

PETRI TIKKA

# CONTROL POLICY TRAINING FOR A SIMULATION-TO-REAL TRANSFER

Simulation-to-real case study

Master of Science Thesis  
Faculty of Information Technology  
and Communication Sciences  
Examiner: prof. Joni Kämäräinen  
December 2021

# ABSTRACT

Petri Tikka: Control policy training for a Simulation-to-Real transfer  
Master of Science Thesis  
Tampere University  
Computing Sciences  
December 2021

---

Robots have been deployed in various fields of the industry, with the expectation of managing more tasks for humans. Simulation-to-real is a relatively new discipline within robotics that offers an alternative approach to the traditional programming methods by training a model of the robot in a simulator and afterwards transferring the knowledge to control the physical counterpart. The knowledge is located in a deep reinforcement learning policy that is carefully selected and tuned for the intended task.

The thesis studied tools and steps that are required to implement a physical system that is trained using the sim-to-real transfer learning. The chosen use case is a Universal Robots UR10e manipulator that is to locate and reach a stationary target in the physical world. Because the scope is to provide a proof-of-concept pipeline for the simulation-to-real process, the only part in the use case requiring adaption is the changing location of the target. However, target reaching is a fundamental task in robotics for which more complex tasks are based upon.

The simulation environment was constructed from a CAD-model of the physical robot cell that was later updated within the chosen simulator CoppeliaSim. For the convenience of having the kinematic chain premade, the manipulator was changed in the simulator to an older version UR10. Also, the gripper was changed to an older version. The control in the simulation environment followed the Markov Decision Process having a manipulator as an agent interacting with the environment. As the agent performed actions in available states, it tried to maximize the total cumulative reward and learned accordingly. The goal was to reach a simulated target that position was randomized along a specified line segment. In practice, the algorithms learned trajectory paths in joint space under given environment constraints while the agent controlled the manipulator with velocity based forward kinematics. The overall process was scripted as Python modules with an interface to the simulator. The considered deep reinforcement learning algorithms were Deep Deterministic Policy Gradient and Soft Actor-Critic.

The algorithms were validated in the simulation and Deep Deterministic Policy Gradient was chosen for the simulation-to-real transfer owing to its better performance. The transfer was based on a zero-shot method where the policy controlled the physical manipulator from the simulation. Control included the joint positions of the simulated manipulator that were forwarded to the physical counterpart via Robot Operating System network. Therefore, the knowledge transfer only considers kinematics. The network conjoined the simulator, the manipulator and the machine vision system, which was responsible of tracking the target, an ArUco marker. The marker position replaced the random position of the simulated target.

The simulation-to-real transfer process demonstrates a working step-by-step pipeline, which at the time of writing this thesis was not publicly provided. The resulted policy learned a redundant kinematics trajectory with geometrical limitations. The manipulator reaches the target within the given precision threshold with a collision free path. At the same time the reality gap between simulation and reality is explained and managed. Although the task related results are not generalizable, the concept of simulation-to-real transfer is applicable to more complex tasks.

Keywords: Simulation-to-Real, Kinematics, Deep Reinforcement Learning, Manipulator

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

# TIIVISTELMÄ

Petri Tikka: Simulaatiosta-todellisuuteen tiedonsiirto robotin ohjaukselle  
Diplomityö  
Tampereen yliopisto  
Tietotekniikka  
Joulukuu 2021

---

Robottien määrä on yleistynyt teollisuuden eri aloilla sillä oletuksella, että ne kykenisivät toteuttamaan entistä enemmän ihmisille tarkoitettuja tehtäviä. Perinteisten menetelmien lisäksi robottien ohjelmointia voidaan lähestyä 'simulaatiosta todellisuuteen' tieteenalan keinoin, joka on suhteellisen uusi robotiikan haara. 'Simulaatiosta todellisuuteen' perustuu robotin kouluttamiseen simulaatiossa ja koulutetun tiedon siirtämiseen fyysiselle vastinparille. Tieto sijoittuu syvään vahvistelun oppimisen menettelytapaan, joka on huolellisesti valittu ja viritetty aiottua tehtävää varten.

Diplomityössä tutkittiin työkaluja ja vaiheita, joita tarvitaan 'simulaatiosta-todellisuuteen' siirto-oppimisen avulla koulutetun fyysisen järjestelmän toteuttamiseen. Valittu ympäristö on Universal Robots UR10e-manipulaattori, jonka tehtävänä on paikantaa ja saavuttaa paikallaan oleva kohde fyysisessä maailmassa. Koska työn tarkoituksena on tarjota konsepti 'simulaatiosta todellisuuteen' prosessille, ainoa mukautumista vaativa osa käyttötapauksessa on kohteen muuttuva sijainti. Huomioitavaa on, että kohteen saavuttaminen on kuitenkin robotiikan perustehtäviä, johon monimutkaisemmat tehtävät perustuvat.

Simulaatioympäristö rakennettiin fyysisen robottisolun CAD-mallista, jota päivitettiin myöhemmin valitussa CoppeliaSim-simulaattorissa. Työn helpottamiseksi, simulaatiossa hyödynnetään vanhempaa manipulaattorimallia UR10:ä, joka sisälsi nativisti kinemaattisen ketjun. Myös tартtuja vaihdettiin vanhempaan malliin simulaattorissa. Simulaatioympäristön ohjaus seurasi Markovin päätösprosessia, jossa agentti eli manipulaattori on vuorovaikutuksessa ympäristön kanssa. Kun agentti suoritti toimintoja mahdollisissa tiloissa, se pyrki maksimoimaan kumulatiivisen kokonaispalkkion ja oppi sen mukaisesti. Simuloidun robotin kohteen paikka vaihteli satunnaisesti työalueella olevalla janalla, ja robotin ohjaus toteutettiin nopeuteen perustuvalla suoralla kinematiikalla. Prosessi toteutettiin Python moduuleilla ja valitut oppivat algoritmit olivat Deep Deterministic Policy Gradient sekä Soft Actor-Critic.

Algoritmit validoitiin simulaatiossa ja Deep Deterministic Policy Gradient valittiin 'simulaatiosta todellisuuteen' siirtoa varten sen tuottamien vakaampien ja turvallisempien liikeratojen johdosta. Tiedon siirto perustui zero-shot menetelmään, jossa menettelytapa ohjasi fyysistä manipulaattoria simulaation välityksellä. Nivelten paikkatiedot välitettiin simulaatiosta fyysiselle robotille Robot Operating System-verkon kautta. Koska ainoa välitetty tieto simulaatiosta fyysiselle robotille on paikkaperusteista, on kehitetty menetelmä käytännössä vain kinemaattinen. Käytetty tietoverkko yhdisti simulaattorin, manipulaattorin sekä konenäköjärjestelmän, joka vastasi ArUco merkkitunnisteen paikantamisesta. Kyseisen tunnisteen sijainti korvasi simulaatiossa olevan satunnaisesti vaihtelevan kohteen sijainnin.

Esitetty 'simulaatiosta-todellisuuteen' tiedonsiirtoprosessi osoittaa toimivan vaiheittaisen toteutusketjun, jollaista ei ollut julkisesti saatavilla tätä diplomityötä kirjoitettaessa. Koulutettu algoritmi kykenee vastaamaan geometrisin rajoituksin ja redundanttisten vapausasteiden mukaisen liikkeen suunnittelun ongelmaan. Fyysinen manipulaattori saavuttaa kohteen törmäysvapaalla liikerajalla annetun tarkkuuskynnyksen rajoissa. Samalla simulaation ja todellisuuden välillä olevan todellisuuseron vaikutukset pystytään selittämään ja vaikutukset esittämään. Vaikka kohteen saavuttamistehtävään liittyvät tulokset eivät ole yleistettävissä muihin tehtäviin, on esitetty 'simulaatiosta todellisuuteen' konsepti sovellettavissa vaativampikin tehtäviin.

Avainsanat: Simulaatiosta-Todellisuuteen, Kinematiikka, Syvä Vahvistettuoppiminen, Manipulaattori

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck –ohjelmalla.

# PREFACE

This thesis was done for the Faculty of Information Technology and Communication Sciences at Hervanta campus of Tampere University (TUNI). The thesis addresses simulation-to-real transfer supporting deep reinforcement learning for a real-world control policy. The work was commissioned by VTT Oy Technical Research Centre of Finland.

I would like to thank VTT for the opportunity to produce a study on the given subject. Atakan Dağ of TUNI I would like to thank for his invaluable advices concerning the simulation environment. Senior Scientist Hannu Saarinen of VTT is to thank for the guidance on manipulator kinematics both theory and practice. Thesis supervisors Professor Joni Kämäräinen of TUNI and Professor Tapio Heikkilä of VTT I thank for their professional inputs on robotics, machine learning, scientific writing, and overall guidance over the course of the project.

Tampere, 3 December 2021

Petri Tikka

# CONTENTS

1 INTRODUCTION .....	1
1.1 Structure and Milestones of the Thesis .....	2
1.2 Identification of the Problem and Objectives .....	4
2 THE MECHANICS AND CONTROL OF MANIPULATORS .....	5
2.1 The Mechanics of a manipulator .....	6
2.2 Kinematics for mechanical manipulation .....	7
2.2.1 Forward kinematics .....	8
2.2.2 Inverse kinematics .....	9
3 LEARNING-BASED CONTROL PARADIGMS .....	10
3.1 Machine learning paradigms .....	10
3.2 Reinforcement Learning .....	12
3.3 The Markov Decision Process .....	13
3.3.1 The reward function and the goal .....	15
3.3.2 Formal definition of the MDP .....	17
3.3.3 Temporal difference learning .....	20
3.4 Model-based and model-free methods .....	22
3.5 Actor-critic framework .....	24
3.6 Reinforcement algorithms for continuous action spaces .....	26
3.6.1 Deep Deterministic Policy Gradient .....	28
3.6.2 Soft Actor-Critic .....	31
4 SIMULATION-TO-REAL LEARNING .....	35
4.1 Simulators for Sim2Real transfer .....	35
4.2 Sim2Real transfer .....	36
4.2.1 Relevant Examples .....	36
4.2.2 Transfer techniques .....	37
4.3 Reality gap .....	38
5 SIMULATION-TO-REAL CASE STUDY .....	40
5.1 Task description .....	40
5.2 Simulation environment .....	41
5.3 Known reality gap factors .....	43
5.4 DRL software architecture .....	45
5.4.1 MDP in practice .....	46
5.4.2 Software modules .....	49
5.5 Training and validation .....	51
5.5.1 Training DDPG policy .....	53
5.5.2 Training SAC policy .....	57
5.5.3 Validating results in simulated environment .....	60
5.5.4 Modifying initial parameters and revalidation .....	62
5.6 Sim2Real transfer process .....	67
5.6.1 Machine vision system .....	68

5.6.2 System architecture for Sim2Real transfer .....	72
5.7 Validating results in physical environment.....	75
6 RECOMMENDATIONS FOR FUTURE WORK .....	79
7 SUMMARY AND CONCLUSIONS .....	80
REFERENCES.....	82
APPENDIX A: INSTANTIABLE DDPG REVALIDATION PARAMETERS .....	91
APPENDIX B: INSTANTIABLE SAC REVALIDATION PARAMETERS .....	92
APPENDIX C: SIM2REAL TRANSFER RESULTS.....	93

# LIST OF FIGURES

Figure 1.1. Conceptual view of a Sim2Real transfer process. Orange and green indicate work done in the physical (orange) and simulated (green) environments. ....	2
Figure 2.1. Classification of robots by application. The red line represents the scope of this thesis. Modified from [10], [11]. ....	5
Figure 2.2. Kinematic terms describing tool frame relative to the base frame as a function of the joint variables (right), modified from [9]. Workspace volume of an anthropomorphic manipulator (left), modified from [11]. ....	6
Figure 2.3. A schematic view of forward and inverse kinematics between joint and cartesian space. Modified from [12]. ....	7
Figure 2.4. Compatible configurations (left skeleton, right graphical) of an anthropomorphic manipulator with given end-effector pose. Modified from [9], [11]. ....	9
Figure 3.1. Overview of main machine learning paradigms and methods, modified from [19]. ....	11
Figure 3.2. The basic structure of reinforcement learning paradigm as the agent-environment interaction process. Modified from [4]. ....	14
Figure 3.3. The taxonomy of reinforcement learning methods. Modified from [1]. ....	23
Figure 3.4. Actor-critic framework with on-policy (left) and off-policy (right), modified from [4], [35]. ....	25
Figure 4.1. Sim2Real transfers for complex motor control with manipulator control (peg-in-the-hole) (left), dexterous hand control (middle) and locomotion (right). Modified from [7], [65] and [66]. ....	37
Figure 4.2. Markov Decision Processes in a simulation and reality depicting state capture, policy inference and action execution processes on a timeline. Modified from [1]. ....	39
Figure 5.1 Case specific robotic cell in simulation (left) and in real-life (right). ....	40
Figure 5.2. Simulation environment for the reach target task in CoppeliaSim (middle, right) with the kinematic chain of the UR10 from the base to the gripper represented with red line (left). Target is represented with a red dot, which position is randomized during the training along the dotted blue line (right). ....	42
Figure 5.3. DH-parameters of UR10 and UR10e (left) and a parameterized diagram of a UR manipulator (right). Modified from [82]. ....	44
Figure 5.4. Base of UR10e in the physical environment (left) and base of UR10 in the CoppeliaSim simulation (right). Pictures are not in scale, and distances are in mm. ....	44
Figure 5.5. Robotiq 2F-85 gripper newer version (left) and older version (right). Distances are in mm. Modified from [83], [84]. ....	45
Figure 5.6. RL policy training as a MDP in simulation. ....	47
Figure 5.7. Simplified representation of the software architecture for the DRL study. Function and parameter names have been modified for the sake of intelligibility. Arrows depict the direction of inheritance. ....	50
Figure 5.8 DDPG episode rewards for Run 1-6 (a-f) as presented in Table 5.3. Red line represents the moving average of 10, and blue line the actual value. ....	54
Figure 5.9 Effect of random seeds to convergence of five DDPG policies with same parameters (Run 1 of Table 5.3). Light shaded regions depict actual values and solid lines moving average of 100 episodes. ....	55

Figure 5.10 Effect of random seeds to convergence of five DDPG policies with same parameters (Run 3 of Table 5.3). Light shaded regions depict actual values and solid lines moving average of 100 episodes. ....	55
Figure 5.11 DDPG success rates for Run 1-6 (a-f) as presented in Table 5.3. Red line represents the moving average of 10, and blue line the actual value. ....	57
Figure 5.12 SAC episode rewards for Run 1-6 (a-f) as presented in Table 5.4. Red line represents the moving average of 10, and blue line the actual value. ....	58
Figure 5.13 SAC success rates for Run 1-6 (a-f) as presented in Table 5.4. Red line represents the moving average of 10, and blue line the actual value. ....	60
Figure 5.14 Validation of SAC (left) and DDPG (right) with dense and augmented reward function while demonstrating hyperextension of the joints. Red line indicates the direct trajectory for the TCP starting from the zero-position. ....	62
Figure 5.15. Favourable direction of motion for joint 5 (left) and for joint 2 (right) in regards the zero-position. Red arrow indicates the favourable direction of motion. ....	63
Figure 5.16 DDPG per episode reward and success rate for policies trained according to Table 5.7. Shaded region depicts standard deviation of performance and solid line moving average of 100 episodes. ....	65
Figure 5.17 SAC per episode reward and success rate for policies trained according to Table 5.8. Shaded region depicts standard deviation of performance and solid line moving average of 100 episodes. ....	65
Figure 5.18 DDPG success rate according to Table 5.7 and Run 6 of Table 5.3 over the course of training. Success rates are given as moving average of 100 episodes. ....	66
Figure 5.19 SAC success rate according to Table 5.8 and Run 6 of Table 5.4 over the course of training. Success rates are given as moving average of 100 episodes. ....	66
Figure 5.20 Cyber-Physical system representation for the concept to formulate Sim2Real transfer. ....	68
Figure 5.21. ArUco marker id 582 (left) and the marker seen from the camera (right). ....	69
Figure 5.22. Standard frames and distance (red) between tool frame and goal frame represented in an illustration (left), modified from [9]. Frame representation in real case environment (right). ....	70
Figure 5.23. ArUco marker localized on the table and the pose is given in accordance to origin in the base of UR10e. Control is managed with inverse kinematics. ....	71
Figure 5.24. The TCP position of X-, Y- and Z-axis compared to the target (marker) position during a 50 second execution on UR10e. Target position is changed three (3) times (illustrated with dotted red lines). Yellow background indicates a period where the manipulator is approaching the target from the initial state. Red background indicates the TCP is at the target and green background indicates a period where the manipulator is withdrawing back to initial state. The three positions of the target are numbered from 1-3 and they are matched with the TCP at the red areas of similar numbering. ....	72
Figure 5.25. The developed ROS communication network for the Sim2Real transfer. ....	73



Figure 5.26. The position of the target dot in the simulation is subscribed from the machine vision system that is tracking the physical ArUco marker. Tracked position is the centre of the ArUco marker. Images 1 and 4, 2 and 5, and 3 and 6 represent the positions of the target in the simulated and physical setup counterparts.....	74
Figure 5.27. Sim2Real performed on UR10e controlled by DDPG policy from CoppeliaSim simulator via ROS network. ArUco marker is at position 1 of Table 5.10. Process follows routine depicted in Figure 5.20. Green boxes implicate actions performed in cyber domain and orange boxes actions in the physical domain. ....	76
Figure 5.28 Five consecutive reaching tasks performed with physical and simulated setup by DDPG policy. Target stays at the same position during all episodes (position 1 of Table 5.10). Target is illustrated with dotted red line. Lighter colours depict the simulated and darker the real manipulator trajectory. Yellow background indicates a period where the manipulator is approaching the target from the initial state. Red background indicates the TCP is at the target and green background indicates a period where the manipulator is withdrawing from the target back to the initial state. ....	77
Figure 5.29 Distance to target measured from simulated and physical manipulator TCP. Black line indicates the precision threshold of 5cm (50mm). ....	78

## LIST OF TABLES

<i>Table 5.1. Joint torque [Nm] and angular velocity [rad/s] limits of UR10/UR10e [81], [3].</i>	43
<i>Table 5.2 Reach task environment dimensions for UR10/UR10e.</i>	46
<i>Table 5.3 DDPG policy training hyperparameters of two scenarios for sparse, dense, and augmented reward functions.</i>	53
<i>Table 5.4 SAC policy training hyperparameters of two scenarios for sparse, dense, and augmented reward functions.</i>	58
<i>Table 5.5 Validation success rates [%] of scenarios in Table 5.3 and Table 5.4 over 1000 episodes.</i>	61
<i>Table 5.6 Final orientations of the manipulator either at the target or at the end of the episode.</i>	61
<i>Table 5.7 Parameter settings for most prominent DDPG policy.</i>	64
<i>Table 5.8 Parameter settings for most prominent SAC policy.</i>	64
<i>Table 5.9 Validation success rates [%] over 1000 episodes and manipulator orientations at the target.</i>	67
<i>Table 5.10 Validation success rates [%] of scenarios (1-6) from Figure 5.26 for the simulated and physical setup over five episodes/repetitions.</i>	75
<i>Table 5.11 Distances [mm] between target, and simulated and physical TCP as target has been reached in five reaching tasks of Figure 5.29.</i>	78

# LIST OF SYMBOLS AND ABBREVIATIONS

ArUco	Augmented reality University of Cordoba
DDPG	Deep Deterministic Policy Gradient
DH	Denavit-Hartenberg
DNN	Deep-Neural Network
DoF	Degree-of-Freedom
DRL	Deep-Reinforcement Learning
MDP	Markov Decision Process
ML	Machine Learning
MSE	Mean Square Error
POC	Proof-Of-Concept
Real2Real	Real-to-Real
RL	Reinforcement Learning
ROS	Robot Operating System
SAC	Soft Actor-Critic
Sim2Real	Simulation-to-Real
TCP	Tool-Center-Point
$A_t / a_t$	an action
$B$	base frame
$D_{KL}$	Kullback-Leibler function
$E_p$	episode consisting of a sequence of states, action and rewards
$E_t$	error equation
$ee$	pose of the end-effector
$G$	goal frame
$G_t$	total discounted reward/return
$J$	expected return of a specific measure
$j_i$	joint angular position
$M$	episodes
$N$	batch size
$P$	a state transition probability function
$pp$	target goal
$q_i$	vector of joint variables
$q_\pi$	an action-value function under a policy (Q-function)
$R_t / r_t$	a reward at time step $t$
$S_t / s_t$	a state at time step $t$
$T$	Tool frame
$T$	steps per episodes
$T_y^x$	transformation operations from x-frame to y-frame
$t$	time
$v_i$	joint velocity
$Wo$	world frame
$Wr$	wrist frame
$\alpha$	learning rate
$\gamma$	discount factor
$\varepsilon_T$	precision factor
$\varepsilon_t$	noise vector
$\theta$	<i>stochastic</i> parameter vector (policy parameters), network parameter (Soft Actor-Critic)

$\theta_{joint}$	joint angle
$\theta^Q$	actor network weight
$\theta^\mu$	critic network weight
$\lambda$	step size
$\mu(s)$	a deterministic policy function
$v_\pi$	a state-value function under a policy
$\pi(a s)$	a stochastic policy function
$\rho$	a distribution
$\sigma_t$	exploration noise at time step t
$\tau$	rate of copying weights (hyperparameter)
$\phi$	value function parameter, network parameter (Soft Actor-Critic)
$\psi$	network parameter (Soft Actor-Critic)
$\mathcal{D}$	replay buffer (distribution of previously sampled states and actions)
$\mathcal{H}$	entropy term
$*$	optimal outcome

# 1 INTRODUCTION

Simulation-to-real (Sim2Real) is a concept for transferring knowledge from simulation to reality. Simulation generates data for robotics that can be utilized for machine learning (ML) models to train robots to see and manipulate objects within a given environment; the training is in simulation, but the deployment is on a real robot [1]. Sim2Real suggests reducing or even removing the need for collecting large quantities of real-world data, which can be not only difficult and dangerous to collect but also expensive in terms of resources [2].

The thesis explores a Sim2real transfer process in the context of a Universal Robots UR10e robotic manipulator [3]. Utilized control policy is based on reinforcement learning (RL), a machine learning paradigm which has gained acknowledgement among robotics domain with the advancement achieved by the ML community [4], [5]. The policy learns in simulation through trial-and-error to perform a fundamental robotic manipulation task - target-reaching.

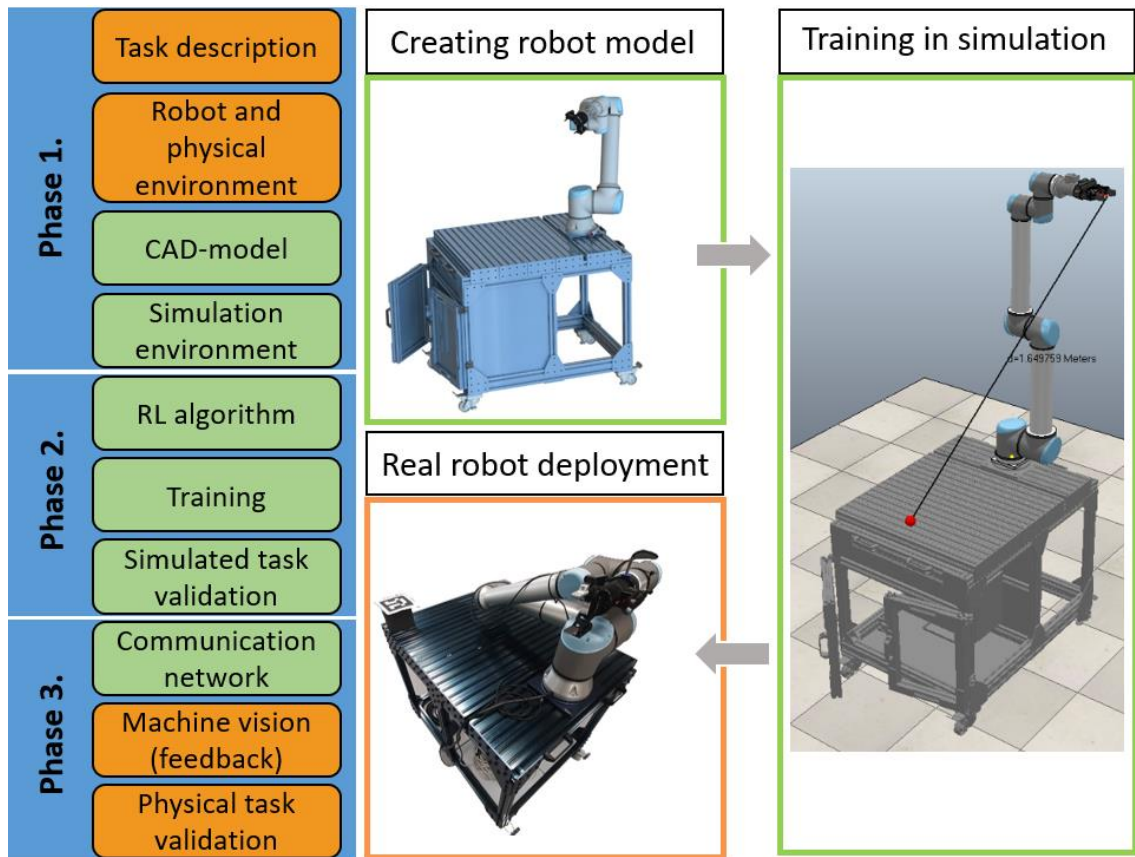
The task involves kinematics and trajectory learning with redundant degree of freedom (DoF); only position is considered leaving orientation out of the scope. The focus is on generating a proof-of-concept (PoC) including a successful control policy that can be validated not only in simulation, but also on the physical manipulator, thus providing a simulation-to-real transfer. Goal is to provide a policy that has a success rate of 100% in achieving the target given the precision threshold and trajectory heuristics.

A problematic aspect of Sim2Real is the reality gap between the simulated and the real-world environment. The developed pipeline brings forward this challenge while the compatibility of two RL algorithms is benchmarked against the presented robotic task. Tools and steps to build a pipeline from the simulation to the real-world are explored with the focus being on training and validating the RL policy.

## 1.1 Structure and Milestones of the Thesis

The work was accomplished at VTT (Technical research centre of Finland) with a government grant. The work was inspired by the industrial renewal concept of the future of robotics concerning adapting robotic manipulators to unforeseen scenarios. Most of the traditional robot programming methods expect that environments are known and structured, which often is not the case [6]. The dilemma raised the question if a robotic manipulator can learn to accomplish a given task independently without operator inserting or teaching the intended trajectories. In this thesis the question is being explored as constructive research with aim to produce an academic Sim2Real workflow that supports reinforcement learning training on a simulated robot to obtain a control policy applicable to a real-world environment. Structure and usability of chosen RL algorithms are studied in practice and through related scientific literature. The study concentrates on the transfer process; hence the environment is known in advance.

Figure 1.1 depicts the Sim2Real workflow on a conceptual level indicating the prime milestones to be achieved in the physical environment with orange colour and milestones to be achieved in the simulated environment with green colour.



**Figure 1.1.** Conceptual view of a Sim2Real transfer process. Orange and green indicate work done in the physical (orange) and simulated (green) environments.

The workflow provides a base for the thesis structure, and it can be divided into three main phases.

**Phase one** consists of setting up the environment in accordance with the specified task description. The task in question is a reach-target, which is presented in Section 5.1 for the task apprehension in the physical and the simulated environment. As the task is identified, proper camera and auxiliary hardware is implemented to the robotic environment. The simulated and physical environment are presented in Figure 5.1 and discussed in Sections 5.1 and 5.2. As the physical setup is validated, a digital representation is generated, and a relevant simulation environment is chosen including digital counterparts of the physical environment. An exception is the camera, which was not digitalized, but the results of the physical camera are imported to the simulation for the Sim2Real transfer.

**Phase two** consists of describing machine learning (ML) paradigm RL and relevant algorithms that are applicable to produce control according to the defined task in the developed environment. Relevant hyperparameters and reward function structures are searched to optimize the behaviour of the chosen method. The RL control algorithm is implemented within the software architecture, and the internal application programming interfaces (API) between the task management script, environment management and the RL algorithm are verified. The control algorithm is trained, and the consistency is validated in the simulation. The process of selecting a proper RL algorithm is discussed in Section 3.6 while the implementation, training and validation is presented in Section 5.5.

Finally, in **phase three** the simulated control is extended to the physical world. Control from simulation to the physical setup is enabled by a network between the hardware. The network includes the physical manipulator, network server machine, simulator machine and a machine for upholding the machine vision system. Machine vision system is implemented to the network to provide feedback for the control while executing localization tasks of the target. Eventually, the physical task validation confirms the transfer of simulation trained control policy within the real-world environment. The structure of the communication network, physical domain feedback implementation and the results of the knowledge transfer validation are presented in Sections 5.6 and 5.7.

The first phase considers settings up the unity of countering environments, whereas the second phase is an iterative process where RL algorithms are benchmarked against each other to clarify the best hyperparameters, reward functions, and eventually policy

for the given study. Success is to be measured in simulation after forking the optimal solution that is validated also in the final phase.

## 1.2 Identification of the Problem and Objectives

Task of Sim2Real transfer is challenging. This is especially true in the domain of robotics where the expectation is for robots to complete more control tasks for industry personnel. Consequences of a simulation trained ML algorithm are not only difficult to predict in a real-world scenario, but also their propagation to the real world is as well. Selecting proper hyperparameters for the right RL algorithm can be laborious since they depend on the used algorithm as well as the application area [6]. Similarly, the construction of the simulated environment to be identical to the real-world counterpart with high fidelity and dynamics is a non-trivial task for which a single simulator of today cannot unambiguously answer [2], [7]. However, with the right abstraction level and application area, simulator selection and environment development is possible [8].

The aim of this thesis is to produce control policy training for a Sim2Real transfer. The process includes describing a pipeline from simulation trained algorithm to a real-world environment while performing a reach target task. RL algorithms are trained and verified in simulation before continuing to a Sim2Real transfer. The main objectives for motivation can be expressed as:

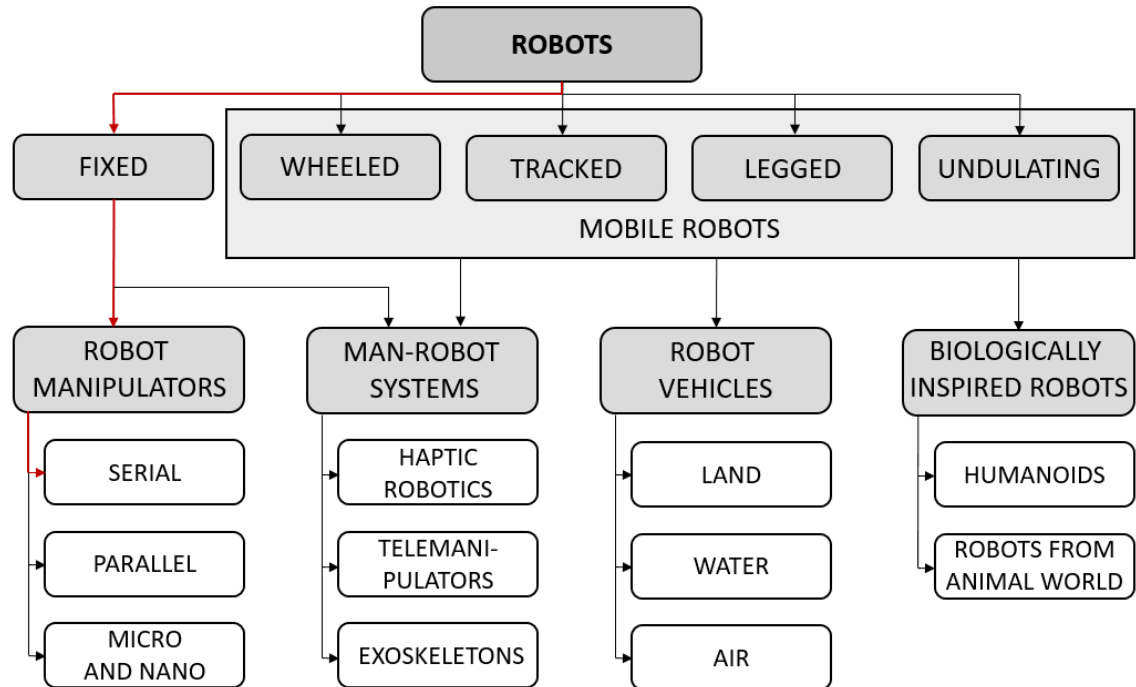
- Verify the applicability of a RL algorithm for a robotic task by training a policy in a case study specific simulation.
- Benchmark chosen RL algorithms against the performed task in simulation by optimizing reward functions and hyperparameters.
- Evaluate the feasibility and practicality of the Sim2Real transfer process for the trained policy on a real UR10e.

The first objective includes selecting and preparing simulation environment with sufficient APIs to support a control algorithm for training and validating the RL policy. Simulator should provide internal APIs for the software architecture based on the training of the control policy, but also provide external APIs to formulate a network for hardware communication. The second objective depends on finding the optimal policy for the presented task. This relates to the state-of-the-art literature review made during the thesis. The third objective is about studying Sim2Real transfer methodologies and evaluating chosen process against the reality gap between the simulation and the physical setup. The level of the achievable adaptation is related to the reality gap, how much it affects the results and is the gap possible to be crossed over.



## 2 THE MECHANICS AND CONTROL OF MANIPULATORS

The scientific discipline of robotics involves simulating human functions with a combination of sensors, actuators and computational power. Combining hardware with software requires a variety of different fields of expertise, but at a high level of abstraction, robotics can be regarded as a combination of mechanical manipulation, locomotion, computer vision and computer science [9]. Together these disciplines have spawned a variety of robotic interpretations that can be classified according to the intended application as depicted in Figure 2.1.

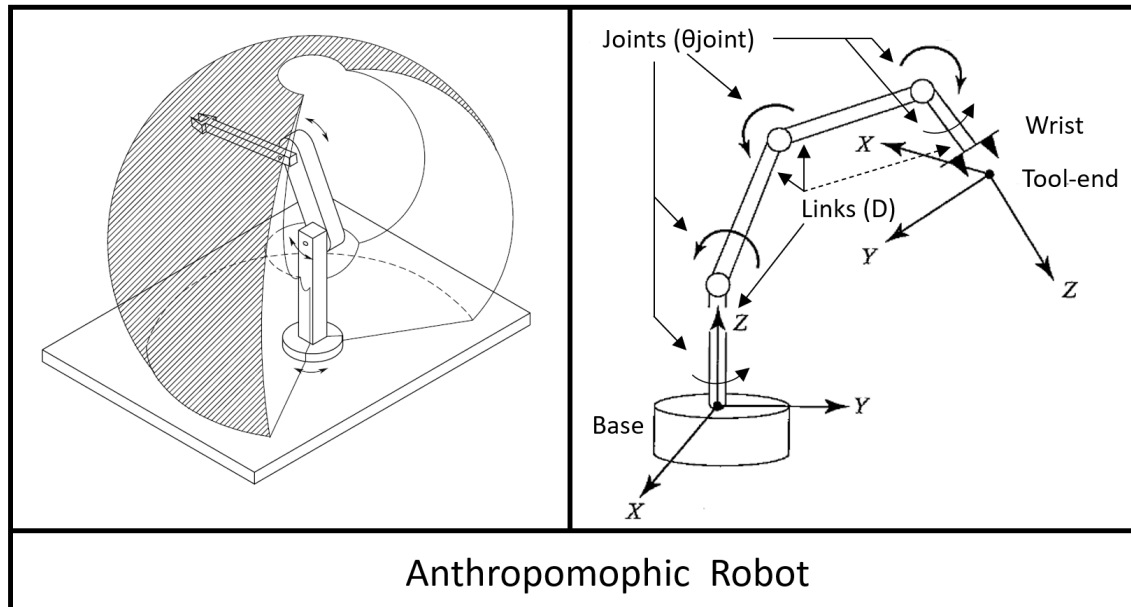


**Figure 2.1.** Classification of robots by application. The red line represents the scope of this thesis. Modified from [10], [11].

This thesis is focused on robotic manipulators and more specifically a fixed serial manipulator UR10e by Universal Robots. Serial manipulators are the most common type of manipulator with a structure that comprises of a serial chain of rigid segments connected by joints. The basic mechanics of the used manipulator structure are presented to emphasize the complexity of the kinematic chain the control has to manage.

## 2.1 The Mechanics of a manipulator

The mechanical structure of a manipulator includes rigid links interconnected by articulative joints. These joints usually include position sensors, which allow the relative motion of neighbouring links to be measured. In general, all articulation between joints can be realized as revolute or prismatic motion, depending on the orientation of the axis. The joint angles are represented by  $\theta_{joint}$ , and the relative positions between the links is represented by distance  $D$  as depicted in Figure 2.2.



**Figure 2.2.** Kinematic terms describing tool frame relative to the base frame as a function of the joint variables (right), modified from [9]. Workspace volume of an anthropomorphic manipulator (left), modified from [11].

The joint configurations define the mobility of a manipulator. The chain of links begins with an arm that ensures general mobility, and with a wrist that enables dexterity. The arm enables position while the wrist typically consists of three revolute joints and orientation. One end of the arm is fixed to a non-moving base with a base frame or fixed frame. At the free end of the chain is the tool-end or end-effector with a tool frame.

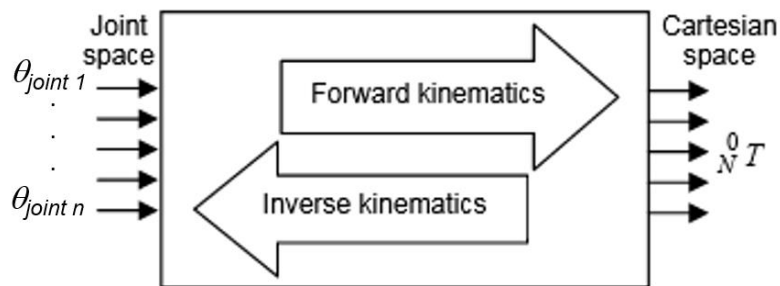
Generally, the position of the manipulator is described by the tool frame relative to the base frame. A task consisting of arbitrarily positioning and orienting an object in three-dimensional space requires six degrees of freedom. The first three degrees of freedom describe the position of a point on the object (X-, Y-, Z-axis) and the other three determine the orientation of the object with respect to a reference frame (pitch, yaw and roll). The term 'pose' is used to describe both the position and orientation of an object [10].

The shape and volume of a manipulator workspace depends on the aforementioned limitations of the joint configurations and hence the structure of the manipulator. Utilized Ur10e manipulator represents an anthropomorphic manipulator depicted in Figure 2.2.

Manipulator control strategy depends on whether the target is concerned with control a single joint independently, or to note the effects of the dynamic interactions between all the joints in order to control their movement. To generalize, joint actuators produce forces and torques, which can be viewed on a timeline as the control of motion while satisfying given transient and steady-state requirements [11]. The control of motion is related to the problem of how to choose a set of control variables to position the manipulator at a desired location with position and orientation. Although, the relative positioning does not concern the motion effects of the manipulator, the kinematics does.

## 2.2 Kinematics for mechanical manipulation

Kinematics is the study of the motion of an object regardless of the forces that cause the motion. Hence, the targets are position, velocity, acceleration, and other higher order derivatives of position variables. More specifically, manipulator kinematics aims to explain a joint space leading to the positioning of the end-effector at a target location on a cartesian space, also called as a workspace, by taking into consideration all the geometrical and time-based properties of the motion. The relationship between joint space and cartesian space is depicted in Figure 2.3.  $\theta_{joint\ n}$  represents the Nth joint of a manipulator and  ${}^0_N T$  describes the position and orientation of the Nth joint with respect to the 0 joint (base) [12].



**Figure 2.3.** A schematic view of forward and inverse kinematics between joint and cartesian space. Modified from [12].

In manipulator centric robotics, kinematics involves the relationship between the joint variables of a manipulator and the position and orientation of the end effector. To reach a specific point in the workspace, one can map the point from the joint space to the end-effector location in the workspace with known joint variables via forward kinematics (direct kinematics). The alternative is to calculate the unknown joint variables for each joint

to place the end-effector at the specific point with inverse kinematics. In other words, for forward kinematics the inputs are the joint angles, and the outputs are the coordinates of the end-effector whereas it is vice versa for inverse kinematics. For a manipulator with multiple degrees-of-freedom, the forward kinematics calculus is straightforward. However, it is more complicated for inverse kinematics since there may be more than one unique configuration for the same end-effector coordinates. This is the case if, for example, the final configuration would pose the manipulator to be redundant from a kinematic viewpoint, i.e., some degrees of freedom are unused.

Section 2.1 explained how the body of a manipulator is thought of as a set of connected links joined by a chain of joints. To solve the kinematic equations of the manipulator mechanism, robotic links are considered as rigid bodies that define the relationship between two connected joint axes of a manipulator. Each degree of freedom is associated with a joint articulation, which constitutes to a joint variable.

### **2.2.1 Forward kinematics**

Forward kinematics utilizes kinematic equations of the manipulator to calculate the pose of the end-effector from the joint parameters of the actuators. Process is essential to detect if the manipulator can collide with the environment. For serial manipulators the process consists of directly substituting the given joint parameters into the kinematics equations for the serial chain. Forward kinematics solves a static geometrical problem for computing the pose of the tool frame relative to the base frame. The process is occasionally considered as changing the representation of a manipulator position from the joint space into a cartesian space description.

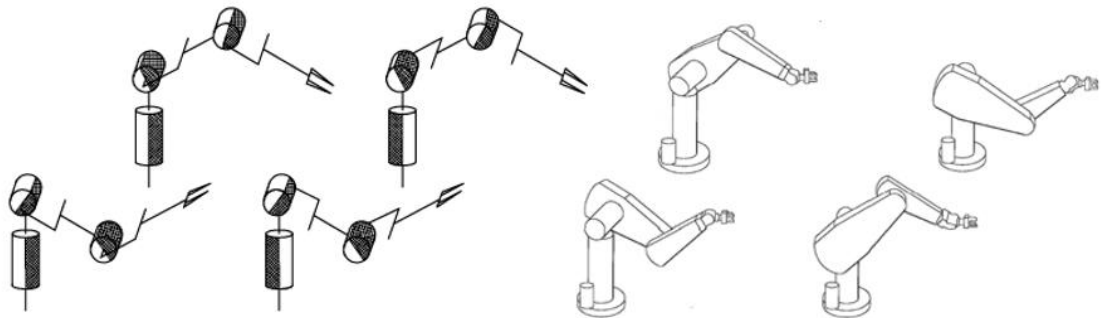
For open chain kinematics the equations are obtained from a rigid transformation of the relative movement at each joint and rigid transformation of each link dimensions as a sequence from the base of the chain to the end link. This specifies the position for the end-effector. The joint and separate link matrices are associated with coordinate frames for spatial linkages. These are formulated into a homogeneous transformation matrix. [11], [13]

Homogeneous transformation means reducing the composition of rigid motions to matrix multiplication. Therefore, the overall description of manipulator kinematics is obtained through a recursive process. The final matrix convention provides forward kinematics calculation through composition of individual link transformations into one homogeneous transformation matrix.

### 2.2.2 Inverse kinematics

The reverse process for computing the joint parameters is called inverse kinematics. The process consists of calculating all possible sets of joint angles that could be substituted to attain the given position and orientation of the end-effector at the operational space. Most of the methods are categorized into complete analytical, also called closed-form, solutions or numerical solutions [12].

The closed-form solutions are faster than numerical, but they are dependent on the structure of the manipulator and the solutions must be developed separately for each manipulator. Yet, for real-time applications closed-form solutions are a necessity as the kinematic equations need to be solved at a rapid rate. If the computation of closed-form solutions, algebraic or geometric method, cannot find a solution or it is difficult to generate equations containing the unknowns, one can resort to numerical solution methods instead. A prime example of numerical solutions and iterative methods is the utilization of a Jacobian matrix, which exploits forward kinematic problem [12]. However, the equations to be solved are nonlinear, and closed-form solutions cannot always be found unlike with forward kinematics where equations could provide a unique solution if the joint variables were known. With inverse kinematics, there might exist multiple to infinite solutions for the pose of the end-effector, especially if the manipulator is kinematically redundant. Figure 2.4 depicts the problem field with an anthropomorphic manipulator like UR10e, that can have multiple compatible configurations with the given end-effector pose.



**Figure 2.4.** Compatible configurations (left skeleton, right graphical) of an anthropomorphic manipulator with given end-effector pose. Modified from [9], [11].

Deriving the inverse equations for a six DoF manipulator is arduous and especially for UR10e owing to its uncommon non-spherical wrist configuration. A note to be made, that there are no general algorithms to be employed to solve a set of nonlinear equations and a solution depends on the given schematic of the manipulator. Exemplary solutions for the inverse kinematics problem can be examined from [14], [11] or [9].

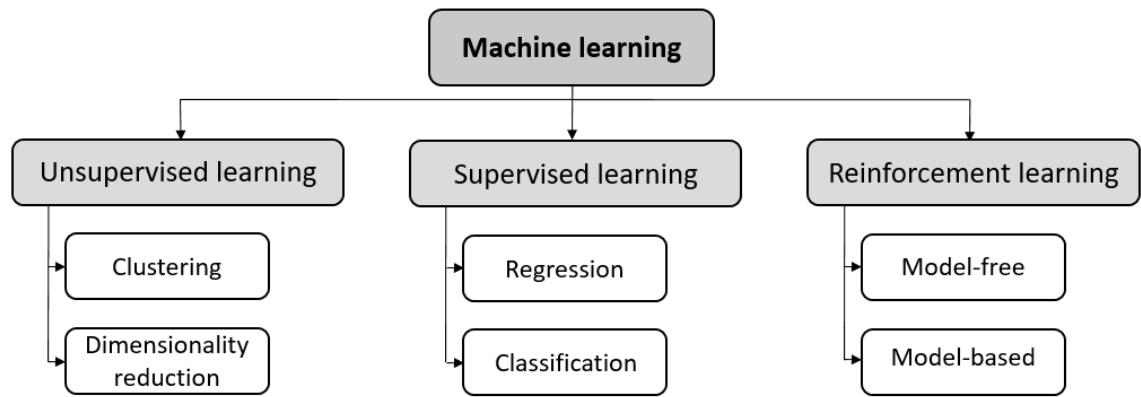
### 3 LEARNING-BASED CONTROL PARADIGMS

Learning accumulates from the fundamental idea of interacting with the surrounding environment, developing conception of the consequences of actions, and how to minimize or maximize the achievable goals [4]. Human way of learning follows these steps. In a similar manner, a robot can be instructed to approach goal-directed learning, but instead of doing this instinctively a machine has to rely on computational methods.

Learning-based paradigms are often related to machine learning domain, where learning refers to a model making predictions on new, unseen data based on either available training data or from data gained from the environment during training. The model itself is fitted to a problem and it is based on some abstract principle of the data related to the problem domain. Learning-based methods can be also found from robotics domain, where processes comparable to learning are traditionally based on feedback control and control methods. As an example, couple of known learning based control methods are repetitive control [15], model predictive control [16] and iterative learning control [17]. Although, these methods utilize tracking error minimization, it is nontrivial to incorporate this kind of optimization for any generic nonlinear system [18]. However, machine learning approach can be used to overcome mentioned issues. This thesis concentrates on demonstrating learning from the point of view of machine learning, hence future references to learning-based paradigms are referring to machine learning paradigms. Machine learning is approached as a unifying discipline of data science fields with a focus on presenting the paradigms especially from the point of view of robotics.

#### 3.1 Machine learning paradigms

Machine learning depends on data to perform results. In addition, the expectations of the results depend on the correct selection of algorithms. Therefore, the quality of the results is affected by the available and collected data, used algorithm, but also the used paradigm. Generally, machine learning and its functionalities have been divided into three paradigms as depicted in Figure 3.1.



**Figure 3.1.** Overview of main machine learning paradigms and methods, modified from [19].

The machine learning paradigms consist of supervisor learning with labelled data either with classification or regression method, unsupervised learning with unlabelled data either with clustering or dimensionality reduction method, and finally of reinforcement learning where algorithm processes unlabelled data and learns either with value or model-based method. All the presented methods and paradigms have in common that they learn from data and produce predictions. This commonality propagates from numerous scientific disciplines such as pattern recognition, data mining, computer vision and statistical signal processing that have come to comprise the term machine learning. What machine learning is used for is to develop a predictive model that tries to discover underlying structures and regularity patterns from the data. Basically, the process is to learn from the data with the created model [20].

A more modern approach to traditional fitting model to describe observable data, is to utilize algorithmic modelling where no explicit model is necessarily specified, but the machine identifies associations in the observed data and creates the model independently. The approach is called deep learning (DL), which is a subfield of machine learning capable of managing higher number of free parameters and complex model architectures with neural networks. Used datasets are often large and incomprehensible to human. The approach has led to many breakthroughs in a variety of applications where describing models was unattainable in the past, because of lack of sufficient computing power. Although, the complexity of the generated model may pose interpretation problems, models being sometimes called a “black box”, owing to the algorithmic transparency, and the amount and correlations of free parameters, significant progress have been made over the recent years to understand the processes of the generated model structures [19], [21]. In robotics domain, DL has been utilized in many fields: natural language processing, perception and motion control, to name a few [22].

From Sim2Real perspective the most utilized machine learning paradigm is reinforcement learning. Where reinforcement learning methods have shown success in the domain of robot control with low dimensional state-action spaces with limitations in continuous cases [23], deep reinforcement learning (DRL) methods with deep neural networks have been shown potential in high-dimensional and continuous action-state spaces [24]. Occasionally in the literature DRL is referenced just as reinforcement learning.

### 3.2 Reinforcement Learning

In reinforcement learning the object is to develop a model that can learn to take optimal actions in a specified environment. The optimal action is defined as an action that maximizes the expected reward over a specific time period. Reinforcement learning may utilize software agents that produce actions while interacting with the environment. In principle, reinforcement learning separates the learning environment or the real-world environment into two main components: an environment and an agent. Agents are tasked to try different strategies to discover the best approach for the given task, which is achieved by maximizing the cumulative reward through interactions with the environment through mapping of states to actions. While conducting interactions with the environment during numerous steps, the algorithm learns from the feedback from the environment. Feedback is not the ground truth, but a measure of how well the performed action was conducted by a reward function in the environment. [4]

The agent learns to perform more in line with the targeted outcome by trying to achieve more positive rewards from the environment. Reinforcement learning algorithms differ from one another according to the decision-making processes, policies, and used reward functions. Yet, the principle of exploratory trial-and-error planning approach is the same within the paradigm. The overall outcome to learn for a reinforcement learning is the dividing factor that branches algorithms into their own sub-classes. Usually learning includes stochastic or deterministic policies, state-value functions, action-value functions and/or environment models. [4]

Compared to supervised and unsupervised machine learning paradigms, reinforcement learning is more focused on goal-directed learning from interaction. Difference between reinforcement learning and supervised learning is that the objective of supervised learning is for the system to generalize or extrapolate generated responses to act correctly in situations not present in the training dataset. This kind of a process is not proficient for learning from interaction, since most of the time it is impractical to collect a labelled training dataset of desired behaviour that is applicable to all situations in which the agent is acting. Instead, in reinforcement learning the agent learns from its own experience and



generally without labelled datasets. This practice is similar to unsupervised learning which did not rely on previously known examples of correct behaviour as its data source. However, unsupervised learning, which aims to discover hidden structures within the data, does not address the reinforcement learning problem of maximizing a reward signal. Yet, both supervised and unsupervised learning can be utilized as part of reinforcement learning if seen fit to address the given problem [25], [26].

Depending on the research domain and the approach on the topic, there are different ways to express the taxonomy and categories of reinforcement learning [27], [1]. This thesis takes the approach specified in Figure 3.1 by making the distinction between model-based and model free methods. To understand the structure and difference of the aforementioned methods, first the general idea of learning within reinforcement learning is specified.

### 3.3 The Markov Decision Process

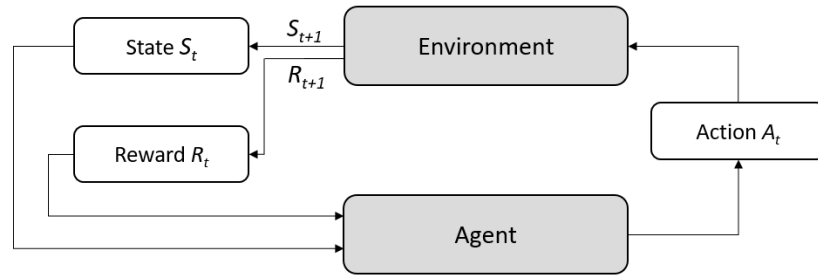
Markov Decision Process (MDP) involves sequential decision-making optimization, which is one of the key challenges in developing machine learning based algorithms. The MDP is a mathematical framework based on goal-directed learning from interaction, where the agent and environment are the basic components. The agent has explicit goals to achieve via choosing actions to influence in the environments. As the agent performs actions  $A_t$  it is learning a policy  $\pi$ , which defines how agent behaves at a given time. In general terms, a policy is a set of rules or network weights which are used for mapping specific actions to perceived discrete states as the agent confronts those states. A state  $S_t$  refers to the current status of the agent within the environment. One could indicate, a state is at the core of reinforcement learning paradigm as it is the input to the state-value and action-value functions. The environment reacts to the taken action, which changes the state of the environment and produces a reward  $R_t$ . Therefore, a state as an input to the agent and as an output from the environment refer to the changing status of the environment model as the agent performs actions. Change of the state causes the environment to return to the agent an observation of the updated state,  $S_{t+1}$ , and the reward,  $R_{t+1}$ , determined according to the quality of the performed action.

Each state is associated with a positive or negative reward. Reward type incorporates how the agent has managed from the perspective of the overall goal. However, even though the optimal action in the current state might result in a negative immediate reward, the action would result to significant positive rewards afterwards in the coming states. Or vice versa, if the action is not optimal, it could lead to a significant immediate reward, but low rewards in the later steps. Hence, the problem of achieving the goal of

learning a policy that maximizes the received rewards over all the steps of an episode, is captured in choosing the correct action in any given step. An episode is a sequence of states, actions, and rewards over all the steps:

$$Ep = (S_0, A_0, R_0, S_1, A_1, R_1, \dots) \quad (3.1)$$

Updated state and reward guide the agent to perform the next action. The process of learning a policy is thereby a feedback loop between the agent and the environment, which guides the improvement of the agent [1]. The agent-environment interaction process is depicted in Figure 3.2.



**Figure 3.2.** The basic structure of reinforcement learning paradigm as the agent-environment interaction process. Modified from [4].

The learning process continues until the expected return of rewards is maximized, which happens either after a specific threshold value or a terminal state is reached. The time between start and terminal step defines a time period that can be described as subsequences within the learning process, or more commonly as episodes. Each episode is followed by a reset to a specified starting state, which is usually a standard for the environment or a sample from a standard set of starting states. Also, each episode begins independently from others and is unaffected by how the previous episodes ended. Hence, each episode has the possibility to get different rewards as they may have different outcomes. For instance, in robotic domain the episode could end if the manipulator has reached a target and the indicative cumulative reward of states is higher than a threshold value. On the other hand, episode could end as an arbitrary limit for the maximum number of time steps have been reached, since the simulated or real manipulator has not managed to accomplish the given task and cumulate high enough reward. It is also possible that episode does not have a pre-planned fixed ending, but the episode ends as the time steps run out.

The MDP promotes that any goal-directed behaviour can be expressed with the aforementioned action, state and reward signals influencing between agent and its environment. However, as a framework, the MDP only defines the status of these components. To solve the MDP for an optimal policy that maximizes the received reward over the

episode, reward and state transition functions in addition to state-value and action-value functions need to be explored.

### 3.3.1 The reward function and the goal

A reward function is typically associated with the environment, which is for the MDP a function that maps the current state and action, and future state to a real value. Thereby, a reward function can be defined as  $R : S \times A \times S \rightarrow \mathbb{R}$ . However, in some cases the reward function depends only on the current state and is defined as  $R : S \times A \rightarrow \mathbb{R}$ . How the reward function is defined, depends on the problem and modelling choice [4]. The first one mentions each component explicitly, while the latter implicitly denotes the sum of regards weighted by a specific probability. The reward function is tied to the aforementioned state transition of the environment, so that if the transition is deterministic or stochastic by nature so is the reward function. This means for a given state, action and future state tuple, the reward function can output always the same deterministic value, or a random stochastic value.

A well designed reward function is crucial to successfully implement reinforcement learning [1]. A reward function provides feedback from the environment to the agent, according to the environment status at every time step in the form of an immediate reward  $R_t$ . The reward is the observed numerical value at each time step  $R_t \in \mathbb{R}$ , when the environment performs a state transition given an action. However, as mentioned in the previous Section 3.3, choosing a ‘correct’ action is balancing between immediate and future rewards.

A solution for estimating future expected rewards in the current state, is the utilization of discount factor  $\gamma$ . With discounting, the agent is guided to select actions that will produce maximized sum of the discounted future rewards. The expected discounted return can be defined as:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots + \gamma^{T-1} R_T = \sum_{t=0}^{T-1} R_{t+1} \gamma^t \quad (3.2)$$

where  $\gamma \in [0, 1]$  is the discount parameter and  $R_t$  is the immediate reward at time  $t$ . The discount parameter depends of the problem and is tuned accordingly for optimal results [1]. The discounted return is a weighted sum of the return, meaning the further away the reward is in the future, the lower its value is in the current state. Therefore, a reward received at a time step  $t$  in the future is now only worth  $\gamma^{t-1}$  times what it would if received in the current step. Rewards from closer time steps are given more appreciation. Though,

as the discount parameter approaches value 1, future rewards are being considered more effectively.

The context of a goal and a reward are bridged together by the reward function, which can be seen as a quantitative form of the goal. The goal is the target for which reinforcement learning is being utilized, and the reward function is manually designed to align with the goal in order for the agent to learn from reward feedbacks. Therefore, the goal and the reward function are to be seen as different entities. If the final learned policy provides different results that were expected after achieving the final goal, it is possible the agent has overfitted to the reward function. A plausible reason for this is the divergence of the reward function and the final goal. To avoid the divergence, the reward function should be designed to be consistent with the goal and cover all corner cases as well. [1]

The reward function is often unknown to the agent, which forces the agent to learn through trial-and-error by observing the environment while performing actions. The received reward feedback enforces the agent to learn the ideal behaviour. The reward function can be sparse or dense in regards of producing reward feedbacks. A sparse reward function is occasionally defined as a binary value being zero in most of its domain, and providing positive values, such as 1, only in a few state transitions. In an environment with a sparse reward function, the agent needs to take many actions before achieving reward feedback on its actions. This may slow down the learning, because without feedback the agent cannot qualify its actions to be good or bad. Regarding the reaching task in this thesis, a sparse reward function for a manipulator could be zero if the Euclidean distance between the end effector and the goal position is greater than the required precision  $\varepsilon_T$ . A successful action is rewarded by 1 if the distance of the end effector is less than the required precision. This can be defined as:

$$R = \begin{cases} 0, & \varepsilon_T < \text{dist}(ee, goal) \\ 1, & \varepsilon_T \geq \text{dist}(ee, goal), \end{cases} \quad (3.3)$$

where *ee* refers to the pose of the end-effector and *goal* to the pose of the reachable target in cartesian space.

A dense reward function refers to a function that provides rewards in most of the state transitions. In an environment with a dense reward function, the agent receives feedback at almost every time step, making the learning process graduate faster. The agent starts to learn to distinguish good and bad actions from the beginning. For a robotic reach target task an example of a dense reward function could be similar to the example of the aforementioned sparse reward being a Euclidean distance between the end-effector and the goal pose that is evaluated. If the pose of the end-effector is smaller than the required

precision  $\varepsilon_T$  then the action is considered as success and the set reward is 1. Otherwise, the set reward penalizes the distance. This can be defined as:

$$R = \begin{cases} -dist(ee, goal), & \varepsilon_T < dist(ee, goal) \\ 1, & \varepsilon_T \geq dist(ee, goal). \end{cases} \quad (3.4)$$

The learning process can be further accelerated by augmenting the reward function with heuristics. These are for guiding the agent to perform actions in a specific way that are inline with the targeted goal. For a robotic reaching task, heuristics could include that the end-effector is approaching the target in a specific angle, the pose of the end-effector is specific and/or the joint angles of the manipulator are within specific thresholds.

The presented expected return  $G_t$  and the reward function  $R$  are important for the learning of the agent and for it to get feedback on its actions. However, the actual evaluation of actions is implemented with the state-value and action-value functions that follow specific policies.

### 3.3.2 Formal definition of the MDP

The formal definition of the MDP can be modelled as a tuple  $\{S, A, P, R, \gamma\}$ .  $S$  represents a finite set of states, and  $A$  is a finite set of actions. At each time step the agent observes a state  $s_t \in S$  and performs a specific action  $a_t \in A$ , hence  $S$  and  $A$  denote the state space and action space of the environment.  $P$  represents a (state) transition probability function, which is the system dynamics by stating the transition probability of the environment from state  $s$  to  $s'$  when the agent performs an action  $a$ :

$$P = P_{ss'a} = p(s'|s, a) = \Pr[S_{t+1} = s' | S_t = s, A_t = a]. \quad (3.5)$$

When the action is performed, the environment returns the next state  $s_{t+1}$  according to the transition function, and an associated immediate reward is produced by the reward function  $R_t = R(s_t, a_t, s_{t+1})$ , which was presented in the previous Section 3.3.1. The final parameter of the tuple is the discount parameter  $\gamma$ . If all the MDP tuple parameters are known, it is possible to train the agent without interaction with the environment i.e. policy iteration or value iteration [1]. However, usually the reward function and transition probability function are left unknown to the agent. The way how the agent decides to perform specific actions to receive maximized cumulative rewards in return, is defined according to a policy  $\pi : S \rightarrow p(A)$  that directs state-value and action-value functions.

The state-value and action-value functions are a common aspect of each RL algorithm and how they are estimated. Estimation of the functions involve exploring whether specific actions are worthy to be performed in the given state or how benefactory for the agent is to be in the given state. The comparison of states is defined in terms of expected

future rewards. More precisely, the state-value function describes the value of being in a state  $S_t$ , where the value is the expected sum of discounted future rewards if the agent starts performing actions from that state. What actions the agent starts to perform affect the rewards the agent can expect to receive in the future. Performed actions that lead to specific future rewards follow policies, which define value functions. A policy  $\pi$  implements mapping the probabilities of selecting specific actions to perceived states of the environment when being in those states. A policy is at the core of the agent and hence the learning process, since policy determines behaviour which guides agent to evaluate actions. Accordingly, as policy comprises of suggested actions the agent should take at possible states it is also linked to the reward function. A state-value function of a state  $s$  under a policy  $\pi$  at a time step  $t$  that represents the expected return of  $R_{t+1}$  is denoted as

$$v_{\pi}(s) \doteq \mathbb{E}_{\pi}[G_t | S_t = s] = \mathbb{E}_{\pi} \left[ \sum_{t=0}^{T-1} R_{t+1} \gamma^t \mid S_t = s \right], \text{ for all } s \in S \quad (3.6)$$

$\mathbb{E}_{\pi}$  explains the expected value of the sum of discounted future rewards  $G_t$  starting from the current state. Basically, the episodes are sampled according to the policy, so that produced actions are conditioned on past states. While the value function is under a policy and there would be two different next states,  $S1$  and  $S2$ , to proceed from the current state, both of the next states would be estimated separately and usually the policy selects the state with the higher expected return. In principle, the state-value function determines the goodness of any given state for an agent who is following a policy  $\pi$ . The expected return calculation depends on the used reinforcement learning method, but the terminal state is always denoted as zero. A simple way to estimate the value function is to utilize Monte Carlo method [4],[1].

In a similar manner, the action-value function comprises of as being in a state  $s$  and taking an action  $a$  at a time step  $t$  and thereafter following a policy  $\pi$ . The action value function denotes an expected return under a state and an action as

$$q_{\pi}(s, a) \doteq \mathbb{E}_{\pi}[G_t | S_t = s, A_t = a] = \mathbb{E}_{\pi} \left[ \sum_{t=0}^{T-1} R_{t+1} \gamma^t \mid S_t = s, A_t = a \right]. \quad (3.7)$$

In principle, the action value function determines the goodness of the action taken by the agent from the given state for a policy  $\pi$ . The action-value function is also often referred to as the Q-function and the output of the action-value function is referred to as Q-value standing for quality. In reinforcement learning literature [27], [4], the core results are expressed with the Bellman equations that can be derived from the expressed action value function by rewriting it in a recursive form

$$q_{\pi}(s, a) = \mathbb{E}_{s' \sim \tau(\cdot|s, a)} \left[ R(s, a, s') + \gamma \mathbb{E}_{a' \sim \pi(\cdot|s')} [Q^{\pi}(s', a')] \right]. \quad (3.8)$$

For instance, optimal policy and value functions can be derived through the Bellman equations. Usually, the Bellman equations are solved with Monte Carlo method, dynamic programming or by temporal difference.

Both state-value and action-value functions are critical for the reinforcement learning algorithm to manage learning as they offer a way to compare actions and states in the evaluation of best results. An important aspect to note is that the selection of future actions in a state are solely based on the information the state can provide on the past agent-environment interactions. If the state does provide all the relevant information to find the optimal action, then the state is said to fulfil the Markov property [4]. In a case the state does not contain enough information, the learning process of the agent is likely to be compromised and the agent fails to learn the optimal policy. The general presentation of the optimal policy, which is the central optimization problem in reinforcement learning is defined as

$$\pi^* = \arg \max_{\pi} J(\pi), \quad (3.9)$$

where “\*” refers to ‘optimal’ and  $J(\pi)$  to the expected return of a specific measure. The optimal policy can be defined as a policy that maximizes the expected return of the action value function  $q_{\pi}(s, a)$  as:

$$\pi^* = \arg \max_{\pi} Q^{\pi}(s, a). \quad (3.10)$$

There is at least one optimal policy, which is better or equal than other policies [4].

In reinforcement learning, the learning process is considered to be based on trial-and-error. The agent observes the environment and learns through multiple trials-and-errors to maximize reward feedback. However, the optimal policy cannot be learned, and therefore the optimal reward achieved, if not all states have been explored. Agent cannot know which actions would produce better rewards than already explored actions, unless the agent tries to discover new unexplored actions from the actions space. Yet, in order to obtain a high cumulative reward, the agent prefers actions that it knows to be effective in producing high rewards. The trade-off of exploiting an action what is already known to be effective and exploring the unknown in an effort to come up with better actions and increasing existing knowledge is called exploration-exploitation dilemma. Neither exploration nor exploitation can be enthroned above the other without failing to achieve the maximum cumulative reward. The trade-off can lead the agent to fall into local value maximums if it only favours found actions. In contrast, if the agent only randomly explores

actions, it most likely does not achieve the best maximized reward. Because the exploration-exploitation trade-off is a well-known problem in reinforcement learning, numerous exploration policies have been developed and there are a variety of policies depending on whether the action space is discrete or continuous.

Discrete and continuous actions define the two types of actions in RL. Discrete action space includes a finite set of discrete actions for agent to perform [1]. As for continuous actions space, actions are expressed with known limits as a real-valued vector. Hence, the agent can perform any action between the known limits [28]. Common solution for discrete action spaces is the utilization of  $\epsilon$ -greedy policy, where the hyper parameter value  $\epsilon$  is chosen randomly and decayed over time to find a balance between exploration and exploitation [4]. Similarly, an example of exploration policy for continuous action space is the policy parameterization where a trained policy produces a mean and standard deviation as an output from which a value is sampled with Gaussian distribution to define an action [4]. Other worthy exploration policies are entropy cost term based policy [29] and adding noise to the greedy policy [30], to name a few. Since robot control is implemented in continuous action space, the scope of this thesis concentrates on those solutions and they are investigated more thoroughly in Section 3.6.

As the agent takes actions in the action space of the environment, the interaction produces information to the agent in the form of rewards. Exploring the environment is a challenging task for the agent if the environment is non-stationary, the action space is large or if the rewards are sparsely available. As mentioned above, learning in the environment is based on the Bellman equations consisting of state-value and action-value functions. These equations can be solved with dynamic programming, Monte Carlo method, or temporal difference method [4]. The reason why these functions need to be solved, is to be able to estimate mentioned functions for a particular policy. Estimation enables the agent to choose an action that will produce the best reward, after being in the given state. Although these methods can be combined in a variety of ways, and because of this interlinked relationship they all are meaningful for reinforcement learning, this thesis concentrates on temporal difference owing to the continuous actions space in robotics domain and the derived algorithms that come from it.

### 3.3.3 Temporal difference learning

The concept of temporal difference learning is one of the central developments to indicate progress in reinforcement learning based algorithms [4]. Temporal difference is an intermediate form between dynamic programming and Monte Carlo Methods combining



ideas from both parties. Similar to the Monte Carlo method, temporal difference provides a solution for the prediction problem of future rewards, states and actions to be taken by not requiring full knowledge of the environment. The prediction problem is extended to the control problem of estimating a policy by introducing generalized policy iteration derived from the dynamic programming method. [31]

The estimation process in temporal difference utilizes a process called bootstrapping. Estimates of state and action value functions are being updated according to estimates of future rewards. If the state-value and action-value functions were to be solved without estimation, the state-action pair values could be updated not until the final reward would be received. After the final reward would be received, the path to the final state would be traced back to the initial state and each state-action value would be updated accordingly. Hence, the update process is done without waiting for the final reward.

In principle, the target value for the temporal difference update is formed from the observed reward and an estimated state value for the next state, expressed as  $R_{t+1} + \gamma V(S_{t+1})$ . This update can be calculated at each step. The difference or error between the target value and the estimated value can be defined in terms of error equation as  $E_t = V(S_t)^* - V(S_t)$ , where  $V(S_t)^*$  is the bootstrapped true value. The generalized error equation is then expressed as

$$E_t = V(S_t)^* - V(S_t) = (R_{t+1} + \gamma V(S_{t+1})) - V(S_t). \quad (3.11)$$

This difference at different time steps depicts learning what is achieved through temporal difference. If this difference is multiplied by a learning rate  $\alpha$ , which is a constant variable at each step, the update rule can be defined for a state value function as

$$V(S_t) \leftarrow V(S_t) + \alpha [R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]. \quad (3.12)$$

The presented method makes the update immediately as it receives the reward and the next state. It also only looks one-step ahead but can be extended to N-steps by extending the target value with discounted rewards. In principle, above equation presents a weighted average across different episodes. Yet, the principle stays the same as temporal difference methods learn directly from the interaction with the environment by utilizing bootstrapping.

The action selection in each step is accomplished according to a policy. The general idea of policies is to balance the trade-off between exploration and exploitation, as discussed in previous Section 3.3.2. This is managed by not always exploiting what has been already learned, which eases the agent to avoid local optima. Policy structures are divided into on-policy and off-policy evaluation. With on-policy algorithms, the agent evaluates

the same policy that is used to explore the environment, whereas off-policy algorithms have a separate sampling behaviour policy in addition to the target policy which is being learned and evaluated. Thereby off-policy algorithms can use the experience of other agents interacting with the environment as well to improve the policy. The general idea of off-line policy is that the agent can freely explore without its actions necessarily corresponding to the best learned policy. The behaviour policy operates by sampling all the actions, while the target policy is deterministic on its action choices. Hence, the target policy evaluates and improves its policy through the exploration of the behaviour policy. These policy evaluation algorithms can be based on other methods such as Monte Carlo. However, temporal difference based algorithms are often favoured owing to their lack of a environment model to learn, their capability to evaluate and learn a policy at every step with online learning, and their lower variance [1], [32].

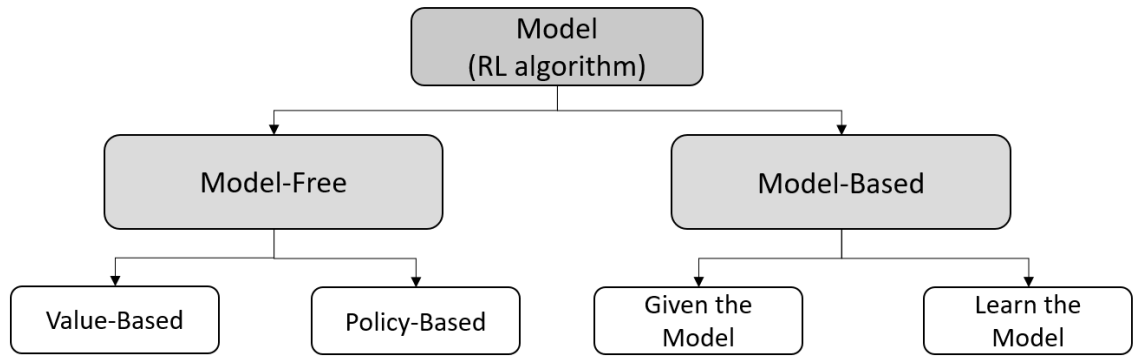
One of the most common on-policy algorithms is Sarsa, which selects an action based on the current policy. After executing the action, generated data is used to update the current policy. From the off-policy side comes one of the most notable breakthroughs in reinforcement learning, an off-policy temporal difference control algorithm known as Q-learning [33]. Q-learning uses  $\epsilon$ -greedy policy, which was discussed in previous explained Section 3.3.2, for selecting actions and a behavioural policy for yielding maximum Q-value. Therefore, the policy that interacts with the environment and the updated policy are not the same policy as Q-learning updates Q-values without making predictions about the actual target policy.

Decision to use a specific policy depends on the context and the quality of data and the available model. The same applies to choosing a reinforcement learning algorithm in general. Yet, with temporal difference the Bellman equations can be solved by estimating the state-value function of the MDP. However, the MDP itself can be expressed either with model-based or model-free methods.

### **3.4 Model-based and model-free methods**

The taxonomy in reinforcement learning can be expressed by separating the concept of a model into model-based and model-free methods. In reinforcement learning, the model refers to either a pre-trained model with initialized parameters or a model such as a neural network that has learned specific parameters. However, depending on the academic source a model does not necessarily mean a neural network or other statistical learning model integrated with the agent, but whether the agent uses predictions of the environment during learning. Predictions can be provided for the agent by an outside source

such as an external algorithm that is designed to understand and uphold the environment, or the agent can learn these predictions making them into approximations. In this thesis, a model is referred to as a function which predicts state transitions and rewards. In addition, the difference between the model-based and model-free method is highlighted by whether the agent has access to or must learn a model of the environment such as the reward and transition functions. Despite reinforcement learning algorithms are occasionally modular, and the terminology and notation may differ between academic sources, the taxonomy of reinforcement learning models can be presented as depicted in Figure 3.3.



**Figure 3.3.** The taxonomy of reinforcement learning methods. Modified from [1].

In model-based method the learning process is achieved by predicting the elements of the environment, although the reward and state transition functions are not known by the agent. Yet, if the agent can collect data samples of the state, action, next state and rewards  $\{S_t, A_t, S_{t+1}, R_t\}$  while performing actions in the environment, it is possible to predict the values of the reward  $R$  and state transition functions  $P$ . These samples are collected by constantly interacting with the environment. After all the elements are known, the agent can plan its actions and locate optimal cumulative reward. The model can be either given or learned. Therefore, the agent either tries to learn the model or tries to refine the given one. In the aforementioned case where the agent has to learn the model, the environment is too complex to form a model that could be given. On the other hand, if the model is given, then the agent has direct access to the models of the reward function and transition function. These are then used by the agent to evaluate and improve its policy and make cognitive predictions of the future returns. [5]

Another solution to approach learning, is to use model-free method by not concentrating on developing the model of the environment, but to directly try to discover the optimal policy. The model of the environment here refers to the dynamics of the environment i.e., reward and state transition functions. The procedure is similar to model-based learning with learning the model as the agent interacts with the environment, but progressively

acquires cached estimates of the long-turn rewards and actions built upon past experience. Value-based method chooses actions with the highest value from the action-value function and converges the result, hence optimizing the action-value function. This enables deriving the optimal policy. Another way is the policy-based method, which iteratively updates the policy until the accumulated return of rewards is maximized. In principle, model-free method either estimates the action-value function or the policy from previous experience on interactions between the agent and the environment, without considering reward or state transition functions. For instance, the previously presented temporal difference forms a category of model-free methods. [4], [5]

The decision on which method to use depends on the model of the environment, whether the agent has access to the dynamics of the environment and whether the action space is continuous or discrete. In model-based methods the agent can execute better planning since future rewards and states can be anticipated owing to the model of the environment. The agent can explicitly decide between options and see beforehand the outcomes of possible choices. The results of planned choices are then conjoined into a learned policy. The downside of the model-based methods is that the ground-truth model of the environment is not usually available, or the dynamics of the environment are too complex to be represented explicitly. In this case, the agent has to learn the model from experience, which may induce bias in the estimation process. The agent exploits this bias resulting in results that perform well according to the learned model, but inaccurately in a real environment. On the other hand, model-free methods are easy to implement, tune and handle dynamical systems with minimal bias since they do not consider the model of the environment [6]. This comes at the price of potential gains in sample efficiency, meaning model-free methods have to explore more intensively to gain experience upon which to develop the policy. Also, the resources required for the exploring in a real-world environment can be high and the exploration can cause safety risks. [4]

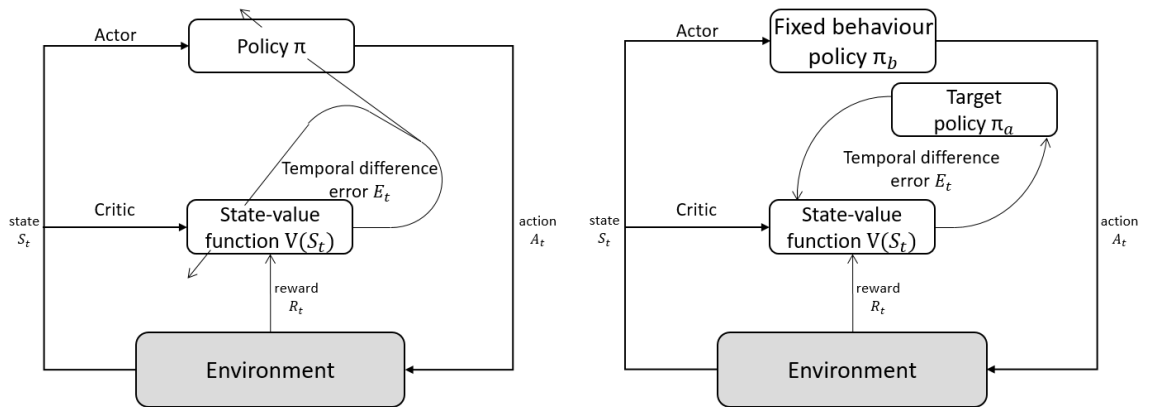
In addition to aforementioned methods, there are methods that combine the characteristics of different learning processes. This thesis considers one of the most popular methods, which is the interpolation of the value-based and policy-based methods from model-free method: the actor-critic framework. Algorithms based on this framework are later implemented in the empirical part of the thesis.

### **3.5 Actor-critic framework**

The Actor-critic method is a combination of value-based and policy-based method by separating the policy and the evaluation of actions into two separate entities. Merits of value-based methods are used to learn the state value function for the sake of sample

efficiency, and policy-based methods are used for learning the policy either for discrete or continuous action space. Basically, actor-critic methods are inherently temporal difference methods with a separate memory structure which enables representing the policy independent of the state-value function. Policy iteration is taken care of with online approximations where state-value parameters are estimated using temporal difference methodology. The policy parameters are updated by a stochastic gradient descent [34]. This framework of the actor-critic method is of special interest in cases which involve large or infinite state spaces in addition of being continuous.

Actor-critic algorithms manage updating estimation and policy parameters in different ways, but the basic structure of the actor-critic framework is based on simultaneous online estimation of the actor and critic parameters [34]. The actor is responsible of learning a conventional action-selection policy by mapping states to actions according to probabilities. In principle, actor is used for selecting actions. The critic represents a conventional state-value function and is responsible of evaluating the policy by mapping states to expected future rewards. The evaluation is accomplished by criticizing the actions made by the actor in order to evaluate the new state and determine the quality of progress. If the critic is estimating the action-value function  $Q(S_t, A_t)$ , it also requires the output of the actor. In other words, the actor is concerned with the control and the critic is addressing the prediction. The optimal policy is discovered by solving the control and prediction problems by the critic giving feedback to the actor, which updates its policy accordingly. Then the critic updates its estimates according to the estimation error of temporal difference learning, Equation 3.11, in which estimates are updated with bootstrapping, discussed in Section 3.3.3. Hence, the output of the critic is responsible of learning in both the actor and the critic. Conventional actor-critic frameworks for on-policy and off-policy are depicted in Figure 3.4.



**Figure 3.4.** Actor-critic framework with on-policy (left) and off-policy (right), modified from [4], [35].

The critic, that is extended from the temporal difference, utilizes bootstrapping with Equation 3.12 on estimates of future state-values to estimate the Q-value, see Equation 3.8. The output of the critic is the update to the policy, which can be approached depending whether the actor-critic method is on-policy or off-policy based as was introduced with temporal difference in Section 3.3.3. On-policy algorithms update the policy and state-value estimates according to the current policy, whereas off-policy algorithms are not dependable on the current policy but use the optimal learned policy by the target policy. Here optimal policy refers to a policy or policies different from the one the agent is executing, which could be a random exploration product by the behavioural policy [35]. Additionally, off-policy algorithms can utilize experience replay buffer to store the experience of the agent. This memory includes the basic MDP interaction parameters as a tuple  $\{S_t, A_t, S_{t+1}, R_t\}$ , which can be utilized in future iterations to improve sample efficiency in state-value function estimations by the critic. On-policy algorithms do not require a replay buffer, since they do not reuse experience sampled from an old policy to estimate the policy gradient.

Although actor-critic methods have the advantages of the value-based and policy-based methods, they inherit the disadvantages as well. The actor is inclined to suffer from insufficient exploration, and the critic suffers from overestimation. These obstacles have been confronted with a variety of algorithms both on on-policy and off-policy based [29], [36], [4]. This thesis is interested on algorithms based on actor-critic framework that can work on continuous action space. This also brings forward the way how policy gradients are dealt with, meaning the scope of the thesis is extended to cover gradient-based optimization.

### 3.6 Reinforcement algorithms for continuous action spaces

Continuous action spaces were for a long time an unsolved problem for the reinforcement learning community. After the advantages made among deep neural networks, namely deep Q-networks (DQN) [37], the research started to concentrate on solving high-dimensional problems with DRL. DQN works with low-dimensional problems in a discrete space, so the next step was to look towards the continuous space. The problem of continuous action spaces is especially relevant considering this thesis, since a variety of characteristics in robotics domain, such as control, are continuous.

### Stochastic policy gradient

The challenge of training a policy in a continuous action space is traditionally cornered with a stochastic policy gradient by representing the policy with a parametric probability distribution

$$\pi_{\theta}(a_t|s_t) = \mathbb{P}[a_t|s_t; \theta], \quad (3.13)$$

where action  $a$  in a state  $s$  is selected stochastically with parameter vector  $\theta$ . The stochastic policy  $\pi_{\theta}$  is then sampled, and the policy parameters adjusted in order to achieve the best cumulative reward. In principle, with stochastic policy for each state in order to take an action a sample of possible actions is taken, and the decision is based on the distribution. Because stochastic policy gradient assumes the policy is a distribution, the algorithm integrates over both action and state spaces to get the gradient of the cumulative reward.

The aforementioned policy gradients are performing under the idea that policy parameters  $\theta$  are adjusted by solving the performance gradient of the policy gradient theorem [4]. The theorem defines the gradient over the cumulative reward (reward function [4]) as

$$\begin{aligned} \nabla_{\theta} J(\pi_{\theta}) &= \int_S \rho^{\pi}(s) \int_A \nabla_{\theta} \pi_{\theta}(a|s) Q^{\pi}(s,a) da ds \\ &= \mathbb{E}_{s \sim \rho^{\pi}, a \sim \pi_{\theta}} [\nabla_{\theta} \log \pi_{\theta}(a|s) Q^{\pi}(s,a)]. \end{aligned} \quad (3.14)$$

The theorem states how the policy optimization is carried out on a policy  $\pi$  with respect to a goal or policy performance metric  $J(\pi_{\theta})$  through a gradient in continuous space. The theorem is applicable for both stochastic  $\pi$  and deterministic policies  $\mu$ , and they both maximize the cumulative reward through estimating repeatably the gradient. When the stochastic gradient of  $J(\pi_{\theta})$  is calculated with respect to the policy parameters  $\theta$ , these parameters can be updated accordingly

$$\theta = \theta + \alpha \nabla_{\theta} J(\pi_{\theta}) \quad (3.15)$$

where  $\alpha$  represents the learning rate. Above Equation 3.15 is known as gradient ascent and results in a policy that will provide higher rewards after each episode finally resulting in a maximized reward.

### Deterministic policy gradient

However, after the introduction of Deterministic Policy Gradient (DPG) [38], it was shown that sample efficiency can be increased by only integrating over the state space. DPG models the policy according to a deterministic decision. Mapping is done deterministically

from state to action, and the result is an action instead of probability. This action selection is defined as

$$a_t = \mu(s_t | \theta^\mu) + \sigma_t, \quad (3.16)$$

where  $\mu(s_t)$  refers to the deterministic policy similar to Equation 3.13 with stochastic policy function.  $\theta^\mu$  represents the deterministic policy parameter. Distinction is made to remove the ambiguity between the policies. Because deterministic policies do not cover exploration over the action space, noise  $\sigma$  is added to the exploration [38].

For deterministic policies the gradient over the reward function is defined through deterministic policy gradient theorem [38]. The theorem defines the gradient over the reward function as

$$\nabla_{\theta} J(\mu_{\theta}) = \int_S \rho^{\mu}(s) \nabla_{\theta} \mu_{\theta}(s) \nabla_a Q^{\mu}(s, a) |_{a=\mu_{\theta}(s)} ds = \mathbb{E}_{s \sim \rho^{\mu}} [\nabla_{\theta} \mu_{\theta}(s) \nabla_a Q^{\mu}(s, a) |_{a=\mu_{\theta}(s)}], \quad (3.17)$$

where the discounted state distribution  $\rho^{\mu}(s)$  is defined analogously to the stochastic policy gradient in Equation 3.14. The defined deterministic gradient enables to update policy parameters to the direction of the gradient.

The policy gradient methods are advantageous in a variety of real world applications, namely robotics domain [39], [40]. The robotics domain has traditionally used deterministic model-based methods for acquiring the policy gradient, though for a complex environment it is extremely difficult to produce a detailed model that includes the simplest details. The case is emphasized with an autonomous system to work adaptively in a changing environment. Hence, the policy gradient is to be estimated with a model-free method by generating the estimation from data collected during the task execution. This thesis concentrates on two specific algorithms that are build off the deterministic policy gradient, and which are commonly utilized within the robotic domain owing to being off-policy and model-free: Deep Deterministic Policy Gradient and Soft Actor-Critic.

### 3.6.1 Deep Deterministic Policy Gradient

Deep deterministic policy gradient (DDPG) is one of the most known deterministic gradient policy-based algorithms having a stochastic behaviour policy, but a deterministic target policy [28]. Stochastic policy is used for enhanced exploration and deterministic policy is easier for the agent to learn. With two different policies the DDPG utilizes in its core an off-policy method, that formulates action predictions for the current state and generates a temporal-difference error signal at each time step. Hence, the DDPG utilizes also an actor-critic framework.



The actor network depicted as  $\mu$  takes the current state as its input and outputs a real-valued action chosen from a continuous action space by establishing a policy function as  $\mu(s_t|\theta^\mu)$ . Here actor network is a deterministic policy network. The critic network depicted as  $Q$  utilizes the Q-function, which is updated with the temporal difference error, and also takes the current state as an input. As an output the critic produces an estimated Q-value with  $Q(s, a|\theta^Q)$ . Both actor  $\mu$  and critic  $Q$  networks have target networks  $\mu'$  and  $Q'$  respectively that are updated according to the original networks. Hence, DDPG utilizes altogether four networks. [28]

In principle, DDPG is an extension of DQN algorithm in continuous action space with a combination of actor-critic framework while concurrently learning a deterministic policy and a Q-function [1]. Both DDPG and DQN have the same Q-function as a critic, and the temporal difference is utilized to update the function. In both, the Q-function is learned using the Bellman equation, see Section 3.3.2 and Equation 3.8. The critic is updated with gradient descent by minimizing the loss between the current estimates and target values defined as

$$L = \frac{1}{N} \sum_i \left( Y_i - Q(s_i, A_i | \theta^Q) \right)^2. \quad (3.18)$$

This loss function represents the Mean Square Error (MSE), where  $Y_i$  is the desired output and an estimate of the Q-value as  $Y_i = R_i + \gamma Q'(S_{t+1}, \mu'(S_{t+1} | \theta^{\mu'})) | \theta^{Q'}$  produced with target actor  $\mu'$  and critic  $Q'$  networks.  $Q(s, a | \theta^Q)$  is the learned Q-function as a predicted output. The target is to minimize the outcome of loss function by updating the policy weights towards the direction where the loss decreases.

$N$  represents the batch size, which is drawn from the experience replay buffer, see Section 3.5. Use of replay buffer is derived from the off-policy nature of the DDPG, as it stores in memory the learned policy by the actor. This learned policy is then used as samples by the behaviour policy. The use of replay buffer eases the instability issues of the actor and critic networks. In addition, by using stochastic policy, such as  $\epsilon$ -greedy policy, as a behaviour policy, the DDPG enforces exploration and tries to confront the exploration-exploitation dilemma. Generally, exploration causes notable challenges for learning in the continuous action space, but DDPG can handle exploration separate from the learning algorithm. The pseudocode of The DDPG is presented in Algorithm 1.

**Algorithm 1:** Deep Deterministic Policy Gradient (DDPG) [28]

---

Randomly initialize actor  $\mu(s|\theta^\mu)$  and critic  $Q(s, a|\theta^Q)$  networks with parameters  $\theta^Q$  and  $\theta^\mu$ ;  
Initialize target actor  $Q'$  and target critic  $\mu'$  networks with parameters  $\theta^{Q'} \leftarrow \theta^Q$ ,  $\theta^{\mu'} \leftarrow \theta^\mu$ ;  
Initialize experience replay buffer  $\mathcal{D}$ ;  
**for** episode = 1, M **do**  
    Initialize Gaussian noise for action selection;  
    Receive initial observation state  $s_1$ ;  
    **for** time increment  $t = 1, T$  **do**  
        Select action  $a_t$  according to Equation 3.16;  
        Execute action  $a_t$  and observe reward  $r_t$  and next state  $s_{t+1}$ ;  
        Store the transition experience  $(s_t, a_t, r_t, s_{t+1})$  in experience replay buffer  $\mathcal{D}$ ;  
        Sample a batch of  $N$  transitions  $(s_t, a_t, r_t, s_{t+1})$  from replay buffer  $\mathcal{D}$ ;  
        Update critic by minimizing the loss according to Equation 3.18;  
        Update actor policy with sampled policy gradient of Equation 3.17:  

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_t \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_t} \nabla_a Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)};$$
  
        Update parameters of the target network according to Equation 3.20;  
    **end**  
**end**

---

At each time step  $T$ , the current action  $a_t$  is calculated with sampled noise  $\sigma$  to boost exploration during training process. In some implementations, such as in the Rllab framework [41], the additive noise is addressed with an independent noise model, such as OU noise [28]. After the critic is updated by minimizing the loss between the target and predicted Q-value with Equation 3.18, the policy function in the actor is updated by utilizing sampled policy gradient theorem defined in the Equation 3.17 with the critic output. Finally, similar to the DQN algorithm, DDPG uses target networks, but with a Polyak averaging [42]

$$\theta_{target} = \tau \theta_{training} + (1 - \tau) \theta_{target}. \quad (3.19)$$

Instead of replacing the full target network parameters or weights at every step, the weights are updated with exponential smoothing, so that the target network (old) slowly updates towards the main training network (new) which is under the evaluation. Both target networks are weighted with hyperparameter soft replacement factor  $\tau \in [0, 1]$ , which is the rate of copying weights. A small  $\tau$  value indicates slow and smooth transition for network weights, which also improves the stability of the learning process [1]. In DDPG the Equation 3.19 of updating target network parameters is presented with critic target  $\theta^{Q'}$ , critic  $\theta^Q$  and actor target  $\theta^{\mu'}$ , actor  $\theta^\mu$  networks as

$$\begin{aligned} \theta^{Q'} &\leftarrow \tau \theta^Q + (1 - \tau) \theta^{Q'} \\ \theta^{\mu'} &\leftarrow \tau \theta^\mu + (1 - \tau) \theta^{\mu'}. \end{aligned} \quad (3.20)$$

After the network and parameter updates, the process starts again for the next time increment or next episode.

Even though DDPG is popular among continuous action space problems, it has its issues. The actor-critic framework may introduce bias because of replacing the actual actor-critic function with the estimated one. In addition, DDPG suffers from brittle convergence properties, which makes the algorithm sensitive towards hyperparameters [41], [43]. Hyperparameters like exploration constants and learning rates have to be set separately for different problem domains to achieve expected results. To combat these shortcomings, an algorithm was developed that incorporates aspects of DDPG, but with a stochastic policy optimization. However, a notification to be made to highlight that algorithm selection depends on the application, data available and the environment.

### 3.6.2 Soft Actor-Critic

Soft Actor-Critic (SAC) is a model-free off-policy stochastic actor-critic RL algorithm designed to be applicable in real-world domains. SAC incorporates a maximum entropy framework, where the optimal policy aims to maximize its entropy augmented reward. Therefore, the policy is trained to maximize the trade-off between expected return and entropy, meaning SAC aims to maximize both cumulative returns and the entropy of the policy. By regularizing the standard maximum reward with an entropy maximization term, the process encourages exploration of the policy and substantially improves robustness [44].

Increasing entropy causes the agent to explore more, which can result in accelerating learning and preventing the policy to converge to a local optimum, because of an action that exploits inconsistencies in the approximated Q-function. Therefore, the policy is encouraged to consider equally actions that have similar Q-values, and thus to avoid assigning high probabilities to marginal range of actions. In principle, introduction of entropy implements randomness in the policy, which is then expected to act more randomly, but still succeed in the given task. Hence, the exploration-exploitation dilemma can be confronted. [44]

The relationship between the reward and the entropy modifies the RL problem of expected return, Equation 3.2, to include an additional reward at each timestep proportional to the entropy of the policy at that specific timestep. Hence, the expected return is defined as

$$J(\pi) = \sum_{t=0}^T \mathbb{E}_{(s_t, a_t) \sim \rho_{\pi}} [r(s_t, a_t) + \alpha \mathcal{H}(\pi(\cdot | s_t))], \quad (3.21)$$

where entropy term  $\mathcal{H}$  is weighted by a regularization coefficient  $\alpha > 1$  called temperature. The coefficient determines the stochasticity of the policy by being a trade-off between the entropy term and the reward. Instead of expressing the optimal policy as in Equation 3.10, with entropy augmented reinforcement learning the policy is defined as

$$\pi^* = \arg \max_{\pi} E_{\tau \sim \pi} \left[ \sum_{t=0}^{\infty} \gamma^t \left( R(s_t, a_t, s_{t+1}) + \alpha \mathcal{H}(\pi(\cdot | s_t)) \right) \right], \quad (3.22)$$

where  $\gamma$  is the discount term. SAC approaches optimization with soft policy iteration, which is a general algorithm for learning the optimal maximum entropy policies [45]. The algorithm consists of policy evaluation and policy improvement in the maximum entropy framework [44].

With soft policy iteration, SAC parameterizes a soft Q-function (action-value)  $Q$  by  $\theta$  as  $Q_{\theta}(s_t, a_t)$ , and a policy function  $\pi$  by  $\phi$  as  $\pi_{\phi}(a_t | s_t)$ . A separate function approximator for the value function  $V$  is seen unnecessary [45]. In practice, SAC concurrently learns a policy function and two Q-functions,  $\theta_1$  and  $\theta_2$ , that utilize two target functions,  $\bar{\theta}_1$  and  $\bar{\theta}_2$ , to mitigate biasness of the networks [45]. The function approximators assist in the convergence of optimal values i.e. policy, where alternative optimization of both  $Q$  and  $\pi$  networks is performed with stochastic gradient descent [44].

The soft Q-function is trained by minimizing the error between the target function and the approximation with MSE similar to Equation 3.18

$$J_Q(\theta) = \mathbb{E}_{(s_t, a_t) \sim \mathcal{D}} \left[ \frac{1}{2} \left( Q_{\theta}(s_t, a_t) - \hat{Q}(s_t, a_t) \right)^2 \right], \quad (3.23)$$

where  $\mathcal{D}$  is a replay buffer having the distribution of previously sampled states and actions, and the approximation is

$$\hat{Q}(s_t, a_t) = r(s_t, a_t) + \gamma \mathbb{E}_{s_{t+1} \sim p} [V_{\bar{\theta}}(s_{t+1})]. \quad (3.24)$$

The value function  $V_{\bar{\theta}}$  is implicitly parameterized by the soft Q-function parameters and is defined as

$$V_{\bar{\theta}}(s_t) = \mathbb{E}_{a_t \sim \pi} [Q_{\bar{\theta}}(s_t, a_t) - \log \pi_{\phi}(a_t | s_t)] \quad (3.25)$$

for a policy  $\pi$ . Here entropy is represented by the negative log of the policy function. Equation 3.23 states that the loss between the prediction of the Q-function and the immediate reward added the discounted expected state-value of the next state is minimized across all the state action pairs from the replay buffer. By further optimizing Equation 3.23 with stochastic gradients, the update rule for the  $\theta$  parameter vector is defined as

$$\hat{\nabla}_{\theta} J_Q(\theta) = \nabla_{\theta} Q_{\theta}(a_t | s_t) (Q_{\theta}(a_t | s_t) - r(s_t, a_t) + \gamma(Q_{\bar{\theta}}(s_{t+1} | a_{t+1}) - \alpha \log \pi_{\phi}(a_{t+1} | s_{t+1}))). \quad (3.26)$$

The update makes use of target network  $Q_{\bar{\theta}}$  where parameter  $\bar{\theta}$  represents exponentially moving average of  $\theta$  [44].

For the policy function  $\pi$ , the parameters are learned by minimizing the expected Kullback-Leibler (KL) divergence [46]

$$J_{\pi}(\phi) = \mathbb{E}_{s_t \sim \mathcal{D}} \left[ D_{KL} \left( \pi_{\phi}(\cdot | s_t) \parallel \frac{\exp(Q_{\theta}(s_t, \cdot))}{Z_{\theta}(s_t)} \right) \right] \quad (3.27)$$

where  $D_{KL}$  is a KL function and  $Z(s) = \sum_a \exp(\frac{1}{\alpha} Q(s, a))$  is the normalizing factor. The policy function distribution is made to look more like the distribution of the exponentiation of the state value function normalized by the normalization factor. By ignoring the log-partition and instead noting the temperature coefficient  $\alpha$ , the policy parameters are defined as

$$J_{\pi}(\phi) = \mathbb{E}_{s_t \sim \mathcal{D}} \left[ \mathbb{E}_{a_t \sim \pi} [\alpha \log \pi_{\phi}(a_t | s_t) - Q_{\bar{\theta}}(s_t, a_t)] \right] \quad (3.28)$$

The minimization of the object function  $J_{\pi}(\phi)$  can be approached with a variety of ways, but with SAC this is done with a reparameterization trick to make the action sampling from the policy a differentiable process

$$a_t = f_{\phi}(\epsilon_t; s_t), \quad (3.29)$$

where  $\epsilon_t$  is a noise vector sampled from a gaussian distribution. After reparametrizing the policy, optimizing with stochastic gradients, and noting how normalization factor does not depend on the parameter  $\phi$  the policy function update for the  $\phi$  parameter vector can be expressed as

$$\hat{\nabla}_{\phi} J_{\pi}(\phi) = \nabla_{\phi} \alpha \log \pi_{\phi}(a_t | s_t) + \left( \nabla_{a_t} \alpha \log \pi_{\phi}(a_t | s_t) - \nabla_{a_t} Q(s_t | a_t) \right) \nabla_{\phi} f_{\phi}(\epsilon_t; s_t) \quad (3.30)$$

Finally, after updating the Q-functions and the policy function, SAC enforces the entropy term by automatically updating the regularization coefficient  $\alpha$  with

$$J(\alpha) = \mathbb{E}_{a_t \sim \pi_{\phi}} [-\alpha \log \pi_{\phi}(a_t | s_t) - \alpha \mathcal{H}]. \quad (3.31)$$

Above Equation 3.31 is to constitute in finding the optimal entropy-constrained expected return. The pseudocode of the SAC algorithm is presented in Algorithm 2.

**Algorithm 2:** Soft Actor-Critic (SAC) [45], [1]

---

Randomly initialize two soft Q-function  $Q_\theta(s_t, a_t)$  and a policy function  $\pi_\phi(a_t|s_t)$  networks with parameters  $\theta_1, \theta_2$  and  $\phi$ ;  
Initialize target Q-function networks  $\bar{\theta}_1$  and  $\bar{\theta}_2$  with parameters  $\bar{\theta}_1 \leftarrow \theta_1, \bar{\theta}_2 \leftarrow \theta_2$ ;  
Initialize experience replay buffer  $\mathcal{D}$ ;  
**for** episode = 1, M **do**  
    Receive initial observation state  $s_1$ ;  
    **for** time increment  $t = 1, T$  **do**  
        Observe state  $s_t$  and sample action  $a_t \sim \pi_\phi(a_t|s_t)$  from the policy;  
        Execute action  $a_t$ , observe reward  $r_t$  and sample transition  $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$ ;  
        Store the transition experience  $(s_t, a_t, r_t, s_{t+1})$  in experience replay buffer  $\mathcal{D}$ ;  
        Sample a batch of transitions from  $\mathcal{D}$  to be used to update function approximators;  
        Update Q-function parameters  $\theta_i \leftarrow \theta_i - \lambda_Q * \text{Equation 3.26 for } i \in \{1, 2\}$ ;  
        Update policy parameters  $\phi \leftarrow \phi - \lambda_\pi * \text{Equation 3.30}$ ;  
        Update regularization coefficient (temperature)  $\alpha \leftarrow \alpha - \lambda_\alpha \widehat{V}_\alpha * \text{Equation 3.31}$ ;  
        Update target Q-function parameters  $\bar{\theta}_i \leftarrow \tau \bar{\theta}_i + (1 - \tau) \theta_i$  for  $i \in \{1, 2\}$ ;  
    **end**  
**end**

---

$\lambda$  represents a step size for the function approximators. SAC does not only collect experience from the environment with the current policy, but also updates the function approximators with the stochastic gradients from the samples saved into the replay buffer. Each step taken in the environment is followed by one gradient step. Because the update iteration is kept as one for the gradient step, update cycle for the function approximators is presented within the time increment (step) loop.

## 4 SIMULATION-TO-REAL LEARNING

Sim2Real refers to techniques to transfer knowledge from simulation to the real world. Although simulation has been widely utilized in testing and prototyping, it is only recently that simulation learned behaviours have been tried to transfer to the real world. Knowledge acquired during simulation is used to solve a real-world problem in a real-world environment. Similar paradigms simulation-to-simulation (Sim2Sim), real-to-simulation (Real2Sim) and real-to-real (Real2Real) describe techniques how knowledge can be transferred between simulations, from real-world to simulation and between real-world problem scenarios [47], [48], [2], [49], [50]. All the mentioned paradigms revolve around knowledge transfer and utilize learning-based methods.

However, translation of progress from simulation to a real-world physical robot is not trivial, as the number of details in the real world may overwhelm the learned capabilities of the simulated model. This is a noted factor in kinematics, where traditional control theory schemes such as adaptive [51] and robust control [52] have been used successfully for robotics since the late 1970s to accomplish manipulation tasks that machine learning based algorithms have, by comparison, only recently managed to accomplish.

### 4.1 Simulators for Sim2Real transfer

Robotic simulation and environment creation can be approached with a variety of simulation tools. Regardless of the utilized methods for transferring knowledge to physical robots, a realistic simulation can help to mitigate the reality gap and increase the quality of the results. Although realistic simulation alone does not mean the learned policy can be directly applied to the real environment, the choice of a relevant simulation for the task is a key aspect in Sim2Real.

Commonly mentioned and used simulators for robotics in the literature *include* CoppeliaSim (V-Rep) [53], MuJoCo [54], Gazebo [55], Unity 3D [56], OpenAI Gym [57] and PyBullet [58]. Aforementioned tools or platforms are designed for different applications with inherently different characteristics. CoppeliaSim and Unity 3D are physics simulators with rendering effects, whereas *Gazebo*, owing to the open-source status, offers modularity in addition to being integrated with the Robot Operating System (ROS) and thus having direct access to a variety of robotic stacks of real robots. MuJoCo and PyBullet manage realistic and accurate physics engines with relatively wide integration with ML

libraries. On the other hand, OpenAI Gym is specifically developed for RL and ML algorithms with simple environments.

Since RL and ML development is usually associated with Python, it is convenient that the chosen simulator has the support and relevant interfaces for the development work. Consequently, most of the simulators have Python counterparts [1], [59], [8].

## 4.2 Sim2Real transfer

Sim2Real can be seen as a sub-discipline within the robotics domain where simulators close the loop from simulation to real, having possibility to control and interact with physical robots. It is a tool to confront prediction problems in robotics while bridging the gaps between simulated and real-world environments. This reality gap can be considered as a confluence of robotics and simulation via machine learning. Hence, the importance of Sim2Real is the relationship of simulation models and the underlying real-world environments. Policies are trained in these simulations and learned policies are transferred to real-world scenarios. In principle, the policy is trained in a source domain within simulation and transferred to a new target domain in the real-world environment assuming different domains share common characteristics so that learned aspects in one domain prove to be useful in the other. Therefore, Sim2Real transfer is an alternative approach to the direct training of algorithms in real world with Real2Real. Though arguable, Sim2Real is taking place in order for simulation to coalesce into new domains. [2]

### 4.2.1 Relevant Examples

The rise of DRL techniques has benefitted robotics field not only on real-world experiments with real2real, but also on robotic tasks otherwise difficult to model explicitly. For instance, incorporating reinforcement learning with simulation has yielded interesting results on manipulation from visual inputs [60], control of complex motor systems [61] and locomotion on complex terrains [62]. However, transferring these control policies learned in simulation to the real-world is a challenging task. Therefore, while there has been accomplishments in attaining precise control of complex tasks, many of the demonstrations have been intentionally limited to simulation [63], [64]. Exemplary implementations of Sim2Real are depicted below in Figure 4.1.





**Figure 4.1.** *Sim2Real transfers for complex motor control with manipulator control (peg-in-the-hole) (left), dexterous hand control (middle) and locomotion (right). Modified from [7], [65] and [66].*

In the context of target-reaching tasks varying DRL algorithms have been used such as hierarchical reinforcement learning [67] and curriculum learning based hindsight experience replay (HER) [68]. HER-style algorithms provide a form of implicit curriculum improving efficiency by learning from fatal experiences. HER has reached notable results when combined with DDPG [69]. Another conventional method includes combining RL with demonstrations thus accelerating the control policy learning process. Coupled with DDPG, demonstrations have managed to reduce training time and also showcase potential results [70]. Other concepts have utilized specifically developed frameworks such as operational space control framework (OSC) [7] and constrained optimization layer [71] or methods like precision-based continuous curriculum learning (PCCL) [39].

#### 4.2.2 Transfer techniques

Modelling a diverse environment for a complex scenario such as a robotic task can be resource-demanding process, requiring considerations for the level of abstractions, generalization, and accuracy. Transfer learning such as progressive nets [72] have showcased a flexible way of using experience collected from simulation in tuning a robotic control system. In other projects, simulator parameters have been optimized via system identification or then differences between the simulation and the real-world scenario have been studied and taught to an identification model to augment the simulation to be closer to the real-world robotic scenario [73]. Other known techniques for Sim2Real transfer

within robotics domain are explicit transferable abstractions, combining analytical modelling with system identification, imitation learning, meta-learning and curriculum learning and knowledge distillation [59]. One of the most common methods used in Sim2Real transfer are the randomization of dynamics [47] or overall domain with domain randomization [74] to augment the training set. A notification to be made here, that some of the mentioned methods and concepts have been in use before the term Sim2Real, but because Sim2Real is a comprehensive concept it intersects the aim of many of the mentioned methods and concepts.

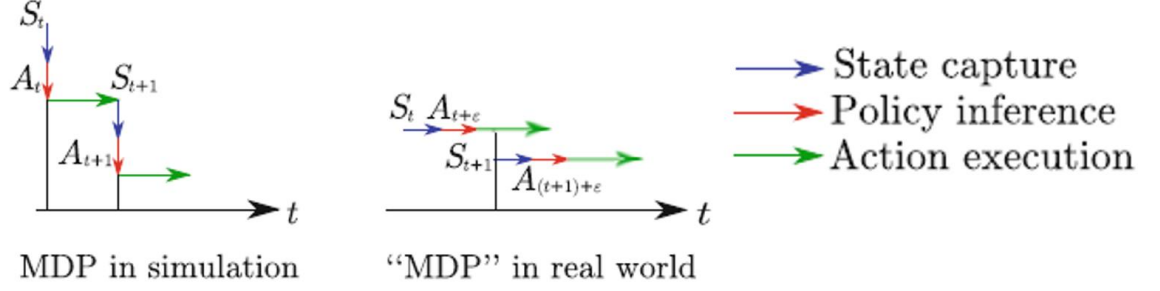
This thesis utilizes one of the most straightforward ways to transfer knowledge from simulation to reality with the zero-shot transfer method [75], [76]. Zero-shot transfers the control of the chosen DRL policy from simulation to the physical domain. The method is built on a realistic simulator, which provides direct implementation of the trained policy into a real-world setup. Owing to its direct approach the method is also called direct transfer. The general idea is that the simulation accounts all the necessary details of the performed task, and the transfer to the physical domain should not introduce any new parameters that could affect the performance. This is managed by setting the abstraction level correctly in regards the performed task. As a contrast to zero-shot transfer, system identification with precise models of the physical world and domain randomization can be regarded as one-shot transfers. It is not uncommon to use a mixture of the aforementioned methods.

### 4.3 Reality gap

When transferring simulation trained DRL policies into a physical setup, the process has to consider the reality gap. The existing reality gap caused, for instance, by the variability between simulated and real observations, model uncertainties and the level of abstraction in regards of physics is a factor for any level of sophistication of simulations. Before highly accurate simulations becomes reality for robotics, reality gap will be an issue affecting the final task performance, requiring fine-tuning as well as, in some cases, learning on the real system. Yet, if the policy works in a simulation, it has the potential to also work in a real-world scenario if the abstraction level is set correctly to mitigate the reality gap.

Reality gap is known to cause limitations due to inaccurate models of deformation, multi-point contact, friction, impact, cutting, and indeterminism in general. Surface friction has inherent indeterminacy, meaning it is undecidable to predict how objects are affected in specific situations [59]. The physical domain gap includes many aspects of physics such as contact dynamics and soft bodies that underly the vast complexity scoped within the

reality gap. The variety of factors to note depends on the specific system, since the reality gap is usually task dependent. A rough example of differences between the system dynamics of simulation and the real-world is depicted in Figure 4.2.



**Figure 4.2.** Markov Decision Processes in a simulation and reality depicting state capture, policy inference and action execution processes on a timeline. Modified from [1].

In the simulation the state capture and the policy inference of MDP are considered to behave instantaneously, while in the real world both cases can actually take considerable amount of time. This simplification is not taking into consideration the richness of the real world, which causes the agent to make action choices based on delayed observation from previous states during the previous action execution. Time delays affecting state capture and policy inference processes can affect applications like manipulator trajectory planning. If object reaching task is considered, in reality the target object is captured and localized with a camera, which consumes time, but in a simulation the process of capturing the state might have been considered to have zero-time consumption. This will induce delay during the observation, and it will create difference between the simulated and the real-world trajectories [1]. Generally, in robotics it is unrealistic to assume the true state is completely observable and free of noise [6].

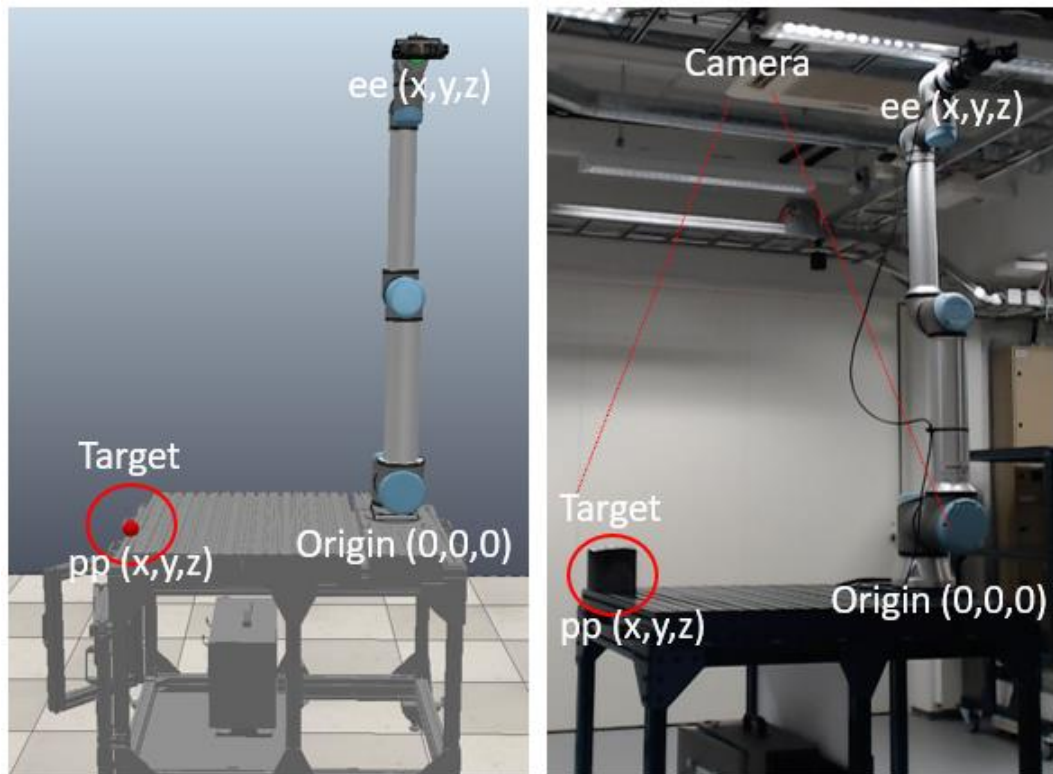
To bridge the gap between simulation and reality requires methods able to confront mismatches in both sensing and actuation. Where actuation can be minimized with more realistic simulation and variability in agent dynamics, sensing can be confronted, for instance, with adversarial attacks of image perturbation on computer vision based algorithms [77]. Both aforementioned cases have been also approached with methods and concepts mentioned in previous section, such as domain randomization in order to generate a policy able to generalize at least on some level to the real-world scenario. Hence, the Sim2Real can be seen to provide a methodology of learning a policy with respect to the reality gap regardless of the accuracy of the simulator. [1], [59]

## 5 SIMULATION-TO-REAL CASE STUDY

The case study implements the presented theory of Chapters 2-4 in practice by demonstrating a pipeline for a Sim2Real transfer. Special interest is put on developing architecture between the RL algorithms and the chosen simulation environment, which follows the structure of an MDP. RL algorithms are first trained with specific reward function and hyperparameter setups to distinguish a baseline to minimize the Euclidean distance between the TCP and the target. Afterwards, more heuristics are included to train the policy to be safer for a Sim2Real transfer.

### 5.1 Task description

The case study is based on an environment defined by UR10e manipulator on a Vention table. The manipulator is a 6-DoF robotic arm that has a Robotiq wrist camera, Buind tool-changer and a Robotiq 2F-85 gripper attached to the wrist. Wrist attachments are not utilized in the project as the scope is not to perform object manipulation. However, they are utilized in other projects by VTT, and it was requested to implement them to the simulation model and keep them as part of the physical manipulator. The robot cell is depicted in Figure 5.1.



**Figure 5.1** Case specific robotic cell in simulation (left) and in real-life (right).

The task is to provide a pipeline for controlling the manipulator with a RL policy while performing a reach-target task, which in this thesis is based on position-only reaching [71]. Target reaching is one of the most common problems in robotics. More complex tasks such as object interaction and grasping are associated with reaching before the actual main task can be performed.

The reaching task is defined as controlling the end-effector (*ee*) to reach a position at the given target goal (*pp*). The origin of the Cartesian coordinate system is located at the base of the manipulator. The manipulator would start from a fixed initial pose and follows the trained policy accordingly to reach the goal. The position of the target goal is desired to be reached from the top-vertical direction. Although this approach is a simplification leaving orientation constraints outside of the study, the scope is not to generate a fully generalizable policy able to interact with poses, but to demonstrate a workflow for implementing a Sim2Real process. At first, during the training the target is defined as a randomized dot (left target in Figure 5.1) in simulation, but eventually in Sim2Real transfer the target is localized with a machine vision system tracking a ArUco (Augmented reality University of Cordoba) marker [78] (right target in Figure 5.1). ArUco marker tracking is discussed in more detail in Section 5.6.1.

## 5.2 Simulation environment

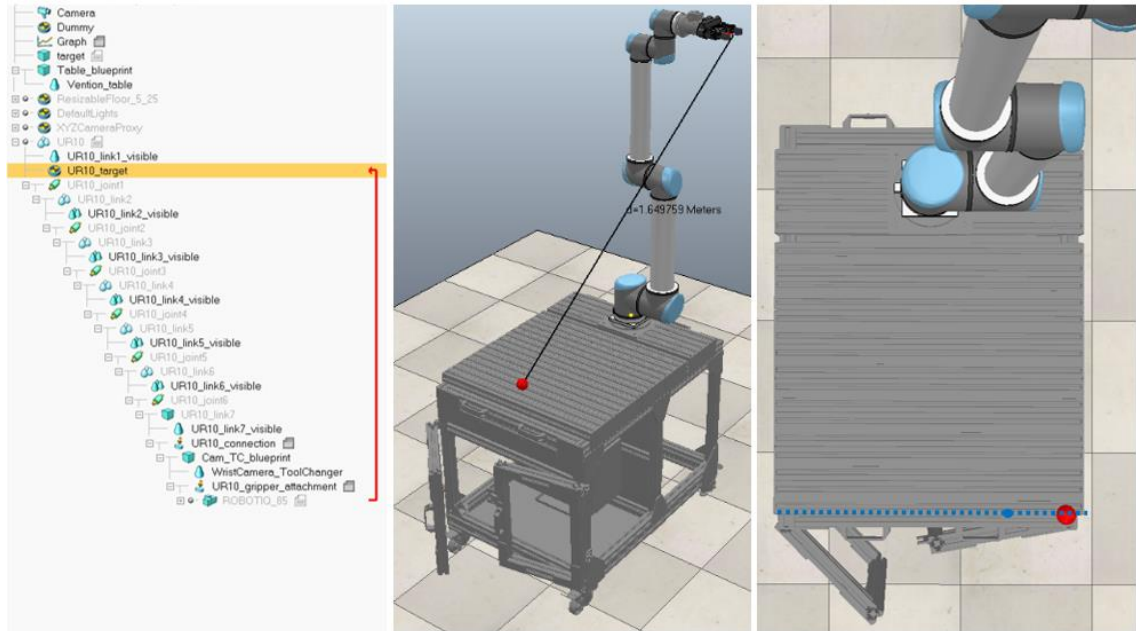
At the time of writing this thesis, OpenAI Gym and MuJoCo did not offer straightforward ways to be utilized with a custom use-case scenario as in this thesis. Unity 3D was also considered, but at the time ML-agents toolkit [56] did not give access to the structure of the utilized RL algorithms, although the interfaces to work with the algorithms were easily accessible. Eventually, the physical system was simulated using CoppeliaSim with Python wrapper PyRep [79] forming a bridge to the RL. Additionally, PyRep enables faster communication with the separate RL scripts than the remote API of CoppeliaSim. The combination proved to be relatively easy to learn and use offering simple conventions to import task-relevant abstractions of the real-world scene into the simulation.

### Simulation components

CoppeliaSim is similar to Gazebo in sense that it is a rigid-body simulator able to control multiple robots in real time. The most convenient aspect of CoppeliaSim and PyRep was the supplied robotics library, which included *Universal Robots* Ur10 with implemented forward and inverse kinematics support. Since Ur10 and UR10e have almost the same specifications including kinematics, UR10 model from CoppeliaSim library was utilized instead of creating a custom model of UR10e. This decision sped up the overall process

of developing the pipeline, since it was not necessary to create UR10e as a new robot within CoppeliaSim and to solve the kinematic equations for it.

To make the simulation environment similar as possible to the physical counterpart, a CAD model of a Vention table is imported to CoppeliaSim on top of which UR10 is situated in same manner as in the physical environment. Same is done to the tool-changer and wrist camera located at the wrist of UR10e. Supplied CoppeliaSim library does not include exact replica of the physical gripper Robotiq 2F-85, ergo the next best choice of older version of Robotiq 85 is chosen. The gripper model was used in hopes for VTT to utilize the created simulation model for grasping tasks in coming projects. Gripper has the same length actuation, but the structure is bulkier and shorter. The final simulated environment with the kinematic chain of the manipulator joints is depicted in Figure 5.2.



**Figure 5.2.** Simulation environment for the reach target task in CoppeliaSim (middle, right) with the kinematic chain of the UR10 from the base to the gripper represented with red line (left). Target is represented with a red dot, which position is randomized during the training along the dotted blue line (right).

### Environment specifications

For the policy training phase, a red dot corresponding to the goal target is included in the environment. At the beginning of each episode, the manipulator is initialized to the ‘zero’-position (depicted middle in Figure 5.2) and the position of the dot is randomized along the dotted blue line (depicted left in Figure 5.2). The target dot is approached from top vertical direction and the Euclidean distance between the target and the TCP is continuously measured. The minimum distance vector is depicted by a black line and a distance value in the middle of above Figure 5.2. The dot is set 6cm above the table working face, having a total offset of 1m from the ground surface. The table has a hitbox set around it

with a 4cm offset. Hence, the hitbox is 2cm below the target. The invisible hitbox functions as a safety measure to protect hardware in case the reality gap causes surprising accidents during validation with physical equipment. Hitbox is utilized as a collision measurement with the augmented reward function discussed in the coming Section 5.5.4.

The simulated manipulator can be controlled either with forward or inverse kinematics as discussed in Section 2.2. In this thesis, the manipulator is controlled with forward kinematics mode having numerical states as observations. The MDP process within the simulation is discussed in more detail in Section 5.4.1. This approach requires manipulator joints to be in ‘*Torque or force mode*’ meaning the joint is simulated by the dynamics module Bullet physics [58]. This physics engine manages interactions within the environment scene that is controlled by a main script consisting of simulation specific and necessary codes.

Actions are executed on the simulated manipulator with joint velocities acquired from a state. Therefore, performed action includes velocity commands for each joint of the manipulator. Consequently, performed actions produce the next numerical state. The joint tries to reach the desired target velocity given the maximum torque it is capable to deliver [80]. Table 5.1 depicts the maximum joint torque and velocity limits of the physical UR10e that are referenced in the simulation. For safety reasons the maximum joint velocity limit for each joint in the simulation is set at 2.09 rad/s.

**Table 5.1.** Joint torque [Nm] and angular velocity [rad/s] limits of UR10/UR10e [81], [3].

Dimension	Joint 1	Joint 2	Joint 3	Joint 4	Joint 5	Joint 6
Max. Torque (Nm)	330	330	150	56	56	56
Max. Velocity (rad/s)	3.33 (2.09)	3.33 (2.09)	3.33 (2.09)	2.29 (2.09)	2.29 (2.09)	2.29 (2.09)

The object bodies are responsible meaning the geometrical shapes will produce a collision reaction. The simulation is a discrete time simulation, which is ran at default time interval of 50ms. This makes the CoppeliaSim simulation time to run at a faster rate than real-time, thus speeding the training phase. Finally, no additional light sources or sensors are implemented within the scene.

### 5.3 Known reality gap factors

Differences between the virtual and the physical counterparts causes the expansion of the reality gap. Due to decisions made in constructing the simulation environment, dimensional differences occurred that are manifested within this gap.

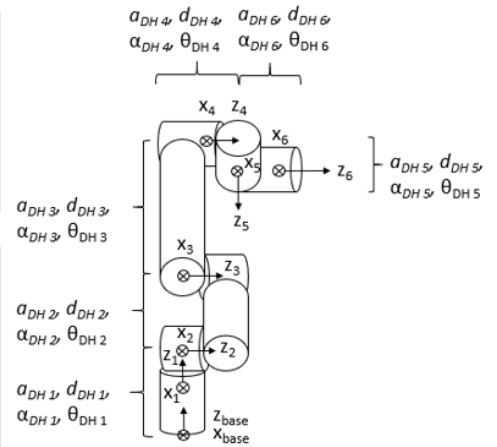


Simulated UR10 model has different DH-parameters compared to UR10e in the physical environment. These parameters include the necessary information for the kinematics discussed in Section 2.2. DH-parameters for both models are presented in Figure 5.3.

UR10e							
Kinematics	theta [rad]	a [m]	d [m]	alpha [rad]	Dynamics	Mass [kg]	Center of Mass [m]
Joint 1	0	0	0.1807	$\pi/2$	Link 1	7.369	[0.021, 0.000, 0.027]
Joint 2	0	-0.6127	0	0	Link 2	13.051	[0.38, 0.000, 0.158]
Joint 3	0	-0.57155	0	0	Link 3	3.989	[0.24, 0.000, 0.068]
Joint 4	0	0	0.17415	$\pi/2$	Link 4	2.1	[0.000, 0.007, 0.018]
Joint 5	0	0	0.11985	$-\pi/2$	Link 5	1.98	[0.000, 0.007, 0.018]
Joint 6	0	0	0.11655	0	Link 6	0.615	[0, 0, -0.026]

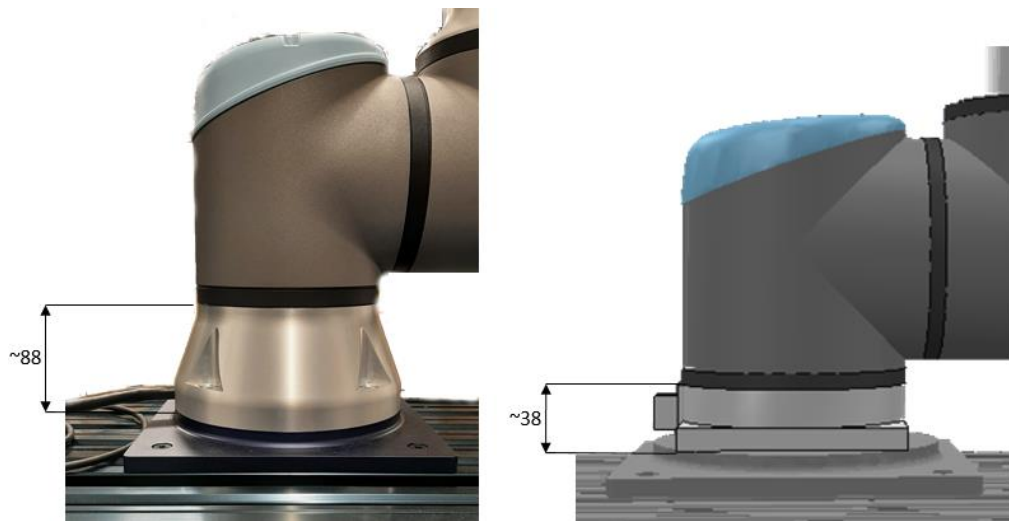
  

UR10							
Kinematics	theta [rad]	a [m]	d [m]	alpha [rad]	Dynamics	Mass [kg]	Center of Mass [m]
Joint 1	0	0	0.1273	$\pi/2$	Link 1	7.1	[0.021, 0.000, 0.027]
Joint 2	0	-0.612	0	0	Link 2	12.7	[0.38, 0.000, 0.158]
Joint 3	0	-0.5723	0	0	Link 3	4.27	[0.24, 0.000, 0.068]
Joint 4	0	0	0.163941	$\pi/2$	Link 4	2	[0.000, 0.007, 0.018]
Joint 5	0	0	0.1157	$-\pi/2$	Link 5	2	[0.000, 0.007, 0.018]
Joint 6	0	0	0.0922	0	Link 6	0.365	[0, 0, -0.026]



**Figure 5.3.** DH-parameters of UR10 and UR10e (left) and a parameterized diagram of a UR manipulator (right). Modified from [82].

Although, joints 2, 3 and 5 are almost identical, the dimensional difference in joint 1 causes UR10e to be about 5cm taller than UR10. The height difference, depicted in Figure 5.4, is roughly noticeable from the different base structures the models have.

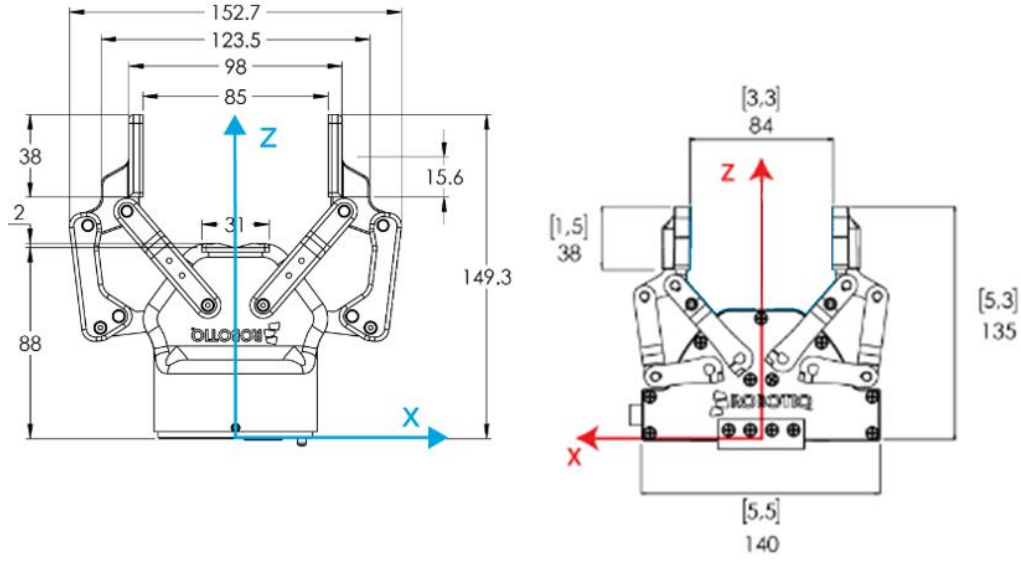


**Figure 5.4.** Base of UR10e in the physical environment (left) and base of UR10 in the CoppeliaSim simulation (right). Pictures are not in scale, and distances are in mm.

In addition to height, differences between joints 4 and 6 results in the TCP to be about 3,5cm further away from the base along the  $Z_6$ -axis for UR10e. The  $Z_6$ -axis is depicted in Figure 5.3 and it is located at the tool-end. In a similar manner, the Robotiq 2F-85 gripper is in the simulation the older version of the actual physical gripper, which most noticeably causes the TCP to be additional 1,4cm further away from the base along the



$Z_6$ -axis for UR10e. The difference in length between the gripper versions is depicted in Figure 5.5.



**Figure 5.5.** Robotiq 2F-85 gripper newer version (left) and older version (right). Distances are in mm. Modified from [83], [84].

In the end, the total cumulated dimensional difference between the simulation and the real-world scenario is approximately 5,3cm in height along the  $Z_{base}$ -axis and 4,9cm along the  $Z_6$ -axis for UR10e. Reference axis are from the Figure 5.3. Calculated dimensional differences are approximations, but they offer a guideline for understanding the reality gap during the validation phase of Sim2Real transfer.

The presented dynamic DH-parameters in Figure 5.3 are roughly the same for both manipulator versions. However, dynamics are not a major factor in this thesis, since in the end the main target for the trained policy is to perform a trajectory path from the zero-position [3] to the target goal. These paths are made by learned joint angular velocities included within the states, that are executed by the forward kinematics. Ultimately each learned joint value can be seen as a static value, since the dynamics of the simulation are not varied during this project. One could argue that this makes the relevance of the dynamic parameters close to nothing.

## 5.4 DRL software architecture

The control of the simulated UR10 is managed by implementing DDPG and SAC DRL algorithms alongside the simulated environment. The reach task is formulated as an MDP process with definitions of state, action, goal, and reward. The software modules and the structure follow the MDP process which is implemented as separate Python scripts on an Ubuntu 18.04 laptop. DRL processes are running on an Intel® Core™ i7-

6600U @ 2.6GHz CPU instead of AMD Opal XT [Radeon R7 M265/M365X/M465] GPU. CPU was utilized, because the available GPU did not provide direct support for Compute Unified Device Architecture (CUDA) platform, which was considered as alternative approach. This limits the computation power and increases training time. Additionally, used CPU cannot provide sufficient multi-processing, which could have speed up computation.

#### 5.4.1 MDP in practice

MDP of Section 3.3 is the basis of training the DRL algorithms. Relevant dimensions regarding environment, state, action, goal, and reward components are presented in Table 5.2.

**Table 5.2** Reach task environment dimensions for UR10/UR10e.

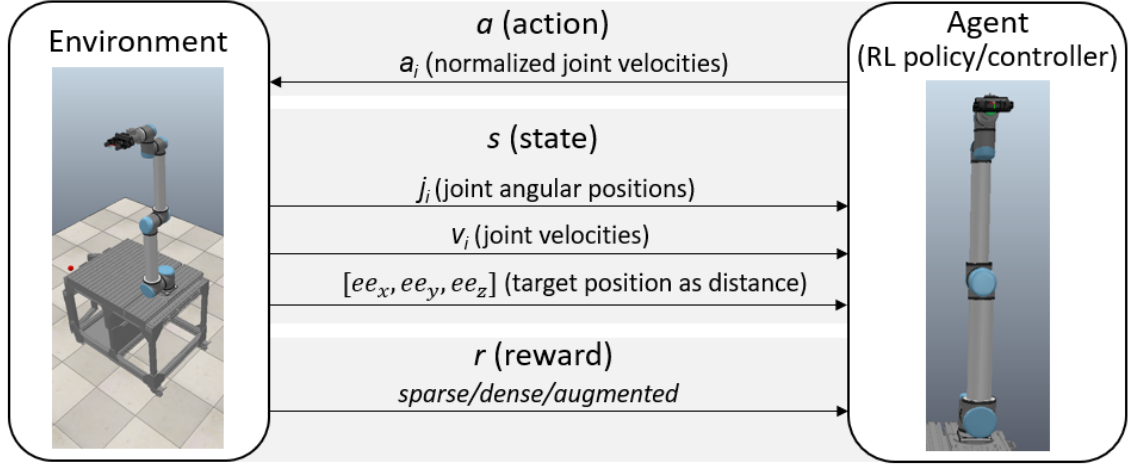
Dimension	Symbol	Value
State dimension	SD	15
Action dimension	AD	6
Action boundary	AB	[-1, 1]

##### State

A discrete step considers what the agent/manipulator does as an action, thus indicating the next state of the agent. Additionally, as an action is taken to alter the state of the environment, the physics simulation of *CoppeliaSim* is proceeded with a step. Performed action alters the state of the environment with trial-and-error to update the knowledge of the algorithm instead of a transition probability distribution or transition model, which is common for model-based algorithms. Here state includes the current status of the manipulator intrinsic joint positions (angular)  $j_i$  and velocities  $v_i$  with  $i$ th-joints  $i \in \{1, \dots, 6\}$ , and the distance to the target position as a target vector  $[ee_x, ee_y, ee_z]$

$$s = [j_i, v_i, [ee_x, ee_y, ee_z]]. \quad (5.1)$$

The distance between the centre of the target goal and the manipulator TCP is calculated in each step as it is the main factor for considering a success in the reach task. The total length of the state vector represents the state dimension (*SD*), which is 6 for  $j_i$ , 6 for  $v_i$ , and 3 for  $ee_x, ee_y, ee_z$  totalling to 15. The RL policy takes the aforementioned state vector as input and outputs an action consisting of joint velocities, which are used to control the manipulator reach the target goal. The training process of a policy is depicted in Figure 5.6.



**Figure 5.6.** RL policy training as a MDP in simulation.

### Action

Actions specify the desired manipulator TCP movement in Cartesian coordinates with joint velocities. The action range or boundary in continuous action space depends on the used activation function, which is hyperbolic tangent ( $\tanh$ ) for DDPG and SAC. Hence, action boundary ( $AB$ ) is between  $[-1, 1]$  meaning the output of predicted values is scaled to a desired range. In this thesis, the action boundary represents normalized joint velocity change of a specific joint. The actual computed actions for specific joints are bounded by the joint limitations and velocity limitations of the manipulator. These are included in the model of the manipulator. In this thesis,  $a_i$  represents normalized joint velocity change of the  $i$ th-joint. Therefore, in this thesis actions are considered to be joint increments of velocities, that are represented as a one-hot vector with the size of 6. 6 comes from the aforementioned joint count of the manipulator, representing the degrees-of-freedom. The DoFs also represent the size of the action space dimension ( $AD$ ). Actions can be presented as

$$a = [a_i]; a_i \in [-1, 1], i \in \{1, \dots, 6\}, \quad (5.2)$$

where the joint velocities are represented in radians per second (rad/s) and normalized between  $[-1, 1]$ . Normalizing the action space especially in RL enables the training process to be more stable as it increases the convergence speed, prevents divergence of parameters and provides for easier hyperparameter tuning [1], [85]. Furthermore, as mentioned above, by limiting the predicted velocities to a desired range, the target goal is reached safely within the limitations of the physical manipulator.

## Goal

The target goal is based on the position of the simulated dot in the Cartesian space. Later, with the physical environment the target position is captured from the Aruco marker. The target goal can be represented as

$$goal = [pp_x, pp_y, pp_z], \quad (5.3)$$

The target goal is reached by sampling joint angles within the limitations of the joints. According to these joint angles, the manipulator joints are controlled with forward kinematics. A more detailed overview of the forward kinematics setting is provided in Section 2.2.1. As the agent/manipulator interacts with the environment in accordance with the target goal, the environment class not only returns the next state, but also a reward.

## Reward

Reward system is approached with sparse and dense reward functions as they are common for reaching tasks [39]. Additionally, augmented reward function is utilized with enhanced heuristics. The success of the trained policy depends heavily on the selection and heuristics of a reward function. The reward function is also engineered according to the performed task meaning the reward system is usually environment specific. The thesis considers each reward function separately to find the best solution for the given reach task. Reward functions are based on the Euclidean distance between the target goal and the manipulator TCP. If the position distance is smaller than the required precision  $\varepsilon_T = 5cm$  (50mm), the action is considered successful.

The sparse reward function gives little information of the transitions in the environment thus the agent does not get proper feedback at every time step. Feedback is considered static. A successful move is rewarded by 1, otherwise a -0.02 punishment is passed. The sparse reward function is presented as

$$r_{sparse} = \begin{cases} -0.02, & \varepsilon_T < dist(ee, goal) \\ 1, & \varepsilon_T \geq dist(ee, goal) \end{cases} \quad (5.4)$$

The dense reward function produces feedback at every step according to the distance to the target. The distance is measured in meters and the positive reward is defined as 1. Otherwise, the reward is set to penalize the distance. The dense reward function is presented as

$$r_{dense} = \begin{cases} -dist(ee, goal), & \varepsilon_T < dist(ee, goal) \\ 1, & \varepsilon_T \geq dist(ee, goal) \end{cases} \quad (5.5)$$

where  $ee$  represents the TCP and  $goal$  the target position in Cartesian coordinates.

Following only the distance to the target may create situations where the manipulator trajectory could cause eventually problems for the physical manipulator or put personnel in dangerous positions. Therefore, the dense reward function is augmented with heuristics to optimize the trajectory of the manipulator by avoiding collisions with the table and approaching the target goal with a vertical orientation. Additional heuristics include penalizing for going beneath the table level and searching outside of the table area. The penalty offset in the aforementioned cases is -0.02. The augmented reward function is presented as

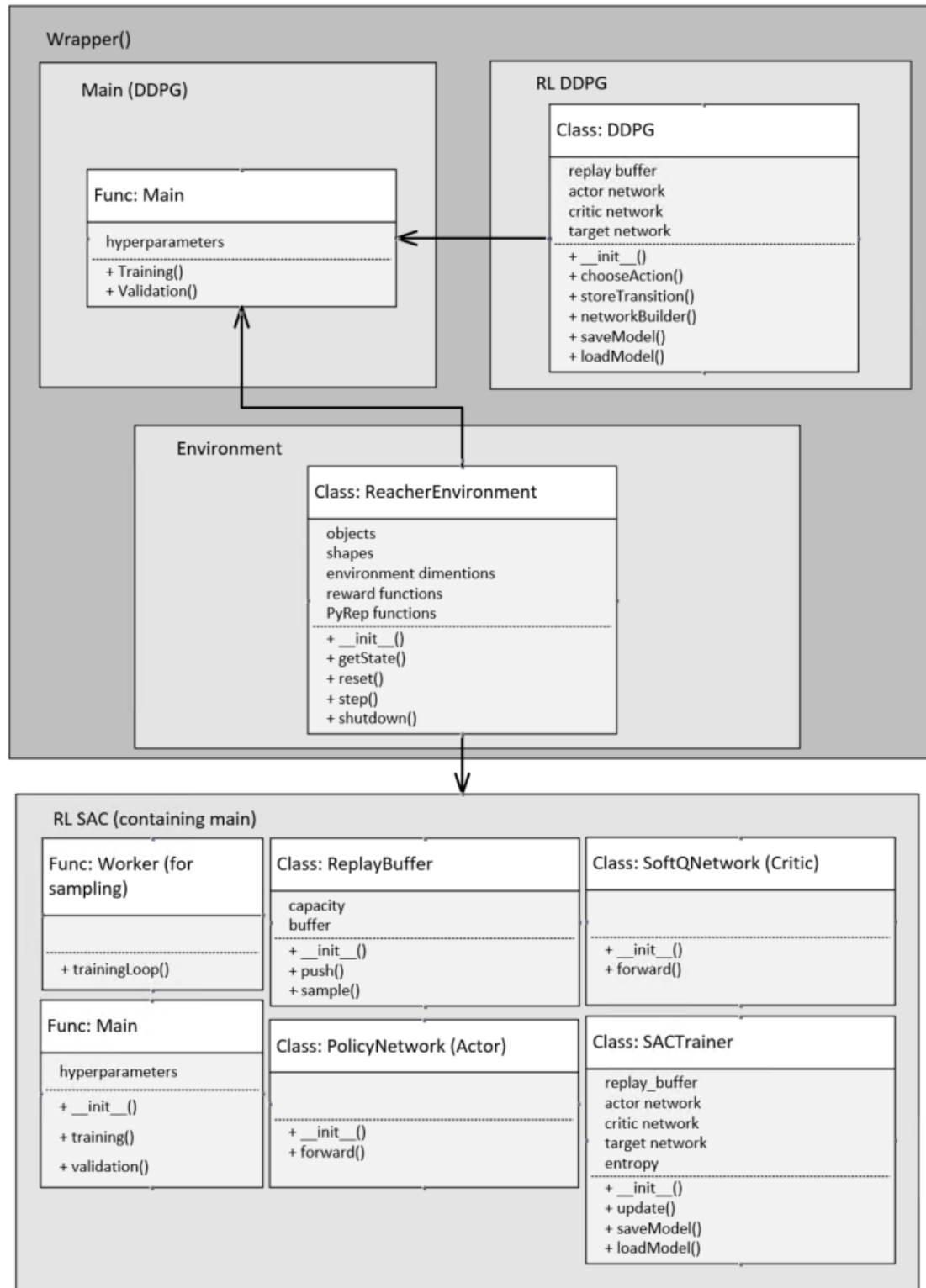
$$r_{augmented} = \begin{cases} -dist(ee, goal), & \varepsilon_T < dist(ee, goal) \\ 1, & \varepsilon_T \geq dist(ee, goal) \\ -(rotation\_penalty * rotation\_norm), & \text{orientation diff. } (ee, goal_z) \\ -0.02, & \text{collision} \\ -0.02, & \text{off\_limit search} \end{cases}, \quad (5.6)$$

where ‘rotation penalty’ is calculated as the orientation difference between the TCP and the z-axis of the target goal normalized with a ‘rotation norm’ of 0.05. Utilized values as well as the considered heuristic formulas are gathered through empirical simulation tests.

### 5.4.2 Software modules

The developed software modules include the basic outline of the presented MDP process. The environment and the state selection are in one module and the control of the agent is in another module. In principle, the environment module separates the architecture according to the used DRL. The DRL modules consist of classes and functions having a variety of parameters and subroutines that follow the structures of DDPG and SAC presented in Sections 3.6.1 and 3.6.2. Developed script for DDPG is based on algorithm 1 and [86], [87], [88], whereas the script for SAC is based on algorithm 2 and [1]. The main components and parameters of the high-level architecture with interactions are depicted in Figure 5.7.

The SAC module consists of the DRL algorithm and the main-function being within the same module while the DDPG consists of the DRL algorithm and the main-function being in different modules. DDPG has one class while SAC is divided into four classes. These classes consist of storing transitions into the replay buffer, training networks and their target networks as well as training the target policy. In SAC entropy regularization is implemented when calculating target Q-value. Reason for the separate architecture approaches comes from the desire to study the structures of the algorithms, and in the SAC case the complexity of the algorithm. The DDPG module can be controlled via a wrapper, which enables running multiple different setup-runs in consecutive order. Regardless of the DRL structure, both approaches inherit routines from the environment class.



**Figure 5.7.** Simplified representation of the software architecture for the DRL study. Function and parameter names have been modified for the sake of intelligibility. Arrows depict the direction of inheritance.

The environment class includes initialization of the Reacher-environment with the CoppeliaSim scene-file and relevant objects such as manipulator and specifications for the

target dot and the table as well as PyRep functions that form a Python wrapper to CoppeliaSim. In addition to spawning and setting up the visual scenery with objects, the environment class considers resetting, closing, obtaining the state of the environment and most of all managing what occurs during a step in the environment.

The DRL algorithm receives its hyperparameters from the main function, which additionally serves as the starting point for the agent training. The main function inherits methods from the environment and the DRL module initializing either training or validation of the policy within the CoppeliaSim scenario.

## 5.5 Training and validation

The DDPG and SAC are trained and validated separately. In training the agent performs exploration and exploitation during definite number of episodes and steps while calling routines from the DRL module to perform learning and storing of parameters. In validation these learned parameters are restored. For both DRL cases the main function performs executing arguments for training or validation after providing hyperparameters. Policy training is initialized in both DRLs in the main function, but whereas DDPG contains the training loop execution within the main, SAC contains this in a separate worker class. Both training routines are similar, and their general structure is presented in Algorithm 3.

---

**Algorithm 3:** Training function for DDPG (training() in Main) and SAC (trainingLoop() in Worker class) from Figure 5.7.

---

```

Initialize networks (DDPG/SAC);
Launch CoppeliaSim scene and initialize UR10 and target dot with PyRep;
for episode = 1, M do
    Reset scene by setting UR10 to initial zero-position and randomizing the position of the
    target dot via PyRep;
    Receive initial observation state  $s_1$  through PyRep from CoppeliaSim;
    for time increment  $t = 1, T$  do
        Select action  $a_t$  with actor network (according to policy);
        Execute a step in CoppeliaSim scene by executing an action  $a_t$  of joint velocities
        on UR10 through PyRep;
        Receive reward  $r_t$  and next state  $s_{t+1}$  from the scene through PyRep after a step
        is executed;
        Store transition experience  $(s_t, a_t, r_t, s_{t+1})$  in experience replay buffer;
        if experience replay buffer is full then
            | Update networks (DDPG/SAC);
        end
        Set  $s_t = s_{t+1}$ ;
        Save metrics data;
    end
end
Save network parameters;
Produce figures of the training;

```

---

The Python script for DDPG initializes the actor-critic framework and the replay buffer for memory storage. SAC performs similarly, except it additionally initializes the regularization coefficient (temperature). Both algorithms predict joint velocities as actions for the agent based on the current state vector, Equation 5.1. PyRep functions forward the joint velocities to CoppeliaSim and progress the simulation scene a step forward. Afterwards, PyRep receives the next state vector and the generated reward from the performed action. The transition is stored to the replay buffer. The process continuous through all the steps until finally PyRep resets the environment for the next episode by setting UR10 to initial configuration and randomizing the position of the target dot. If the experience replay buffer is full, the collected experience is utilized to update the DRL networks. During the training process if the distance between the TCP and the target dot is within the precision before the maximum step count is reached, the step is regarded as successful.

During validation, the learned behaviour of the policy is restored for examination. The process is similar to training, except the policy makes predictions according to learned parameters for the observed state. Algorithm 4 depicts the validation process of DDPG and SAC. Success is measured same way in both training and validation being that performed action leads to reaching the target. In validation, 30 successful actions are required to be produced to qualify that the policy stays on the target given the precision threshold.

---

**Algorithm 4:** Validation function for DDPG (validation() in Main) and SAC (validation() in Main) from Figure 5.7.

---

```

Restore network (DDPG/SAC);
Launch CoppeliaSim scene and initialize UR10 and target dot with PyRep;
for episode = 1, M do
    Reset scene by setting UR10 to initial zero-position and randomizing the position of the
    target dot via PyRep;
    Receive initial observation state  $s_1$  through PyRep from CoppeliaSim;
    for time increment  $t = 1, T$  do
        Select action  $a_t$  with actor network (according to policy);
        Execute a step in CoppeliaSim scene by executing an action  $a_t$  of joint velocities
        on UR10 through PyRep;
        Receive next state  $s_{t+1}$  and information of successful action from the scene
        through PyRep after a step is executed;
        if 30 consecutive steps  $t$  produce successful action then
            End current episode with success;
        end
        Set  $s_t = s_{t+1}$ ;
        Save metrics data;
    end
end
Produce figures of the validation;

```

---

DDPG is trained with 2000 episodes and 300 steps. Chosen values enable easier follow-up of the learning process. For SAC 800 episodes with 200 steps seemed to be sufficient



to highlight the learning process. In general, hyperparameters are task and algorithm dependant [89]. Deeper insight of the hyperparameters and speculation of their correlation is out of the scope of this thesis.

Different reward functions produce different scales of rewards, meaning it may not be practical to compare learning curves purely on cumulated rewards. Hence, in addition to the total reward per episode, the success rate is also displayed. In principle success rate depicts how many times in each episode agent has managed to accumulate a successful action. The total reward metric depicts the cumulative reward the agent has gained during an episode.

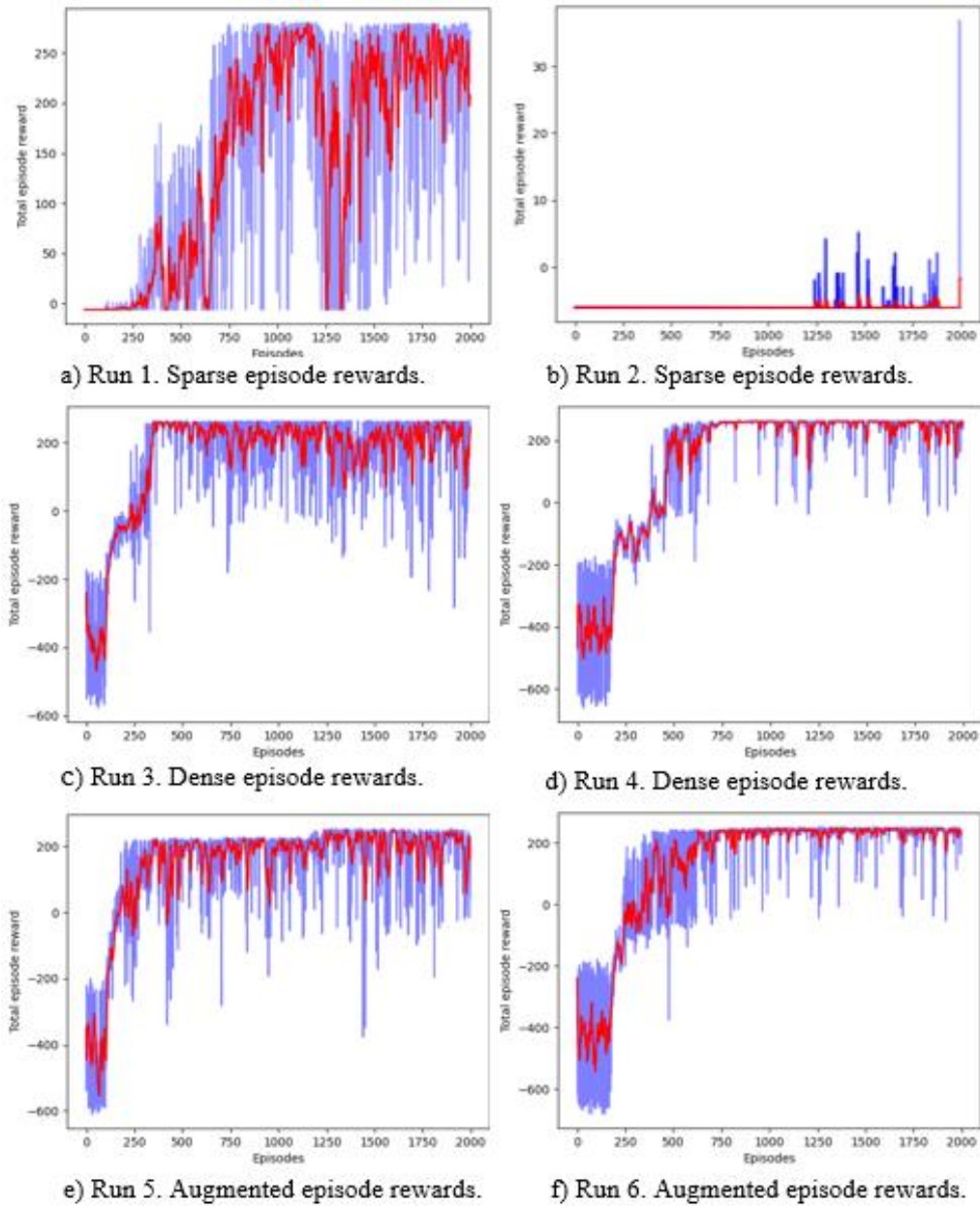
### 5.5.1 Training DDPG policy

The initial training process is divided into six cases. Hyperparameters are presented in Table 5.3 as two different setups, which are utilized for the policy training with the sparse, dense, and augmented reward function settings presented in Section 5.4.1. Hyperparameters are derived from the literature and modified to fit the reach task [28], [90].

Episodes are executed until the maximum step count regardless if a success has occurred. Alternative approach would be to terminate episode after the first successful step. Difference between the approaches is that, if the episode is not immediately terminated, the agent may learn to stay in the vicinity of the target or even correct its approach in accordance with the set precision. In this case, it is not required to reinitialize the simulation immediately after the target is reached. Since the results of these two settings do not deviate much from each other, episode termination is not considered as an option in this thesis.

**Table 5.3** *DDPG policy training hyperparameters of two scenarios for sparse, dense, and augmented reward functions.*

Hyperparameters	Symbol	Run 1.	Run 2.	Run 3.	Run 4.	Run 5.	Run 6.
Actor learning rate	$\mu_Q$	0.001	0.0001	0.001	0.0001	0.001	0.0001
Critic learning rate	$\mu_\pi$	0.001	0.0001	0.001	0.0001	0.001	0.0001
Discount factor	$\gamma$	0.9	0.99	0.9	0.99	0.9	0.99
Target update ratio	$\tau$	0.01	0.001	0.01	0.001	0.01	0.001
Replay buffer size	$\mathcal{D}$	30000	50000	30000	50000	30000	50000
Batch size	N	32	64	32	64	32	64
Gaussian noise	$\sigma$	2	2	2	2	2	2
Episodes	M	2000	2000	2000	2000	2000	2000
Steps per episode	T	300	300	300	300	300	300
Reward function	R	Sparse	Sparse	Dense	Dense	Aug.	Aug.

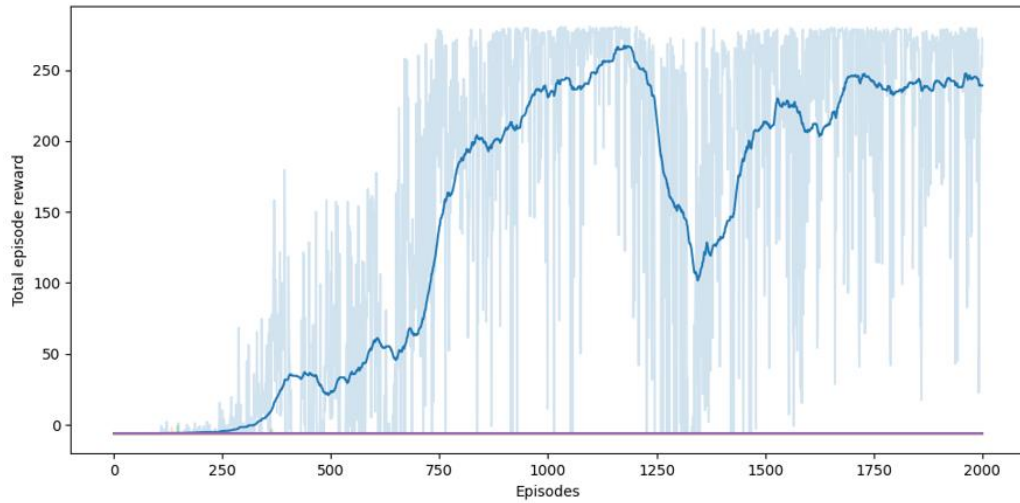


**Figure 5.8** DDPG episode rewards for Run 1-6 (a-f) as presented in Table 5.3. Red line represents the moving average of 10, and blue line the actual value.

Figure 5.8 depicts training process of selected best DDPG policy for sparse (Run 1-2), dense (Run 3-4) and augmented (Run 5-6) reward functions when observing the accumulated episode rewards. Blue line represents the actual value of each episode and red line the moving average of 10 consecutive episode values. Because of the scaling, if there are many rewards close to each other, the colour scheme darkens.

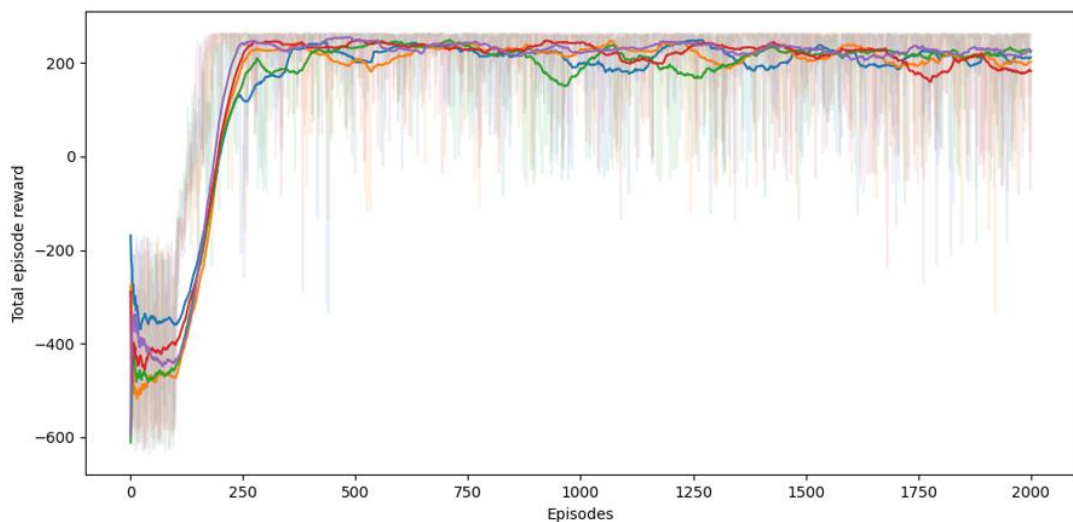
The results of Run 1, which stand out most of Figure 5.8, demonstrate how randomness in initializing the network parameters and in action selection can produce different solutions. Run 2 is performed with different initial parameters, but the difference to Run 1 is vast even when comparing dense and augmented rewards of Run 3 and 5 to Run 4 and

6. The effects of random seeds to convergence of five different policy solutions are depicted in Figure 5.9. All policies are trained with the same parameters as Run 1.



**Figure 5.9** Effect of random seeds to convergence of five DDPG policies with same parameters (Run 1 of Table 5.3). Light shaded regions depict actual values and solid lines moving average of 100 episodes.

Even though sparse reward gives little feedback of the environment, one of the policies of Run 1 of Table 5.3 has managed to find the target early on during the exploration phase before the experience replay is full, so that these positive rewards have been exploited for the rest of the training. On the other hand, four other policies have learned to exploit the static negative feedback and never or rarely gains positive feedback. As the level of feedback increases, the effects of extreme randomness can dilute as can be seen from Figure 5.10 with five policies of dense reward function trained with the same parameters as Run 3 of Table 5.3.



**Figure 5.10** Effect of random seeds to convergence of five DDPG policies with same parameters (Run 3 of Table 5.3). Light shaded regions depict actual values and solid lines moving average of 100 episodes.

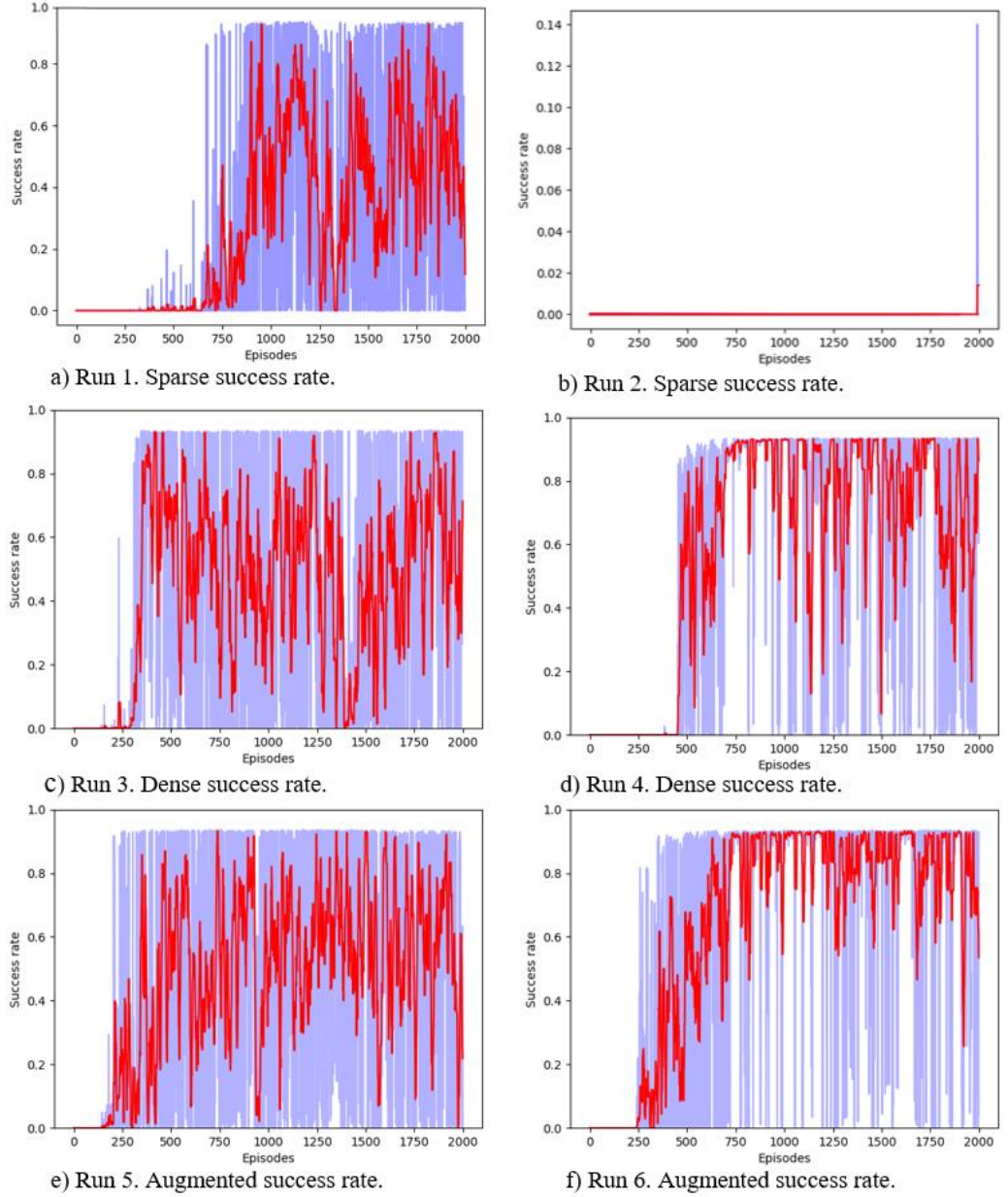
Although, Figure 5.10 depicts how all five policies have converged to more similar solutions, randomness in initializing parameters and in action selection of the actor network still can be seen affecting to the speed at which the policy achieves an optimal solution. The difference between the policies is especially visible at the beginning of the training before the experience replay buffer is full. The episode at which this occurs can be estimated as a relation between the experience replay buffer capacity and the step count per episode. Hence the average episode should be 100 for Run 1, 3 and 5 and 167 for Run 2, 4, and 6 in Figure 5.8. After this, the policy starts to update the network parameters from the experience replay buffer, which is visible in Figure 5.8. Even with sparse reward in Run 1 the rewards eventually start to rise after episode 250 as more steps begin to exploit the positive feedback.

As the network parameters are being updated the cumulated rewards per episode start to rise. The maximum total reward per episode is at best the product of available positive feedback and step count per episode. Therefore, for all scenarios in Figure 5.8 the maximum reward is 300. However, since the simulated manipulator starts from a position where it already gets negative feedback per step, the total cumulative reward cannot reach the aforementioned maximum. If the unique result of Run 1 is excluded, then the rewards seem to converge around 220 for a successful episode.

Corresponding success rates for scenarios in Table 5.3 are depicted in Figure 5.11. In principle, the success rate during the training describes how fast the policy learns to exploit positive feedback and thus how quickly the agent begins to move towards reaching the target. Success rate also depicts the stability of the policy between episodes as the policy learns to reach the target more frequently.

From Figure 5.11 one can see how policies begin to learn after transition experience from the experience replay buffer is utilized for updating the network parameters as all policies produce no success before episodes 100 and 167 for related hyperparameter setups. Blue line represents the actual value of each episode and red line the moving average of 10 consecutive episode values.

Afterwards, policies quickly begin to exploit positive feedback as the success rates start to rise. This behaviour is coherent with total episode rewards of Figure 5.8. Most prominent results seem to be produced by dense and augmented reward functions for Run 4 and Run 6 respectively. Reason for the more stable success rates and rewards could be explained by the larger experience replay buffer, which states that larger size should indicate less likelihood for sampling correlated elements.



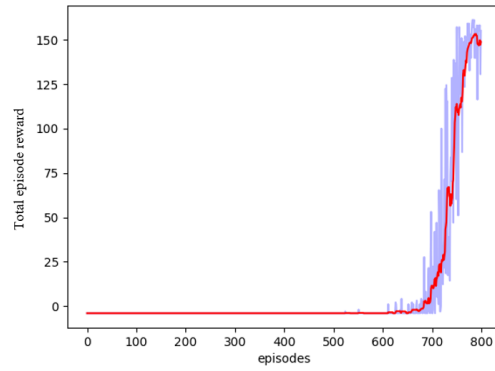
**Figure 5.11** DDPG success rates for Run 1-6 (a-f) as presented in Table 5.3. Red line represents the moving average of 10, and blue line the actual value.

### 5.5.2 Training SAC policy

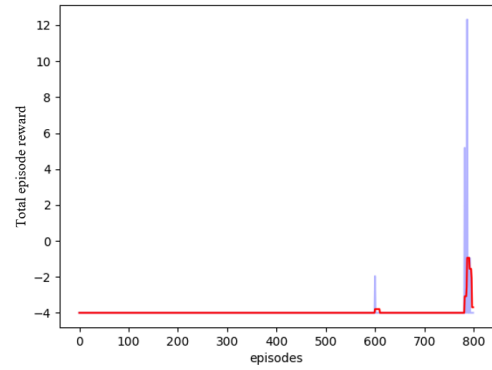
SAC training follows the same arrangement as it was presented for DDPG. Hyperparameters are presented in Table 5.4 as two different setups, which are utilized for the policy training with the sparse, dense, and augmented reward function settings presented in Section 5.4.1. Hyperparameters include entropy, which is explained in Section 3.6.2. The target entropy is equal to the negative number of action dimensions  $AD$ , see Table 5.2. In this thesis  $\alpha_T$  is -6. Hyperparameters are derived from the literature and modified to fit the reach task [45], [44], [1].

**Table 5.4** SAC policy training hyperparameters of two scenarios for sparse, dense, and augmented reward functions.

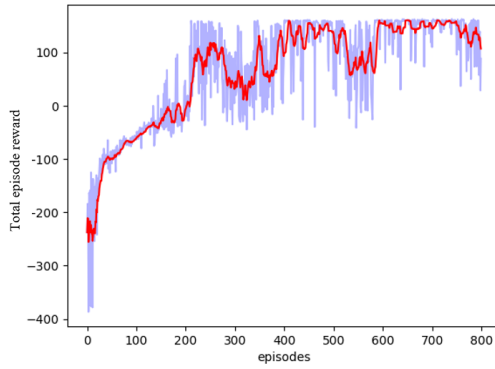
Hyperparameters	Symbol	Run 1.	Run 2.	Run 3.	Run 4.	Run 5.	Run 6.
Actor learning rate	$\mu_Q$	0.0003	0.0001	0.0003	0.0001	0.0003	0.0001
Critic learning rate	$\mu_\pi$	0.0003	0.0001	0.0003	0.0001	0.0003	0.0001
Entropy learning rate	$\mu_a$	0.0003	0.0001	0.0003	0.0001	0.0003	0.0001
Entropy target	$\alpha_T$	$-dim(AD)$	$-dim(AD)$	$-dim(AD)$	$-dim(AD)$	$-dim(AD)$	$-dim(AD)$
Discount factor	$\gamma$	0.99	0.9	0.99	0.9	0.99	0.9
Target update ratio	$\tau$	0.01	0.001	0.01	0.001	0.01	0.001
Replay buffer size	$\mathcal{D}$	100000	1000000	100000	1000000	100000	1000000
Batch size	N	64	128	64	128	64	128
Episodes	M	800	800	800	800	800	800
Steps per episode	T	200	200	200	200	200	200
Reward function	R	Sparse	Sparse	Dense	Dense	Aug.	Aug.



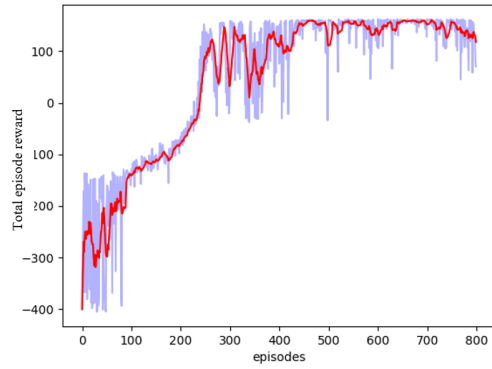
a) Run 1. Sparse episode rewards.



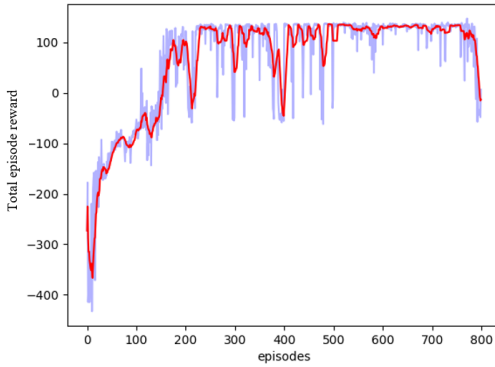
b) Run 2. Sparse episode rewards.



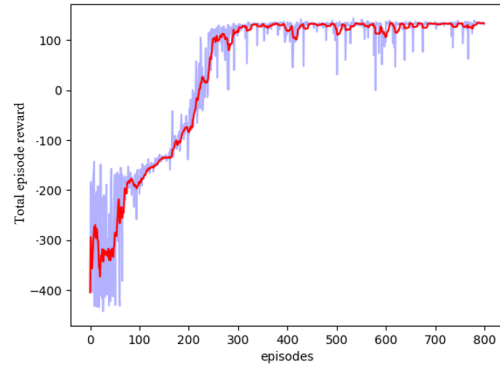
c) Run 3. Dense episode rewards.



d) Run 4. Dense episode rewards.



e) Run 5. Augmented episode rewards.



f) Run 6. Augmented episode rewards.

**Figure 5.12** SAC episode rewards for Run 1-6 (a-f) as presented in Table 5.4. Red line represents the moving average of 10, and blue line the actual value.

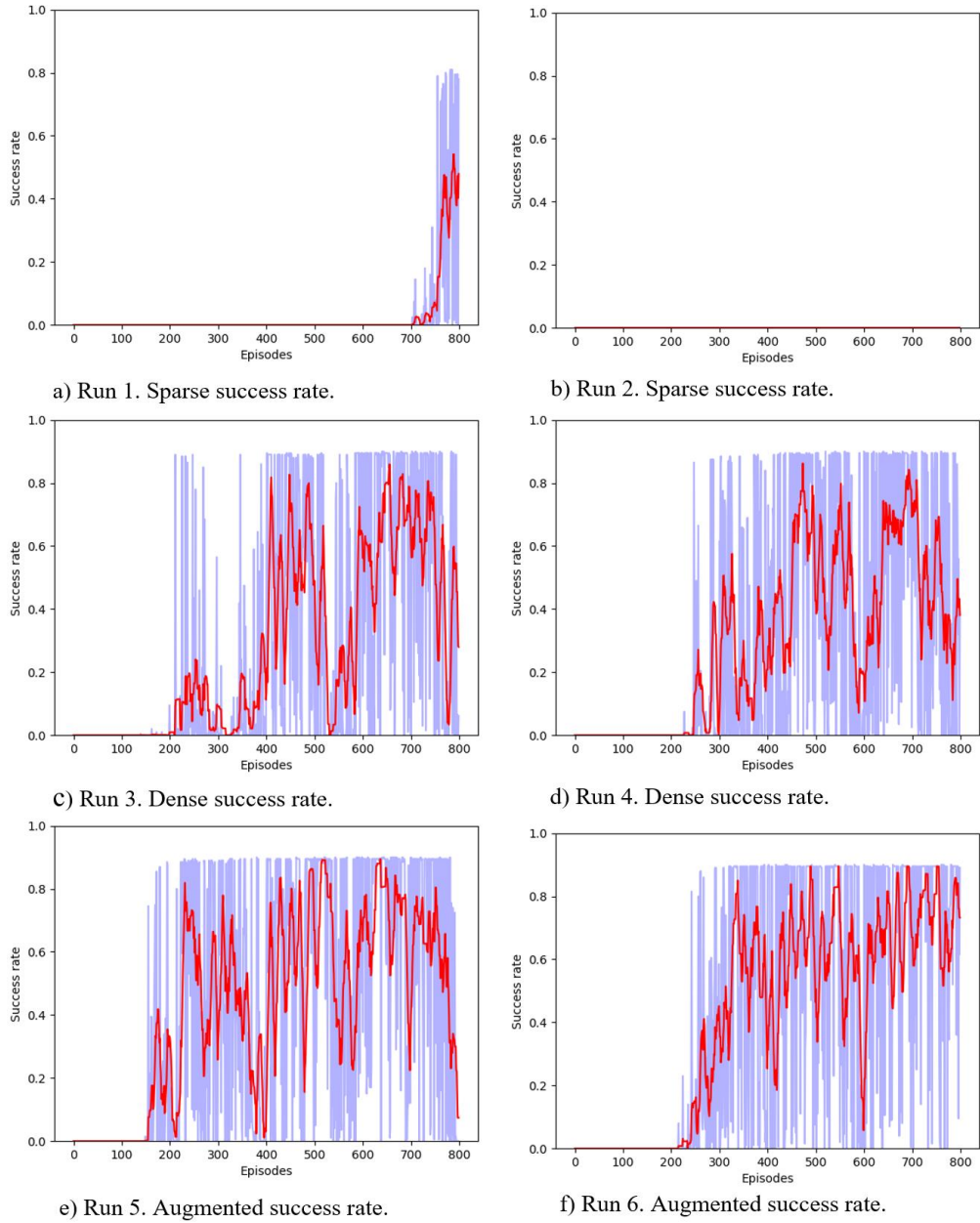
Figure 5.12 depicts training process of selected best SAC policy for sparse (Run 1-2), dense (Run 3-4) and augmented (Run 5-6) reward functions when observing the accumulated episode rewards. Blue line represents the actual value of each episode and red line the moving average of 10 consecutive episode values. Episodes are also executed until the maximum step count regardless if a success has occurred.

The results of the training are comparative to the results of the DDPG algorithm. The sparse reward does not seem to receive enough feedback from the environment to localize the target, which results in an exploration failure. Although Run 1. showcases that towards the end of the run the agent has managed to receive some positive feedback with the entropy boosted exploration. The dense and augmented rewards are also increasingly similar to the runs of DDPG. Albeit the graph trends are similar, the total cumulated reward per episode is almost 100 units lower on SAC. This could be caused by the entropy-based exploration, when the agent reaches the target, it still explored around the target to see if it could achieve even better results.

The total episode reward indicates SAC starts to learn with an increasing upward trend from an early phase and reaches the reward average of 100 around 200 episodes in Run 3 and 5, and around 250 in Run 4 and 6. As contrast to DDPG where network parameters are being updated after the experience replay buffer is full, SAC beings to update parameters after the experience replay buffer contains more transitions than a set batch size. Hence, for Run 1, 3 and 5 network parameters are being updated after there are 64 transitions in the replay buffer and Run 2, 4 and 6 after 128 transitions. In practice, this means SAC starts to immediately update parameters, which is visible in Runs 4-6 in Figure 5.12.

Corresponding success rates for scenarios in Table 5.4 are depicted in Figure 5.13. The success rates start to increase roughly after the 200- and 250-episode marks for Runs 3, 4 and 5, 6 accordingly. This is coherent with the total episode rewards in Figure 5.12. By visually inspecting both episode rewards and success rates, it can be indicated that augmented reward function produces the best results.





**Figure 5.13** SAC success rates for Run 1-6 (a-f) as presented in Table 5.4. Red line represents the moving average of 10, and blue line the actual value.

### 5.5.3 Validating results in simulated environment

Validation is realized in the simulation by allowing the trained policy to demonstrate its capabilities during a validation set size of 1000 episodes. Similar to the training process, the position of the target is randomized in the beginning of each episode, but this time the agent restores the learned policy and performs according to what it has learned during the training. Validation success rates for both DDPG and SAC policies in each scenario of Table 5.3 and Table 5.4 are depicted in Table 5.5.

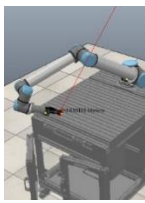
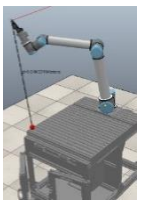
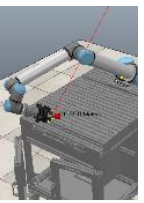
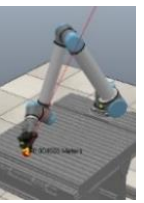
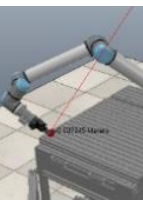
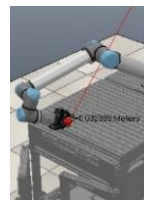
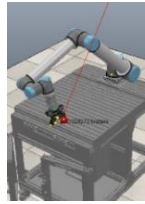
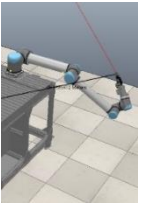

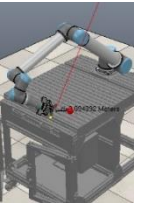
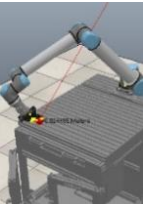
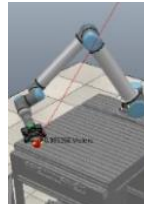


**Table 5.5** Validation success rates [%] of scenarios in Table 5.3 and Table 5.4 over 1000 episodes.

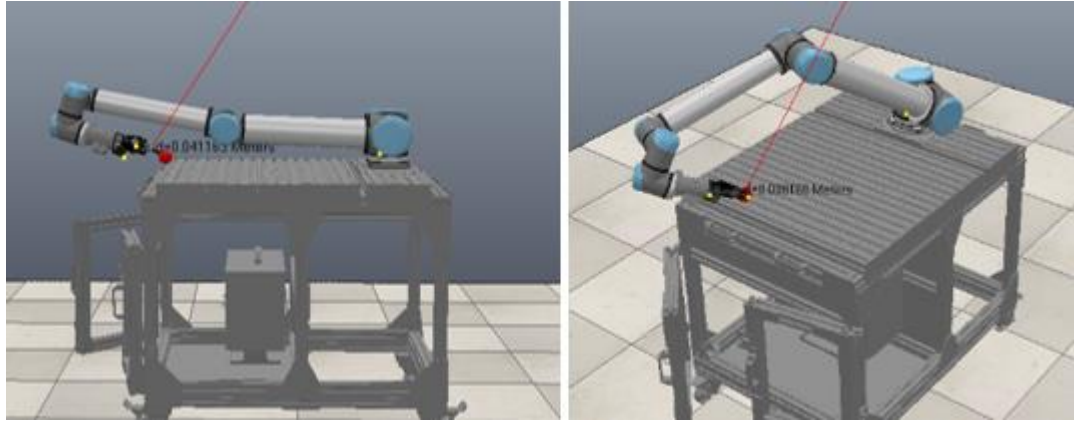
	Sparse reward		Dense reward		Augmented reward	
	Run 1.	Run 2.	Run 3.	Run 4.	Run 5.	Run 6.
<b>DDPG</b>	95.12	0.00	98.96	90.50	96.40	100.0
<b>SAC</b>	99.57	1.03	99.50	90.57	99.94	100.0

Surprisingly both algorithms have managed to produce a comparable policy with the sparse reward function although the initial position of the manipulator is not favourable to scarce feedback. Especially Run 1. results on SAC are interesting considering both metrics during the training on Figure 5.12 and Figure 5.13. Generally, the success rate seems to be aligned with the amount of feedback reaching 100% rate with the augmented reward function. Although the target is reached straightforwardly for both DDPG and SAC without unnecessary deviation from the optimal, shortest, trajectory path, such as revolving a specific joint in reciprocating motion, the final orientation of the joints is often not optimal. Table 5.6 depicts the final orientation of the manipulator either at the target or at the end of the episode for chosen episodes of each scenario of Table 5.5.

**Table 5.6** Final orientations of the manipulator either at the target or at the end of the episode.

	Sparse reward		Dense reward		Augmented reward	
	Run 1.	Run 2.	Run 3.	Run 4.	Run 5.	Run 6.
<b>DDPG</b>						
<b>SAC</b>						

In most cases DDPG reaches the target with the second and third joint almost extended to a horizontal inclination. SAC demonstrates same behaviour, and this is demonstrated on both dense and augmented reward type. Figure 5.14 depicts the hyperextension issue presenting SAC on the left and DDPG on the right with dense and augmented reward type.



**Figure 5.14** Validation of SAC (left) and DDPG (right) with dense and augmented reward function while demonstrating hyperextension of the joints. Red line indicates the direct trajectory for the TCP starting from the zero-position.

The augmented reward type does not present as extreme results as the dense reward because it has been implemented with the orientation check of the TCP. Though, the results showcase a success for both the DDPG and the SAC in completing the reach-target task, the initial parameters should be refined, and additional heuristics added to accumulate a safer policy for Sim2Real transfer. The manipulator should never reach an orientation where the operator or the equipment would be in danger.

#### 5.5.4 Modifying initial parameters and revalidation

The goal for the manipulator is not to hit the table, not to overextend any of its joints, approach from top vertical direction, avoid heavy oscillation and provide safe reasonable trajectory paths while approaching the target. Reaching the target is still the primary task, but the safety aspects are heavily regarded during the revalidation. Therefore, only augmented reward function is considered henceforward.

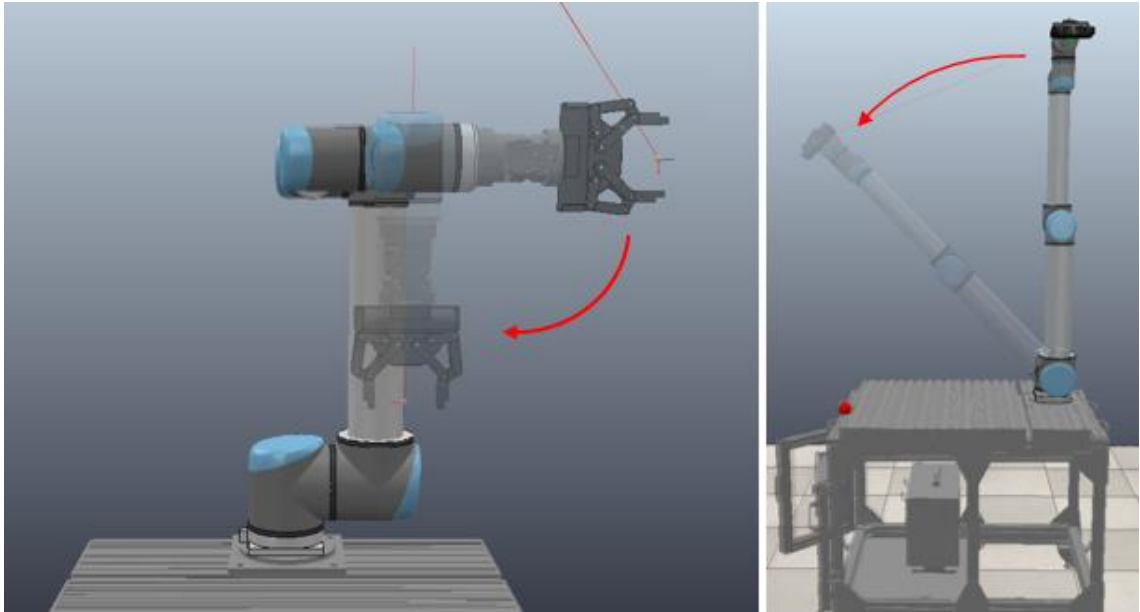
##### Parameter and reward function modifications

DDPG was re-trained with ~250 and SAC with ~150 different variations where alterations mainly included modifying hyperparameters and the reward function, but also experimenting with the height of the target. Primarily the target height is set at 1.0m, but to generate a policy that does not accidentally hit the table, the height was also experimented with. Selected instantiable revalidation parameters of 10 chosen scenarios for DDPG and SAC are presented in Appendix A and Appendix B to illustrate the variability of training parameters.

Reward offset for all the runs is set at 1, whereas the penalty offset is increased from -0.02 to -0.2. Modifications for these values were also carried out, but because of legibility, the base values are kept static. Therefore, in Appendix A and Appendix B some rewards

and penalties are presented with division or multiplication signs to indicate difference to the base offset value. For instance, the gripper was observed to occasionally hit the table during training, which was indicated as a highly negative aspect and hence the collision penalty was increased to -5 thus having a 25 modifier.

Reward function parameters include previously presented off-limit search, collision with the table, the vertical orientation difference between z-axis of the target and the manipulator TCP, and the distance to the target. To enhance the vertical orientation to the target, the rotation norm was experimented with varying values. Additional experiments are carried out by studying parameters for unnecessary joint direction of motion. Direction of motion is especially relevant for the 5<sup>th</sup> joint, which occasionally causes unnecessarily complex motions by not rotating directly towards the target when applicable. Favourable direction of motion for the Joint 5 is depicted in Figure 5.15. From the zero-position [3], where the 5<sup>th</sup> joint is 0 degrees, the favourable angle is at -90 degrees.



**Figure 5.15.** Favourable direction of motion for joint 5 (left) and for joint 2 (right) in regards the zero-position. Red arrow indicates the favourable direction of motion.

Similarly, the 2<sup>nd</sup> joint is observed to produce best trajectories when it is rotating between 0-45 degrees. Both joint motions are either regarded with a reward or with a penalty depending on whether the favourable motion is reached. As a note, all reward function parameters can be set either as a penalty or a reward. The procedure of when, what, and how much to reward depends on the application and the modeller [91]. For instance, the 5<sup>th</sup> joint motion can be penalized if it is not at the favourable -90-degree angle or it can be rewarded if it is. With penalty option, the agent would receive most of the time

negative feedback whereas with a reward option agent would not receive any feedback unless it does reach the favourable angle.

### Training and revalidation

Table 5.7 and Table 5.8 depict the most prominent parameter and incentive settings.

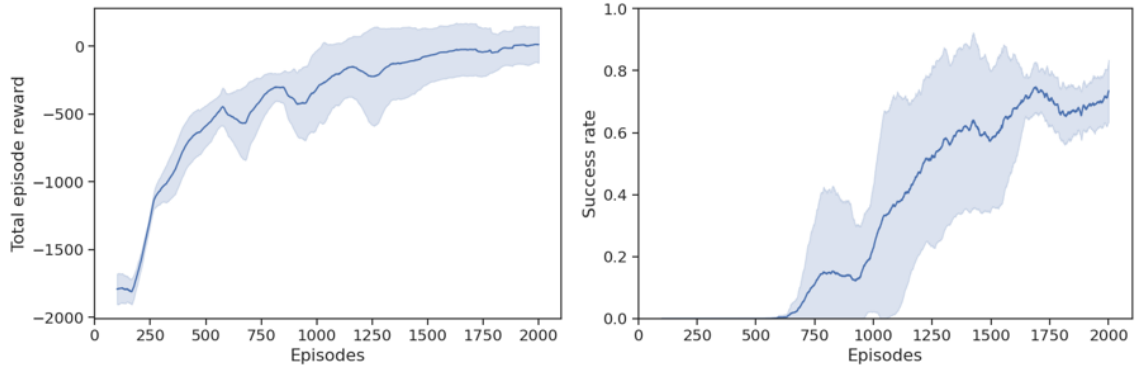
**Table 5.7** *Parameter settings for most prominent DDPG policy.*

Hyperparameters	Symbol	Run
Actor learning rate	$\mu_Q$	0.001
Critic learning rate	$\mu_\pi$	0.001
Discount factor	$\gamma$	0.99
Target update ratio	$\tau$	0.001
Replay buffer size	$\mathcal{D}$	50000
Batch size	N	64
Gaussian noise	$\sigma$	2
Episodes	M	2000
Steps per episode	T	300
Reward function	R	Augmented
Collision with the table		Penalty*25
Off-limit search		Penalty
Search beneath table		Penalty
Vertical orientation to target		Rotational penalty
Rotation norm		0.4
2 <sup>nd</sup> joint not between 0-45deg		Penalty
5 <sup>th</sup> joint not between -(85-95)deg		Penalty
Distance to target < Precision threshold		Reward
Distance to target > Precision threshold		Penalty
Precision threshold	$\varepsilon_T$	5cm
Hight of target (from floor) in Z-axis		100cm

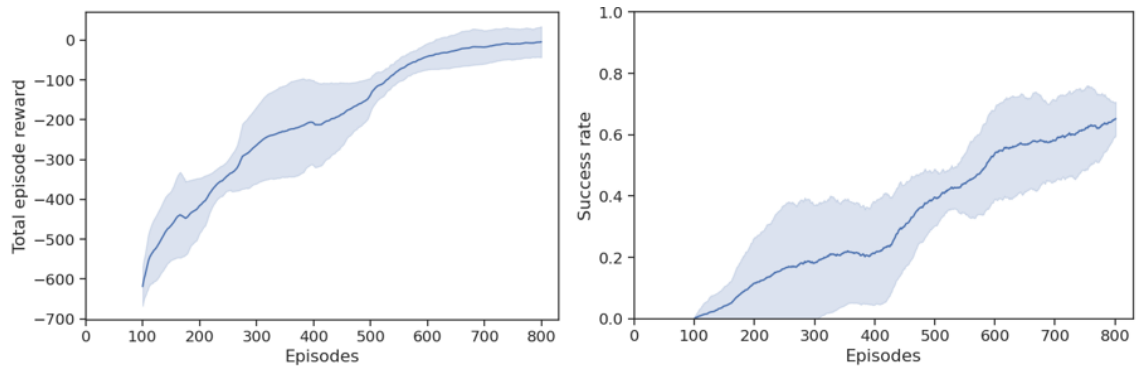
**Table 5.8** *Parameter settings for most prominent SAC policy.*

Hyperparameters	Symbol	Run
Actor learning rate	$\mu_Q$	0.0003
Critic learning rate	$\mu_\pi$	0.0003
Entropy learning rate	$\mu_a$	0.0003
Entropy target	$\alpha_T$	-dim(AD)
Discount factor	$\gamma$	0.99
Target update ratio	$\tau$	0.01
Replay buffer size	$\mathcal{D}$	100000
Batch size	N	64
Episodes	M	800
Steps per episode	T	200
Reward function	R	Augmented
Collision with the table		Penalty*25
Off-limit search		-
Search beneath table		-
Vertical orientation to target		Rotational penalty
Rotation norm		0.4
2 <sup>nd</sup> joint not between 0-45 deg		-
5 <sup>th</sup> joint not between -(85-95) deg		Penalty
Distance to target < Precision threshold		Reward
Distance to target > Precision threshold		Penalty
Precision threshold	$\varepsilon_T$	5cm
Hight of target (from floor) in Z-axis		100cm

Both DDPG and SAC are trained multiple times with the corresponding settings of Table 5.7 and Table 5.8 and the best policies were chosen to reduce the effects of randomness. The precision threshold  $\varepsilon_T$  is kept constant 5cm for all trained policies. Figure 5.16 and Figure 5.17 depicts the training results as a standard deviation of performance and a moving average of 100 episodes



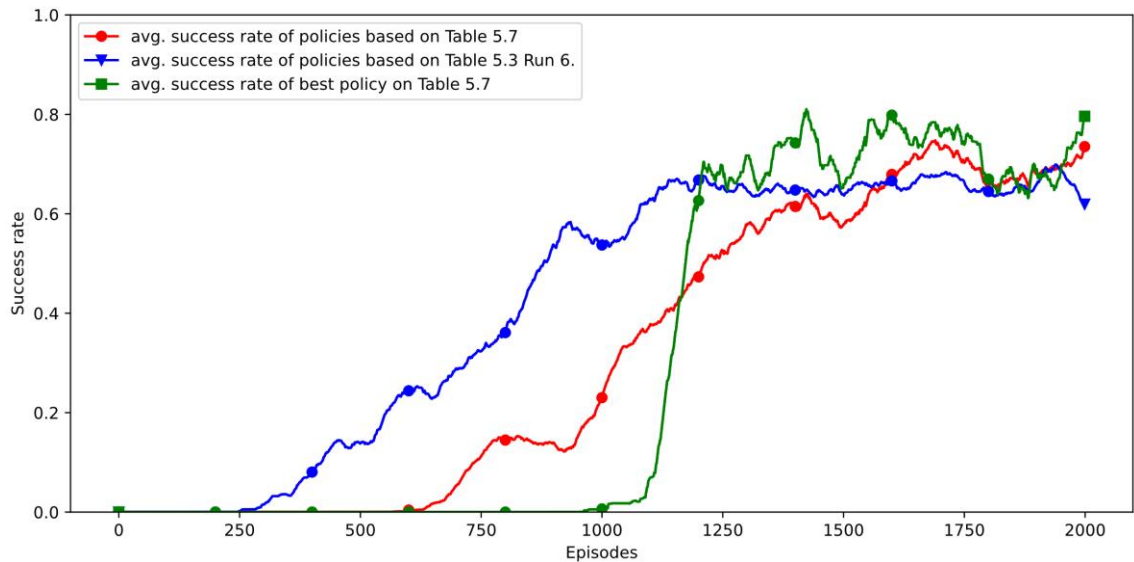
**Figure 5.16** DDPG per episode reward and success rate for policies trained according to Table 5.7. Shaded region depicts standard deviation of performance and solid line moving average of 100 episodes.



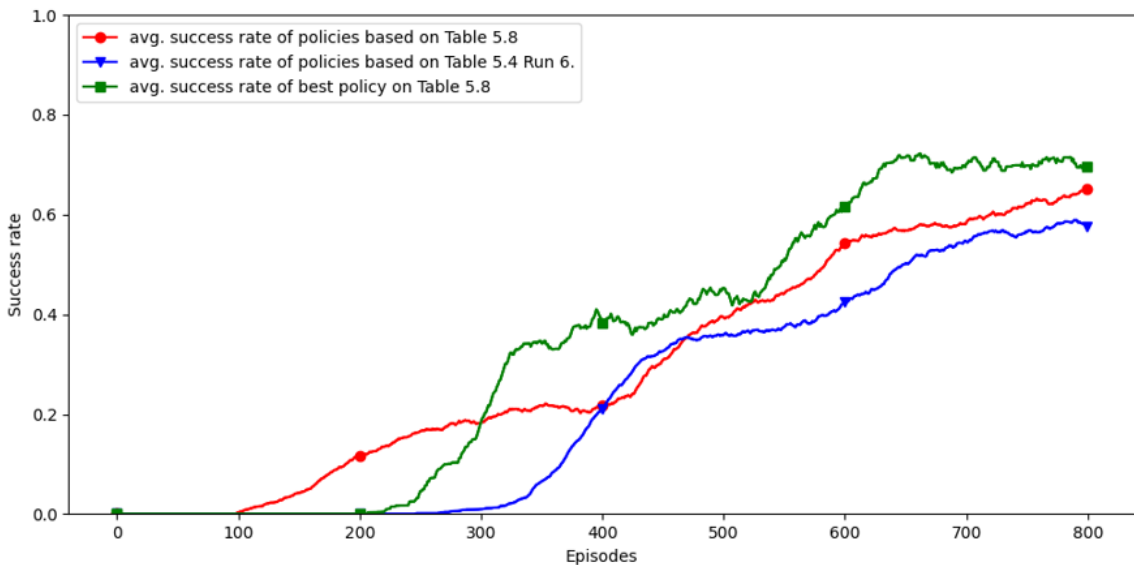
**Figure 5.17** SAC per episode reward and success rate for policies trained according to Table 5.8. Shaded region depicts standard deviation of performance and solid line moving average of 100 episodes.

The per episode reward is not comparable between DDPG and SAC, because the reward heuristics differ. However, similarly to the success rate, the per episode reward demonstrates progressive learning which is visible for both algorithms.

The average success rate of DDPG is slightly better than SAC with **74.38%** to **65.43%** (read from Figure 5.16 and Figure 5.17). To indicate progress between the training results of previous Sections, average success rates are compared to multiple selected best-trained policies of Run 6 of Table 5.7 and Table 5.8 over the course of training. Figure 5.18 and Figure 5.19 depict the success rate comparison and additionally include the best-performed policy of Table 5.7 and Table 5.8.



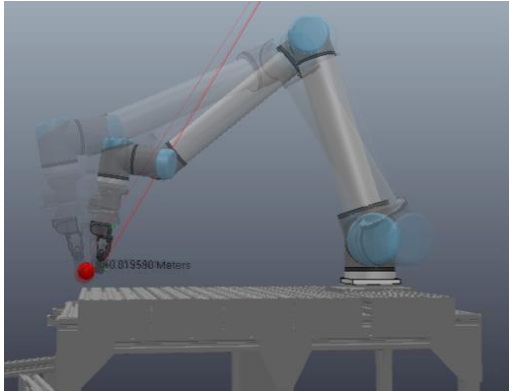
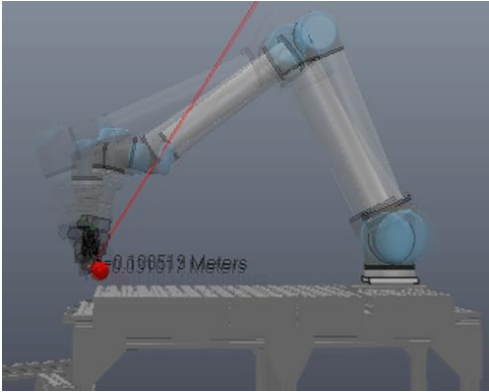
**Figure 5.18** DDPG success rate according to Table 5.7 and Run 6 of Table 5.3 over the course of training. Success rates are given as moving average of 100 episodes.



**Figure 5.19** SAC success rate according to Table 5.8 and Run 6 of Table 5.4 over the course of training. Success rates are given as moving average of 100 episodes.

Comparison graphs illustrate how the policies of Table 5.7 and Table 5.8 outperform the average best policies of Table 5.3 and Table 5.4. Revalidation in a 1000-episode environment testing, as in Section 5.5.3, demonstrate a **100.0%** success rate for both DDPG and SAC policy. The success rate is the same as it was with DDPG and SAC policies of Run 6 in Table 5.5. However, the retrained policies manage more naturally within the vicinity of the target and do not overextend any joints as depicted in Table 5.9.

**Table 5.9** Validation success rates [%] over 1000 episodes and manipulator orientations at the target.

Augmented reward (according to Table 5.7 and Table 5.8)		
	DDPG	SAC
[%]	100.0	100.0
		

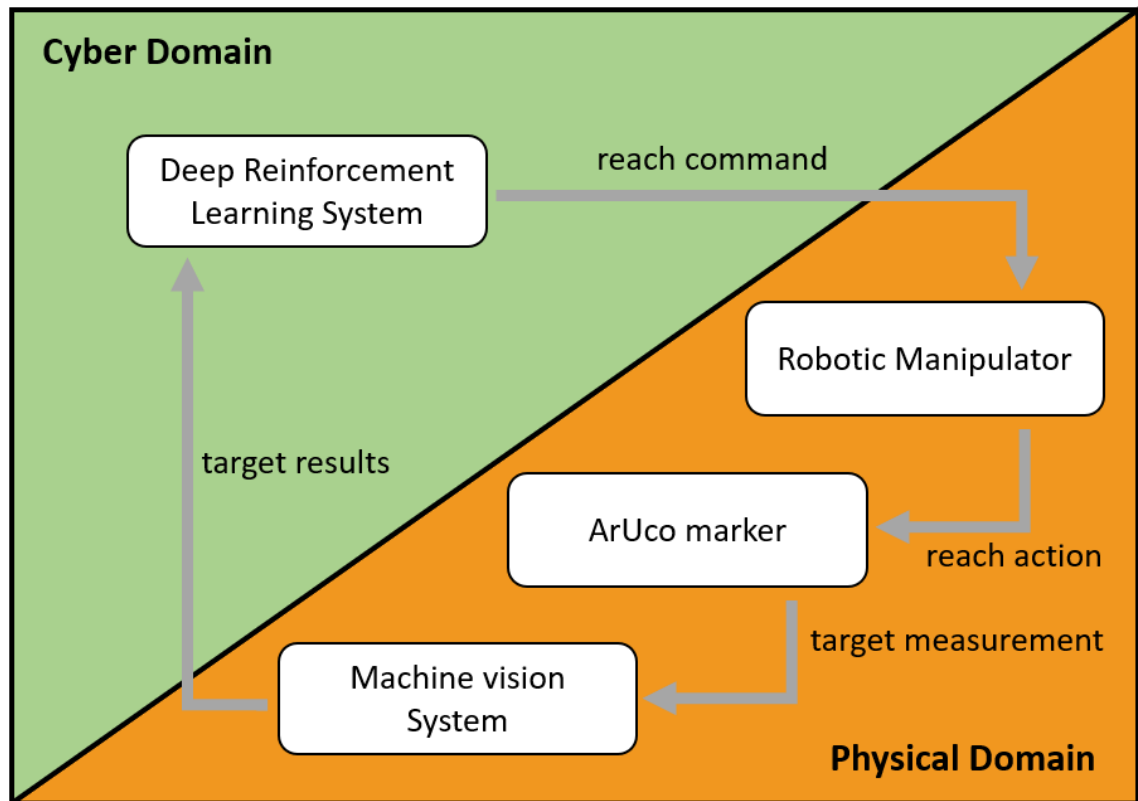
Both DDPG and SAC policies indicate some orientation differences at the target as the target location changes over the course of validation. However, for the Sim2Real transfer DDPG is chosen due to better performance indicated by the standard deviation and moving average of success rate in Figure 5.16 and Figure 5.17 (**74.38%** vs. **65.43%**). Chosen policy is validated in Figure 5.18 with the performance comparison to average of previous policies.

## 5.6 Sim2Real transfer process

Policy control is transferred from the simulation to the physical setup with a direct approach where the simulation is kept as part of the control process. Chosen method is similar to the zero-shot transfer where the scope is to build a realistic simulator and directly apply the policy to a real-world setup [59]. Figure 5.20 depicts the thesis approach as a Cyber-Physical System where the simulated cyber domain is interlinked with the physical domain.

ArUco marker is detected with the machine vision system and the localized results are supplied to the DRL system. The DRL system includes the simulator and the trained DDPG policy controller. The policy produces joint velocity values that are given to the forward kinematics of the robotic manipulator. The manipulator starts to move towards the target.





**Figure 5.20** Cyber-Physical system representation for the concept to formulate *Sim2Real* transfer.

When the target has been reached the results are verified with the machine vision system by checking if ArUco marker is covered. In success and failure, the operator is informed with a notification and the system is initialized to the initial zero-position state [3].

The ensemble comprises of three laptops in addition to the control box of the UR10e. The third laptop communicates with the manipulator through its control box, and it upholds the multimachine communication network. Communication is approached with ROS and network sockets to conjoin hardware APIs.

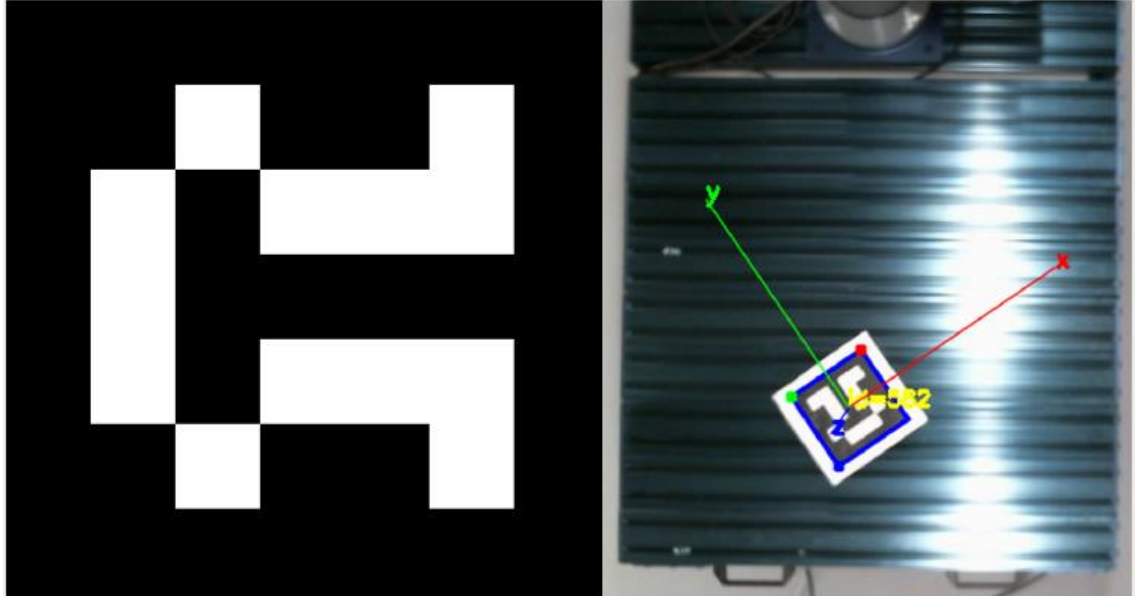
ROS is a framework for robotic software including a collection of tools and libraries to ease collaboration of different software modules and hardware interfaces. In its core, ROS is an open-source system that enables services including hardware abstraction, low-level device control, messaging between processes and package management [92]. ROS provides packages not only for controlling UR10e, but also for interlinking machine vision system with ArUco markers to the *Sim2Real* process.

### 5.6.1 Machine vision system

Machine vision system is used to detect and localize the target on the robot work surface. The thesis utilized a Logitech C615 HD webcam, which was set up above the work surface of the physical UR10e. Target to be detected is a ArUco marker [78]. which is a



square marker consisting of a wide black border with an inner binary matrix determining its identifier. The detection process utilizes ArUco library [78] for estimating the pose of the marker. Used ArUco marker and the detected pose of the marker as seen from the camera are depicted in Figure 5.21.



**Figure 5.21.** ArUco marker id 582 (left) and the marker seen from the camera (right).

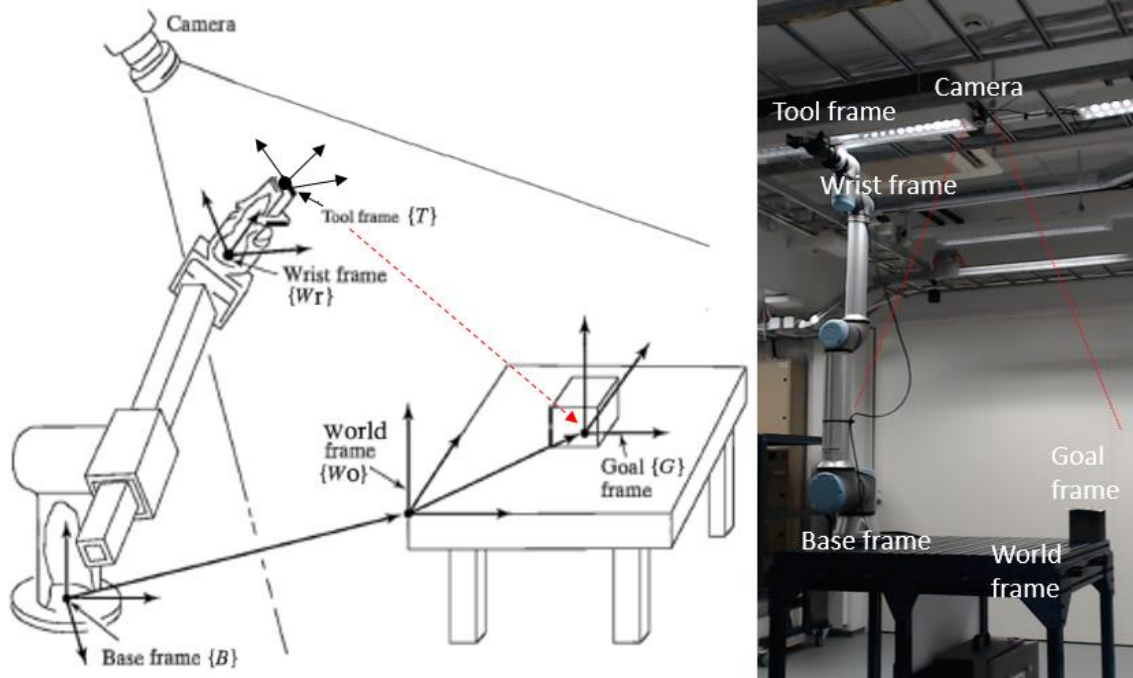
The pose is illustrated in a cartesian coordinate system that has its origin stationed according to the calibration of the camera. The camera is calibrated to recognize the marker on the surface of the table.

### Camera calibration

For the manipulator to know the location of the marker within its own cartesian coordinate system, the marker frame is to be brought to coincidence with the tool frame by calculating the value of the tool frame relative to the world frame. In other words, the camera is calibrated to bring the ArUco marker to a coordinate system where the origin is at the base of UR10e by first calibrating camera to recognise the surface of the table with commonly used checkerboard. In manipulator centric robotics these relative frames are base frame, world frame, wrist frame, tool frame and goal frame. These frames are depicted in Figure 5.22. The centre of the marker represents the goal frame, and the tool frame is specified with respect to the wrist frame, and it defines the TCP. This is defined with its origin between the jaws of the gripper. The world frame is defined with respect to the base frame with its origin at the corner of the table.

For the manipulator to accomplish motion between different frames, geometric transformations are required. The position and orientation of the tool frame with respect to the

world frame places the end-effector to the same area from where the goal frame should be found according to the information from the camera.



**Figure 5.22.** Standard frames and distance (red) between tool frame and goal frame represented in an illustration (left), modified from [9]. Frame representation in real case environment (right).

Representing the marker in the manipulator coordinate system is accomplished by implementing cartesian transforms to the matrix representations of the frames. Frames are connected by vectors that reveal the offset between frames. From Figure 5.22 one can notice that the matrix transform operations are presented in the following order

$$\{G\}^{-1} * \{Wo\}^{-1} * \{B\}^{-1} * \{Wr\} * \{T\} = 0. \quad (5.7)$$

The outcome produces a closed loop, that illustrates transforms from the goal frame to the tool frame in a situation where the process ends back to the starting point - the goal frame. The order of transform equations to match the Equation 5.1 can be presented as:

$$T_{wo}^B * T_G^{wo} = T_{wr}^B * T_T^{wr}. \quad (5.8)$$

The Equation 5.8 describes how transformations from base frame to goal frame through world frame are the same as transformations from base frame to tool frame through wrist frame. In this thesis the goal frame is gained from the camera. To simplify the transformation process, both the camera and the manipulator base can be positioned on the world frame by specifying three points from the world frame, separately with manipulator and with camera, to create a homogenous transformation matrix representation for each of them. These matrix representations define the pose for each of them with respect to the world frame. If the matrix representation of the calibrated camera frame is presented

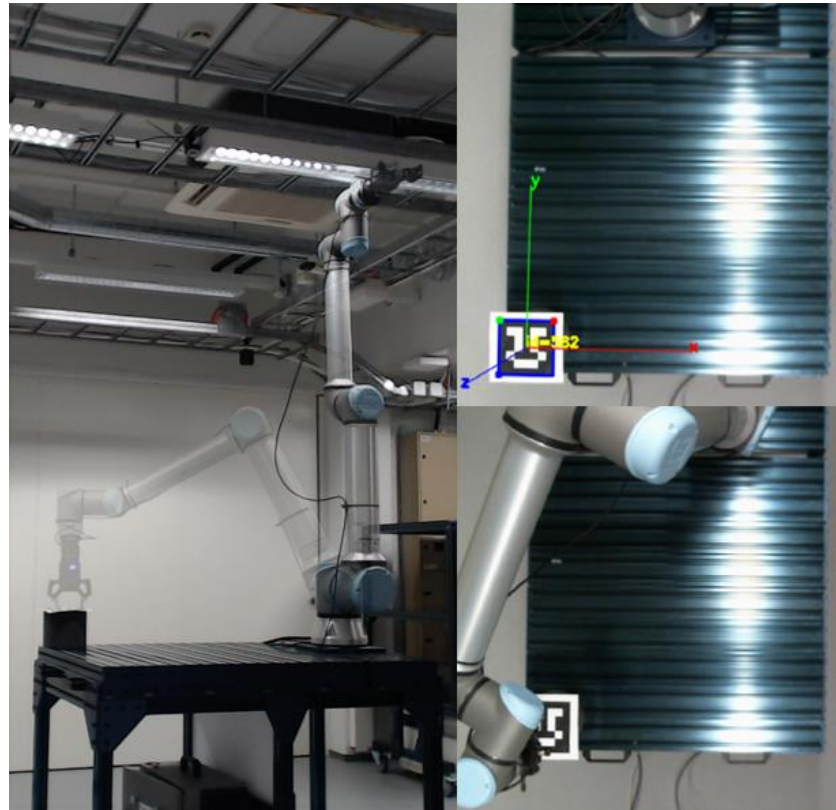
as  $T_{cam0}$  and the marker pose detected by the camera as  $T_{cam}$ , then the pose of the goal frame relative to the world frame can be represented as

$$T_G^{wo} = T_{cam0}^{-1} * T_{cam}, \quad (5.9)$$

where  $T_{cam0}$  is taken as inverse to get from world frame to the camera frame. Also, if the base frame with respect to the world frame is still presented as  $T_{wo}^B$ , then with Equations 5.7-5.9 the transform process can be presented as

$$T_T^B = T_{wr}^B * T_T^{wr} = T_{wo}^B * T_{cam0}^{-1} * T_{cam}. \quad (5.10)$$

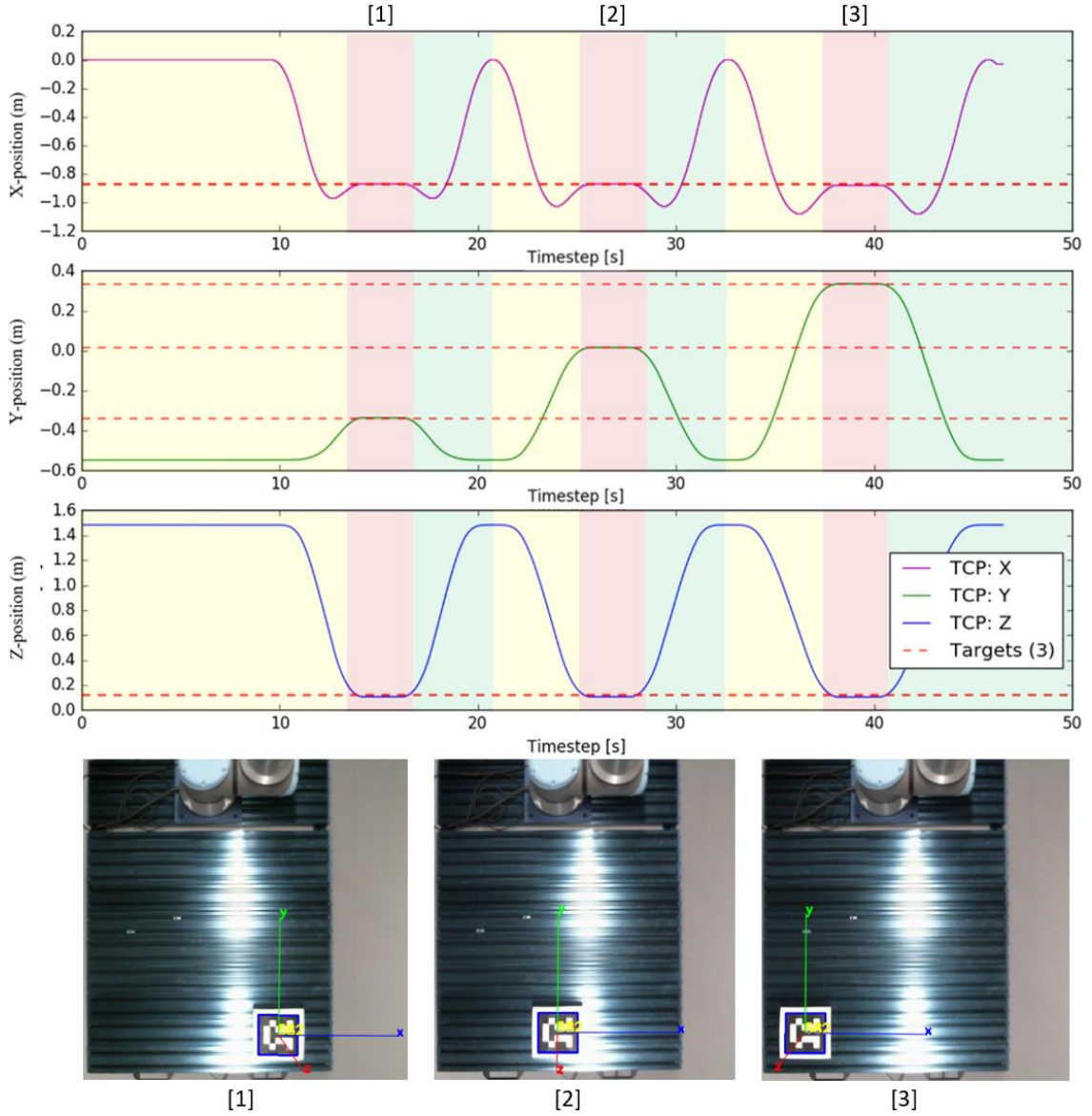
$T_T^B = T_{wr}^B * T_T^{wr}$  represents cartesian transformation from base frame to the tool frame, which is computed by the kinematic equations to enable motion of the end-effector in relation to the base of the manipulator. Because the pose of the marker,  $T_{cam0}^{-1} * T_{cam}$ , and the base frame,  $T_{wo}^B$ , are now also known within the world frame, the end-effector can be controlled with respect to the marker and the tool frame can be matched with the goal frame [9]. Thus, the controller of UR10e can implement kinematic calculations to operate the TCP to reach the ArUco marker as depicted in Figure 5.23.



**Figure 5.23.** ArUco marker localized on the table and the pose is given in accordance to origin in the base of UR10e. Control is managed with inverse kinematics.

The camera precision is verified by checking the distance change from the origin to the ArUco marker by following the TCP of UR10e. In Figure 5.24 the marker position is changed (num. 1-3) and UR10e is automatically controlled to the target by inverse kinematics. Positions are given as a function of time. In Figure 5.24 yellow colour indicates

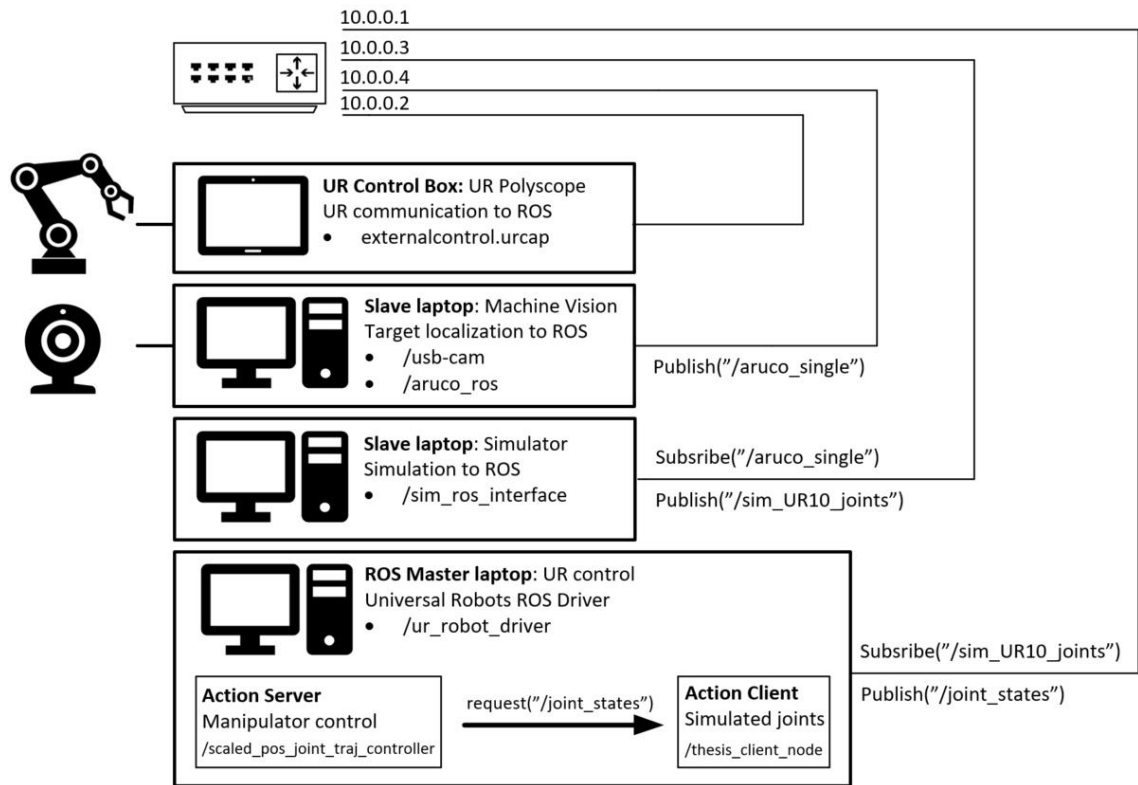
approach, red being at target, and green withdrawal to initial zero-position. UR10e is not located at the centreline of the table and the marker is on top of a 10cm foam cylinder.



**Figure 5.24.** The TCP position of X-, Y- and Z-axis compared to the target (marker) position during a 50 second execution on UR10e. Target position is changed three (3) times (illustrated with dotted red lines). Yellow background indicates a period where the manipulator is approaching the target from the initial state. Red background indicates the TCP is at the target and green background indicates a period where the manipulator is withdrawing back to initial state. The three positions of the target are numbered from 1-3 and they are matched with the TCP at the red areas of similar numbering.

### 5.6.2 System architecture for Sim2Real transfer

The developed ROS communication network incorporates structure according to Figure 5.20. Proposed ROS multimachine network is presented in following Figure 5.25.



**Figure 5.25.** The developed ROS communication network for the Sim2Real transfer.

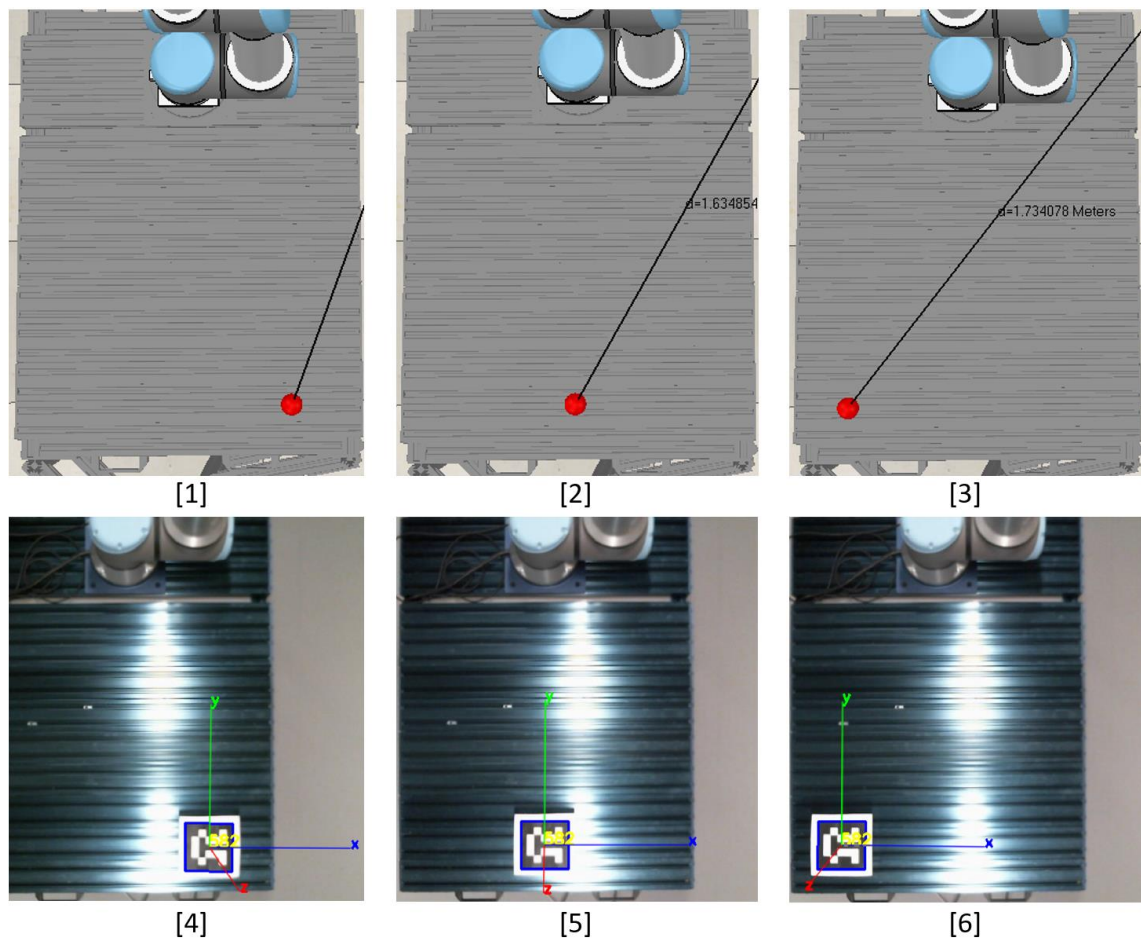
The ROS master laptop upholds the communication of the system. For communicating with UR10e, a UR robot ROS driver package is initialized. The package requires helper packages for the calibration, state control and controllers that provide functionalities for the actual driver package. Additionally, the package includes the action server for the scaled position joint trajectory control, which forwards the given joint states to the actuators of the physical joints. The ROS master laptop initializes the action client node, which subscribes to the simulated UR10 joints topic, "`sim_UR10_joints`". When the action server receives the "`/joint_states`" applicable to the UR10e, the server publishes them for the physical UR10e hardware interface. The UR control box is running the Polyscope operating system, where an external control application "`externalcontrol.urcap`" is running to enable a bridge between the Polyscope and ROS.

The machine vision laptop controls the camera to locate the ArUco marker. Camera is initialized with "`/usb-cam`" node while the calculation for the ArUco marker localization is done at "`/aruco_ros`" node. Camera checks the location of the marker when the operator starts the Sim2Real application and publishes the results to a "`/aruco_single`" topic. Similarly, when the simulated manipulator claims to have reached the target, the application verifies this from the "`/aruco_single`" topic, which should indicate there is no marker data published because the marker is covered by the physical manipulator. Otherwise, the



application produces an error message of a contradiction between the simulated and the physical environment

Finally, the simulator laptop runs CoppeliaSim simulator with DDPG control policy. The architecture is same as in Figure 5.7, except the policy is in evaluation mode thus utilizing what it has learned of the environment. CoppeliaSim provides external APIs to communicate with ROS via Lua interface. The simulation environment module subscribes to the aforementioned ArUco marker topic, which embeds the data of the target position. Thus, the random position of the target dot in the simulation is replaced with the calculated position of the physical ArUco marker. The origin of the simulated environment is scaled to be same as in the physical setup being the base of the manipulator. The coalition of simulated and physical target is depicted in Figure 5.26.



**Figure 5.26.** The position of the target dot in the simulation is subscribed from the machine vision system that is tracking the physical ArUco marker. Tracked position is the centre of the ArUco marker. Images 1 and 4, 2 and 5, and 3 and 6 represent the positions of the target in the simulated and physical setup counterparts.

When the position has arrived and the environment is ready, the simulated UR10 is controlled by DDPG policy to reach the target. As the simulated UR10 moves its joints according to target position as distance, joint velocities, and joint angular positions, UR10

publishes new joint states as `/sim_UR10_joints` topic via `/sim_ros_interface`. This is the package providing ROS to communicate with the simulation. The physical UR10e controller reads the joint positions of the published joint states and moves accordingly. Note, that physical UR10e reads joint positions not velocities. Published joint states also include velocity so this could be used, but at the time the used `/ur_robot_driver` did not seem to provide sufficient results with joint velocities.

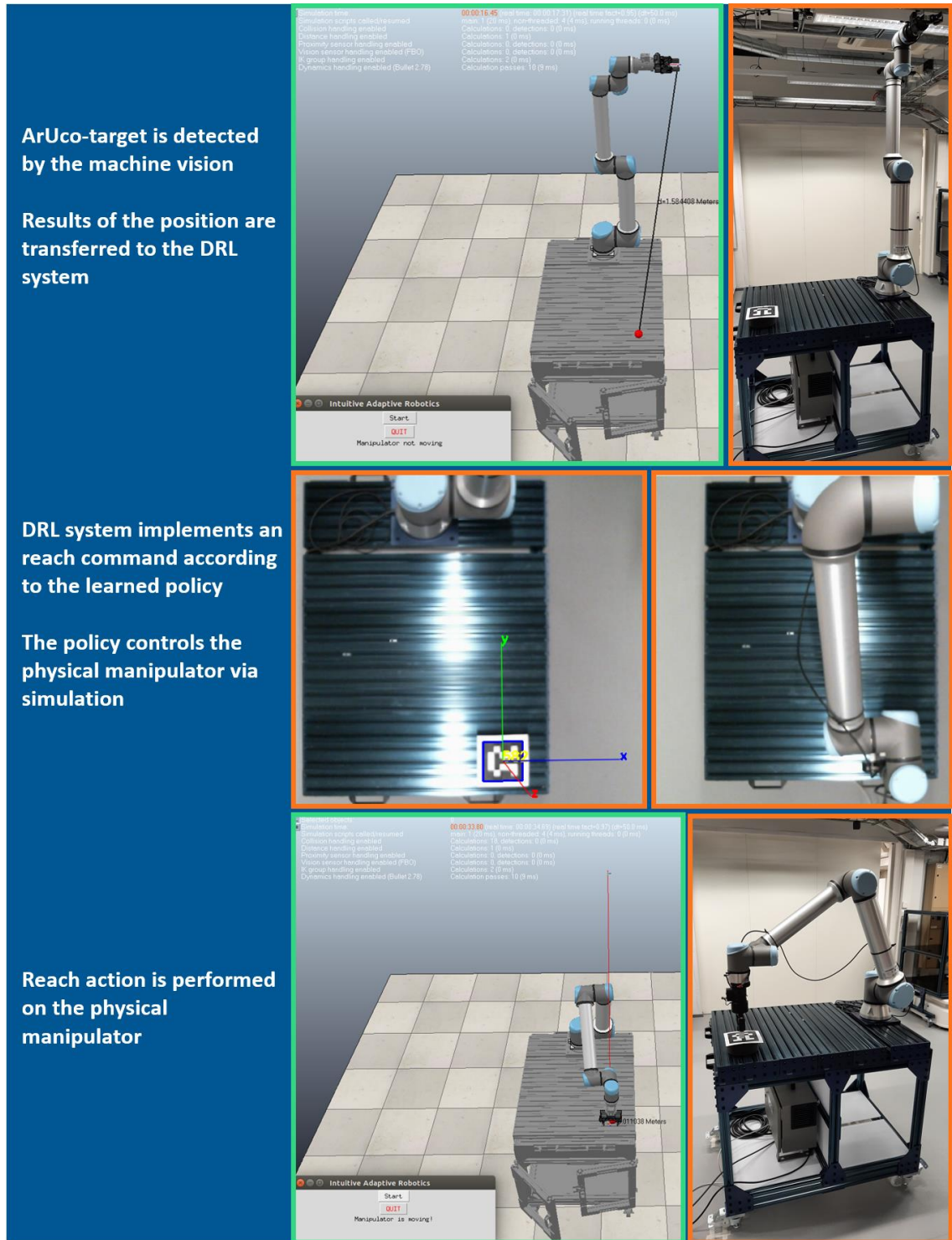
## 5.7 Validating results in physical environment

Chosen DDPG policy was validated in Section 5.5.4 in a simulated environment. Visual inspection and the success metrics demonstrated that it would be safe to proceed with the policy in a real-world scenario. Unlike in simulation, it is not practical to validate the control policy with a 1000-episode scenario in the physical environment. The process would be time-consuming and would require personnel to overwatch each episode. Instead, the comparison of simulated and physical setups is carried out by performing the reaching task with placing the ArUco marker five times to each of the three positions specified in Figure 5.26. Success is measured same as in Algorithm 4 with 30 consecutive steps producing an action that leads to keeping the TCP within the precision threshold vicinity of the target. This way it is possible to follow the orientation of the gripper at the target. The results are summarized in Table 5.10. Figure 5.27 depicts one execution of the validation scenario indicating how Cyber-Physical setup of Figure 5.20 is implicated in practice and how the validation test was displayed.

**Table 5.10** Validation success rates [%] of scenarios (1-6) from Figure 5.26 for the simulated and physical setup over five episodes/repetitions.

	Simulation	Real
<b>Position 1</b> (place 1&4 in Figure 5.26)	100.0	<b>100.0</b>
<b>Position 2</b> (place 2&5 in Figure 5.26)	100.0	<b>100.0</b>
<b>Position 3</b> (place 3&6 in Figure 5.26)	100.0	<b>100.0</b>

The results are as expected considering how well DDPG policy managed during previous validations of Section 5.5. Among RL, reaching is considered a relatively simple task [69] having no multi-objective aspects such as picking and placing that would possibly translate differently to the physical world owing to contact forces etc. Although, the physical environment managed equally well as the simulated when considering the success metric, there seemed to be difference between the trajectory paths of simulated and physical setup. This reality gap is validated by observing the distance variation between the TCP and the target over consecutive approaches.

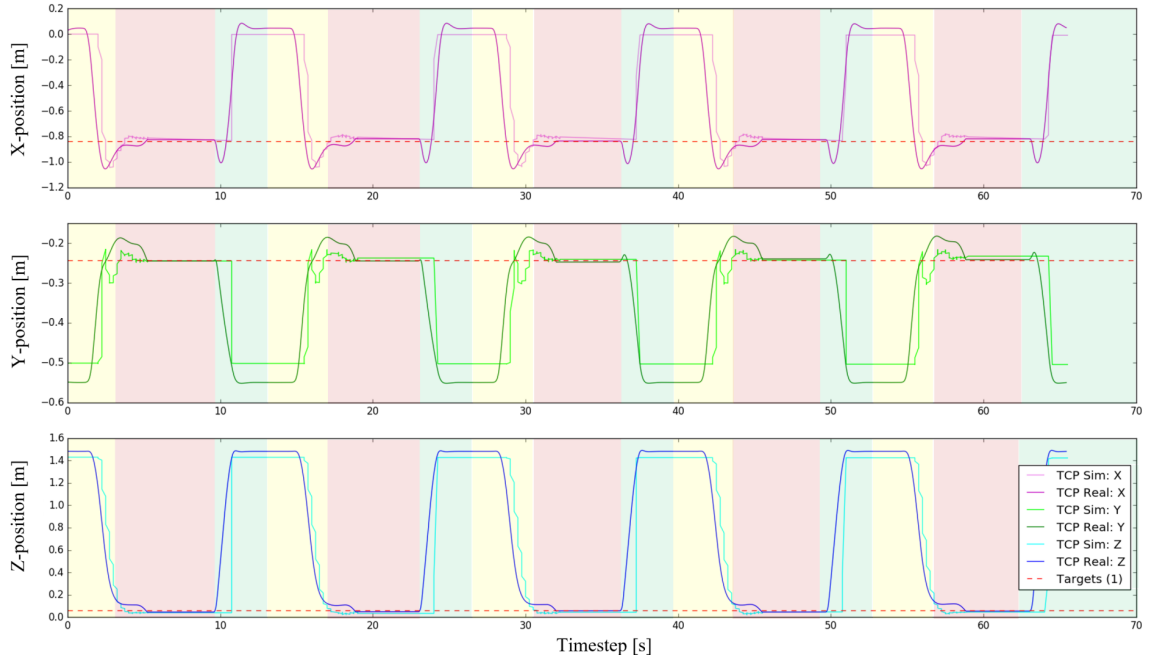


**Figure 5.27.** Sim2Real performed on UR10e controlled by DDPG policy from CoppeliaSim simulator via ROS network. ArUco marker is at position 1 of Table 5.10. Process follows routine depicted in Figure 5.20. Green boxes implicate actions performed in cyber domain and orange boxes actions in the physical domain.

Figure 5.28 depicts five approaches to position 1 of Table 5.10 in Cartesian space for both simulated and physical setup. Dimensions are represented in meters because of convenience and a larger scaled image is presented in Appendix C: Sim2Real transfer results. Simulation setup produces data at 20Hz while physical setup at 170Hz, which is



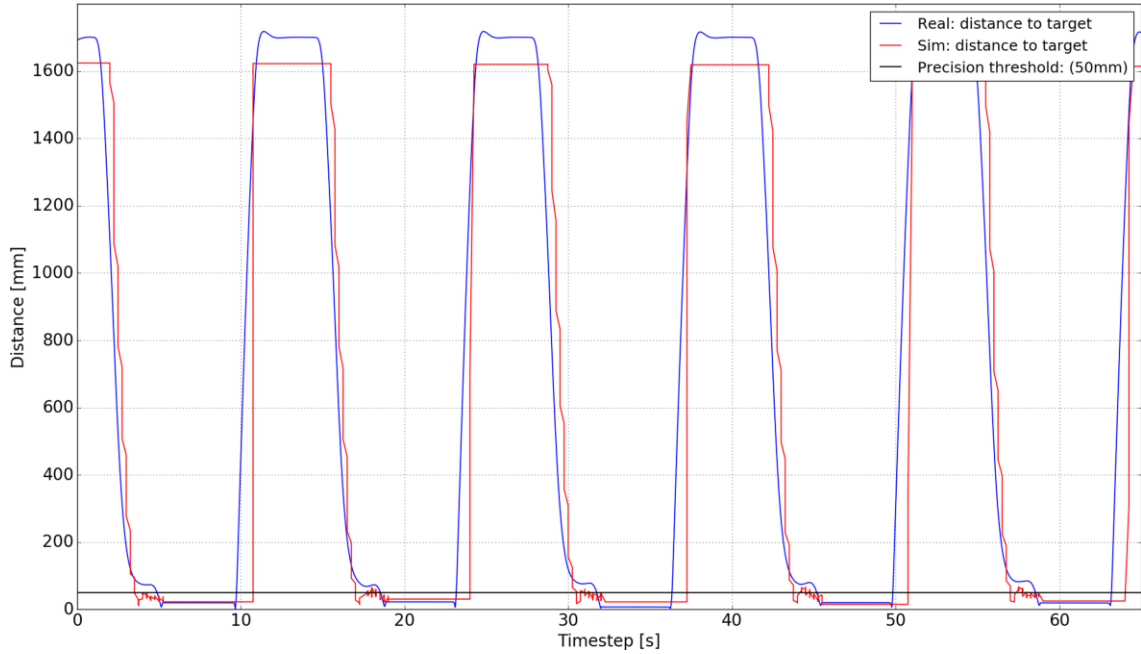
why simulation graph has more rough appearance. After 30 successful steps the manipulator is paused at the target for 5s before returning to the initial zero-position. In simulation the return phase occurs as a sudden movement because the environment is restarted. This gives the simulation graph its sharp edges. On the other hand, because of the restart, the physical manipulator reaches the initial zero-position with a sudden move causing small spikes in the physical graph, since DDPG policy controls the physical UR10e through the simulator.



**Figure 5.28** Five consecutive reaching tasks performed with physical and simulated setup by DDPG policy. Target stays at the same position during all episodes (position 1 of Table 5.10). Target is illustrated with dotted red line. Lighter colours depict the simulated and darker the real manipulator trajectory. Yellow background indicates a period where the manipulator is approaching the target from the initial state. Red background indicates the TCP is at the target and green background indicates a period where the manipulator is withdrawing from the target back to the initial state.

Differences between the simulated (light) and the real (dark) manipulator trajectory graphs demonstrate the width of the reality gap. Known reality gap factors, which were discussed in Section 5.3, are visible in the Y- and Z-axis, where the differences between the graphs are the known 4.9cm and 5.3cm at the initial state of the manipulator. The 3-4cm difference of the TCP among the X-axis is also explainable by the different DH-parameters of UR10 and UR10e [82], as the physical UR10e tries to compensate this difference by including 2.23deg to all joints. The angular difference is visible at the initial state of the manipulators, and it mainly affects the X-axis having only max. 2mm difference on Y-axis.

From the targets point-of-view, the positional difference at the initial zero-position seems to mitigate considerably as both the simulated and the physical manipulator reach the target within the given 5cm (50mm) precision threshold. Figure 5.29 describes Figure 5.28 as point-to-point distance between the target and the manipulator TCP in both simulated and physical setup during the five reaching tasks.



**Figure 5.29** Distance to target measured from simulated and physical manipulator TCP. Black line indicates the precision threshold of 5cm (50mm).

At the target, the distance variations between the setups are explained by the precision threshold parameter, which can be imagined as a sphere around the target. Table 5.11 summarizes distances from the target to the simulated and the physical TCP, while the target has been reached in Figure 5.29.

**Table 5.11** Distances [mm] between target, and simulated and physical TCP as target has been reached in five reaching tasks of Figure 5.29.

	Reach 1	Reach 2	Reach 3	Reach 4	Reach 5
<b>Simulation</b>	22.38	30.09	22.16	15.13	24.70
<b>Real</b>	20.05	22.77	9.41	19.67	19.17
<b>Difference</b>	2.33	7.32	12.75	4.54	5.53

The difference in Table 5.11 describes roughly about the accuracy of the Sim2Real transfer. In optimal situation the difference would be close to zero but considering the average difference at the target is only ~6.5mm at a 1700mm distance with a 50mm precision threshold, the overall results are satisfactory. Since DDPG policy can reach the target with safe trajectories in both simulation and reality, the Sim2Real transfer is a success.

## 6 RECOMMENDATIONS FOR FUTURE WORK

The performed robotic task brought forward the comparison between the traditional control techniques and the DRL based Sim2Real transfer approach. For a simple position control task, that the reach target represents, traditional techniques involving inverse kinematics are generally the easiest solution. If the pose of the target is known, it is relatively simple to forward this information directly to a manipulator and perform a position-based task. However, if the task involves an environment with picking and placing of objects, contact dynamics and adjusting contact forces, a DRL based algorithm with Sim2Real transfer becomes more applicable. Therefore, outside of the presented reaching task, the task specification should include operations where the adaptive nature of DRL and Sim2Real could be showcased more clearly. Similarly, the setup should showcase the generalizability that is expected from the policy and from the Sim2Real process.

The level of dynamics depends on the expected input-output behaviour of a simulator on a specific level of abstraction. Although, simulators like CoppeliaSim can offer an abstraction of the physical world, in this thesis only kinematics were considered. This was the decided level of abstraction. With the right abstraction level, the simulation can train a robot to perform as it is desired in the real world. This would also diminish the reality gap, which was inherently present in the thesis because the simulation model included different models than the real-world setup. With a correct setup and level of abstraction, the reality gap can be managed even though the physical domain gap is vast and contact dynamics modelling is not yet at an optimal level. Common practices among Sim2Real transfer are domain or dynamics randomization for preparing the simulation trained policy to the real world.

Though simplicities in simulators can lead to policies, which will fail to generalize, the structure and depth of the algorithms themselves can affect this as well. Hence, a key aspect of the Sim2Real process is the selection of DRL algorithms, correct hyperparameters and reward functions. Git repositories like OpenAI Gym [93] and RL Baselines Zoo [94] offer verified RL and DRL algorithms with Python APIs that could be utilized in a Sim2Real process. In this thesis the DRL algorithms were scripted by hand to understand the underlying structure, and the hyperparameters were manually tuned. However, implementing algorithms directly from a verified library reduces the possibility of a human-mistake when constructing the agent and the environment. Also, a more stable practice for hyperparameter tuning is to do automatic hyperparameter optimization [94] and perform manual tuning afterwards if seen necessary.

## 7 SUMMARY AND CONCLUSIONS

The thesis studied the tools and steps required to implement a physical system that is trained using Sim2Real robot learning. In robotics, Sim2Real can be seen as a convention to confront the prediction problem and as a way of bridging the reality gap with techniques concerning knowledge transfer. The theory basis of the thesis unravelled the DRL and control terminology, problem field around Sim2Real and concentrated on describing the physical robotic system and the simulated counterpart first as separate ensembles and finally as a unite process within the Sim2Real context.

The physical environment consisted of a Universal Robots manipulator UR10e, which was located on a table from which it was to locate and reach a target Aruco marker. To manage control, the goal in the background was to study how to choose, train and validate a DRL algorithm. DRL is used for sequential decision-making problems, which can be estimated with an MDP. The scope was on two model-free algorithms: DDPG and SAC. Model-free implies the algorithm does not require an accurate environment model and does not use generated predictions of the state transitions and rewards to update its behaviour. This is aligned with the complex environments related to robotics domain. At the core of the study was a set of heterogeneous tools consisting of CoppeliaSim simulator and open-source Python packages.

Algorithms were trained in a simulated counterpart of the physical environment. At first, the training consisted of sparse, dense, and augmented reward functions with two sets of hyperparameters to highlight their influence on the exploration and exploitation of the agent. Augmented reward function demonstrated best results on both algorithms, but the orientation of the manipulator was horizontally inclined posing safety issues if manifested on a physical manipulator. Tuning the parameters with additional heuristics required considerable effort and time. After training the policies multiple times, because individual results do not necessarily converge similarly owing to randomness in the action selection and network initialization, DDPG demonstrated better performance and was chosen as control policy for the Sim2Real transfer.

The solution was based on a zero-shot transfer methodology where DDPG policy controlled the physical manipulator from the simulation via ROS multimachine network. The system included a machine vision system to track the position of the target marker. For the physical UR10e to locate the target it was necessary to calculate the target frame in relation to the base of the manipulator. The position of the target must be same in the

physical setup and in the simulation to minimize the reality gap. The reality gap was demonstrated during the study, which was explainable through dimensional differences of the simulated and physical setup. Additionally, the Sim2Real process was simplified to concentrate on only kinematics meaning the policy does not capture the real dynamic processes of the physical setup. However, the decision was supported by the chosen classical position-control target-reaching task. The trained policy is capable to control UR10e and perform a collision free trajectory path with redundant DoF, meaning no orientation is considered at the target, and with geometrical limitations. Limitations pertain to the location of the target being at the end of the table.

The result and the resources put to develop the Sim2Real approach demonstrated that basic position control tasks such as the chosen target-reaching are more convenient to perform with traditional control approaches. However, if the task involves more requirements for adaptation such as picking and placing then Sim2Real approach becomes more applicable solution. Nevertheless, most of the robotic tasks involve reaching for which the thesis provided a verified functionality of an algorithmic approach for robot control, which performs equally well in simulation as in reality. Videos of the exploration and exploitation phase as well as the final Sim2Real transfer are available at [95].

## REFERENCES

- [1] H. Dong, Z. Ding, and S. Zhang, *Deep reinforcement learning: Fundamentals, research and applications*. Springer Singapore, 2020.
- [2] S. Höfer *et al.*, “Perspectives on Sim2Real Transfer for Robotics: A Summary of the R:SS 2020 Workshop,” Dec. 2020.
- [3] “Universal Robots - Service manual - e-Series - English.” <https://www.universal-robots.com/download/manuals-e-series/service/service-manual-e-series-english/> (accessed Nov. 08, 2021).
- [4] A. G. SUTTON, R. S., & BARTO, *Reinforcement learning: an introduction.*, 2nd ed. tchester Publishing Services, 2018.
- [5] H. Dong, Z. Ding, and S. Zhang, *Deep reinforcement learning: Fundamentals, research and applications*. Springer Singapore, 2020.
- [6] J. Kober, J. A. Bagnell, and J. Peters, “Reinforcement learning in robotics: A survey,” *Int. J. Rob. Res.*, vol. 32, no. 11, pp. 1238–1274, Sep. 2013, doi: 10.1177/0278364913495721.
- [7] M. Kaspar, J. D. Munoz Osorio, and J. Bock, “Sim2Real transfer for reinforcement learning without dynamics randomization,” in *IEEE International Conference on Intelligent Robots and Systems*, Oct. 2020, pp. 4383–4388, doi: 10.1109/IROS45743.2020.9341260.
- [8] J. Collins, S. Chand, A. Vanderkop, and D. Howard, “A review of physics simulators for robotic applications,” *IEEE Access*, vol. 9. Institute of Electrical and Electronics Engineers Inc., pp. 51416–51431, 2021, doi: 10.1109/ACCESS.2021.3068769.
- [9] C. J. J., *Introduction to robotics : mechanics & control*, 3rd ed. Addison-Wesley Pub. Co., , Reading, Mass.
- [10] M. Mihelj *et al.*, *Robotics: Second edition*. Springer International Publishing, 2018.
- [11] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo, *Robotics*. London: Springer London, 2009.
- [12] S. Küçük and Z. Bingül, “The inverse kinematics solutions of industrial robot manipulators,” in *Proceedings of the IEEE International Conference on Mechatronics 2004, ICM’04*, 2004, pp. 274–279, doi:

10.1109/icmech.2004.1364451.

- [13] S. Baglioni, F. Cianetti, C. Braccesi, and L. Landi, "Parametric multibody modeling of anthropomorphic robot to predict joint compliance influence on end effector positioning," in *ASME International Mechanical Engineering Congress and Exposition, Proceedings (IMECE)*, 2013, vol. 4 A, doi: 10.1115/IMECE2013-64815.
- [14] K. P. Hawkins, "Analytic Inverse Kinematics for the Universal Robots UR-5/UR-10 Arms," Georgia Institute of Technology, Dec. 2013.
- [15] L. Cuiyan, Z. Dongchun, and Z. Xianyi, "A survey of repetitive control," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2004, vol. 2, pp. 1160–1166, doi: 10.1109/iros.2004.1389553.
- [16] M. Krämer, C. Rösmann, F. Hoffmann, and T. Bertram, "Model predictive control of a collaborative manipulator considering dynamic obstacles," *Optim. Control Appl. Methods*, vol. 41, no. 4, pp. 1211–1232, Jul. 2020, doi: 10.1002/oca.2599.
- [17] J. Hätönen, T. J. Harte, D. H. Owens, J. Ratcliffe, P. Lewin, and E. Rogers, "Iterative learning control - What is it all about?," in *IFAC Proceedings Volumes (IFAC-PapersOnline)*, Aug. 2004, vol. 37, no. 12, pp. 547–553, doi: 10.1016/S1474-6670(17)31526-4.
- [18] Y. P. Pane, S. P. Nagesh Rao, J. Kober, and R. Babuška, "Reinforcement learning based compensation methods for robot manipulators," *Eng. Appl. Artif. Intell.*, vol. 78, pp. 236–247, Feb. 2019, doi: 10.1016/j.engappai.2018.11.006.
- [19] S. Badillo *et al.*, "An Introduction to Machine Learning," *Clin. Pharmacol. Ther.*, vol. 107, no. 4, pp. 871–885, Apr. 2020, doi: 10.1002/cpt.1796.
- [20] S. Theodoridis, *Machine Learning: A Bayesian and Optimization Perspective*,. Elsevier Science & Technology, 2015.
- [21] Z. C. Lipton, "The mythos of model interpretability: In machine learning, the concept of interpretability is both important and slippery.," *Queue*, vol. 16, no. 3, pp. 31–57, May 2018, doi: 10.1145/3236386.3241340.
- [22] V. M. Aparanji, U. V. Wali, and R. Aparna, "Robotic motion control using machine learning techniques," in *Proceedings of the 2017 IEEE International Conference on Communication and Signal Processing, ICCSP 2017*, Feb. 2018, vol. 2018-Janua, pp. 1241–1245, doi: 10.1109/ICCSP.2017.8286579.
- [23] P. Kormushev, S. Calinon, and D. G. Caldwell, "Reinforcement learning in

- robotics: Applications and real-world challenges,” *Robotics*, vol. 2, no. 3, pp. 122–148, Sep. 2013, doi: 10.3390/robotics2030122.
- [24] N. Liu, Y. Cai, T. Lu, R. Wang, and S. Wang, “Real–Sim–Real Transfer for Real-World Robot Control Policy Learning with Deep Reinforcement Learning,” *Appl. Sci.*, vol. 10, no. 5, p. 1555, Feb. 2020, doi: 10.3390/app10051555.
  - [25] T. Hester *et al.*, “Deep Q-learning from Demonstrations,” *32nd AAAI Conf. Artif. Intell. AAAI 2018*, pp. 3223–3230, Apr. 2017.
  - [26] X. Zhou, T. Bai, Y. Gao, and Y. Han, “Vision-based robot navigation through combining unsupervised learning and hierarchical reinforcement learning,” *Sensors (Switzerland)*, vol. 19, no. 7, p. 1576, Apr. 2019, doi: 10.3390/s19071576.
  - [27] T. M. Moerland, J. Broekens, and C. M. Jonker, “Model-based Reinforcement Learning: A Survey,” *Proc. Int. Conf. Electron. Bus.*, vol. 2018-December, pp. 421–429, Jun. 2020, Accessed: Apr. 29, 2021. [Online]. Available: <http://arxiv.org/abs/2006.16712>.
  - [28] T. P. Lillicrap *et al.*, “Continuous control with deep reinforcement learning,” in *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, Sep. 2016.
  - [29] V. Mnih *et al.*, “Asynchronous Methods for Deep Reinforcement Learning,” *33rd Int. Conf. Mach. Learn. ICML 2016*, vol. 4, pp. 2850–2869, Feb. 2016, Accessed: Apr. 26, 2021. [Online]. Available: <http://arxiv.org/abs/1602.01783>.
  - [30] M. Plappert *et al.*, “Parameter Space Noise for Exploration,” *arXiv*, Jun. 2017, Accessed: Apr. 26, 2021. [Online]. Available: <http://arxiv.org/abs/1706.01905>.
  - [31] R. S. Sutton, “Learning to predict by the methods of temporal differences,” *Mach. Learn.*, vol. 3, no. 1, pp. 9–44, Aug. 1988, doi: 10.1007/bf00115009.
  - [32] A. Givchi and M. Palhang, “Off-policy temporal difference learning with distribution adaptation in fast mixing chains,” *Soft Comput.*, vol. 22, no. 3, pp. 737–750, Feb. 2018, doi: 10.1007/s00500-017-2490-1.
  - [33] C. J. C. H. Watkins, “Learning from Delayed Rewards,” Cambridge, UK, 1989.
  - [34] S. Bhatnagar, R. S. Sutton, M. Ghavamzadeh, and M. Lee, “Natural actor-critic algorithms,” *Automatica*, vol. 45, no. 11, pp. 2471–2482, Nov. 2009, doi: 10.1016/j.automatica.2009.07.008.
  - [35] T. Degris, M. White, and R. S. Sutton, “Off-Policy Actor-Critic,” *Proc. 29th Int. Conf. Mach. Learn. ICML 2012*, vol. 1, pp. 457–464, May 2012, Accessed: May



- 19, 2021. [Online]. Available: <http://arxiv.org/abs/1205.4839>.
- [36] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing Function Approximation Error in Actor-Critic Methods," *35th Int. Conf. Mach. Learn. ICML 2018*, vol. 4, pp. 2587–2601, Feb. 2018, Accessed: May 19, 2021. [Online]. Available: <http://arxiv.org/abs/1802.09477>.
  - [37] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015, doi: 10.1038/nature14236.
  - [38] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic Policy Gradient Algorithms," in *Proceedings of the 31st International Conference on Machine Learning*, 2014, vol. 32, no. 1, pp. 387–395, [Online]. Available: <https://proceedings.mlr.press/v32/silver14.html>.
  - [39] S. Luo, H. Kasaei, and L. Schomaker, "Accelerating Reinforcement Learning for Reaching using Continuous Curriculum Learning," Feb. 2020, doi: 10.1109/IJCNN48605.2020.9207427.
  - [40] V. Gullapalli, J. A. Franklin, and H. Benbrahim, "Acquiring Robot Skills via Reinforcement Learning," *IEEE Control Syst.*, vol. 14, no. 1, pp. 13–24, 1994, doi: 10.1109/37.257890.
  - [41] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, "Benchmarking Deep Reinforcement Learning for Continuous Control," *33rd Int. Conf. Mach. Learn. ICML 2016*, vol. 3, pp. 2001–2014, Apr. 2016, Accessed: May 27, 2021. [Online]. Available: <http://arxiv.org/abs/1604.06778>.
  - [42] B. T. Polyak, "Some methods of speeding up the convergence of iteration methods," *USSR Comput. Math. Math. Phys.*, vol. 4, no. 5, pp. 1–17, 1964, doi: 10.1016/0041-5553(64)90137-5.
  - [43] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep Reinforcement Learning that Matters," *32nd AAAI Conf. Artif. Intell. AAAI 2018*, pp. 3207–3214, Sep. 2017, Accessed: Jun. 09, 2021. [Online]. Available: <http://arxiv.org/abs/1709.06560>.
  - [44] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *35th International Conference on Machine Learning, ICML 2018*, Jan. 2018, vol. 5, pp. 2976–2989.
  - [45] T. Haarnoja *et al.*, "Soft Actor-Critic Algorithms and Applications," Dec. 2018, Accessed: Jul. 26, 2021. [Online]. Available: <http://arxiv.org/abs/1812.05905>.

- [46] S. Kullback and R. A. Leibler, "On Information and Sufficiency," *Ann. Math. Stat.*, vol. 22, no. 1, pp. 79–86, Mar. 1951, doi: 10.1214/aoms/1177729694.
- [47] X. Bin Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel, "Sim-to-Real Transfer of Robotic Control with Dynamics Randomization," *Proc. - IEEE Int. Conf. Robot. Autom.*, pp. 3803–3810, Oct. 2017, doi: 10.1109/ICRA.2018.8460528.
- [48] S. James *et al.*, "Sim-to-Real via Sim-to-Sim: Data-efficient Robotic Grasping via Randomized-to-Canonical Adaptation Networks," *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 2019-June, pp. 12619–12629, Dec. 2018.
- [49] D. F. Gomes, P. Paoletti, and S. Luo, "Generation of GelSight Tactile Images for Sim2Real Learning," Jan. 2021.
- [50] L. Pinto and A. Gupta, "Supersizing Self-supervision: Learning to Grasp from 50K Tries and 700 Robot Hours," *Proc. - IEEE Int. Conf. Robot. Autom.*, vol. 2016-June, pp. 3406–3413, Sep. 2015.
- [51] J. E. Gaudio, T. E. Gibson, A. M. Annaswamy, M. A. Bolender, and E. Lavretsky, "Connections Between Adaptive Control and Optimization in Machine Learning," *Proc. IEEE Conf. Decis. Control*, vol. 2019-Decem, pp. 4563–4568, Apr. 2019, doi: 10.1109/CDC40024.2019.9029197.
- [52] "(PDF) A synthesis of reinforcement learning and robust control theory." .
- [53] E. Rohmer, S. P. N. Singh, and M. Freese, "V-REP: A versatile and scalable robot simulation framework," in *IEEE International Conference on Intelligent Robots and Systems*, Jan. 2013, pp. 1321–1326, doi: 10.1109/IROS.2013.6696520.
- [54] E. Todorov, T. Erez, and Y. Tassa, "MuJoCo: A physics engine for model-based control," in *IEEE International Conference on Intelligent Robots and Systems*, 2012, pp. 5026–5033, doi: 10.1109/IROS.2012.6386109.
- [55] N. Koenig and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2004, vol. 3, pp. 2149–2154, doi: 10.1109/iros.2004.1389727.
- [56] A. Juliani *et al.*, "Unity: A General Platform for Intelligent Agents," Sep. 2018, Accessed: Jul. 13, 2021. [Online]. Available: <http://arxiv.org/abs/1809.02627>.
- [57] G. Brockman *et al.*, "OpenAI Gym," Jun. 2016, Accessed: Jul. 13, 2021. [Online]. Available: <http://arxiv.org/abs/1606.01540>.
- [58] E. Coumans and Y. Bai, "Pybullet, a python module for physics simulation for

- games, robotics and machine learning.” 2016, [Online]. Available: <http://pybullet.org>.
- [59] W. Zhao, J. P. Queralta, and T. Westerlund, “Sim-to-Real Transfer in Deep Reinforcement Learning for Robotics: a Survey,” *2020 IEEE Symp. Ser. Comput. Intell. SSCI 2020*, pp. 737–744, Sep. 2020, Accessed: Jul. 07, 2021. [Online]. Available: <http://arxiv.org/abs/2009.13303>.
  - [60] S. Levine, C. Finn, T. Darrell, and P. Abbeel, “End-to-End Training of Deep Visuomotor Policies,” *J. Mach. Learn. Res.*, vol. 17, Apr. 2015, Accessed: Jul. 02, 2021. [Online]. Available: <http://arxiv.org/abs/1504.00702>.
  - [61] E. Theodorou, J. Buchli, and S. Schaal, “Reinforcement learning of motor skills in high dimensions: A path integral approach,” in *Proceedings - IEEE International Conference on Robotics and Automation*, 2010, pp. 2397–2403, doi: 10.1109/ROBOT.2010.5509336.
  - [62] N. Heess *et al.*, “Emergence of Locomotion Behaviours in Rich Environments,” Jul. 2017, Accessed: Jul. 02, 2021. [Online]. Available: <http://arxiv.org/abs/1707.02286>.
  - [63] J. Schulman, P. Moritz, S. Levine, M. I. Jordan, and P. Abbeel, “High-dimensional continuous control using generalized advantage estimation,” in *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, Jun. 2016.
  - [64] N. Heess *et al.*, “Learning Continuous Control Policies by Stochastic Value Gradients,” 2015.
  - [65] OpenAI *et al.*, “Learning Dexterous In-Hand Manipulation,” *Int. J. Rob. Res.*, vol. 39, no. 1, pp. 3–20, Aug. 2018, Accessed: Jul. 06, 2021. [Online]. Available: <http://arxiv.org/abs/1808.00177>.
  - [66] J. Tan *et al.*, “Sim-to-Real: Learning Agile Locomotion For Quadruped Robots,” Apr. 2018, Accessed: Jul. 06, 2021. [Online]. Available: <http://arxiv.org/abs/1804.10332>.
  - [67] O. Nachum, S. Gu, H. Lee, and S. Levine, “Data-Efficient Hierarchical Reinforcement Learning,” *Adv. Neural Inf. Process. Syst.*, vol. 2018-December, pp. 3303–3313, May 2018, Accessed: Jul. 08, 2021. [Online]. Available: <http://arxiv.org/abs/1805.08296>.
  - [68] M. Andrychowicz *et al.*, “Hindsight Experience Replay,” *Adv. Neural Inf. Process. Syst.*, vol. 2017-December, pp. 5049–5059, Jul. 2017, Accessed: Jul. 08, 2021.

- [Online]. Available: <http://arxiv.org/abs/1707.01495>.
- [69] M. Plappert *et al.*, “Multi-Goal Reinforcement Learning: Challenging Robotics Environments and Request for Research,” Feb. 2018, Accessed: Jul. 09, 2021. [Online]. Available: <http://arxiv.org/abs/1802.09464>.
- [70] A. Nair, B. McGrew, M. Andrychowicz, W. Zaremba, and P. Abbeel, “Overcoming Exploration in Reinforcement Learning with Demonstrations,” *Proc. - IEEE Int. Conf. Robot. Autom.*, pp. 6292–6299, Sep. 2017, Accessed: Jul. 09, 2021. [Online]. Available: <http://arxiv.org/abs/1709.10089>.
- [71] T.-H. Pham, G. De Magistris, and R. Tachibana, “OptLayer - Practical Constrained Optimization for Deep Reinforcement Learning in the Real World,” *Proc. - IEEE Int. Conf. Robot. Autom.*, pp. 6236–6243, Sep. 2017, Accessed: Jul. 01, 2021. [Online]. Available: <http://arxiv.org/abs/1709.07643>.
- [72] A. A. Rusu, M. Vecerik, T. Rothörl, N. Heess, R. Pascanu, and R. Hadsell, “Sim-to-Real Robot Learning from Pixels with Progressive Nets,” Oct. 2016, Accessed: Jul. 06, 2021. [Online]. Available: <http://arxiv.org/abs/1610.04286>.
- [73] F. Golemo, A. Taïga, P.-Y. Oudeyer, A. Courville, A. C. Sim, and A. A. Taïga, “Sim-to-Real Transfer with Neural-Augmented Robot Simulation,” Oct. 2018, Accessed: Jul. 06, 2021. [Online]. Available: <https://hal.inria.fr/hal-01911978>.
- [74] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, “Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World,” *IEEE Int. Conf. Intell. Robot. Syst.*, vol. 2017-September, pp. 23–30, Mar. 2017, Accessed: Jul. 12, 2021. [Online]. Available: <http://arxiv.org/abs/1703.06907>.
- [75] S. Genc, S. Mallya, S. Bodapati, T. Sun, and Y. Tao, “Zero-Shot Reinforcement Learning with Deep Attention Convolutional Neural Networks,” Jan. 2020, Accessed: Nov. 19, 2021. [Online]. Available: <http://arxiv.org/abs/2001.00605>.
- [76] I. Higgins *et al.*, “DARLA: Improving Zero-Shot Transfer in Reinforcement Learning,” *34th Int. Conf. Mach. Learn. ICML 2017*, vol. 3, pp. 2335–2350, Jul. 2017, Accessed: Nov. 19, 2021. [Online]. Available: <http://arxiv.org/abs/1707.08475>.
- [77] N. Akhtar and A. Mian, “Threat of Adversarial Attacks on Deep Learning in Computer Vision: A Survey,” *IEEE Access*, vol. 6, Institute of Electrical and Electronics Engineers Inc., pp. 14410–14430, Feb. 16, 2018, doi: 10.1109/ACCESS.2018.2807385.

- [78] S. Garrido-Jurado, R. Muñoz-Salinas, F. J. Madrid-Cuevas, and M. J. Marín-Jiménez, "Automatic generation and detection of highly reliable fiducial markers under occlusion," *Pattern Recognit.*, vol. 47, no. 6, pp. 2280–2292, Jun. 2014, doi: 10.1016/j.patcog.2014.01.005.
- [79] S. James, M. Freese, and A. J. Davison, "PyRep: Bringing V-REP to Deep Robot Learning," Jun. 2019, Accessed: Jul. 13, 2021. [Online]. Available: <http://arxiv.org/abs/1906.11176>.
- [80] Coppeliarobotics, "Joint Types and Operations." <https://www.coppeliarobotics.com/helpFiles/en/jointDescription.htm> (accessed Oct. 24, 2021).
- [81] "Universal Robots - Max. joint torques." <https://www.universal-robots.com/articles/ur/robot-care-maintenance/max-joint-torques/> (accessed Mar. 09, 2022).
- [82] UniversalRobots, "DH Parameters for calculations of kinematics and dynamics," *Denavit Hartenberg Parameters - DH Parameters*, 2021. <https://www.universal-robots.com/articles/ur/application-installation/dh-parameters-for-calculations-of-kinematics-and-dynamics/>.
- [83] Robotiq Inc, "Robotiq 2-Finger Adaptive Robot Gripper - 85 Instruction Manual," 2014. [Online]. Available: [support.robotiq.com](http://support.robotiq.com).
- [84] Robotic inc., "Robotiq 2F-85 & 2F-140 for e-Series Universal Robots Instruction Manual," 2018. [Online]. Available: [support.robotiq.com](http://support.robotiq.com).
- [85] "Stable Baselines: Reinforcement Learning Tips and Tricks," *Stable Baselines Revision f877c85b*, 2021. [https://stable-baselines.readthedocs.io/en/master/guide/rl\\_tips.html#tips-and-tricks-when-creating-a-custom-environment](https://stable-baselines.readthedocs.io/en/master/guide/rl_tips.html#tips-and-tricks-when-creating-a-custom-environment).
- [86] P. Dhariwal *et al.*, "OpenAI Baselines." Github, 2017, [Online]. Available: <https://github.com/openai/baselines>.
- [87] N. Raffin, Antonin and Hill, Ashley and Ernestus, Maximilian and Gleave, Adam and Kanervisto, Anssi and Dormann, "Stable Baselines3." Github, 2019, [Online]. Available: <https://github.com/DLR-RM/stable-baselines3>.
- [88] Z. Morvan, "train-robot-arm-from-scratch." GitHub, 2019, [Online]. Available: <https://github.com/MorvanZhou/train-robot-arm-from-scratch>.
- [89] J. Ibarz, J. Tan, C. Finn, M. Kalakrishnan, P. Pastor, and S. Levine, "How to train

- your robot with deep reinforcement learning: lessons we have learned,” *Int. J. Rob. Res.*, vol. 40, no. 4–5, pp. 698–721, Apr. 2021, doi: 10.1177/0278364920987859.
- [90] A. Sehgal, H. M. La, S. J. Louis, and H. Nguyen, “Deep Reinforcement Learning using Genetic Algorithm for Parameter Optimization,” *Proc. - 3rd IEEE Int. Conf. Robot. Comput. IRC 2019*, pp. 596–601, Feb. 2019, Accessed: Jul. 27, 2021. [Online]. Available: <http://arxiv.org/abs/1905.04100>.
  - [91] C. P. Janssen and W. D. Gray, “When, What, and How Much to Reward in Reinforcement Learning-Based Models of Cognition,” *Cogn. Sci.*, vol. 36, no. 2, pp. 333–358, Mar. 2012, doi: 10.1111/j.1551-6709.2011.01222.x.
  - [92] “ROS/Introduction - ROS Wiki.” <http://wiki.ros.org/ROS/Introduction> (accessed Mar. 17, 2021).
  - [93] “GitHub - openai/gym: A toolkit for developing and comparing reinforcement learning algorithms.” <https://github.com/openai/gym> (accessed Nov. 12, 2021).
  - [94] “GitHub - DLR-RM/rl-baselines3-zoo: A training framework for Stable Baselines3 reinforcement learning agents, with hyperparameter optimization and pre-trained agents included.” <https://github.com/DLR-RM/rl-baselines3-zoo> (accessed Nov. 12, 2021).
  - [95] “Petri Tikka - YouTube.” <https://www.youtube.com/channel/UCdK32Xz9VGR2bNfXLPX1Y7A> (accessed Dec. 17, 2021).

## APPENDIX A: INSTANTIABLE DDPG REVALIDATION PARAMETERS

	Symbol	Run 1.	Run 2.	Run 3.	Run 4.	Run 5.	Run 6.	Run 7.	Run 8.	Run 9.	Run 10.
Hyperparameters											
Actor learning rate	$\mu_Q$	0.001	0.0001	0.0001	0.001	0.001	0.001	0.001	0.0001	0.001	0.0001
Critic learning rate	$\mu_\pi$	0.001	0.0001	0.0001	0.001	0.001	0.001	0.001	0.0001	0.001	0.0001
Discount factor	$\gamma$	0.9	0.99	0.9	0.99	0.9	0.9	0.99	0.99	0.9	0.99
Target update ratio	$\tau$	0.01	0.001	0.01	0.01	0.01	0.01	0.001	0.001	0.01	0.001
Replay buffer size	B	30000	50000	30000	30000	50000	30000	50000	50000	30000	50000
Batch size	N	32	64	32	32	32	64	64	64	32	64
Gaussian action noise	$\sigma$	2	2	2	2	2	2	2	2	2	2
Episodes of training	M	2000	2000	2000	2000	2000	2000	2000	2000	2000	2000
Steps per episode	T	300	300	300	300	300	300	300	300	300	300
Reward function	R	Augmented	Augmented	Augmented	Augmented	Augmented	Augmented	Augmented	Augmented	Augmented	Augmented
Collision with the table		-	-	Penalty	Penalty	Penalty	Penalty	Penalty*5	Penalty*25	Penalty*10	Penalty*25
Off-limit search		-	-	-	-	Penalty	Penalty	Penalty	Penalty	Penalty	Penalty
Search beneath the table surface level		-	-	-	-	-	Penalty	Penalty	Penalty	Penalty	Penalty
Vertical orientation to the target		Rotation penalty	Rotation penalty	-	-	Rotation penalty	Rotation penalty	Rotation penalty	Rotation penalty	Rotation penalty	Rotation penalty
Rotation norm		0.05	0.05	-	-	0.02	0.1	0.4	0.2	0.2	0.4
2nd joint between / not between 0-45deg		-	-	Reward	Reward	Reward/3	Reward/2	Reward	Penalty/3	Penalty	Reward/2
5th joint motion favourable / unfavourable		-	-	-	-	Reward	Penalty	Reward/2	Penalty	Penalty	Penalty
Distance to target < Distance threshold		Reward	Reward	Reward	Reward	Reward	Reward	Reward	Reward	Reward	Reward
Distance to target > Distance threshold		Penalty	Penalty	Penalty	Penalty	Penalty	Penalty	Penalty	Penalty	Penalty	Penalty
Distance threshold (precision)		5cm	5cm	1cm	2cm	3cm	4cm	5cm	5cm	5cm	5cm
Environment adjustment											
Hight of the target (Z-axis)		100cm	100cm	100cm	100cm	105cm	105cm	100cm	103cm	104cm	100cm

Penalty offset = -0.2

Reward offset = 1.0

## APPENDIX B: INSTANTIABLE SAC REVALIDATION PARAMETERS

	Symbol	Run 1.	Run 2.	Run 3.	Run 4.	Run 5.	Run 6.	Run 7.	Run 8.	Run 9.	Run 10.
<b>Hyperparameters</b>											
Actor learning rate	$\mu_Q$	0.0003	0.0001	0.0003	0.0001	0.0003	0.0003	0.0003	0.0001	0.0003	0.0001
Critic learning rate	$\mu_c$	0.0003	0.0001	0.0003	0.0001	0.0003	0.0003	0.0003	0.0001	0.0003	0.0001
Entropy learning rate	$\mu_{\pi}$	0.0003	0.0001	0.0003	0.0001	0.0003	0.0003	0.0003	0.0001	0.0003	0.0001
Entropy target	$a_{\pi}$	$-dim(AD)$	$-dim(AD)$	$-dim(AD)$	$-dim(AD)$	$-dim(AD)$	$-dim(AD)$	$-dim(AD)$	$-dim(AD)$	$-dim(AD)$	$-dim(AD)$
Discount factor	$\gamma$	0.99	0.9	0.99	0.9	0.99	0.99	0.99	0.9	0.99	0.9
Target update ratio	$\tau$	0.01	0.001	0.01	0.001	0.01	0.01	0.01	0.001	0.01	0.001
Replay buffer size	B	100000	1000000	100000	1000000	100000	100000	100000	1000000	100000	1000000
Batch size	N	64	128	64	128	64	64	64	128	64	128
Gaussian action noise	$\sigma$	2	2	-	-	-	2	2	2	2	2
Episodes of training	M	800	800	800	800	800	800	1600	1600	800	800
Steps per episode	T	200	200	200	200	300	300	300	200	200	200
Reward function	R	Augmented	Augmented	Augmented	Augmented	Augmented	Augmented	Augmented	Augmented	Augmented	Augmented
Collision with the table		-	-	Penalty*5	Penalty	Penalty*5	Penalty*25	Penalty*25	Penalty*25	-	Penalty
Offlimit search		-	-	Penalty	Penalty	Penalty	-	-	-	-	-
Search beneath the table surface level		-	-	Penalty	Penalty	Penalty	-	-	-	-	-
Vertical orientation to the target		Rotation penalty	Rotation penalty	Rotation penalty	Rotation penalty	Rotation penalty	Rotation penalty	Rotation penalty	Rotation penalty	Rotation penalty	Rotation penalty
Rotation norm		0.05	0.05	0.4	0.4	0.4	0.2	0.2	0.4	0.4	0.4
2nd joint between / not between 0-45deg		-	Reward	Penalty	Penalty	-	-	-	Penalty	Penalty	Reward
5th joint motion favourable / unfavourable		-	-	Penalty	Penalty	Penalty	Penalty*5	Penalty*5	Penalty*5	Penalty*5	Reward
Distance to target < Distance threshold		Reward	Reward	Reward	Reward	Reward	Reward	Reward	Reward	Reward	Reward
Distance to target > Distance threshold		Penalty	Penalty	Penalty	Penalty	Penalty	Penalty	Penalty	Penalty	Penalty	Penalty
Distance threshold (precision)		5cm	10cm	7cm	5cm	5cm	5cm	5cm	5cm	5cm	5cm
<b>Environment adjustment</b>											
Hight of the target (Z-axis)		100cm	101cm	100cm	105cm	103cm	102cm	101cm	100cm	100cm	100cm

Penalty offset = -0.2

Reward offset = 1.0



## APPENDIX C: SIM2REAL TRANSFER RESULTS

