEE Conference on Decision and Control*, Dec. 2014, pp. 81–87. DOI: 10.1109/CDC.2014.7039363.

[10] X. Qin, N. Aréchiga, A. Best, and J. Deshmukh, "Automatic Testing and Falsification with Dynamically Constrained Reinforcement Learning," Oct. 2019. [Online]. Available: http://arxiv.org/abs/1910.13645.

[11] Y. Yamagata, S. Liu, T. Akazaki, Y. Duan, and J. Hao, "Falsification of Cyber-Physical Systems Using Deep Reinforcement Learning," *IEEE Transactions on Software Engineering*, 2020, ISSN: 0098-5589. DOI: 10.1109/TSE.2020.

2969178. [Online]. Available: https://ieeexplore.ieee.org/document/8967146/.

[12] R. S. Sutton and A. G. Barto, *Reinforcement Learning, Second Edition: An Introduction - Complete Draft*. 2018, ISBN: 9780262039246.

[13] J. van Rijn and F. Hutter, "Hyperparameter importance across datasets," Oct. 2017.

[14] D. Amodei and J. Clark, *Faulty Reward Functions in the Wild*. [Online]. Available: https://openai.com/blog/faulty-reward-functions/.

[15] O. Lindberg and A. Shokrian, "Sequence optimization using reinforcement learning in RobotStudio," Department of Electrical Engineering, Gothenburg, Tech. Rep., 2019.

[16] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, Oct. 2017, ISSN: 0028-0836. DOI: 10.1038/nature24270. [Online]. Available: http://www.nature.com/articles/nature24270.

[17] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, ISBN: 9780262035613. [Online]. Available: http://www.deeplearningbook.org.

[18] G. E. Hinton, *Neural Networks for Machine Learning - Lecture 6e rmsprop: Divide the gradient by a running average of its recent magnitude*, 2012. [Online]. Available: https://www.cs.toronto.edu/~hinton/coursera/lecture6/lec6.pdf (visited on 06/10/2021).

[19] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," Dec. 2014. [Online]. Available: http://arxiv.org/abs/1412.6980.

[20] H. Robbins, "A stochastic approximation method," *Annals of Mathematical Statistics*, vol. 22, pp. 400–407, 2007.

[21] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, Jul. 2011.

[22] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. 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*, vol. 518, no. 7540, pp. 529–533, Feb. 2015, ISSN: 0028-0836. DOI: 10.1038/nature14236. [Online]. Available: http://www.nature.com/articles/nature14236.

[23] L.-J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Machine Learning*, vol. 8, no. 3-4, pp. 293–321, 1992, ISSN: 0885-6125. DOI: 10.1007/BF00992699. [Online]. Available: http://link.springer.com/10.1007/BF00992699.

[24] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor," Jan. 2018. [Online]. Available: http://arxiv.org/abs/1801.01290.

[25] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine, "Soft Actor-Critic Algorithms and Applications," Dec. 2019. [Online]. Available: http://arxiv.org/abs/1812.05905.

[26] J. M. Joyce, "Kullback-leibler divergence," in *International Encyclopedia of Statistical Science*, M. Lovric, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 720–722, ISBN: 978-3-642-04898-2. DOI: 10.1007/978-3-642-04898-2_327. [Online]. Available: https://doi.org/10.1007/978-3-642-04898-2_327.

[27] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016*, 2016, pp. 265–283, ISBN: 9781931971331.

[28] Master Thesis - HADK, *Falsification of Robot Safety Systems using Deep Reinforcement Learning - Master's Thesis*, YouTube, Jun. 2021. [Online]. Available: https://youtu.be/iwkAcOj9bSk.

[29] ABB Robotics, *New braking distance simulator improves safety and reduces robotic cell footprint by up to 25 per cent*, 2021. [Online]. Available: https://new.abb.com/news/detail/74174/prsrl-new-braking-distance-simulator-improves-safety-and-reduces-robotic-cell-footprint-by-up-to-25-per-cent (visited on 03/11/2021).

[30] T. Wang, X. Bao, I. Clavera, J. Hoang, Y. Wen, E. Langlois, S. Zhang, G. Zhang, P. Abbeel, and J. Ba, "Benchmarking Model-Based Reinforcement Learning," Jul. 2019. [Online]. Available: http://arxiv.org/abs/1907.02057.

[31] A. Joshua, "Spinning Up in Deep Reinforcement Learning," 2018.

[32] A. V. Clemente, H. N. Castejón, and A. Chandra, "Efficient Parallel Methods for Deep Reinforcement Learning," May 2017. [Online]. Available: http://arxiv.org/abs/1705.04862.

[33] ABB Robotics, *PADME - Process Automation for Discrete Manufacturing Excellence*, 2021. [Online]. Available: https://new.abb.com/se/padme (visited on 04/22/2021).

[34] A. Hanna, K. Bengtsson, P. L. Götvall, and M. Ekström, "Towards safe human robot collaboration - Risk assessment of intelligent automation," *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, vol. 2020-Septe, pp. 424–431, 2020, ISSN: 19460759. DOI: 10.1109/ETFA46521.2020.9212127.

[35] M. Lewis, K. Sycara, and P. Walker, "The role of trust in human-robot interaction," in *Foundations of Trusted Autonomy*, H. A. Abbass, J. Scholz, and D. J. Reid, Eds. Cham: Springer International Publishing, 2018, pp. 135–159, ISBN: 978-3-319-64816-3. DOI: 10.1007/978-3-319-64816-3_8. [Online]. Available: https://doi.org/10.1007/978-3-319-64816-3_8.

[36] H. van Hasselt, Y. Doron, F. Strub, M. Hessel, N. Sonnerat, and J. Modayil, "Deep Reinforcement Learning and the Deadly Triad," *CoRR*, vol. abs/1812.0,

Dec. 2018. arXiv: 1812.02648. [Online]. Available: http://arxiv.org/abs/1812.02648.

[37] F. Ding, G. Ma, Z. Chen, J. Gao, and P. Li, "Averaged Soft Actor-Critic for Deep Reinforcement Learning," *Complexity*, vol. 2021, N. Cai, Ed., pp. 1–16, Apr. 2021, ISSN: 1099-0526. DOI: 10.1155/2021/6658724. [Online]. Available: https://www.hindawi.com/journals/complexity/2021/6658724/.

DEPARTMENT OF ELECTRICAL ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY