Petter Rasmussen
Ole Jørgen Gether Rise

Sim-to-Real Transfer in Deep Reinforcement Learning for Vision-Based Robotic Grasping

The Implementation and Configuration of Simulation Training and Physical Testing

Master's thesis in Mechanical Engineering
Supervisor: Lars Tingelstad
Co-supervisor: Eirik Njåstad
June 2022

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

NTNU
Norwegian University of
Science and Technology

Petter Rasmussen
Ole Jørgen Gether Rise

Sim-to-Real Transfer in Deep Reinforcement Learning for Vision-Based Robotic Grasping

The Implementation and Configuration of Simulation Training and Physical Testing

Master's thesis in Mechanical Engineering
Supervisor: Lars Tingelstad
Co-supervisor: Eirik Njåstad
June 2022

Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

NTNU
Norwegian University of
Science and Technology

Preface

This thesis concludes our master's degree in Mechanical Engineering at the Norwegian University of Technology and Science. It is conducted for the Robotics and Automation group at the Department of Mechanical and Industrial Engineering in relation to the MANULAB facilities.

We want to say thank you to our supervisors for helping us throughout this last year. Thanks to Lars Tingelstad for guiding us through the first half of this last semester. Also, a big thanks to Eirik Njåstad for helping us in finalizing our project.

We want to thank "Robotgjengen og co." at Valgrinda for keeping us sane throughout the stressful parts of the semester. A special thanks to Ludvik Kasbo for helping with some of the code implementation related to the simulations. We look forward to meeting you all in the everyday work life in the years to come.

Finally, we would like to express our gratitude towards our partners, families, and friends for their continuous moral support. The last five years in Trondheim have been challenging, but you have kept us motivated till the very end.

Trondheim, 10.06.2022

Petter Rasmussen & Ole Jørgen Gether Rise

Summary

In this thesis, a sim-to-real framework was developed for the task of vision-based robotic grasping of a cube with a Deep Reinforcement Learning (DRL) agent. Model-free reinforcement learning with Proximal Policy Optimization (PPO) was used to learn an end-to-end policy that mapped visual observations to continuous actions in operational space. RGB images, gripper position, and gripper status were utilized for observations. All frameworks used in this thesis support the use of other observations, including 3D data. A Convolutional Neural Network (CNN) was designed to extract visual features of the RGB images.

A simulation environment for robotic grasping was developed on top of the Robosuite framework using a MuJoCo physics engine. Sim-to-real transfer with a trained policy was made possible by imitating the physical environment. In addition, domain randomization was used during training to limit the sim-to-real gap.

Because of the Robosuite architecture, we propose a sim-to-real solution where the simulated environment runs parallel to the physical environment. This was done to enable a sim-to-real transfer with two different robot controllers. The Robot Operating System 2 (ROS2) framework is used to enable communication between different hardware at the physical lab.

Results of the experimental evaluation indicate that our suggested setup for sim-to-real can be applied to transfer a PPO agent trained for vision-based robotic grasping on a cube. Agents proposing different sim-to-real methods were tested. The agents were tested as proof of concept and confirmed the system's potential. Our best agent trained with cartesian actions, calibrated camera observations, and domain randomization reached a success rate of 45% when transferred directly from simulation. Our results indicate that these three methods help in closing the reality gap.

During testing were, different factors and obstacles discovered that led to subpar results. Among these are joint limit-, friction-, and collision- errors on the physical robot and differences in gripper-cube interaction on the simulator compared to the physical environment.

Sammendrag

I denne avhandlingen ble det utviklet et sim-til-real rammeverk for å løse visjonsbasert robot plukking ved hjelp av Deep Reinforcement Learning (DRL). Modellfri DRL med Proximal Policy Optimization (PPO) algorithmen ble benyttet til å lære en agent å overføre visuelle observasjoner til kontinuerlige bevegelser. RGB-bilder, griperposisjon og griperstatus ble brukt som observasjoner. Alle rammeverk som benyttes i denne avhandlingen støtter bruk av andre observasjoner, inkludert 3D-data. Et Convolutional Neural Network (CNN) ble designet for å trekke ut visuelle trekk i RGB-bildene.

Et simuleringsmiljø for robot griping ble utviklet på toppen av Robosuite - rammeverket som bruker fysikkmotoren MuJoCo for sine kalkulasjoner. Sim-til-real overføring av en ferdig trent agent ble gjort mulig ved å etterligne det fysiske miljøet. I tillegg ble domene randomisering brukt under trening for å begrense sim-til-real gapet.

På grunn av Robosuite arkitekturen foreslår vi en sim-til-real løsning der det simulerte miljøet går parallelt med det fysiske miljøet. Dette ble gjort for å muliggjøre en sim-til-real overføring med to forskjellige robotkontrollere. Rammeverket for Robot Operating System 2 (ROS2) brukes til å muliggjøre kommunikasjon mellom forskjellig maskinvare på det fysiske laboratoriet.

Resultatene av den eksperimentelle evalueringen indikerer at vårt foreslåtte oppsett for sim-til-real kan brukes til å overføre en PPO-agent opplært til visjonsbasert robotgriping på en kube. Agenter som foreslår forskjellige sim-til-real-metoder ble testet. Agentene ble også testet for bekreftet systemets potensial og robusthet. Vår beste agent, opplært med kartesiske handlinger, kalibrerte kameraobservasjoner og randomisering av domene nådde en suksessrate på 45% når den ble overført direkte fra simulering. Våre resultater tyder på at disse tre metodene bidrar til å lukke sim-til-real gapet.

Under testingen ble det oppdaget forskjellige faktorer og hindringer som førte til svekkede resultater. Blant disse er robotledd grenser, for store friskjonskrefte, kollisjonsdefekter og forskjeller i griper-kube interaksjon i simulatoren sammenlignet med det fysiske miljøet.

Contents

Preface i

Summary iii

Sammendrag v

1. Introduction 1
- 1.1. Background and motivation . . . 1
- 1.2. Problem description . . . 2
- 1.3. Previous work . . . 2
- 1.4. Related work . . . 3
- 1.5. Thesis structure . . . 4

I. Fundamentals 7

2. Preliminaries 9
- 2.1. Reinforcement Learning . . . 9
  - 2.1.1. Markov Decision Process . . . 9
  - 2.1.2. Key Concepts of reinforcement learning . . . 10
  - 2.1.3. Difference from other machine learning paradigm . . . 11
  - 2.1.4. States and Observations . . . 11
  - 2.1.5. Action Space . . . 12
  - 2.1.6. Policies . . . 12
  - 2.1.7. Episodes . . . 13
  - 2.1.8. Reward and Return . . . 13
  - 2.1.9. Value Function . . . 14
  - 2.1.10. Advantage Function . . . 15
  - 2.1.11. Model . . . 15
- 2.2. Deep Reinforcement Learning . . . 16
  - 2.2.1. Deep Learning . . . 16
  - 2.2.2. Convolutional Neural Networks . . . 17

2.3. Taxonomy of Reinforcement algorithms 18
2.3.1. Model based and model free methods 18
2.3.2. Model-Free RL Methods 20
2.4. Robot Learning 22
2.4.1. Deep Reinforcement Learning and robotics 22
2.4.2. Sample Inefficiency 23
2.4.3. Exploration vs Exploitation 23
2.4.4. Optimization challenges in model-free on-policy algorithms 23
2.4.5. Use of simulation 24
2.4.6. Sim-to-Real, closing the Reality Gap 24
2.4.7. Domain Randomization 25
2.4.8. Domain Adaptation 26
2.4.9. Exploration in robotic DRL 26
2.4.10. Reward shaping 28
2.5. Proximal Policy Optimization Algorithms 28
2.5.1. Policy Optimization 29
2.5.2. Trust Region Policy Optimization 30
2.5.3. Clipped Surrogate Objecttive 31
2.6. ROS2 32
2.6.1. General 32
2.6.2. Nodes,topics, services and messages 33
2.6.3. DDS communication 33

II. System design 35

3. Simulation 37
3.1. Simulators 37
3.1.1. Simulation Environment 37
3.1.2. Simulator used 40
3.2. Simulation Setup 40
3.2.1. Robosuite 40
3.2.2. The basearena 41
3.2.3. KUKA iiwa 14 r820 in simulation 42
3.2.4. OSC controller in simulation 43
3.2.5. Robotiq 2f-85 in simulation 43
3.2.6. The Object 46
3.3. Observation Space 46
3.3.1. Image observation 46
3.3.2. Observations in Robosuite 48
3.4. Action Space 48
3.5. Reward Function 50

3.6. Parameters . . . 51
3.6.1. Output parameter . . . 51
3.6.2. Control frequency . . . 52
3.7. Domain Randomization . . . 52
3.7.1. Visual Randomization . . . 53
3.7.2. Dynamic Randomization . . . 54
3.7.3. Further randomization . . . 55

4. Training **57**
4.1. Frameworks . . . 57
4.1.1. Algorithm used . . . 57
4.1.2. Stable Baselines . . . 59
4.1.3. Wrapping the environment . . . 60
4.1.4. High Performance Computing . . . 61
4.2. Network Structure . . . 62
4.2.1. Hyperparameters . . . 63

5. Physical environment **65**
5.1. KUKA LBR iiwa 14 R820 . . . 65
5.1.1. Description . . . 65
5.1.2. KUKA Sunrise cabinet . . . 65
5.1.3. Motion control . . . 66
5.1.4. KUKA iiwa in MANULAB . . . 67
5.2. Robotiq 2F-85 Gripper . . . 67
5.2.1. Description . . . 67
5.2.2. Software and communication . . . 68
5.2.3. Gripper in MANULAB . . . 69
5.3. Zivid Two Camera . . . 69
5.3.1. Description . . . 69
5.3.2. Software and Communication . . . 70
5.3.3. Hand-eye Calibration . . . 71
5.3.4. Zivid in MANULAB . . . 72
5.4. Physical Setup . . . 74
5.4.1. MANULAB environment . . . 74
5.4.2. ROS2 communication . . . 74

6. Sim-to-Real Transfer **79**
6.1. Solution . . . 79
6.1.1. Robot actions . . . 79
6.1.2. Gripper actions . . . 81
6.1.3. Observations . . . 81
6.2. PolicyNode communication . . . 81

6.3. Safety . . . 82

III. Experiments 85

7. Hand-eye Calibration 87
7.1. Results . . . 88
7.1.1. Estimated transformation matrix . . . 88
7.1.2. Calibrated transformation matrix . . . 88
7.1.3. Comparison . . . 89
7.2. Discussion . . . 90

8. Training in simulation 91
8.1. Results . . . 91
8.2. Discussion . . . 95
8.2.1. Agent steps during training . . . 96
8.2.2. The Baseline, Custom Camera *Cus* . . . 96
8.2.3. Custom Camera with Joint Limits . . . 97
8.2.4. Calibrated Camera . . . 98
8.2.5. Limiting the action space . . . 98
8.2.6. Training with Domain Randomization . . . 98
8.2.7. Notes on results . . . 99

9. Physical experiments 101
9.1. Experiment design . . . 101
9.2. Results . . . 103
9.3. Discussion . . . 103
9.3.1. Joint limits . . . 105
9.3.2. Calibrated camera . . . 106
9.3.3. Domain randomization . . . 107
9.3.4. Limiting the action space . . . 109
9.3.5. Placement and orientation of the cube . . . 110
9.3.6. Sources of error . . . 111

IV. Discussion and Conclusion 113

10. Discussion 115
10.1. Zivid Two . . . 115
10.2. Sim-to-real solution . . . 116
10.3. Errors during physical testing . . . 116
10.4. Robotiq 2F-85 . . . 116
10.5. General discussion . . . 117

11. Conclusion and Further Work 119
11.1. Conclusion . . . 119
11.2. Further work . . . 120
11.2.1. Further work in simulation . . . 120
11.2.2. Further work in training . . . 121
11.2.3. Further work in physical environment . . . 122

A. Test setup 133
A.1. Contacts and box placement . . . 133

B. Training Plots 135

C. Digital appendix 139
C.1. Github code . . . 139
C.1.1. Training code . . . 139
C.2. ROS2 . . . 139
C.2.1. ROS2 package . . . 139
C.3. Attached .zip file . . . 139
C.3.1. hand_eye_calibration . . . 140
C.3.2. Videos . . . 141
C.3.3. Raw data . . . 142

D. Hyperparameters 143

E. Hardware communication 145
E.1. Communication with the KUKA iiwa 14 . . . 145
E.1.1. General info . . . 145
E.1.2. Procedure for turning it on . . . 145
E.1.3. Network configuration with Linux . . . 145
E.2. Communication with the Zivid camera . . . 146
E.2.1. General info . . . 146
E.2.2. How to connect a Zivid Two with Linux or Windows . . . 146
E.2.3. Network configuration with Linux . . . 146
E.3. Communication with the Robotiq 2F-85 gripper . . . 147
E.3.1. General info . . . 147
E.3.2. Procedure for setting it up . . . 147
E.3.3. How to connect with python package . . . 148
E.3.4. Use the python package . . . 148
E.4. Run the ROS2 system . . . 149

F. Edited Robosuite code 151

List of Figures

2.1. The agent-environment interaction loop 10
2.2. Overview of the information flow through a node 16
2.3. Convolutional neural network comparison 17
2.4. Taxonomy of algorithms in modern RL 19
2.5. Domain randomization visualization 25
2.6. Proximal Policy Optimization 29
2.7. Surrogate function plots 32

3.1. Physical and simulated environment comparison 41
3.2. Improved KUKA iiwa model 42
3.3. Robotiq 2F-85 revisions 44
3.4. Gripper-table interaction 45
3.5. Gripper-object interaction 45
3.6. Physical and simulated image observation comparison 47
3.7. Matrix to convert rotation matrix from lab to simulation 47
3.8. Object exceeding reward limit 50
3.9. End effector position of the gripper 51
3.10. Domain randomization examples 53

4.1. Framework Architecture 58
4.2. Multi Input Policy architecture 62

5.1. KUKA iiwa communication 66
5.2. Under-actuated gripper 68
5.3. Optimal working distance for Zivid 2 camera 70
5.4. Hand-eye calibration 71
5.5. Zivid placement 73
5.6. ROS2 communication 75

6.1. Agent-Environment interaction 80
6.2. PolicyNode communication 82
6.3. Gripper crash 83

7.1. Hand-eye calibration example 87

8.1. Global agent steps 92
8.2. Mean Episodic Reward, without Domain Randomization 93
8.3. Success Rate without Domain Randomization 93
8.4. Mean Episodic Reward with Domain Randomization 94
8.5. Success Rate with Domain Randomization 94
8.6. Mean and STD with and without Domain Randomization 96

9.1. Physical test result graph 104
9.2. Cube placement graph 104
9.3. Cube orientation graph 104

A.1. Gripper-object contacts 133
A.2. Cube position in experiments 134

E.1. Wire configuration for Robotiq controller 147

List of Tables

3.1. Used friction forces 46
3.2. Robosuite observations 48
3.3. Robosuite actions 49
3.4. Robosuite action clipping 51

5.1. KUKA iiwa joint limits 68

8.1. Number of agent steps to 100% success rate 95
8.2. Agents tested in simulator 95

9.1. Physical test result 103
9.2. Termination of physical tests 105

C.1. Video attachments 142

Acronyms

A3C Asynchronous advantage actor- critic. 3, 20

C51 51-atom agent. 21

CNN Convolutional Neural Network. iii, v, 17, 60, 62

DDPG Deep Deterministic Policy Gradient. 3, 21, 22, 27, 59

DEF Singularity Definition File. 61, 62

DQN Deep Q-Network. 21

DRL Deep Reinforcement Learning. iii, v, viii, 1–3, 12, 16, 21, 22, 25, 26, 59, 101, 119

FOVY Field of View Y-axis. 53

GAN Generative Adversarial Network. 26

HDR High Dynamic Range. 70, 72, 73

HPC High Performance Computing. 61, 119

IPG Interpolated Policy Gradients. 59

KL Kullback–Leibler Constraint. 31

MDP Markov decision Process. 9, 10, 16, 22

NN Neural Network. 25, 60

NTNU Norwegian University of Science and Technology. 1

OSC Operational Space Controller. 43, 48

PPO Proximal Policy Optimization. iii, v, 3, 12, 20, 21, 28, 29, 31, 32, 57, 59, 63, 91, 92, 97, 98

PTP Point-to-Point. 67, 80

RL Reinforcement Learning. xiii, 9–11, 18, 19, 22–24, 37, 40, 59

ROS2 Robot Operating System 2. iii, v, 32, 119

SAC Soft Actor-Critic. 3, 22, 59

SIF Singularity Image File. 61, 119

SSH Secure Shell. 61

TCP Tool Center Point. 67

TRPO Trust Region Policy Optimization. 3, 21, 29–31, 59

URDF Unified Robot Description Format. 42

Chapter 1.

Introduction

1.1. Background and motivation

The Norwegian University of Science and Technology (NTNU) has a mission to develop the technological foundation for the future society. One of the initiatives initiated to encourage research towards this mission is the MANULAB facilities and its Industry 4.0 Laboratory. With its robotic manipulators, the facilities can be used for research within warehouse automation and autonomous manufacturing. Our thesis is mainly a contribution to the Industry 4.0 Laboratory competence in the work towards NTNU mission.

Robotic grasping is an important field of research within robotics and an essential part of the transition to Industry 4.0. By finding reliable grasping solutions that can be placed in an arbitrary environment, the Norwegian industry can benefit from manipulators to a higher degree than today.

Deep Reinforcement Learning (DRL) can help control robots by training an agent through experience. Ideally, the essence of a robotic grasp can be learned and used for grasping various objects in arbitrary environments without the need to program the robot motion for every task. The algorithms and tools used for training differ in complexity, but frameworks are continuously being built to make grasping research available to more researchers.

One of the powerful tools in DRL is the use of simulation for testing and training. Training an agent can be time-consuming because of the need for a high number of training steps to get an agent with satisfactory performance. Simulation can be used for training, but it is challenging to use the simulation-trained agent directly in the physical environment because of the visible and dynamic differences. These differences are often referred to as the sim-to-real gap. This thesis utilizes the hardware available in MANULAB and the open-source frameworks for simulated DRL training to build a platform for further sim-to-real research within robotic

grasping and reinforcement learning.

1.2. Problem description

The main objective of this thesis is to perform a sim-to-real transfer of a DRL agent trained for solving a vision-based robotic grasping task and compare different sim-to-real techniques. Specifically, the task is to train a vision-based DRL agent in simulation to control a robot arm to grasp and lift a cube in the MANULAB facilities and further compare agents trained with sim-to-real techniques against each other. To do this is, the task separated into three parts.

Firstly a viable simulated training must be implemented. It can be time-consuming and error-prone to build an environment and implement a DRL algorithm from scratch. Due to this, is the use of modular frameworks that already have reliable solutions for making environments preferable.

Secondly, a setup for training in the simulated environment must be implemented. To be able to train a policy, a DRL algorithm must be chosen. By utilizing high-performance computing platforms, simulation training can be completed at a faster pace. The simulation framework must be altered to fit the chosen computing platform.

Finally, a physical test setup must be made to measure the agents' performance in a physical environment. To configure a setup to handle the sim-to-real transition, must an action and observation space that fits with both the framework and the physical environment be found. The communication structure, hardware placement, and the experimental design for collecting empirical data must also be considered.

1.3. Previous work

This thesis is an extension of two separate specialization reports written by the authors last fall. *Deep Reinforcement Learning In Robotic Manipulation* was a literature review of DRL in robotic manipulation. It introduced different algorithms and theories within robotics. The thesis discussed the challenges and solutions in the field and evaluated the state-of-the-art research done on the topic. It lastly looked at different simulators used in research and discussed how they could be used to close the reality gap.

Development Of A Test Setup For Pick And Place Tasks With Deep Reinforcement Learning discusses different solutions for communication and configuration

of a physical test setup. Except for a change in the robot manipulator, the hardware proposed was the same are used in this thesis. Parts of the communication architecture based on ROS2 were also used in this thesis.

1.4. Related work

The field of Deep Reinforcement Learning (DRL) is extensive, and many sources presents relevant theory. This thesis has used [80], [37], and [19] as the primary sources of theory. These three present everything from basic theory to state-of-the-art use cases.

To get a good overview of different challenges and possibilities in the robot learning field. in the field is [27] used. They present several case studies involving robotic DRL and discuss common challenges and how other papers have addressed them.

Model-free algorithms are the focus of this task. Papers comparing and showing a promising result of sim-to-real transfer with domain randomization have been studied to get inspiration for which algorithms perform well in this field. [27] discuss the latest research on model-free DRL, among these algorithms are SAC [22], PPO [72], A3C [51], DDPG [46] and TRPO [71]. Papers like [98], [16] that discuss PPO used on partial observation are used for inspiration.

For inspiration on the observation and action space are [16] [36] and [5] is used. The QT-opt algorithm proposed by Kalashnikov et al. [36] shows high-performance policy in a physical environment through large-scale physical training. In addition to using the RGB data as an observation, they show how gripper status and height of the end effector can improve the performance substantially. They also have a small action space that limits the complexity of the grasp and training needed. [5] present a suggestion to action space and argue for its benefit. This action space is further used in this paper.

The paper [99] covers the fundamental background behind sim-to-real transfer in Deep Reinforcement Learning (DRL) and an overview of the main methods being utilized at the moment. A categorization of the most recent papers is also presented, as well as the use of different DRL algorithms. As an inspiration to the domain randomization employed in this thesis is [87] used. This paper explores different domain randomization techniques for a good transfer from the simulator to the real world.

For comparison of different simulator frameworks, have the papers [40] and [15] been looked at. These two compare different physics simulators on the task of robotic manipulation, speed, and accuracy.

Simulation training is widely used in robotics both for testing before deploying

the system in real life and for training a DRL policy. Zhu et al. [100] present Robosuite, a modular framework for robotic simulation built on Mujoco as a physics engine. They propose a system based on torque control of the robot with different robots and environments available. An example of a paper using this framework is Zhu et al. [16]. They show how the framework can be used for training a high-performance policy. The network, hyperparameters, and reward shaping used in this paper are used as a foundation for our training.

1.5. Thesis structure

The thesis is separated into four parts with a total of 11 chapters.

Fundamentals

Chapter 2 explains the fundamental theory related to this thesis. Deep reinforcement learning theory and algorithms are presented as well as tools for improving the agent performance. The ROS2 framework is also presented.

System design

This part describes the implementations in every segment of the system. It presents the solutions and argues for why the choices were made.

Chapter 3 describes the choices made when building a simulated environment. Robosuite is presented as the simulation framework, and measures to imitate the physical environment are discussed. The observation space, action space, and reward function related to the training is also presented in this chapter because they are implemented in Robosuite.

Chapter 4 describes the frameworks network structure and hyperparameters used for the training of an agent in simulation.

Chapter 5 presents hardware and software used in the project and their configuration. The communication between the separate parts is also described.

Chapter 6 presents the final solution for the sim-to-real transfer of the trained agent. The communication, actions, and observations are addressed.

Experiments

This part is separated into three chapters. Each chapter presents results from experiments and discusses the findings individually.

Chapter 7 presents results and discussion of the hand-eye calibration performed to find the placement of the camera in the physical environment.

Chapter 8 presents results and discussion from the simulation training of several agents. The differences between the agents are presented and discussed.

Chapter 9 presents the results from the physical testing related to the sim-to-real transfer of the trained agents. The agents are compared and common errors are discussed.

Discussion and Conclusion

As an ending to the thesis, we present our conclusion and suggestions to further work.

Chapter 10 discusses the segments of the system that was not discussed in chapter 9 part of the thesis.

chapter 11 presents concluding remarks and suggests what can be improved and further researched.

Part I.

Fundamentals

Chapter 2.

Preliminaries

This chapter presents relevant theory within RL used in this thesis. The specialization project *Deep Reinforcement Learning In Robotic Manipulation* influences some parts. The ROS2 chapter is influenced by the *Development Of A Test Setup For Pick And Place Tasks With Deep Reinforcement Learning* specialization project.

2.1. Reinforcement Learning

Reinforcement Learning (RL) is the task of learning an agent to solve a specific problem through rewards. An agent can be a computer, a pet, or anything that can learn through trial and error. The agent gets observation from an environment and can interact with the environment through actions as shown in fig. 2.1.

The basic idea of reinforcement learning is to capture the most critical aspects of the problem. This is done by interacting with an environment to achieve a goal. A learning agent must be able to sense the state of its environment to some extent and must be able to take actions that affect the state. The agent must have a goal or goals relating to the state of the environment. Any method well suited to solving such problems is considered a reinforcement learning method.

2.1.1. Markov Decision Process

This subsection introduces the problem of Markov decision Process (MDP). This is the standard mathematical formalism of reinforcement learning and refers to the fact that the system obeys the *Markov property*, meaning transitions only depend on the most recent state and action and no previous history.

An MDP is a 5-tuple (S, A, R, P, ρ_0)

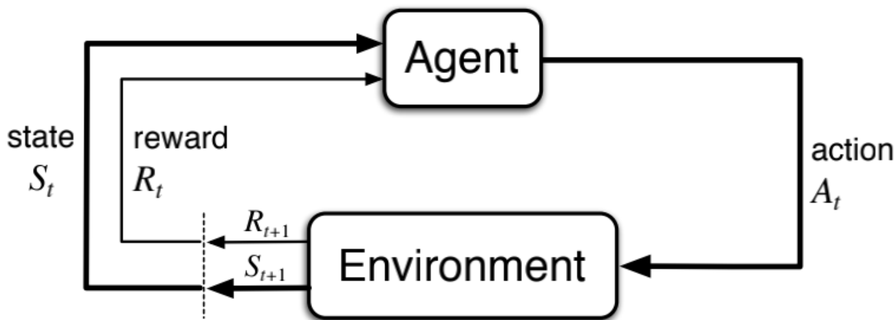


Figure 2.1.: The agent-environment interaction loop

- S is the set of all valid states
- A is the set of all valid actions
- $R : S \times A \times S \to \mathbb{R}$ is the reward function, with $r_t = R(s_t, a_t, s_{t+1})$
- $P : S \times A \to \mathcal{P}(S)$ is the transition probability function, with $P(s'|s, a)$ being the probability of transitioning into state s' if you start in state s and take action a
- ρ_0 is the starting state distribution.

At each time step, the process is in some state **s**, and the agent may choose any action **a** that is available in state **s**. The process responds at the next time step by moving into a new state **s'** given by the state transition probability function $P(s'|s, a)$. The agent receives a reward of r_t depending on how good it is to be in the new state.

MDPs are a classical formalization of sequential decision making, where actions influence not just immediate rewards but also subsequent situations or states and through those future rewards. Thus MDPs involve delayed rewards and the need to trade off immediate and delayed rewards. In MDPs, we estimate the action-value $q_*(s, a)$ of each action a in each state s, or we estimate the value $v_*(s)$ of each state given optimal action selections.

These state-dependent quantities are essential to accurately assigning credit for long-term consequences to individual action selections.

2.1.2. Key Concepts of reinforcement learning

The agent and the environment are the main elements of RL. The environment is the world the agent lives in and interacts with. The agent and environment

interact continuously. At every step, the agent receives a state or partial observation from the environment and then decides an action to take based on those observations. The environment responds to the actions and returns a reward signal to the agent. A reward is a number that tells how good or bad the current environment state is.

The agent's goal is to maximize the cumulative reward, also known as the return.

Beyond the agent and the environment, one can identify four main sub-elements of a reinforcement learning system according to [80]: a *policy*, a *reward signal*, a *value function*, and, optionally, a *model* of the environment. These will be explained in closer detail further down in this section.

2.1.3. Difference from other machine learning paradigm

RL is considered to be a third machine learning paradigm, alongside supervised learning and unsupervised learning.

In supervised learning, training data with the exact solution is given to the function approximator. This is not the case with reinforcement learning, where no exact solution is given. In interactive problems, it is challenging to obtain examples of desired behavior that are both correct and representative of all the situations in which the agent must act. Sometimes the different states and actions an agent can take are so big that it would take way too much time to give out all this labeled data. Because of this, an agent must be able to learn from its own experiences when exploring uncharted territory.

Reinforcement learning is also different from unsupervised learning, where the goal is to find a hidden structure of the unlabeled data given. One might think these two are pretty similar, but reinforcement learning tries to maximize a reward signal instead of finding a hidden structure.

2.1.4. States and Observations

A state s is a complete description of the state of the world. When an agent can observe the complete state of the environment, the environment is *fully observed*. An observation o is a partial description of a state, meaning it may neglect some information from the world. When the agent only receives partial observations, the environment is partially observed.

2.1.5. Action Space

The set of valid actions in an environment is called the *action space*. Actions spaces can be divided into discrete- and continuous-action spaces. In discrete, only a finite number of moves are available to the agent per action. In contrast, actions are real-valued vectors in continuous action spaces, meaning the agent has many more options when choosing an action for each state.

2.1.6. Policies

A policy defines how the agent acts at a given time. One can think of a policy as a mapping from observed states of the environment to actions in those states. The policy determines the agent's behavior and can vary from a simple function or a lookup table to a major computation such as a search process.

It can be either deterministic or stochastic. Deterministic policies map observations directly to actions, while stochastic policies output a probability distribution over actions. This allows the agent to explore the state space without always taking the same action and is helpful in the exploration vs exploitation trade-off described in section 2.4.3.

There are different kinds of stochastic policies. The stochastic policy used in this thesis is a *diagonal Gaussian policy*. This is one of the most common kinds of stochastic policies in DRL when it comes to continuous action spaces.

There are two important computations when training with stochastic polices:

1. Sample actions from the policy
2. Compute log likelihood of particular actions $log(\pi_\theta(a|s)$

For diagonal Gaussian policies, a multivariate Gaussian distribution is described by a mean vector μ and a covariance matrix, $\sum$. Diagonal Gaussian refers to a special case where the covariance matrix only has entries in the diagonal. This means it can be represented as a vector.

The mean actions $\mu_\theta(s)$ are mapped with a neural network from observations. In PPO the covariance matrix is represented as a single vector of log standard deviations, $log(\sigma)$. It is important to note that this is not a function of the state but standalone parameters. Log standard deviation is used because it can take on any values in $(-\infty, \infty)$ compared to the standard deviation, which must be non-negative. According to [37] is it easier to train parameters when you do not have any constraints.

The formula for sampling actions is given by eq. (2.1)

$$a = \mu_\theta(s) + \sigma_\theta(s) \cdot z \tag{2.1}$$

z is a noise vector from spherical Gaussian ($z \sim N(0, I)$ multiplied element-wise with the standard deviation. The calculation of the Log-likelihood of a k -dimensional action a, for a diagonal Gaussian with mean $\mu = \mu_\theta(s)$ and standard deviation $\sigma = \sigma_\theta(s)$, is given by eq. (2.2)

$$\log \pi_\theta(a|s) = -\frac{1}{2}\left(\sum_{i=1}^{k}\left(\frac{(a_i - \mu_i)^2}{\sigma_i^2} + 2\log\sigma_i\right) + k\log 2\pi\right). \tag{2.2}$$

2.1.7. Episodes

Episodes, also often called *rollouts* or *trajectories* is a sequence of states, actions and rewards who end in a terminal state.

$$E = (s_0, a_0, r_1, s_1, a_1, r_2, ...) \tag{2.3}$$

The initial state is randomly sampled from the *start-state distribution*. This is often a distribution set by the environment.

The state transitions, which describe what happens between the state at time t, and the state at time $t + 1$ are controlled by the laws of the environment. In the case of robot grasping, can this be gravitation, collision detection, limits on the robot, and more. The states can be either deterministic or stochastic.

2.1.8. Reward and Return

A reward signal defines the goal of a reinforcement learning problem. For each time step, the environment sends the reward, in the form of a number, to the reinforcement learning agent. The agent's objective is to maximize the total reward it receives in the long run. This means that the reward signal defines what is good and bad for the agent. The reward affects how the policy changes. If a low reward follows an action selected by the policy, then the policy may be changed to select some other action in that state in the future. One can think of the reward signal as a stochastic function dependent on the state of the environment and the actions taken.

Return is the total reward over an episode. There are different kinds of returns, but the two most common are *finite-horizon undiscounted return*. This is the sum of rewards obtained in a certain amount of steps eq. (2.4).

$$R(\tau) = \sum_{t=0}^{T} r_t. \tag{2.4}$$

Another common return is the *infinite-horizon discounted return* eq. (2.5). This return looks at an infinite time horizon. However, it adds a discount factor $\gamma \in (0,1)$ which determines the present value of future rewards: a reward received k time steps in the future is worth only γ^{k-1} times what it would be worth if it were received immediately.

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t. \tag{2.5}$$

2.1.9. Value Function

A value function specifies what is good in the long run. The value function returns a value of each state that is the total amount of reward an agent can expect to earn in the future when starting from that state. Whereas rewards determine an agent's immediate wish to be in a specific state, values determine the long-term desirability of states after taking into account the states that are likely to follow and the rewards available in those states. Without rewards, there could be no values, and the only purpose of estimating values is to achieve more rewards.

We seek actions with the highest value, not the highest reward because these actions obtain the highest reward in the long run. Unfortunately, it is much harder to determine values than determine rewards. Rewards are given directly by the environment, but values must be estimated and re-estimated from the sequences of observations an agent makes over its entire lifetime. The most critical component of almost all reinforcement learning algorithms we consider is a method for efficiently estimating values.

According to [37] are there four main value functions to note. The list below is from [37]:

1. The On-Policy Value Function, $V^{\pi}(s)$, which gives the expected return if you start in state s and always act according to policy π:

$$V^{\pi}(s) = \underset{\tau\sim\pi}{E}[R(\tau)\,|s_0 = s] \tag{2.6}$$

2. The On-Policy Action-Value Function, $Q^{\pi}(s,a)$, which gives the expected return if you start in state s, take an arbitrary action a (which may not have come from the policy), and then forever after act according to policy π:

$$Q^{\pi}(s,a) = \underset{\tau \sim \pi}{E}[R(\tau)\,|s_0 = s, a_0 = a] \tag{2.7}$$

There is also an Optimal Value Function, $V^*(s)$ and Optimal Action-Value Function, $Q^*(s,a)$ where it is assumed to act after the optimal policy.

These four value functions obey special self-consistency equations called **Bellman equations**. The Bellman equation tries to give a value of the state the policy is in. This is done by adding together the expected reward of the current state and the value of wherever the policy land next.

The optimal policy in s will select the action that maximizes the expected return from starting in s. The optimal action $a^*(s)$ from the optimal Action-Value Function _Action-Value_Function@cref_Action-Value_Function@cref_Action-Value_Function@
will be:

$$a^*(s) = \arg\max_a Q^*(s,a). \tag{2.8}$$

If there are multiple optimal actions, the optimal policy may randomly select one of them.

2.1.10. Advantage Function

Advantage functions estimate how much better an action is than others on average. In other words the advantage function $A^{\pi}(s,a)$ eq. (2.9), corresponding to a policy π, gives a calculation on how much better it is to take a specific action a in state s compared to a random action in that state.

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s). \tag{2.9}$$

2.1.11. Model

Some reinforcement learning systems need a model. The model tries to mimic the environment, or in other words, tries to predict how the environment will behave. For example, given a state and action, the model might predict the resultant next state and next reward. Models are used for planning; this means deciding a course of action by looking at different future situations before they are actually experienced. Methods for solving reinforcement learning problems that use models and planning are called model-based methods, as opposed to more straightforward model-free methods that are explicitly trial-and-error learners-

2.2. Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) combines reinforcement learning and deep learning. When the number of states in our MDP's is of high dimensionality, we can no longer solve the problem with traditional RL algorithms. This is where deep RL comes into play. The neural network often represents a policy $\pi(a|s)$ taking in the high dimensional state space (images from a camera or raw sensor stream from a robot), sending it through the network, and returning an action based on the states or observations given.

With DRL, we only need to tune some parameters in our Neural Network to get a good policy instead of going through every state for each step.

2.2.1. Deep Learning

Deep learning is learning based on neural networks of several layers. An overview of how this communication might look is shown in fig. 2.2. The layers consist of nodes that combine input from the input data with a set of weights. The weights either amplify or dampen the input, assigning how important that input is in regards to the task the network is trying to solve. The product between the input and the weights is summed together and sent to an activation function. This activation function's purpose is to determine to what extent the signal sent in should progress further down through the network.

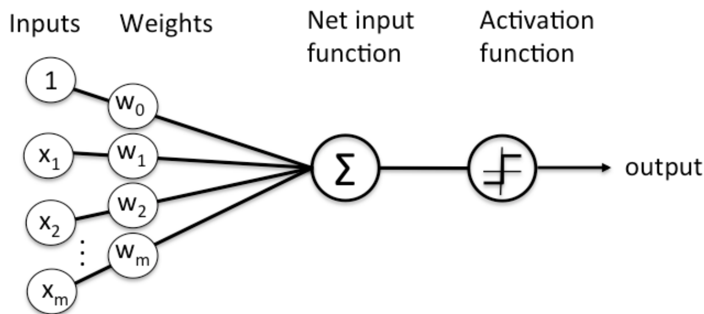


Figure 2.2.: Overview of the information flow through a node

A DRL is defined as having at least one hidden layer, meaning there is a layer of nodes between the input and output layer.

A forward pass is used to move forward through the network. This takes us from the input through the network until we reach the output. To measure how good the output is, a cost function indicates how well the network performs.

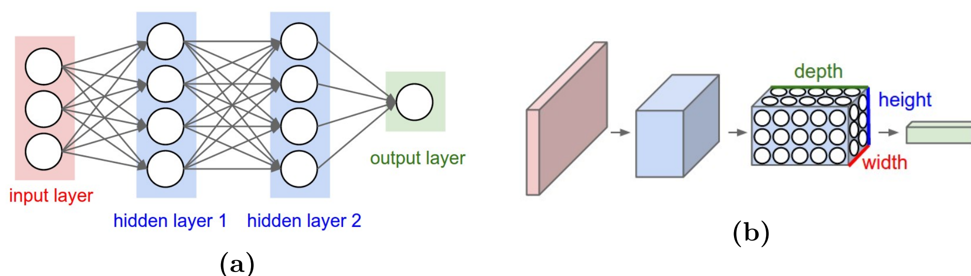


Figure 2.3.: From [12]. Left: A regular 3-layer Neural Network. Right: A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels)

After the cost function is calculated, we go backwards through our network and adjust our weights and biases to optimize the cost function. This is called a backward pass and is how the network improves over time.

2.2.2. Convolutional Neural Networks

Convolutional Neural Network (CNN)s make the assumption that the inputs are images. This allows the encoding of certain properties into the architecture, making the forward function more efficient to implement. Consequently, it reduces the number of parameters in the network.

Unlike regular Neural Networks, the layers of the CNN have neurons arranged in 3 dimensions: **width, height, depth**. The neurons in a layer will only be connected to a small region of the layer before it. In contrast, a fully-connected layer has a connection to all the neurons in the layer before.

Layers in CNNs

This subsection describes the individual layers used in our thesis.

Convolutional Layer . The CONV layer's parameters consist of a set of learnable filters. Every filter is small in width and height but extends through the full depth of the input volume. During the forward pass, each filter is convoluted across the width and height of the input volume. Furthermore, the dot product between the entries of the filter and the input at any position is computed. The filter produces a 2-dimensional activation map after

sliding over the width and height of the input volume. This 2d-map gives the responses of that filter at every spatial position. The sizes of these filters times the depth of the input volume define the number of weights (plus 1 in bias) for this specific filter.

The size of the output volume is controlled by the input volume size and three hyperparameters: **depth, stride** and **zero-padding**. The depth corresponds to the number of filters used on the input volume. Stride defines the number of pixels to slide between every calculation. Zero-padding is wrapping zeros around each depth element in the input volume. This is so that the size of the output volume can be controlled.

The backward pass for a convolution operation, for both the data and the weights, is also a convolution but with spatially-flipped filters.

Relu and TanH activation function . The RELU layer applies an element-wise activation function, such as the **max(0,x)**. This does not change the size of the volume. The TanH activation function takes any real value as input and outputs values from -1 to 1. The larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to -1.0.

$$TanH(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.10}$$

Fully-connected Layer Neurons in a fully connected layer have full connections to all activations in the previous layer. Their activations can hence be computed with a matrix multiplication followed by a bias offset

2.3. Taxonomy of Reinforcement algorithms

This section points out some of the differences of RL algorithms and discusses some trade-offs between the different categories.

Figure 2.4 categorises different RL algorithms into different branches of the Reinforcement Learning (RL) field. It is worth mentioning that it is a simplified overview. It is challenging to draw an accurate taxonomy of RL algorithms because the modularity of algorithms is not well-represented by a tree structure.

2.3.1. Model based and model free methods

In deep learning, a model means a specific function with initialized parameters (pre-trained model) or learned parameters (well-trained model), such as a deep

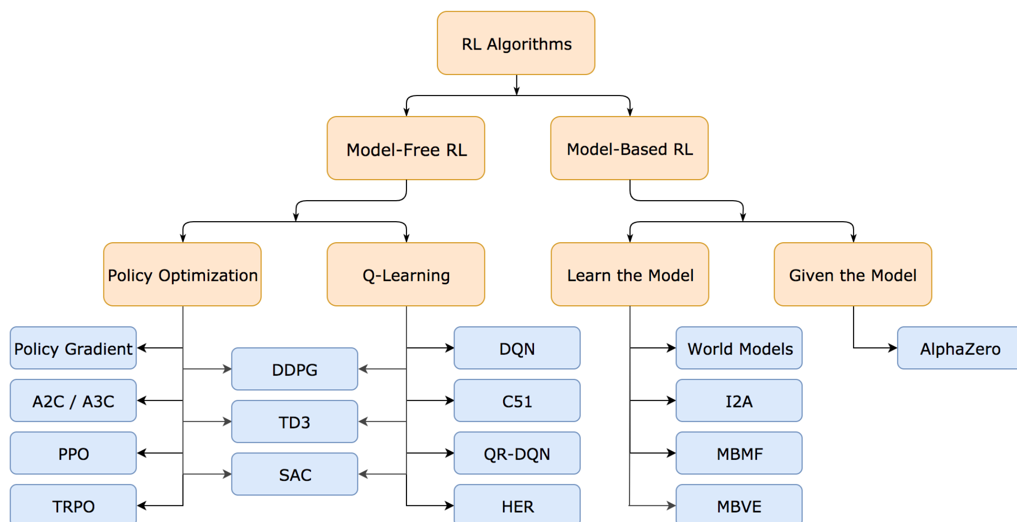


Figure 2.4.: Taxonomy of algorithms in modern RL from [37]

neural network. However, in model-based reinforcement learning, a "model" is the ensemble of acquired environmental knowledge.

We can split model-based methods into two categories. Methods that work with a given model and methods that learn the model. All five elements in the Markov decision process are known for the methods that work with a given model, meaning we can use model-based RL algorithms. Consequently, we can use value - and policy - iteration directly without interacting with the environment. An example of model-based methods is the AlphaGo algorithm [73] where the rules of the Go game are specified to the computer. This means that Go's transition- and reward functions are all known to the agent to evaluate and improve its policy.

In the second category, the methods cannot directly acquire the model due to the complexity of the environment. Instead, the agent can learn a model from interactions with the environment and then apply the model in policy improvement.

The critical advantage of model-based methods is that the future states and rewards can be anticipated in advance via the environmental model. This helps the agent to do better planning.

The disadvantage of model-based methods is that they can be hard to represent explicitly if the environment has complex dynamics. The learned models are usually inaccurate in practice, which induces estimation bias. The policy estimated and improved based on a biased model usually fails when applied in the physical environment.

Because of the challenges mentioned above, a model-free algorithm was chosen. We do not know the transition probabilities from state to state or the reward function in model-free learning. The model-free algorithms either estimate a "value function" or the "policy" directly from experience. In other words, the agent interacts with the environment directly and improves its performance based on the explored states.

Model-free methods are simpler to implement than model-based methods because they do not need a model, which can be hard to learn. However, model-free methods also have some challenges. The cost of exploring the physical environment can be high in terms of time consumption, wear and tear on the equipment, and safety risks. By training in a simulator, we can explore several steps per second without worrying about wear and tear or safety risks. Therefore, we can bypass some of the biggest challenges in both model-free - and model-based - learning by choosing a model-free algorithm.

2.3.2. Model-Free RL Methods

There are two main approaches to representing and training agents with model-free RL.

Policy Optimization

This family represents a policy explicitly as $\pi_\theta(a|s)$. The parameters θ are updated directly by gradient ascent on the performance objective $J(\pi_\theta)$, or indirectly, by maximizing local approximations of $J(\pi_\theta)$. Commonly, this optimization is done **On-Policy**. This means that only collected data while acting according to the most recent policy version is used to update the parameters. Policy optimization also usually involves learning an approximator V_ϕ(s) for the on-policy value function $V^\pi(s)$, which gets used in figuring out how to update the policy.

Two common policy optimization methods are Asynchronous advantage actor-critic (A3C) [51] and Proximal Policy Optimization (PPO) [72]. A3C performs gradient ascent to maximize performance directly. On the other hand, PPO updates indirectly maximize performance by maximizing a surrogate objective function, which estimates how much $J(\pi_\theta)$ will change as a result of the update.

Challenges in model-free DRL

One of the problems model-free reinforcement learning is suffering from is that the training data generated is itself dependent on the current policy. This is because

our agent generates its own training data by interacting with the environment rather than relying on a static data set which is the case in supervised learning.

This means that the data distribution of our observations and rewards are constantly changing as our agent learns, which is a major cause of instability in the training process.

Reinforcement learning also suffers from a high sensitivity in hyperparameter tuning and initialization. One example of this could be a learning rate that is too large. This would result in a policy update that pushes the policy network into a region of the parameter space where it will collect the next batch of data under a poor policy. The data gathered is hard to learn anything from, meaning the policy will have a hard time recovering. PPO and TRPO limits this problem by clipping too big updates.

Q-Learning

This family learn an approximator $Q_\theta(s, a)$ for the optimal action-value function. An objective function based on the **Bellman equation** section 2.1.9 is often used. This optimization is often performed **off-policy**, meaning that data collected at any point during training can be used to update the agent. The action taken by the Q-learning agent are given by eq. (2.8) only with the approximator $Q_\theta(s, a)$ instead of the optimal action value function. Some examples of Q-learning methods are Deep Q-Network (DQN) [52], which revolutionized the field of DRL and 51-atom agent (C51) [3] a variant that learns a distribution over return whose expectation is the optimal action value function.

Combining Policy Optimization and Q-Learning

The primary strength of policy optimization is that it directly learns the optimal policy. Instead of making a function that takes a state as input and outputs Q values for all actions, the policy instead learns a function that outputs the best action that can be taken from that state. This tends to make them more stable and reliable than Q-Learning. A discussion on how Q-Learning methods can fail is further discussed in sections 4.3.2 and 3 in the following paper [81].

The strength of Q-Learning methods is that they are more sample efficient upon learning. This is because they can reuse data more effectively than policy optimization methods.

Some algorithms use both policy optimization and Q-Learning. These algorithms can trade off between the strengths and weaknesses of either side. Deep Deterministic Policy Gradient (DDPG) [46] is an algorithm that learns a deterministic

policy and a Q-function jointly by using each to improve the other. Soft Actor-Critic (SAC) [22] is another variant which uses stochastic policies, entropy regularization, and a few other tricks to stabilize learning. SAC score higher than DDPG on standard benchmarks.

2.4. Robot Learning

This section discusses some important topics and challenges when training robots with deep reinforcement learning.

2.4.1. Deep Reinforcement Learning and robotics

A robot is an inherently active agent that interacts with the physical world and often operates in uncontrolled or detrimental conditions. Robots must perceive, decide, plan, and execute actions based on incomplete and uncertain knowledge. Within the robotic manipulation context, DRL offers a framework and a set of tools for learning dexterous manipulation directly from sensor data or raw pixels.

A robotics problem is characterized by defining a state and action space and the dynamics that describe how actions influence the system's state. The state-space includes the robot's internal states and the environment's state. The robot can be thought of as the agent in a RL interface. Quite often, the state is not directly observable - instead, the robot is equipped with sensors, which provide observations that can be used to infer the state, and thus, we have a partial observable MDP.

The goal may be defined either as a target state to be achieved or as a reward function to be maximized. We want to find a controller, otherwise known as policy, from a deep neural network that maps states to actions in a way that maximizes the reward when executed.

When combined with robotics, reinforcement learning is often represented with continuous high-dimensional action and state space. For robot manipulation in the real world, collecting samples is often expensive and time-consuming. Experiences are also sensitive to a variety of noise and difficult to reproduce. To collect a single training sample, it might take a few minutes for a robot to move around or perform the tasks. Because of this, we need sample-efficient algorithms when it comes to DRL with robotics.

2.4.2. Sample Inefficiency

Sample inefficiency concerns the algorithms' abilities to learn from limited data and is one of the main reasons that seriously limit the applications of RL in robot manipulation. Even some of the best current RL algorithms can still be impractical due to sample inefficiency. There are multiple causes for the problem. Many algorithms try to learn to perform a task from scratch and therefore need a lot of data to learn. Another thing is that some algorithms are not good enough to take advantage of current data. On-policy algorithms require new data for every update step. So when data collection in robotics is already a time-consuming affair, we are in significant need of algorithms that can exploit current data as much as possible.

Some classes of RL algorithms are more sample efficient than others. Off-policy methods are about an order of magnitude more data-efficient than on-policy methods. Model-based methods could be another order of magnitude more data-efficient than their model-free counterparts [27].

2.4.3. Exploration vs Exploitation

A central challenge in RL is the exploration-exploitation problem, i.e. to exploit the solution we know to give the max reward or explore new solutions that might give even higher rewards. Let's say that the agent maintains estimates of the action values. Then at any time step, there is at least one action whose estimated value is greatest. These actions are referred to as the *greedy actions*. Exploiting is when one of these greedy actions is chosen, meaning the agent exploits its current knowledge. If instead, a nongreedy action is chosen, we say that the agent is exploring. This enables the agent to improve its estimate of the nongreedy action's value.

The downside of exploring is that the maximum amount of reward possible is not achieved. The benefit is that higher rewards are expected in the long run because after better actions are discovered, they can be exploited many times in the future. Because it is impossible to both explore and exploit with any single action selection, one often refers to the "conflict" between exploration and exploitation.

2.4.4. Optimization challenges in model-free on-policy algorithms

One of the problems that model-free on-policy RL is suffering from is that the training data generated is dependent on the current policy. This is because the agent generates training data by interacting with the environment rather than

relying on a static data set which is the case in supervised learning. This means that the data distribution of observations and rewards constantly changes as the agent learns. This is a major cause of instability in the whole training process.

Model-free on-policy RL also suffers from a high sensitivity in hyper parameter tuning and initialization. One example of this could be a learning rate that is too large, meaning the policy update that pushes the policy network into a region of the parameter space where it will collect the next batch of data under a poor policy. The data now gathered is hard to learn anything from, meaning the policy will have a hard time recovering.

2.4.5. Use of simulation

Considering that simulators are becoming more and more accurate over the years, it is a good tool to use on the step toward training real robots. One solution to the sample-inefficiency challenge is to collect more data. While collecting enough data on the physical system is slow and expensive, simulation can run orders of magnitude faster than in real-time and start many instances simultaneously. Data can also be collected continuously without human intervention. Experiments can be reset automatically in simulation, and safety is not a problem. Thus, prototyping in simulation is faster, cheaper, and safer than experimenting on the real robot. The rapid pace of experiments allows us to efficiently shape the reward function, sweep the hyper-parameters, fine-tune the algorithm, and test whether a given task falls within the robot's hardware capability.

2.4.6. Sim-to-Real, closing the Reality Gap

Because simulation is an abstraction of real-world conditions, policies learned in simulation typically perform worse when transferred onto hardware. This is what we call the *reality gap* and is one of the most important considerations when selecting a simulator for reinforcement learning purposes. Different methods have been employed successfully for sim-to-real or, in other words, closing the reality gap.

The first solution is to address the partial observation in the real world by not training on any inaccessible states for the real world in the simulator. This is done in [82]. You could also apply state estimators to get more information on your state space. Alternatively, add more sensors on the robot, meaning we have more data to train on. Lastly, you can utilize your extra information/states in the simulator by training a privileged agent on the extra states. Then, use this privileged agent as a teacher that trains a purely vision-based sensorimotor agent. This is done in [7].

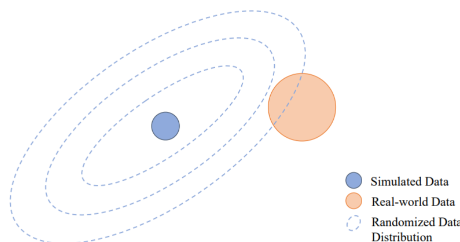


Figure 2.5.: Shows how domain randomization can help in training a model that incorporates the physical environment to its repertoire of familiar environments [99].

Another technique to close the reality gap is domain randomization.

2.4.7. Domain Randomization

Domain randomization is a tool used in DRL to train a policy to handle environmental variations. This is done by changing environmental parameters. Without domain randomization, the sim-to-real transfer would require a high-precision description of every part of the physical system. Minor inaccuracies in visual or dynamical parameters could affect the performance of the policy. The goal of the randomization is to have a broad variation in parameters so that the physical parameters are familiar to the policy before it is implemented in the physical environment, as illustrated in fig. 2.5.

Domain randomization in the simulation training can be separated into visual and dynamic randomization [99]. Visual randomization can be implemented when a visual representation of the environment is a part of the observation. Camera placement, lighting, object colors, and textures are examples of what can be randomized. Dynamic randomization can be implemented for the policy to handle differences in how the environment responds to an action. Moments of inertia, masses, and friction for both the actuators and the other objects in the environment are examples of parameters that can be changed.

Supporting multiple physics engines is another domain randomization technique that prevents learned policies from overfitting to the simulation environment used [18].

There is a tradeoff here as more environmental diversity may cause the policy to perform poorly. Often this can be alleviated with a larger and better network architecture or by giving the agent more data per update. An example is [36] where a larger and deeper NN was required for the Q-function to deal with the

large variety when domain randomization was introduced.

2.4.8. Domain Adaptation

Instead of reducing the reality gap by modifying the simulation, domain adaptation uses data from the source domain to improve the performance of a learned model on a different target domain. An example of this is [97] where they generate synthetic images of a robot arm based on real-time readings of the robot's joint angle positions similar to the training data used. Another solution is done by [4] that used a Generative Adversarial Network (GAN) during training to make the simulated images more closely resemble the real-world domain.

Real-To-Sim

One other technique to close the reality gap is to flip it around. Meaning you go from real to sim. This approach was done by [33], where they use an adaption network to convert real-world images to simulation images. This allowed a policy only trained on simulation data to be applied to the real world with a grasping success of 70 % with the QT-Opt algorithm right of the bat. It reached a 91 % success rate after fine-tuning on just 5 000 real-world grasps: which previously took over 500 000 grasps to achieve.

2.4.9. Exploration in robotic DRL

The state and action space in robotic manipulation is so big that exploring them all is not an option. Therefore, we must find a good and effective solution to explore our state-space but not visit all states. The task gets even more complicated when sparse rewards are introduced. Meaning that many steps need to be taken in the state space before getting a reward.

For this reason, several prior works have focused on studying exploration for sparse reward robotic tasks. [2] introduced Hindsight experience replay (HER), which allowed sample-efficient learning from sparse binary rewards and therefore avoided the need for complicated reward engineering. The training scenario is that they send an initial state and a target state/goal to the policy. The idea is that after experiencing some episode, every transition is stored in the replay buffer. Not only with the original goal used but also with a subset of other goals. Using an off-policy algorithm, one can replay this trajectory with an arbitrary goal.

[70] showed that methods that combine RL with prior information, such as classical controllers or demonstrations, can solve these tasks from a reasonable amount of real-world interaction.

Demonstration

Instead of improving exploration, we can try to sidestep this problem by combining simple manual engineering and demonstration data. There are different ways of incorporating demonstrations into the learning process. One of these is imitation learning, where a policy is pre-trained with demonstrations of desired behavior. It should be mentioned that there are some challenges with imitation learning. There is no guarantee of performance both in theory and practice. Imitation learning can suffer from "compounding error," where a small mistake in the pre-trained policy sends the training policy into an unexpected state where it makes a more significant mistake. There is also a big chance that the pre-trained network can be forgotten as it is common to start RL algorithms with a high exploration factor.

Off-policy model-free RL algorithms are also able to utilize demonstrations. *Data aggregation* can be used by sending demonstration data to the training data on which the algorithm is trained. [90] did exactly this where they both sent in demonstrations and actual interactions to fill a replay buffer. This was used as data for a DDPG algorithm to train on. This outperformed the regular DDPG, and it does not require engineering rewards.

Although there have been some good results with *data aggregation*, some problems need to be addressed. Value function estimation used in Q-Learning needs to see both good and bad examples to learn which actions that are desirable [35]. This means that the value function might fail to learn which actions must be taken to reach the demonstrated states.

The solution to this problem is to train the imitation policy together with the objective policy instead of pre-training the imitation policy. This allows us to add the loss from the policy gradient objective with the loss from the imitated policy. This is done in [24]. This method succeeds in making the learner stay close to the demonstrations. The problem with this joint training is that the algorithm will have difficulty finding better solutions than those given as demonstrations. So the learning speed might be accelerated, but if the best demonstrations are not given to the learner, the algorithm might never learn them.

Scripted Policies

Another method to overcome the exploration challenge is by designing a "scripted" policy. This can be seen as a reasonably good policy for solving the task at hand. Scripted policies were used to pre-populate the replay buffer with a higher proportion of successful episodes. QT-Opt [36] used a scripted policy to collect 200 000 grasp attempts. This scripted policy had a success rate of 15 - 30 % meaning

that the Q function had some successful data to learn from instead of having to explore by itself and use a very long time before it even got some valuable data. We do not use an excellent policy as our "scripted" policy because we want to keep the ratio of successful and unsuccessful episodes close to 50 %. This is because a Q-function requires both good and bad attempts to learn a good ranking of what a good or bad action is. To keep this ratio close to 50 % [27] states that you should wait to use data from your Q function in the replay buffer until it reaches a 20 % success rate.

2.4.10. Reward shaping

Reward shaping leads to faster learning by sidestepping the exploration challenge. In practice, reward shaping uses prior knowledge to give intermediate rewards for actions that lead to desired outcome [19]. This additional guidance during the exploration can be beneficial in settings with sparse and delayed rewards and can help the learning process [44]. As mentioned in [27] reward shaping is very effective for any task where the agent has to go to a specific location. One example of this is peg insertion as shown in [45]. Since our task requires the robot gripper to reach the position of the cube, we decide to use reward shaping to help guide the exploration.

One challenge about reward shaping is to weigh the shaping terms properly to avoid any greedy and unintentional sub-optimal behavior. An example given by [27] describes the situation where you want to open a door with a robot arm. One might think it is good to give a negative reward to the robot if it is far away from the door. This could be a good idea since one may want to get close to the handle. However, if the robot is already very close to the door but cannot open it, it may require some distance from the handle to take a different approach with the gripper. The engineered reward would restrict the robot from finding such a solution.

The object's location must be known to give a reward based on the object's distance. This is often available in simulators but not in real-world scenarios where only camera information is available. Meaning the object's location is not a part of the observations.

2.5. Proximal Policy Optimization Algorithms

This section introduces the theory behind Proximal Policy Optimization, which is the algorithm used in this thesis.

Proximal Policy Optimization (PPO) [72] introduces a new branch of policy gra-

Algorithm 1 PPO, Actor-Critic Style

for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

Figure 2.6.: Proximal Policy Optimization (PPO) algorithm as described in [72].

dient methods. These methods alternate between sampling data through interaction with the environment and optimizing a "surrogate" objective function with a stochastic gradient ascent. Instead of only doing one gradient update per data sample, which is what the standard policy gradient methods do PPO propose an objective function that enables multiple epochs of mini-batch updates. So this method has some of the benefits of Trust Region Policy Optimization (TRPO), but is also much simpler to implement, more general, and has better sample complexity. The algorithm is shown in fig. 2.6.

PPO is designed to fix the challenges introduced in section 2.4.4, and its core purpose is to strike a balance between ease of implementation, sample efficiency, and ease of tuning.

PPO is an on-policy algorithm, meaning it does not learn from stored data in an experience replay buffer. Instead, it learns directly from what its agent encounters in the environment. Once a batch of experiences has been used to do a gradient update, the experience is discarded. Consequently, it is less sample efficient than Q-learning methods because they only use the collected experience once when performing an update.

2.5.1. Policy Optimization

General policy optimization methods usually start by defining the policy gradient laws as the expectation over the log of policy actions times an estimate of the advantage function as shown in eq. (2.11)

$$L^{PG}(\theta) = \hat{E}_t[log\pi_\theta(a_t|s_t)\hat{A}_t] \tag{2.11}$$

Where π_θ is our policy. In other words, a neural network takes the observed states from the environment as input and suggests actions to take as an output. The

second term is the advantage function $\hat{A}_t$ which tries to estimate the relative value of the selected action in the current state.

In order to compute the advantage function $\hat{A}_t$ eq. (2.12) we need two terms. The first is the discounted sum of rewards G_t shown in eq. (2.13) minus the second term which is the baseline estimate or in other words the value function $V(s)$.

$$\hat{A}_t = G_t - V(s) \tag{2.12}$$

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.13}$$

The discount factor γ in eq. (2.13), which is usually in the range of 0.9 and 0.99, is a variable that quantifies the importance of future rewards. If Gamma is closer to zero, the agent will tend to consider more immediate rewards.

The advantage estimate is calculated after the episode sequence is collected from the environment as shown in fig. 2.6. This means we know all the rewards when calculating this value and avoid guessing when calculating the discounted return.

The value function is trying to guess what the final return will be for this episode starting from the current state. The neural network representing the value function will be frequently updated using the experience the agent collects in the environment, close to how supervised learning works. It is worth mentioning that these estimates are noisy because our value function is a neural network.

To sum up, the advantage estimate answers how much better the taken action was based on the expectation of what would typically happen in the state it was in. This tells if the action taken was better or worse than expected.

2.5.2. Trust Region Policy Optimization

When running gradient descent on one batch of the collected experience, one challenge is that you will update the parameters in the network outside of the range where this data was collected. This will lead to an already noisy advantage function which is an estimate of the real advantage to be completely wrong. The policy will be ruined if you keep running gradient descent on a single batch of collected experiences.

To solve this issue Trust Region Policy Optimization (TRPO) [71] are introduced. This method ensures that the updated policy never moves too far away from the old policy. To do this they divide the old policies $\pi_{\theta_{old}}$ estimate on the new as

shown in eq. (2.14). An Kullback–Leibler Constraint (KL) is also added to the optimization objective eq. (2.15). This constraint limits the updated policy from moving too far away from the old.

$$\underset{\theta}{\text{maximize}} \; \hat{E}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t \right] \tag{2.14}$$

$$\text{subject to} \;\; \hat{E}_t \left[KL \left[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t) \right] \right] \tag{2.15}$$

The downside is that the KL constraint adds additional complexity to the optimization process. This might sometimes lead to undesirable training behavior. To counteract this PPO adds this constraint directly into the optimization objective.

2.5.3. Clipped Surrogate Objecttive

Here we introduce the optimization objective of the PPO algorithm. Their loss function is:

$$L^{CLIP}(\theta) \;=\; \hat{E}_t \left[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right] \tag{2.16}$$

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{2.17}$$

The first thing to mention is that PPO optimizes an expectation operator. This means that it is computed over batches of trajectories, and an expectation operator is taken over the minimum of two terms as shown in eq. (2.16). The first term is equal to the objective in the TRPO objective function eq. (2.14) where the variable $r_t(\theta)$ the fraction of the new over old policy which is shown in eq. (2.17). Epsilon is a hyperparameter usually with a value of $\epsilon = 0.2$. The second term, modifies the surrogate objective by clipping the probability ratio, which removes the incentive for moving r_t outside of the interval $[1-\epsilon, 1+\epsilon]$. The minimum of these two terms are taken, which [72] describes will make the final objective *"a lower bound (i.e., a pessimistic bound) on the unclipped objective"*.

The advantage estimate A_t can both be positive and negative and changes the main operator's effect as shown in fig. 2.7. Positive estimates are all the cases where the selected action had a better-than-expected effect on the outcome. Negative values are all the cases where the action had an estimated negative effect on the outcome.

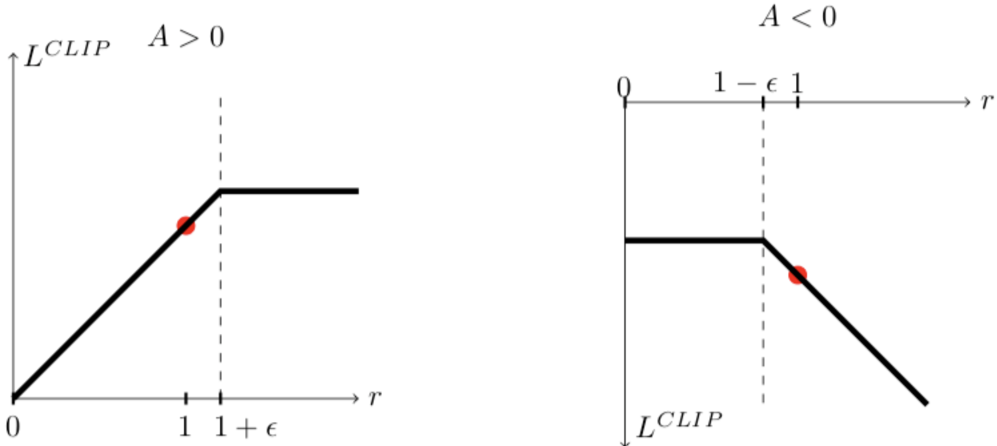


Figure 2.7.: Plots showing one term (i.e., a single timestep) of the surrogate function L^{CLIP} as a function of the probability ratio r, for positive advantages (left) and negative advantages (right). The red circle on each plot shows the starting point for the optimization, i.e., r = 1. Note that L^{CLIP} sums many of these terms

Still looking at the left graph, if the action was good and $t_t(\theta)$ yields a higher value, then one notices that the surrogate function flattens out. This is to limit the effect of the gradient update. We do this because the advantage function is noisy, so we do not want to update too far in either direction.

The same goes for the graph on the right side of fig. 2.7. This is where the action had an estimated negative value. The objective function here flattens when r goes closer to zero. This part of the graph corresponds to actions that are much less likely now than in the old policy and we don't want to overdo an update because of our lack of trust in the noisy advantage function. Lastly, the right side of the right graph in fig. 2.7 corresponds to when the last gradient step made the selected action more probable while also making the policy worse since the advantage function here is negative. This is an update we would like to undo, which is what the PPO algorithm allows because the function is negative here, which makes the gradient update in the other direction.

2.6. ROS2

2.6.1. General

Robot Operating System 2 (ROS2) is a collection of tools for making robot applications. In this project, communication and driver tools are being used to make

the communication architecture of the physical system.

2.6.2. Nodes,topics, services and messages

Nodes are one of the core concepts of ROS2. It is the building block for the communication between the different parts of a robot system. The simplest type of communication between ROS2 nodes is through topics. A publisher node pushes a message, .msg, to a topic, and a subscriber node pulls the messages from the topic. There can be several publishers and subscribers connected to the same topic. What is essential is that the .msg files that are sent over a topic have the same format. This architecture makes it easy to set up communications with new nodes to extend the network.

In addition to topics, ROS2 uses services as a way of communication between nodes. First, a client node sends a request to a sever node. The server node send its response back to the client. In contrast to the topics, the request and response often has different formats.

ROS2 uses packages to structure the code for different nodes and programs. The packages should work in other systems if appropriately structured, simplifying the developers' sharing of code. A package can contain launch files and .msg structure in addition to all the system nodes.

2.6.3. DDS communication

DDS, Data Distributing Service, is the middleware that ROS2 is built upon. The wire protocol for the DDS communication is RTPS, Real-Time Publish-Subscribe. With a distributed discovery, the DDS communication is more rigid than ROS communication because of the lack of a central point of failure [67].

Part II.

System design

Chapter 3.

Simulation

3.1. Simulators

This section gives an introduction to important factors for a good physic simulator and introduces different physic simulators relevant for robot grasping. Moreover, it argues why the selected physic simulator was chosen.

3.1.1. Simulation Environment

As presented in section 2.4.5, simulators are often used for RL training in order to significantly increase the rate at which data can be collected, as well as do it safely. In order to implement a virtual setup for the training of robotic grasping based on the task described in section 1.2, a simulation had to be capable of accurately modeling the physical interactions between a robot and the manipulated objects as well as featuring a high-quality visual sensor in the form of RGB images. A simulated model of the hardware in the lab was also needed. Therefore, selecting a robotics simulator was of great importance because it directly influenced the robustness of sim-to-real transfer and determined the additional steps that had to be taken to achieve such transfer. Some of the popular simulators for robotics RL research are therefore described with the aim to select one that will be used to implement the environment.

MuJoCo

MuJoCo [88] (Multi-Joint dynamics with Contact) is a simulator very often used within research and commonly known for RL applications. One of the reasons it is so popular comes from its contact stability [65]. MuJoco has been used to train policies both for proof of concept [60] and for later transferring into the real

world [86] [8] [69]. Its physics engine is focused on robotic and biomechanic simulation, animation, and machine learning applications. MuJoCo models are based on an XML format that is readable and editable. MuJoCo recently announced that they, in cooperation with DeepMind, have made the physics engine an open-source. Another advantage is the different simulation frameworks using MuJoCo specifically built for robot learning. Some of these are Dm_control [83], Robosuite [101] and Robomimic [49][1]. They provide tutorials and examples, making it intuitive to set up an environment.

Pybullet

Pybullet is used in studies from object collision [47] pick and grasp dynamics [96] and for deformable object manipulation [50]. The latter case was done with manipulation in the simulator. Pybullet is based on the Bullet physics-based simulation environment. Pybullets main focus is machine learning applications in combination with robotic applications. This, combined with a large community [1] means that it is a simulation environment in continuous development with good support for beginners. It supports model formats such as SDF, URDF, and MJFC, giving a good foundation for customizing an environment.

Gazebo

Gazebo [39] is one of the oldest open-source simulators and provides a simulation environment with the necessary actuators and sensors for robotic manipulation. It supports four physic engines: Bullet [11], Dynamic Animation and Robotics Toolkit (DART) [13], Open Dynamics Engine (ODE) [54] and Simbody [74]. Gazebo can switch between these physic engines, meaning it has one of the best physics-based domain randomization, as it would allow randomizing physics parameters and the entire physics implementation. It also provides support for ROS which provides packages for forward and inverse kinematics, as well as path and motion planning. Since the foundation of the Open Source Robotics Foundation (OSRF) [58] in 2012, OSRF has been leading the development and is supported by a large community [20].

Gazebo provides support for noise models, which can be applied to sensor outputs. This can be an essential feature in solving the reality gap.

[1] Robomimic also supports Pybullet

Ignition Gazebo

Due to the limitations in Gazebo there is a development of the next generation of Gazebo named Ignition Gazebo. Ignition supports the DART physics engine and has upcoming support for Bullet. Ignition support the latest updates of OGRE [55] when it comes to rendering. This enables PBR (physical-based rendering), which can be of good use to tackle the reality gap. Ignition Gazebo is in a relatively early stage, meaning there is a limited amount of RL research conducted with it, although some has been done [57]. It is worth mentioning that Gym-Ignition [17] is introduced as a framework that simplifies its usage for RL research.

Nvidia Isaac

Nvidia is developing a promising robotics simulator called Isaac Sim [31]. This simulator utilizes PhysX [61] physics engine and has support for SOTA PBR rendering. Isaac sim also comes with a software framework designed for RL called Isaac Gym [30]. One of the significant advantages of Isaac Sim is that physics computations, rendering, as well as the process of determining rewards can be offloaded to GPU in order to enable running a large number of environments in parallel. As of May 2022, Isaac Gym is only available as early access, and its functionalities are limited. They provide a ROS API, which makes it possible to set up code similar to real physical code.

CoppeliaSim

CoppeliaSim is a robotics simulator with a range of user-centric features, including sensor and actuator models, as well as motion planning and forward and inverse kinematics support. CoppeliaSim recently introduced a python toolkit for robot learning called PyRep. This has been used for pick and place [32].

Webots

Webots is an open-source and multi-platform desktop application used to simulate robots. Webots comes with a complete development environment where you can model, program, and simulate robots [92]. It has an extensive library of sensors and robot models, which can be used when modeling the environment. It supports C, C++, Python, Java, MATLAB, or ROS, for programming robots with detailed API documentation for each option. Webots uses a customized ODE physics engine at its core. Webots has a node called PBRApperence [93] that can be used for physical-based visual appearance of an object. This might help when it comes to domain randomization.

3.1.2. Simulator used

Of the considered robotic simulators, Mujoco was selected for the following reasons:

Mujoco was chosen over Gazebo, Ignition Gazebo, Nvidia Isaac, and Webots because of the strong documentation on RL and the software framework Robosuite [101] that support models of the robot, gripper, and environment present at the lab. This makes it more reliable to solve unforeseen challenges and problems. There was some downsides in making this choice. Webots was the physics simulator that scored best on speed and accuracy in [40] for robot grasping tasks. Mujoco was not too far of, but the fastest simulator was not chosen. Another challenge is the lack of ROS in Mujoco. Because the code on the physical setup was based on ROS, we lost some similarity in code structure between simulator and physical setup by choosing Mujoco over the other four mentioned simulators, who all support ROS.

Pybullet was a close contender, but after looking at papers that compare different physics simulators when tested on robotic grasping and different accuracy and speed tests, Mujoco scored better than Pybullet on categories important for this task. The categories that were looked at were the performance of the gripper on small and precise movements [40], and accuracy and speed when it comes to robotic applications [15]. It is worth mentioning that Pybullet scored very well on handling multiple physics collisions simultaneously by looking at a scenario of falling spheres. Because our environment only consists of one cube, this was not given too much attention but could be important if multiple objects were to be interacted with simultaneously.

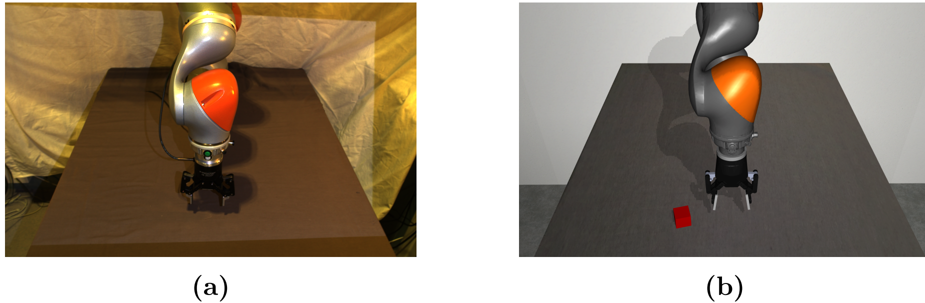
Coppeliasim fell short because of the lack of studies comparing its performance to other simulators. This made it safer to choose a simulator with already promising results.

3.2. Simulation Setup

This section describes the simulation framework used and how the environment is built in the code. I also presents the choices made along the way.

3.2.1. Robosuite

Robosuite is an open-source framework for robot simulations with Mujoco as its physics engine. It is a modular framework with different robots, grippers, objects, controllers, renderers, and tools to adjust the robotic environment. It is easily



(a) (b)

Figure 3.1.: Shows the visual differences between the simulated and physical environment.

implemented and can be used with the Stable-Baselines3 implementations of the PPO algorithm as mentioned later in section 4.1.2.

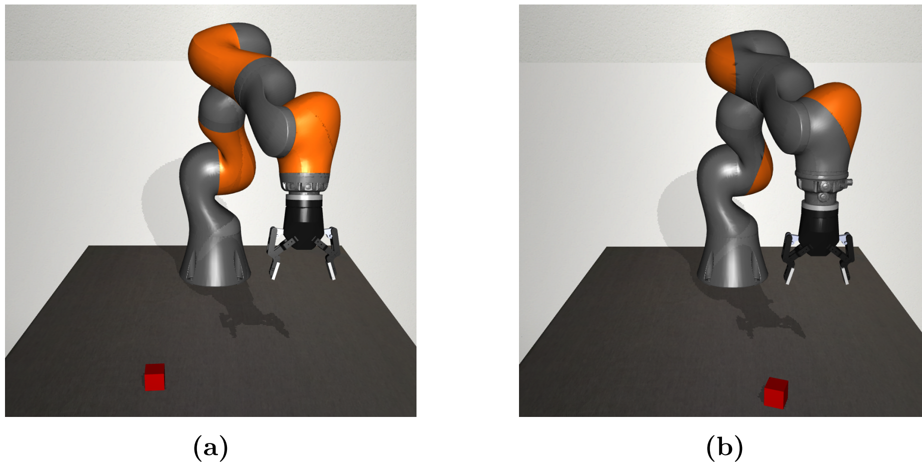
Models for the KUKA iiwa and the Robotic 2F-85 gripper are conveniently available, and tools to mimic the MANULAB environment are in place. The camera perspective, the field of view, and the resolution of Zivid can be imitated. It has operational space, joint space, and joint velocity controllers to be used on the simulated robot. With the different controllers, it could be possible to change the controller if a solution seemed more promising throughout the project.

Robosuite was chosen as the framework because of its adaptability, intelligibility, the presence of relevant hardware models, and its Mujoco physics engine.

3.2.2. The basearena

At the base of the environment lies an XML file called lab_arena.xml. This file placed everything in the environment except the robot, gripper, and the objects. Here, the table was given the exact size and placement as the table in the lab. Its surface texture was based on an image taken of the table in MANULAB. The lighting in the room was also set here. A white texture for the walls was chosen to mimic the white bed sheet in the physical setup.

There were used two light sources in the environment. One was placed at [x y z] = [3 1 4] to mimic the lighting from the lab's roof and, consequently, the shadows on the table. The other was placed at [x y z] = [1.6 0 1.45]. This was to mimic the reflections from the projector on the Zivid camera. The projector of the zivid camera is described in section 5.3. A comparison of the lighting and shadows can be seen in fig. 3.1.



(a) (b)

Figure 3.2.: Shows the improved KUKA iiwa model. The link meshes are split to mimic the design of the physical KUKA iiwa.

3.2.3. KUKA iiwa 14 r820 in simulation

The KUKA iiwa 14 r820 was not implemented in Robosuite, but the KUKA iiwa 7 r800 was available. The robot was represented with visual and collision .stl files, where the visual models were of higher resolution than the collision models. The .stl files were referred to in an XML file with the appropriate parameters to represent the robot accurately.

The XML file from the iiwa 7 was used as the foundation for implementing the iiwa 14. Meshes, for both visual and collision models, were collected from the Unified Robot Description Format (URDF) provided in the ros-industrial Github repository [68]. The collision .stl files were directly transferred to the mesh directory referenced by the XML. As visual meshes .dae files from ros-industrial were converted to .stl before being placed in Robosuite. We had to remove double faces in the meshes to prevent shadow artifacts in the vicinity of the joints. These artifacts were visible because of shadow aliasing [95]. The URDF file was used to find the appropriate parameters for the XML file.

We were not satisfied with the KUKA iiwa 7 representation in Robosuite. All joints containing orange details were fully colored 3.2a, which is not satisfactory when we rely on an accurate visual representation to limit the sim-to-real gap. By splitting the relevant meshes and saving them in separate files, we could refer to the part independently in the XML file. The color could then be set for the parts separately, which ended in the model shown in fig. 3.2b

In the XML file, the joint limits were set to the same values as the physical robot, as shown in table 5.1.

3.2.4. OSC controller in simulation

A Operational Space Controller (OSC) was used to control the robot. The controller has a 6D input dimension consisting of the position and orientation of an end-effector site placed between the finger of the gripper. This site can be seen in fig. 3.9.

The controller in our environment follows the formalism from [38]. The OSC framework "computes the necessary joint torques to minimize the error between the desired and the current pose of the end effector site with the minimal kinematic energy"[10]. The controller is further explained in [10].

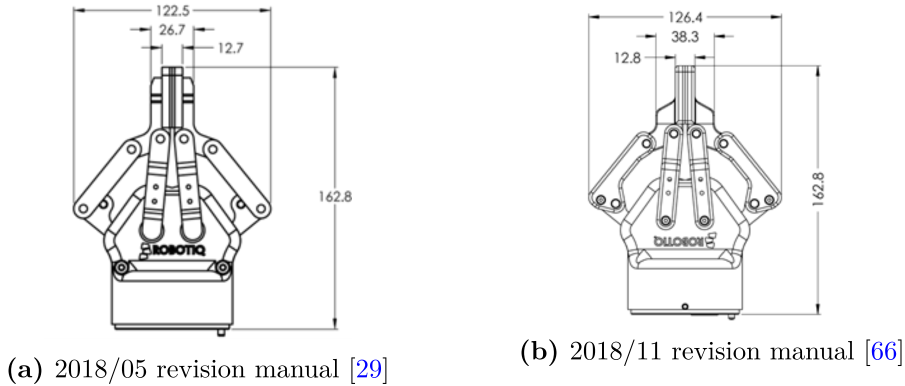
The default control settings from Robosuite were used in this work, except a position limit in the z-direction and the control frequency. The exceptions are explained in chapter 6 and section 3.6.2 respectively. With these settings, actions from the agent were clipped in a ± 0.05 m range for translations and a ± 0.5 radians range for rotation, as mentioned in section 3.6.1. The torques were then calculated by the OSC and sent to the robot. From testing, we saw that the robot moved with a maximum of 0.0003 m and 0.0045 radians for every time-step. With our control frequency defining that an agent-step is 50 time-steps, every agent step's movement was a maximum of 15 m and 12 degrees.

3.2.5. Robotiq 2f-85 in simulation

The Robotiq 2f-85 was already available in the Robosuite framework. The representation of the gripper in Robosuite is analogous to the KUKA iiwa representation with .stl files and an XML file. It was necessary to add the in-house made aluminum adapter plate, and the coupling [66] that enables communication and power supply for the gripper. The .stl files for the two parts were added to the gripper folder in Robosuite. Furthermore, the parts' parameters for placement and color were implemented in the XML for the gripper.

The end effector site used as the end effector position by the robot controller is shown in fig. 3.9. The site was defined in the Gripper class in Robosuite. This class read the gripper XML file and served as the core modeling component in the simulation. I addition to reading the XML file and defining the end-effector position, this class defined the gripper action, speed, initial position, and more.

It was found that the gripper used in the simulation was an old version of the 2F-85 gripper. There were minor variations in the dimensions, but they had the same under-actuated structure and had almost an identical way of interacting with the objects. The main difference was the fingertips, as shown in fig. 3.3. The fingertips are thinner in the simulated version, as seen in fig. 3.3a.



(a) 2018/05 revision manual [29]

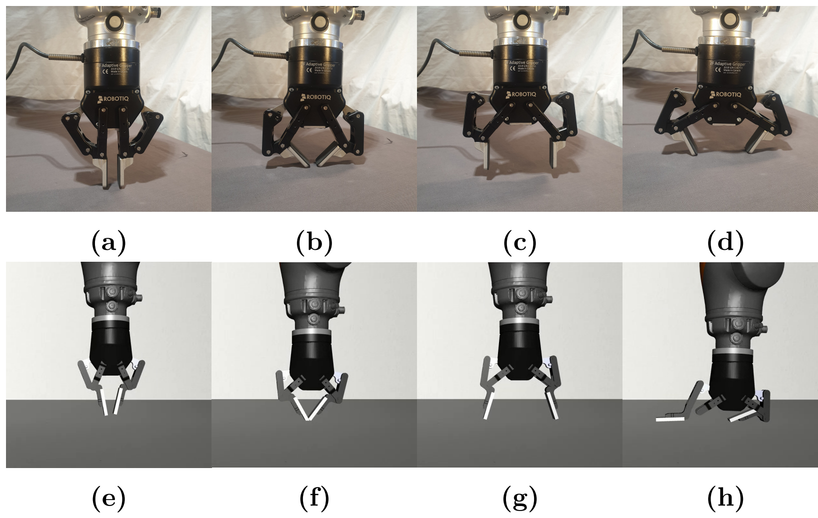
(b) 2018/11 revision manual [66]

Figure 3.3.: Shows the revisions of the Robotiq 2F-85 gripper.

Through measuring, we found a 10 mm difference in height for the simulated gripper model relative to the drawings in fig. 3.3, and effectively our physical gripper. We think this difference came from the loose inner joints in the Robosuite illustrated in fig. 3.4h. It can be seen how the inner knuckle aligned the fingers and consequently stretched them downwards by comparing fig. 3.4a and fig. 3.4e. Both the 2f-85 and the 2F-140 Robotiq gripper in Robosuite had loose joints between the inner knuckle and the inner finger. To make up for this difference, we made the fingertips 10 mm longer.

Figure 3.4 shows how the interaction with the table was almost identical for the simulated and physical gripper when the gripper was closed. There were only small angular and translation differences. In contrast, the open gripper in the simulation collapsed when interacting with the table. This was because of its loose joint between the inner knuckle and the inner finger.

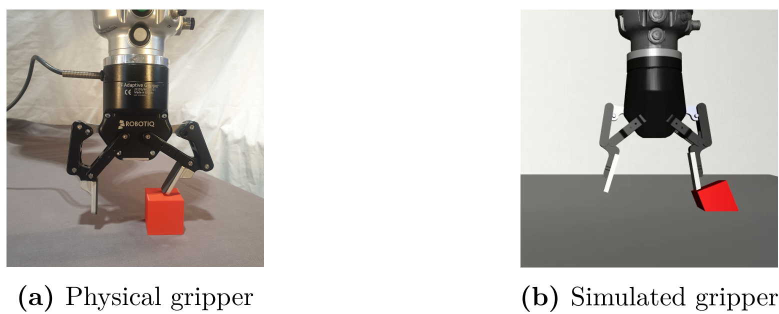
There were other dynamical differences between the physical and simulated gripper. Figure fig. 3.5 shows that the interaction between the gripper and an object was non-identical. This applied to the gripper both in the closed and open configuration. When the physical gripper was in contact with the box it "gave in" because of its under-actuated structure. The gripper did not push down on the cube with much force. With horizontal gripper translation, the cube only moved to a small degree because of the low friction coefficient between the gripper and the cube. These described dynamics were mismatched in the simulation because the under-actuated abilities were not implemented. If the gripper was in contact with the cube, it pushed the cube without movement in the gripper joints. In some cases, it could push the box into the table as fig. 3.5 shows, and even push the cube through the table.



(a) **(b)** **(c)** **(d)**

(e) **(f)** **(g)** **(h)**

Figure 3.4.: Gripper-table interaction in physical and simulated environment.



(a) Physical gripper **(b)** Simulated gripper

Figure 3.5.: Difference in gripper-object interaction.

3.2.6. The Object

The cube's density, color, and friction were set to mimic the cube in the lab as close as possible. After weighing the cube, the density was set to $375\frac{kg}{m^3}$. The sliding, torsional and rolling friction is shown in table 3.1 is empirically tested by comparing the behavior of the objects when pushing the cube from different points in the physical and simulated environment. The sliding friction was set by comparing pushes on a low point on the cube. The rotational and torsional friction was set by looking at how the cube tipped over and rolled in the physical world when being pushed at a high point and then imitating this in simulation.

Table 3.1.: Used friction forces

Type of Friction	Friction value
Sliding	0.01
Torsional	0.005
Rotational	0.0001

3.3. Observation Space

The observation space used in this thesis are RGB images

3.3.1. Image observation

The image observations were chosen to be RGB with a resolution of 84x84. This is the exact resolution they used in Fan et al. [16], and we found this to be an adequate size. Both the physical and simulated images are shown in fig. 3.6.

The placement of the Zivid camera in the simulation was based on the physical placement. During testing, both an approximate placement and calibrated placement were used. These placements were described with a transformation matrix as explained in section 5.3.3. Because the camera in the lab and simulator use different coordinates to represent the rotation matrix, a conversion between the two coordinate frames was used. This matrix is shown in fig. 3.7 is used to go from lab to simulator.

An 84x84 image size was set and, consequently, set the pixel size. The image's width is found based on the pixel size and given horizontal resolution, in our case, 84. The middle of the image is defined by the transformation matrix describing the camera frame position.

(a) (b)

Figure 3.6.: Shows the visual differences between the simulated and physical image observation.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

Figure 3.7.: Matrix to convert rotation matrix from lab to simulation

3.3.2. Observations in Robosuite

There are proprioceptive observations available to be used in the Robosuite framework. Proprioceptive observations are all the observations that not can be considered as an image. A set of these observations are shown in table 3.2. The joint position and end effector observations was used throughout the testing process. During training, the cube position observation was used in the calculations of the policy reward.

In addition to the observations, the authors implemented an observation. The g_obs is a continuous variable that sends the gripper's current position where -1 is entirely open, and 1 is completely closed or in contact with an object. The gripper observation is a continuous variable because we did not manage to implement discrete variables in Stable Baselines.

Table 3.2.: List of Robosuite observations. The *gripper_status* observation was implemented by the authors.

Observation name	Variables
image	(84x84)
robot_eef_pos	(x,y,z)
gripper_status	(g_obs)

3.4. Action Space

The action space for the end-to-end robotic grasping consists of continuous actions in Cartesian space and a gripper command.

OSC was used to control the robot with an action dimension of 6 as mentioned in section 3.2.3. The reason for this was better learning [6] and the ability to limit the action space. Furrer et al. [6] argue that an action space in the operational space makes for better learning than joint space because of the similar relative poses between objects and gripper. They further mention that velocity and torque control demands higher accuracy for the physical models compared to position controller. In addition, it is simple to reduce the action space to three or four variables and still have a model that can pick an object when using operational action space. Restrictions can also be set to where the robot can move to avoid crashing in its environment.

In our environment, we sat a restriction in height for the end-effector. This was to prevent collisions between the gripper and the table in the lab. The restriction also prevented contact between the gripper and table in simulation, as illustrated in fig. 3.4h. By testing, we found that 0.819 was a preferable height. Both the

simulated and physical robot had a small gap of about 2 mm to the table with this restriction.

Table 3.3.: List of Robosuite actions.

Action name	Variables	Comment
robot_eef_pos	(x,y,z,c)	Delta values
gripper_pos	(g_pos)	Continous

The full potential action space was of dimension 7 $[x, y, z, a, b, c, g]$ where the first 6 variables describe the end-effector position of the robot. These commands describe the difference between the current pose and the desired pose. The first three variables are described in Cartesian coordinates. $[a, b, c]$ are calculated with rotation encoding and relates to rotation about the x,y, and z axis's. Lastly, g relates to the gripper command.

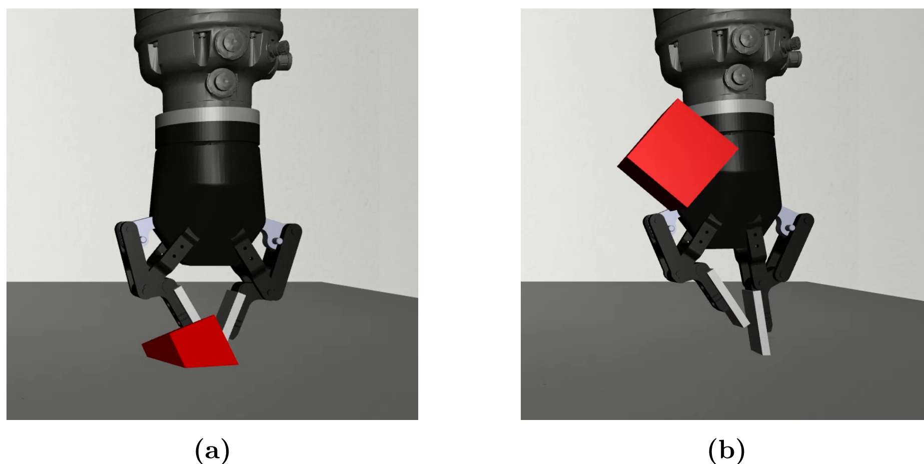
Our action space is continuous, and each variable's upper and lower value is set from $[-1, 1]$ and of type float32. This is the same for the gripper action, where negative values open the gripper and positive values close the gripper. By default, the environment clips the gripper action according to eq. (3.1) in Robosuite.

$$a_c = \begin{cases} a_c + s * sign(a) & \text{if -1} < a_c < 1 \\ -1 & \text{if } a_c <= \text{-1} \\ 1 & \text{if } a_c >= 1 \end{cases} \tag{3.1}$$

where a_c is the current action, s is the speed telling how big of steps to take each update, and a is the new action given. $sign()$ checks if a is a positive or negative value.

The speed used in this work was set to $s = 0.01$. Different values were tested, but this was chosen because it allowed the gripper to have smooth and controlled motion. Setting the speed too high led to the gripper being too aggressive and often hitting the cube out of the frame. The control frequency sat in the environment also affects the movement of the gripper, and this parameter is discussed in detail in section 3.6.2.

Limiting the action space gives the agent fewer parameters to optimize. One way to limit the number of variables is to neglect a and b, which are horizontal rotations from our action space. This is done in other state-of-the-art papers like [36] and [6]. This would limit our action space to 5 dimensions [x,y,z,c,g]. We also had one test where the action space was set to [x,y,z,g] to reduce our action space further. An argument for why this reduction is appropriate is that most objects can be picked without the two last degrees of freedom.



(a) (b)

Figure 3.8.: Shows how the gripper can make the gripper bounce over the reward limit.

3.5. Reward Function

Reward shaping was used to speed up the learning and guide the exploration, as mentioned in section 2.4.10. A reward function that combined the task's three phases was made. The agent must first *approach* the object, then *grasp*, and finally $lift$ it to perform a successful lift.

Each episode lasted for 200 timesteps. Each agent-step received a maximum reward of 1, meaning the maximum possible reward for one episode was 200.

The agent got a reward $r = r_1 + r_2 + r_3$ for each step. $r_1 \in [0, 0.444]$ was 0.444 when the gripper's grip site shown in fig. 3.9 was at the cubes position. It decreased as the grip site was further away. $r_2 = 0.111$ if both insides on the fingers of the gripper were touching the cube and $r_2 = 0$ otherwise. $r_3 = 0.444$ if the cube's center was above 4 cm and the inside of the fingers of the gripper were in contact with the cube such the robot was lifting the cube. It was zero otherwise.

The KUKA iiwa robot was initially rewarded a full score if it could get the cube 4 cm over the tabletop. This reward threshold was inspired from [16]. One downside of this was that the robot occasionally could hit the cube and make it reach this height, as shown in fig. 3.8. To prevent this, we added the criteria that both insides of the gripper fingers had to be in contact with the cube while it was lifted 4 cm over the table.

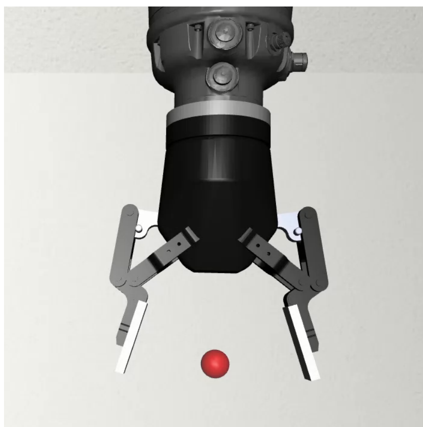


Figure 3.9.: End effector position of the system

3.6. Parameters

An essential part of the Robosuite environment is the parameters that define how the simulator behaves during training. In this section, two parameters related to the gripper and robot movement are presented.

3.6.1. Output parameter

The *output_min* and *output_max* parameters relate to the clipping of actions of the robot. They define how far the robot moves between every time-step when an action is given. A *time-step* is defined as the frequency at which the simulation is updated. These parameters are presented in table 3.4 with their respective values.

Table 3.4.: List of the clipping parameters for the actions given to the Robosuite environment.

Parameter name	Values
output_min	(-0.05, -0.05, -0.05, -0.5, -0.5, -0.5)
output_max	(0.05, 0.05, 0.05, 0.5, 0.5, 0.5)

With the values presented in table 3.4 a positional action from the agent is clipped to a value between -0.05 and 0.05. The angular agent action is clipped to a value between -0.5 and 0.5. With a control frequency at 10 Hz, these values give an approximately maximum movement of 15 mm and 12 degrees. Raw data from these tests can be found in the digital appendix appendix C.3.3.

3.6.2. Control frequency

The control frequency defines how many time-steps there are in every agent step. An *agent-step* is how often observations and actions are given to the agent in the simulated environment. By default, the control frequency is set to 20 Hz by Robosuite. Each time-step in the Robosuite simulation is 0.002 seconds, giving 25 time-steps for each agent step. This parameter can be changed but will interfere with several parts of the system.

Robot and gripper movement is affected by the control frequency. Both set an end goal and move towards their goal at a specific speed. If the control frequency is high, the gripper and robot may not reach their goal position in time for a new observation and action. A new action from the policy may contradict the previous action for the robot and gripper and move them in another direction.

With a control frequency of 10 the eq. (3.1) runs 50 times for each action given. With the given speed, the gripper takes 4 action steps to reach its goal from open to closed. It frequency limits the robot movement to the values described in section 3.6.1.

After testing with different control frequency values, we found that the training speed decreases substantially with a low frequency. This is because the low frequency demands more motion steps before a new observation and action is taken. The number of action steps for each episode is set to 200. With a control frequency of 1 every episode has $200 * 500 = 100000$ time-steps, while a control frequency of 20 has $200 * 25 = 5000$ time-steps. This resulted in training with updates taking 110 minutes and 43 minutes, respectively.

In this work, a control frequency of 10 Hz was used. High-performing agents have already used this frequency [16]. It was low enough to limit the gripper movement to 4 steps between each agent-step while at the same time keeping the robot movement distance low between each agent-step. It also gave an acceptable training time.

3.7. Domain Randomization

Domain randomization was implemented as a tool to handle the sim-to-real transfer. Why this is the case is discussed in section 2.4.7. Domain randomization was applied for several properties at each environment reset, i.e. before the beginning of every episode.

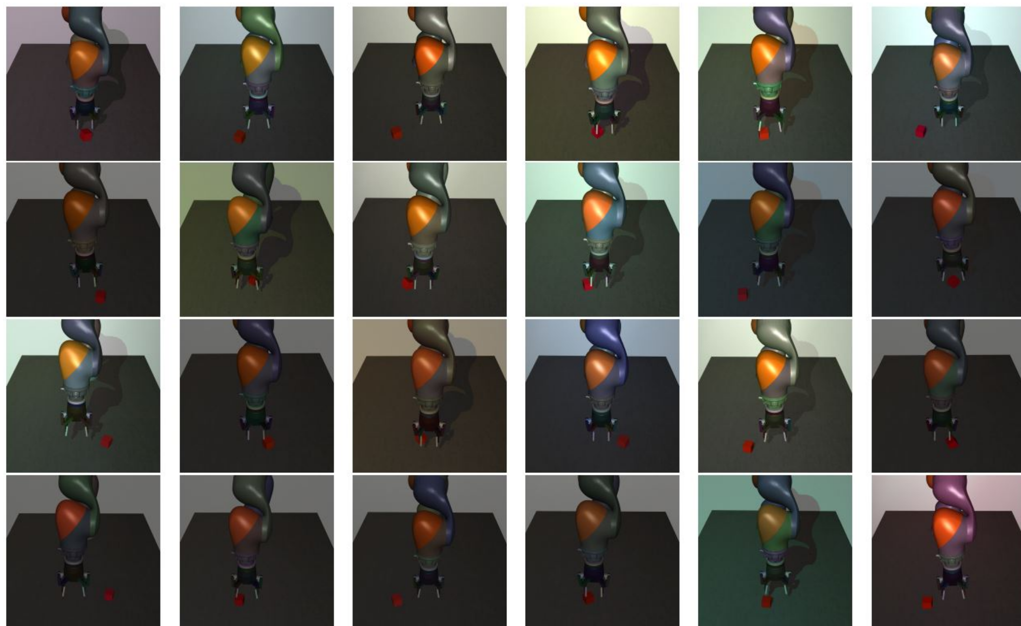


Figure 3.10.: Examples of domain randomization applied to the implemented simulation environment for robotic grasping

3.7.1. Visual Randomization

Visual randomization is an important part of the domain randomization. It is important because of the visual differences in the physical and simulated environment. Camera placement, lighting, and color were the main parameters that were randomized. Different examples of domain randomization used in this work are shown in fig. 3.10

Randomize camera configurations

Because of differences between the camera placement in the simulator and the physical world, the camera position, orientation, and FOVY were randomized upon initializing each environment. The position was uniformly randomized with a range of 1 cm in the positive and negative direction for the [x,y,z] position of the camera. The FOVY also used a random uniform distribution with the range of 1 degree in positive and negative value from the initial FOVY which was 36 degrees. For rotation, a random axis with a uniformly random angle was chosen. The random angle had a lower limit of 0 and an upper limit of 0.01 radians to rotate around this axis.

Random color and texture

The color of the robot and gripper were randomized in each episode. The color was decided by using the original colors for the robot and gripper as the base and then using an RGB interpolation of size 0.2 to change the color from here. The formula for the randomized color $\vec{rgb}_n$ is shown in eq. (3.2)

$$\vec{rgb}_n = (1 - i_n) * \vec{rgb} + U(0, 1)_3 * i_n \tag{3.2}$$

where $i_n = 0.2$ is the interpolation value, $\vec{rgb}$ is the default color of the object and $U(0, 1)_3$ is a vector of size 3 with uniformly random numbers in the range from 0 to 1.

The reflectance, shininess and specular describing the texture of the walls, table, and floor were randomized in each episode. The formula for deciding texture was the same used for deciding color eq. (3.2). The difference was the vector $\vec{rss}$ describing the base reflectance, shininess, and specular value of the texture, and the interpolation value used was $i_n = 0.3$

Random lighting

The position and direction of the light source were randomized as well as the specular, ambient and diffuse attributes.

The position of the light source was uniformly randomized with a length of 1 cm in all directions from its original position. The direction of the light was changed by rotating a uniformly random angle in the range of 0 to 0.2 radians around a random axis.

A uniformly random delta value in the range of 0 to 0.1 was added or subtracted from the default size of the light's specular, ambient, and diffuse attributes.

3.7.2. Dynamic Randomization

Random Pose and orientation of object

The pose and orientation of the cube were set uniformly randomly at the start of each episode. The pose was placed within a space of width 30 cm and depth of 10 cm, with the space's center located at the center of the table. Moreover, the object was uniformly rotated around the z axis.

Size on cube

The cube size was randomly sampled between $38mm^2$ and $42mm^2$ at the start of each episode. This forced the agent to adapt to slight deviations in the cube.

Random initial Joint Configuration

At the beginning of each episode, Gaussian noise, with mean 0 and variance multiplied by a magnitude of 0.02, was added to each joint to give slight deviations to the robot starting configuration.

3.7.3. Further randomization

Further dynamic randomization was not implemented. This was because of the architecture we chose for the system. The policy did not interact with the physical system directly, but all actions went to the Robosuite environment before it was sent to the physical environment. By doing this, the robot's dynamics were the same during testing and training.

Chapter 4.

Training

This chapter describes the frameworks used in training as well as the decisions made for training.

4.1. Frameworks

An overview of the framework architecture, which uses the Robosuite and Stable Baselines framework as the foundation for the environment and training process, is shown in fig. 4.1. The MuJoCo engine instantiates a simulation model referred to as an environment. The policy then sends a set of actions to the low-level controller and a rollout buffer. The controller uses these actions to compute a set of torques. Based on the torques, the MuJoCo engine performs internal calculations to determine the next state of the simulation. Sensors within the Robosuite framework retrieve information from a new simulation state and generate corresponding visual and proprioceptive observations. These observations are sent back to the current policy and, together with a reward, dependent on the recent action, are sent to the rollout buffer for gathering off data. The data stored in the rollout buffer is trajectories consisting of multiple action, observation, reward, and termination tuples. The dashed lines represent the data flow during the training of a PPO agent. The training loop consists of two main parts. First is the desired amount of data gathered in the rollout buffer by the current policy. Secondly, is this data used to optimize the parameters in the networks. This is done repeatedly until the agent's desired behavior is reached.

4.1.1. Algorithm used

PPO was used as the model-free on-policy algorithm for training during this thesis. The reason for this is its ability for easy hyperparameter tuning and clipped trust-region updates, which restricts poor updates as introduced in section 2.5. PPO

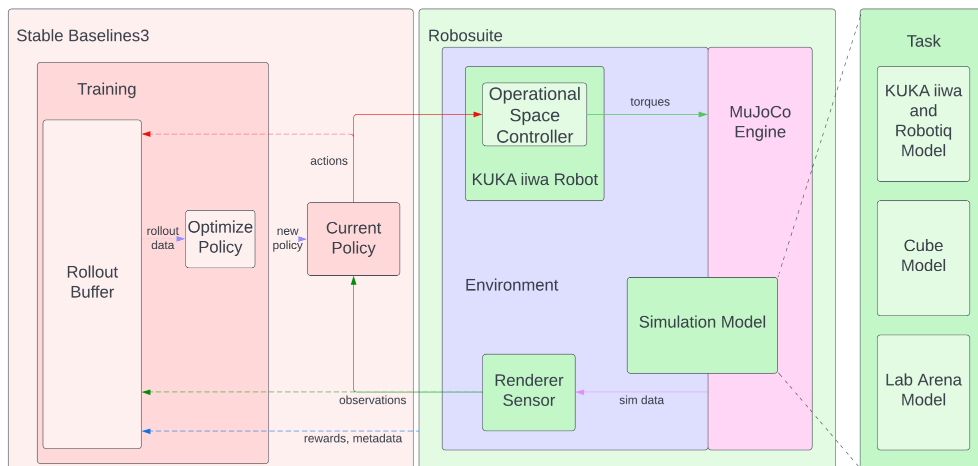


Figure 4.1.: Framework architecture. Actions are sent from the current policy to the robot's controller and the rollout buffer. The controller converts actions into low-level torque commands. The MuJoCo engine uses the torque commands to calculate a new simulation state. The sensor and renderer interpret the new simulation state and convert them into observations, which are sent back to the current policy and the rollout buffer. The rollout buffer is filled until a given amount before the data of action, observation, reward, and termination tuples are used to optimize the parameters in the policy network. The dashed lines illustrate the training loop

has also shown promising results in other papers relevant to the task at hand, like [16] that uses Robosuite and PPO to solve a similar grasping task a presented in this paper. [26] uses PPO to train a policy to reach, grasp and re-grasp different objects.

Different papers have been evaluated when comparing PPO 's performance to other algorithms. [43] compare SAC, PPO and Interpolated Policy Gradients (IPG) [21] for solving the vision-based robotic grasping problem *KukaDiverseObject* by OpenAI/Gym. The environment provides observations in the form of RGB images and takes continuous action inputs, which makes it relevant for this thesis. PPO outperform SAC by some margin. IPG scores the best, but due to lack of algorithm implementation in Stable-Baselines3 and PPOs easily tunable hyperparameters where PPO chosen above IPG.

[48] compare the performance of TRPO, PPO and DDPG on different robotics task on real-world robots. Among these tasks where a reacher task with a UR5 robot evaluated. The agent's objective was to reach arbitrary target positions by exercising low-level control over a six-joint robotic arm. DDPG performed poorly on the reaching task while TRPO slightly outperformed PPO. However, [48] also looked at the different algorithm's ability to transfer between tasks where the same hyperparameters for the original task are used on the second task. Here did PPO outperform TRPO. This result indicates that although hyper-parameter optimization is likely necessary for the best performance on a new task, a good configuration based on one task can still provide good baseline performance for another. Considering we have available hyperparameters from Stable-Baselines3 for PPO was this another strong argument as to why go for PPO as our algorithm.

It is worth mentioning that most of the other algorithms presented in this thesis have shown promising results on robot grasping and would possibly work for this task as well.

In this thesis was PPO in an actor-critic fashion used. With the actor consisting of a policy $\pi_\theta(st)$ parameterized by Θ and a critic consisting of estimated value function $V_\phi(st)$ parameterized by ϕ.

4.1.2. Stable Baselines

It can be very time-consuming and error-prone to implement DRL algorithms from scratch. Therefore, a framework with pre-existing implementations of the algorithms described in section 2.3.2 i.e PPO, SAC and DDPG was utilized. PPO was the main priority as mentioned above, but if it didn't achieve desirable results, was there a wish to have the other algorithms available.

After investigating available frameworks for model-free RL was Stable Baselines3

[63], hereafter referred to as Stable Baselines, chosen. This was due to its reliable implementation of the wanted algorithms, open-source nature, and extensive documentation. PyTorch [59] was used as the machine learning backend, which is an optimized tensor library for deep learning using GPUs and CPUs. This helped speed up the learning process.

Weights and Biases [94] was used to monitor and plot the training. The policy used a callback function during training to send relevant data.

4.1.3. Wrapping the environment

To make the environment from Robosuite compatible with Stable Baselines, the environment needed to be wrapped to follow the OpenAI Gym interface. Robosuite already had a gymwrapper, but a new class called *GymWrapper_multiinput* was made. The class was made because certain wanted features were missing. This enabled us to use CNN on the image observations while NN on proprioceptive observations (gripper status and end-effector position), limit the action space, and give information on wanted parameters like successful grasps.

To speed up the training and gathering of data, 64 parallel environments were run simultaneously. A run-through of the wrapping done for each environment before training was started is presented here:

1. Each Robosuite environment was wrapped with the gymwrapper_multiinput class. This makes the environment compatible with Stable Baselines as mentioned above.
2. The environment was then wrapped with a domain randomization wrapper. This enabled the visual randomization described in section 3.7.1. This was only done if training with domain randomization was wanted.
3. A Monitor wrapper was used for logging different data relevant for training.
4. All of the 64 environments were stacked together to a single Vectorized environment with the SubprocVecEnv wrapper from Stable Baselines. This is the wrapper that enables multiprocessing of the parallel environments.
5. Lastly, a wrapper called *VecTransposeImage* was used. This Re-ordered image channels from HxWxC to CxHxW and is required for PyTorch convolution layers.

The proprioceptive observations and reward were normalized. This was to speed up the convergence of the training process and reduce the estimation error when training with Neural Networks [77]. This was also recommended by Stable Baselines when training with on policy algorithms [85]. The image was normalized to

lie in the range of 0 to 1 by dividing every pixel by 255.

4.1.4. High Performance Computing

To train the agents, a high-performance computing cluster called Idun was used. Idun is a cluster of GPUs and CPUs available for research with High Performance Computing (HPC) at NTNU [28].

The different GPU's available are:

- P100 (40 available GPUS) with 16GB
- V100 (38 available GPUS) with 16GB and 32GB
- A100 (64 available GPUS) with 40GB and 80GB

Idun's operating system is CentOS, meaning we could not install software that was packaged for Ubuntu. Because Mujoco and Robosuite are built for a Linux operating system a containerized applications using Singularity needed to be used.

The Idun cluster's communication is based on SSH communication. Upon training, gathering data for the agent was done with 64 CPUs, while optimizing the parameters was done with a A100 80GB GPU.

Singularity

Singularity [75] is a container platform. It allows you to create and run containers that package up pieces of software in a way that is portable and reproducible. It can be built on a local laptop and then run on large HPC clusters. The container is a single compressed read-only Singularity Image File (SIF) file containing all needed software for your desired operating system. It is built from a Singularity Definition File (DEF).

The DEF file is like a set of blueprints explaining how to build a custom container. It includes specifics about the base OS to build or the base container to start from. It also includes software to install, environment variables to set at runtime, files to add from the host system, and container metadata.

A Singularity Definition File (DEF) is divided into two parts:

1. **Header**: The Header describes the core operating system to build within the container. Here you will configure the base operating system features needed within the container. You can specify the Linux distribution, the specific version, and the packages that must be part of the core install (borrowed from the host system).

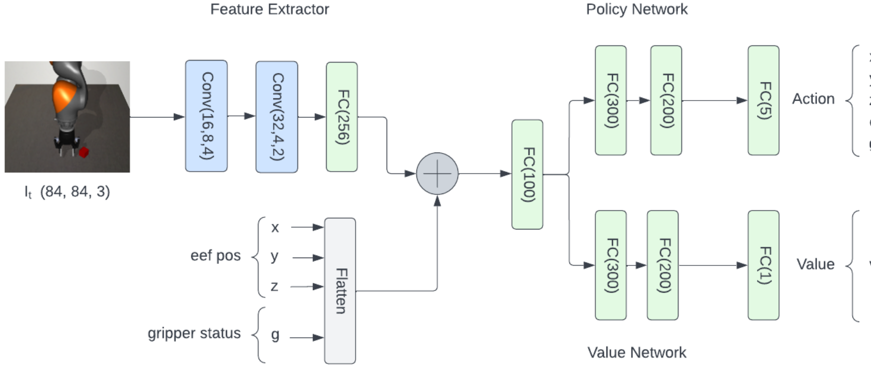


Figure 4.2.: The architecture of the Multi Input Policy

2. **Sections**: The rest of the definition is comprised of sections (sometimes called scriptlets or blobs of data). Each section is defined by a % character followed by the name of the particular section. All sections are optional, and a def file may contain more than one instance of a given section. Sections that are executed at build time are executed with the /bin/sh interpreter and can accept /bin/sh options. Similarly, sections that produce scripts to be executed at runtime can accept options intended for /bin/sh

The DEF file used in this thesis and instructions for connecting and using the Idun cluster is available on our Github repository [56].

4.2. Network Structure

The network used in this thesis was separated into two main parts, as shown in fig. 4.2. One *feature extractor* and one *fully connected* network.

The feature extractor : which is shared between the actor and the critic to save computation, extracts features from high dimensional input. In our case, an RGB image of size (84,84,3).

Fully Connected Networks that map the features to actions/value.

Our multi-input policy was inspired by [16]. The network took a monocular 84 × 84 × 3 RGB image as input, which was fed into a CNN feature extractor. The extractor consisted of an 8 × 8 convolution with 16 filters and stride 4, followed by a 4 × 4 convolution with 32 filters and stride 2, with ReLU activations in-between. The convolution outputs were flattened and passed into a linear layer

of size 256, which was concatenated with the normalized end-effector and gripper status observations.

The concatenated features were sent to a shared linear layer of size 100 before it was further fed into two separate networks for actor and critic. Both these networks had hidden layers of sizes 300 and 200 with TanH activations. The actor-network output an action means and a log-likelihood eq. (2.2) for the taken action. The critic network output a scalar which was an estimate of the value function in that state. The actions from the actor were sampled from a Gaussian distribution with a diagonal covariance matrix before being fed back to the environment.

The critic and actor-network was updated through backpropagation upon every update.

4.2.1. Hyperparameters

The PPO algorithm described in section 2.5 was used for training. The focus of this study was not to optimize hyperparameters for the algorithm, so unless stated otherwise, were the default hyperparameters from Stable Baselinses used [14]. These values were empirically yielded good results for a variety of environments [64].

PPO is an on-policy algorithm meaning it only learns from its own generated data per update. When training without domain randomization, each of the 64 environments collects 2048 tuples of data. This results in 131 072 tuples of observation, action, reward, and terminal info to optimize over for every update. When training with domain randomization, each environment takes 3072 steps before each update, which results in 196 608 tuples of data per update. More data is used when training with domain randomization to limit the trade-off discussed in section 2.4.7.

Upon every second update are, the agent was evaluated for 20 episodes. The agent was evaluated on successful grasps and episodic mean reward. An instance of the agent was saved if it beat either the previous best successful grasp or means episodic reward.

The batch size was set to 512, compared to the standard 64. This corresponded to how many experiences/tuples used for each gradient update. With the chosen batch size, the agent had the opportunity to look at all the data leading up to a successful lift. Considering that our episodes were 200 tuples long, could a batch size of 64 clip a successful lift trajectory at an unwanted place. This means that the agent only would learn from the last 64 steps leading up to the lift. It would therefore miss out on the opportunity to learn what was good to do on step 65 and further backward. So the agent would not be able to learn the optimal

action when given the observation in step 65 and further back. By using a batch size bigger than 200 this was not a problem, and the agent would not lose any information leading up to the successful lift.

The same random seed was used to train all agents. This was to prevent random parameter initializing from being a factor when comparing results.

A learning rate scheduler decreased the learning rate for every update. It started at $\alpha = 1x10^{-4}$ and linearly decreased to 0 with the remaining training process.

A summary of the hyperparameters used is shown in appendix D.

Chapter 5.

Physical environment

This chapter describes the main components of the experimental setup used in this thesis and the configuration of these. The ROS2 communication for the system is also presented. The chapter is highly influenced by the *Development Of A Test Setup For Pick And Place Tasks With Deep Reinforcement Learning* specialization project.

5.1. KUKA LBR iiwa 14 R820

5.1.1. Description

The KUKA LBR iiwa 14 R820, from now on referred to as KUKA iiwa, is a 7-degree of freedom robot manipulator delivered by the German robot manufacturer KUKA AG. The manipulator has a maximum payload of 14 kg, a precision of 0.1 mm, and a reach of 820 mm [42]. It is a lightweight robot that is aimed at doing tasks in collaboration with humans with a high degree of safety.

The KUKA iiwa has integrated torque sensors in its joints. These can be used for force control of the end effector, moving the end effector manually, monitoring the force applied on the end effector, and stopping the robot immediately if it crashes in itself or the environment.

5.1.2. KUKA Sunrise cabinet

The KUKA iiwa robot is connected to a KUKA Sunrise Cabinet. It has a KUKA.OS operating system that enables communication with third party computers as well as tools to convert commands to motion commands. The Sunrise cabinet works as the crossroad for all communication between a KUKA iiwa

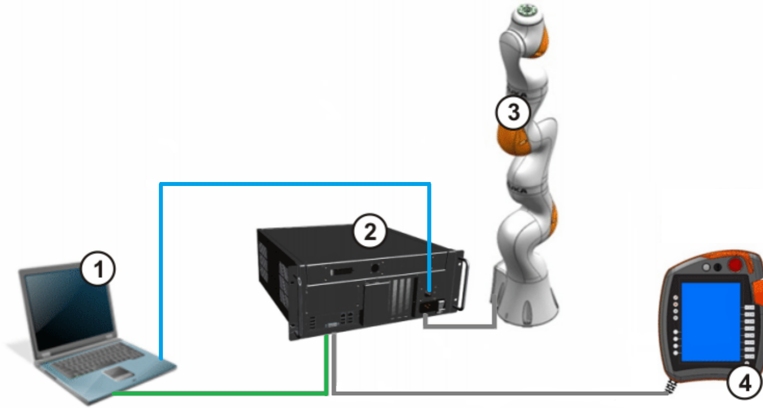


Figure 5.1.: Shows the hardware related to the KUKA robot. 1: Development computer, 2: KUKA SUnrise Cabinet, 3: KUKA iiwa, 4:KUKA smartPad. The green cable illustrates the ROS2 communication, and the blue cable illustrates the cable used for Java communication. Image from: [41]

robot, a development computer, and a KUKA smartPad control panel, as shown in fig. 5.1.

The third-party computer is used for building robot applications with the KUKA Sunrise.Workbench software. The Sunrise Cabinet in MANULAB runs on Windows 7 and is programmable with Java 1.06 through robot applications. When building an application, any interchangeable parameters for the robot can be controlled, e.g. the workspace, robot speed, and joint limits can be restricted to improve the environment's safety. In addition to changing parameters, the communication interfaces can be configured.

As shown in fig. 5.1 a KUKA smartPad control panel is also connected to the Windows 7 part of the Sunrise Cabinet. This can be used to start up applications uploaded to the Sunrise Cabinet. It can reset any applications, override them, and be used for repositioning the robot if it moves to the outer limits of the joints or the workspace, as well as being an interface for surveillance of joint and end-effector position. Error messages also appear if something is out of place.

5.1.3. Motion control

Actions can be sent to the Sunrise controller as an individual pose or a trajectory. These actions can be sent in joint space or operational space. There are 7 variables for a joint space action, one for each joint. For an operation space action, there are 6variables; 3 for the Cartesian position and 3 for the rotation of the end effector. The actions can be represented as absolute values, where the base frame of the

robot is used as the reference, or as Tool Center Point (TCP) values where the current state of the end effector is used as reference.

The Sunrise controller can calculate the actions into motions. The robot can be controlled with linear motion or Point-to-Point (PTP) motion. With a linear motion the robot moves in a linear path in the Cartesian coordinate system. With a PTP, the path is optimized for fast movements with the robot joints. Both the linear and PTP movements have SmartServo versions. These motion planners can have vision-based collision avoidance and can continuously correct the path of the robot.

5.1.4. KUKA iiwa in MANULAB

The KUKA iiwa in MANULAB was placed 30 cm from a wall on a table. Safety restrictions in the robotic configuration were in place to limit the workspace, preventing the robot from moving close to the wall or outside of the table parameter. These restrictions prevented the robot from crashing into hardware or people. There were no restrictions to avoid collisions between the gripper and table, but the torque sensors could detect a collision and shut down the robot. In addition, the joint limits were set to the values shown in table 5.1.

The application uploaded to the Sunrise Cabinet from the third-party computer enabled ROS2 communication with the robot [84]. It was located on the computer used in MANULAB and synchronized through an Ethernet cable connected to the Sunrise Cabinet. When the application was launched from the smartPad, the controller constructed a ROS2 node that enabled DDS 2.6.3 communication over another Ethernet cable connected between the computer and the Sunrise Cabinet. The cables are illustrated in fig. 5.1. Both the subscriber and publisher of the robot node communicated with joint position messages.

The actions were sent to the robot with individual absolute joint space positions. The robot was moved with the PTP Smartservo controller mentioned in section 5.1.3. Arguments for why this was chosen are presented in chapter 6.

5.2. Robotiq 2F-85 Gripper

5.2.1. Description

The Robotiq 2F-85 gripper is a two-finger gripper made for repetitive tasks with robot manipulators. It has an 85 mm stroke, a pinching force between 20 and 235 newton force, and weighs 0,9 kg [78]. The gripper has an under-actuated structure with one actuator. Consequently, the fingers adapt to the shape of the objects

Table 5.1.: Shows the joint limits for the KUKA iiwa. These are used both for the physical and simulated robot.

Link	Min (deg)	Max (deg)
Link 1	-170	170
Link 2	-120	120
Link 3	-170	170
Link 4	-120	120
Link 5	-170	170
Link 6	-120	120
Link 7	-175	175

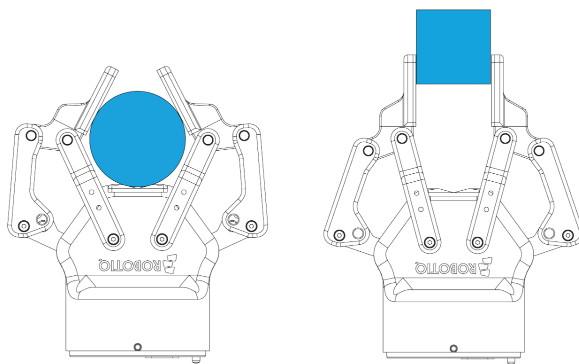


Figure 5.2.: Shows the under-actuated abilities of the gripper. ([66], page 12

it is picking, as shown in fig. 5.2. The fingers are replaceable and can perform internal and external gripping if needed.

5.2.2. Software and communication

The Robotiq gripper has a Robotiq User Interface that can be used to test the gripper's connection and functions before connecting to it through custom-made code. Several communication protocols are available for use, but in this project, the Modbus RTU communication was used. After sending a startup signal to the gripper, it can be controlled by transmitting force, position, and speed with values between 0 and 255. A value of 0 corresponds with a closed gripper, and a value of 255 corresponds with an open gripper. Variables like requested position, current movement, current position, and fault status can be sent to the user as feedback throughout a gripping process.

5.2.3. Gripper in MANULAB

The gripper was connected to the KUKA iiwa end effector flange with an aluminum adapter plate. A Robotiq coupling with cable inlet was placed between the gripper and the adapter plate. The power and communication were delivered with a custom Robotiq device cable from a Robotiq controller box. The cable was fastened to the KUKA iiwa robot with a rubber band to prevent it from being in the way of the robot's movements. The Robotiq control box had its power supply from a Siemens SITOP PSU100S and was connected to the third-party computer for communication through a USB2 cable. Modbus RTU protocol was used to communicate with the gripper as mentioned in section 5.2.

As mentioned in section 5.4.2, a ROS2 node enabled communication between the agent and the gripper. In addition to setting up the Modbus RTU communication, the motion parameters for the gripper can be set. The speed was set to 200, corresponding to 120 mm/s according to [66]. The force was set to 1, corresponding to the lowest force of the gripper, 25 N. The low force was set to mimic how the object slips between the simulated fingers.

It was desired that the physical gripper acted identical to the simulated gripper when given an action from the agent. Therefore, as done in simulation training, the node clipped the agent action according to eq. (3.1). When a closing signal was sent from the policy node, the gripper moved towards its goal in four movements to mimic the simulated gripper. This is further discussed in section 3.6.2. After moving all four steps towards a closed position, the fingertips still had a 12 mm space between them. Firstly, this was done to mimic how the gripper looks when closed in the simulation fig. 3.4. Secondly, it stopped the physical gripper from getting stuck when in contact with the table. Because it is an under-actuated gripper, it gave in when pushed if not fully closed, as shown in fig. 3.4. This would not be possible with a fully closed gripper. The four steps from open to closed were only approximately identical because the 12 mm offset for a closed physical gripper is not the same as the simulated gripper. However, the gripper should be in contact with the cube in the same states.

5.3. Zivid Two Camera

5.3.1. Description

Zivid Two is a high-definition 3D vision camera that can provide point clouds with RGB information with a resolution of 1944 x 1200 [102]. It uses structured light to create its point clouds and collects both the point cloud and the RGB image with the same sensor. This fact makes a precise XYZ and RGB matching. The camera can send images with a frequency of 10 Hz.

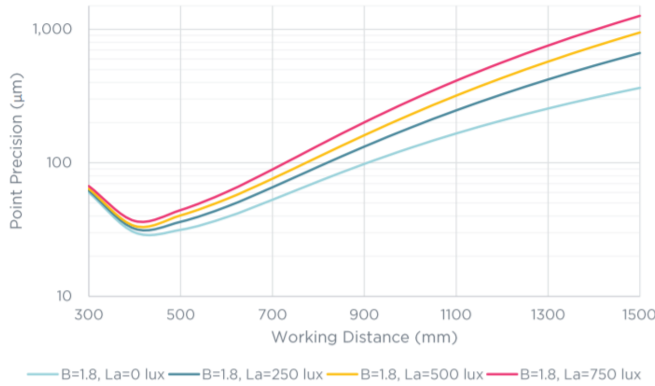


Figure 5.3.: Optimal working distance for Zivid 2 camera [102].

In addition to the XYZ and RGB output, the Zivid camera has an SNR output for every image pixel. The SNR value is a Signal-to-Noise Ratio that describes the camera's confidence in the distance to a specific pixel. A low SNR value means a low confidence in the distance and subsequently less useful data [76].

The camera's weight is 945 g with a size of 169 mm x 122 mm x 56 mm. Zivid Two has an optimal working distance of 500-1100 mm and a recommended working distance of 300-1500 mm. The fig. 5.3 shows how the point precision changes with distance. This is important to consider when choosing the placement of the camera relative to the object being captured.

An important concept used while capturing an image is the camera acquisitions. An acquisition is a set of parameters that determine the outcome of the capture. The parameters used by Zivid are aperture, brightness, exposure time, gain, and color balance. High Dynamic Range (HDR) images referrer to images where several acquisitions are combined to get higher quality data. In a scene with both dark and light spots, a HDR image can combine images to get high-quality data from both of these spots.

The structured light used by the camera to capture 3D data comes from a projector. It can be seen in the 2D image from fig. 3.1 that the projector cast light on the environment. This can cause reflections from hardware in the environment, and be a disturbance when 2D images are used as input to an agent.

5.3.2. Software and Communication

The Zivid SDK has tools that enable communication with the Zivid camera. The simplest way of communicating with the camera is to use Zivid Studio. This is

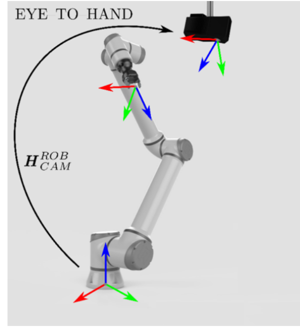


Figure 5.4.: Hand-eye calibration for stationary mounted cameras. [89].

software where the user can capture, analyze and adjust 3D and 2D images from the camera. It is a useful test tool to find what parameters and acquisitions that fits the project's purpose and is used during the hand-eye calibrations in this project. Official Python [104] and unofficial ROS2 [62] packages are also available as options for the camera communications.

5.3.3. Hand-eye Calibration

Hand-eye calibration can be used to describe how the robot and camera are placed in relation to each other. Figure 5.4 illustrates how we have a robot base frame, an end effector frame, and a camera frame in an environment. By finding the transformation matrix between the base frame of the robot and the camera frame H_{CAM}^{ROB}, we have a description of the camera placement, which can be used to place the camera accurately in a simulation.

H_{CAM}^{ROB} can also be convenient when collecting 3D data of an object. By finding this transformation matrix, it is possible to present the 3D data from the camera in the robot base frame. This can be useful when the robot is controlled with operational control because the end-effector positions correspond with the 3D data from the camera. The relationship between the transformation matrices can be seen in equation eq. (5.1).

$$H_{OBJ}^{ROB} = H_{CAM}^{ROB} H_{OBJ}^{CAM} \tag{5.1}$$

Zivid SDK versions after 1.6 include an API for hand-eye calibration in 3D. It is done by capturing between 10 to 20 images of a checkerboard [23] and further using the images with the Zivid CLI tool for Hand-Eye calibration. With the help of a custom Zivid checker board attached to the end effector, Zivid studio, and

the python package with calibration scrips [104] a calibration can be executed.

5.3.4. Zivid in MANULAB

By using the Zivid camera, 3D information was available to be used as observations. Despite being used solely as an RGB camera in our experiments, we wanted to have a viable configuration for depth data because of the high-quality 3D abilities of the Zivid camera. Therefore, this section presents solutions that can be used for collecting RGB and 3D data.

The camera was placed in front of the objects and the robot on the table in MANULAB. By placing the camera out of reach of the robot, we could sustain a higher degree of safety than having it in the robot's workspace. Another way of solving this would be to limit the workspace through the robot application. We also discovered that the robot could interfere with the object's visibility in some over-the-shoulder configurations.

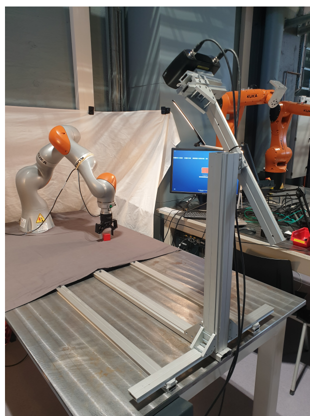
The Zivid camera was placed approximately 800 mm from link 6 of the robot and 1100 mm from the objects, just within the recommended range of 500-1100. If the camera was to be used for 3D images, the distance could be decreased to be further within the range.

A fixed position was chosen over an on-wrist configuration in our system. An end-effector view could make the simulation implementation more complex and demand a higher degree of restriction to the physical robot movement to prevent the robot from colliding with the camera. It also simplified the configuration of the Zivid camera parameters because of the fixed distance and light conditions. The HDR acquisitions did not need to be changed during a task with this fixed configuration.

The camera's tilt was chosen so that the objects and large parts of the robots were visible. The camera was panned to place the scene perpendicular to the 2D camera lens. This means that the camera was panned with three degrees to compensate for the offset angle mentioned in [102].

The camera also has an angle looking downwards at the objects. Zivid recommends placing the camera at an angle to avoid reflections from the objects, and background [103]. This could also increase the depth perception from the 2D images compared to a placement closer to the table. The distance between the gripper and camera could be harder to interpret with a lower placement. Looking downwards with an angle, several edges of the cube were visible, which could help in interpreting the cube's orientation.

The Zivid camera was placed on an aluminum adapter plate. This plate was


(a)

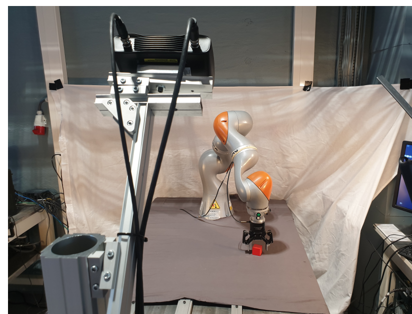

(b)

Figure 5.5.: Shows how the Zivid camera is placed in the physical environment.

attached to a modular Montech [53] setup that enabled precise translation and rotation of the camera. Figure 5.5 shows how the setup had a base with three aluminum profiles screwed to the table. This prevented any movements of the camera after a hand-eye calibration.

The capture parameters had to be tuned to fit the captured environment and get high-quality images. This goes for both 2D and 3D images. The ROS2 driver enabled modification of the High Dynamic Range (HDR) acquisitions to improve the quality when the lighting differed within the same image. Single-acquisitions were available for both 2D and 3D images, but multi-acquisition images were only available for 3D images [105]. We used 2D images and could only use single acquisition capture with the following values:

Aperature:	2.83
Brightness:	1
Exposure Time:	25000
Gain:	1
Blue color balance:	1.4
Green color balance:	1
Red color balance:	1

5.4. Physical Setup

5.4.1. MANULAB environment

Other parts of the physical environment than the robot, camera, and gripper had to be configured. The lighting of the environment was mainly from the MANULAB roof lighting. In addition, we supplemented with lights at the neighbour tables as shown in fig. 5.5. This was to make the objects more visible and to prevent shadows from disturbing the image. By doing this, the final lighting was closer to the uniform light in the simulated environment. All test in the physical environment was done with daylight present to different degrees.

The object to be grasped was chosen to be a 3D printed cube with an approximated size of 40 mm. This size fits the 85 mm opening of the gripper with a fair margin. The color of the cube was chosen to be red, a contrasting color to the grey table surface and black gripper. The 3D printed surface has some degree of reflection that could interfere with the image quality.

The table initially had a metallic surface. A grey bed sheet was used to limit the reflection from the surface. The reflections can impact the performance of the Zivid camera when capturing depth images. The grey bed sheet contrasted the colorful objects and had a high coefficient of friction. The wall was covered with a white bed sheet as shown in fig. 5.5 to prevent the noisy wall from being visible in the image observation.

5.4.2. ROS2 communication

The communication used in the physical test setup was based on ROS2. The topics and nodes for the communication are illustrated in fig. 5.6. In the heart of the communication, we have the PolicyNode that subscribes and publishes to all the other nodes. In this section, we will present all the nodes of the architecture as well as the topics and services.

As mentioned in section 2.6 the ROS2 communication is based on DDS communication. The implementation of the CycloneDDS on KUKA Sunrise.OS [84] was used for enabling ROS2 communication with the robot controller. The application constructs a ROS2 node with subscribers and publishers with their associated topics. Both the /command and /state topics have information about the robot's seven joints.

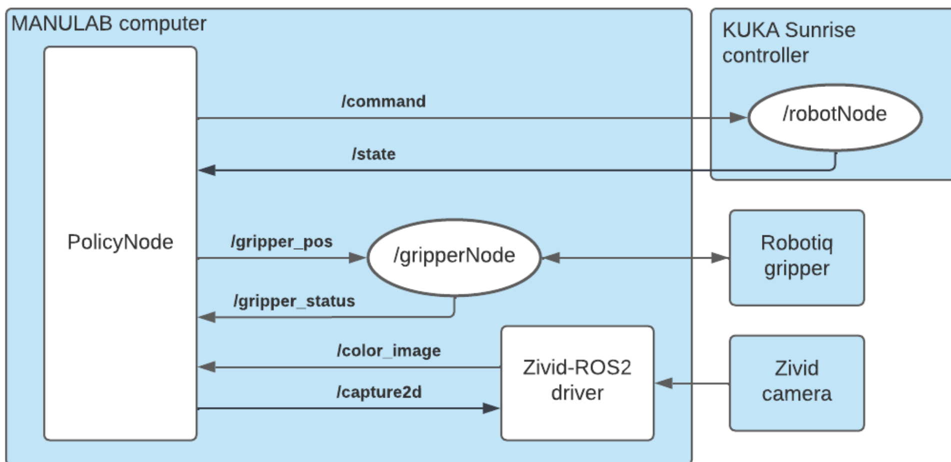


Figure 5.6.: Shows the ROS2 nodes and the topic communication.(should the ZividNode be placed inside the Zivid-ROS2-Driver? Mention that /capture2d is a service?)

/command and */state*

```
float32 joint_1
float32 joint_2
float32 joint_3
float32 joint_4
float32 joint_5
float32 joint_6
float32 joint_7
```

The /gripperNode was run from the MANULAB computer. It received /gripper_pos message, a float that contains a requested position for the gripper. This message was a float between -1 and 1. This number was, in turn, transformed into gripper commands based on the sign of the number.

/gripper_pos

```
float32 pos
```

The movement variable of /gripper_status message 5.4.2 is the object detection status variable described in [66]. Its values corresponds to the following:

0: Gripper is in motion

1: Gripper has stopped while opening

2: Gripper has stopped while closing

3: Gripper is at the requested position

In the policy node, a statement checked whether the gripper was in motion. The node did not proceed in taking a new step before the value was different from 0 and 1. A value of 2 was interpreted as a grasp. Both 2 and 3 was considered successful movements to the requested action.

/gripper_status

```
int8 movement
```

The Zivid node based on the Zivid-ROS2 driver was launched on the third-party computer. An empty service message, Capture2d, was sent to the Zivid node to capture an image. This service call initiated a capture, and an image was sent to the /color/image topic.

/color_image

```
sensor_msgs.msg.Image
```

A *Capture* service call for capturing 3D data was also available. The *Capture* service could initiate a capture, and a point cloud could be sent to the depth/-pointCloud topics. This can be extended to point clouds with RGB data and RGBD topics as the ROS1 driver for Zivid has available [105].

Chapter 6.

Sim-to-Real Transfer

This chapter presents how the sim-to-real transfer is executed in our experiments. Firstly the main solution is presented and argued. Secondly, the communication architecture is presented in detail.

6.1. Solution

A typical zero-shot transfer is done by switching all the simulated actions and observations to physical observations and actions. Our solution ran a simulation in parallel, which was used as a "translator" between the agent and the physical environment. The simulation was identical to the one used in training. Figure 6.1 shows how the actions from the agent were sent to a simulated environment. The physical hardware copied the joint angles of the simulated robot after the execution of an action in the simulation. These joint angles were sent to the controller in the physical lab. The observations given to the agent were from the simulation except for the image taken in the physical environment. In this chapter, we will argue why this architecture is chosen.

6.1.1. Robot actions

The torque-based controllers in Robosuite made it challenging to make the agent communicate directly with the physical robot. Robosuite suggests in their paper [100] to use torque-based control to use the agent in a physical environment. "Our controllers facilitate sim-to-real transfer ability, as torque-based controllers are common to most real-world existing robotic platforms". KUKA iiwa can be controlled with torque commands, but we did not have this control option available in the ROS2 interface. An effort to implement torque control in the ROS2 interface was not made because the action space was chosen to be in operational space as discussed in section 3.4.

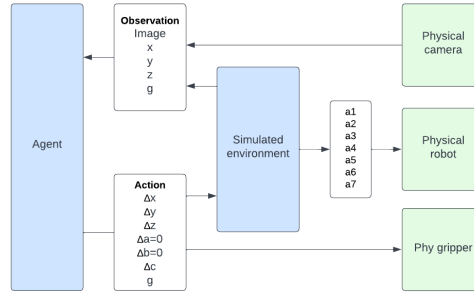


Figure 6.1.: The agent-environment interaction during physical testing

When transferring the agent to a physical environment, it was essential that every action from the agent had the same reaction as in the simulated environment. That is, the relationship between the agent's action and the robot's movement had to be the same in the simulated and physical environments. The solution was to use a simulation to translate the operational space actions from the agent to joint space positions to the physical robot. Joint positions were used in favor of end-effector positions because they were unambiguous.

Even though the simulation controller translated the actions to joint commands, the robot's movement between each point was somewhat different. The section 3.2.4 controller calculated joint torque commands by minimizing the error between the desired and current pose with the minimal kinematic energy. In contrast, the physical robot controller moved the robot with the PTP SmartServo controller, where the fastest path is calculated and executed.

It was not conducted tests on how different the trajectories from these controllers were, but it was assumed that the trajectories were nonidentical because of their computational differences. To reduce the difference a low degree of movement between every agent step was used. As mentioned in section 3.2.4, the simulated robot moves between every agent step with a maximum movement of 15 mm and 12.9 degrees.

One way to make the movements of the simulated and physical robot more identical would be to record the joint position of the simulated robot for every time-step instead of every agent-step. Every agent-step consists of 50 intermediate time-steps as discussed in section 3.6.2. These joint positions could be used to make a detailed trajectory for the physical robot and therefore make the similarity between simulation and physical robot behavior closer.

6.1.2. Gripper actions

The gripper action from the agent was sent directly to the physical gripper and not through the simulation. The same clip function 3.1 used in the simulation was implemented for the physical gripper to secure the same movement in the two environments.

6.1.3. Observations

We wanted the observations in the physical implementation to be similar to the observations in training to get a high-performance sim-to-real transfer. Because the simulation ran in parallel with the physical environment, observations from the simulation were available. In our system, the agent got the image from the physical environment and was supplemented with simulated observations.

The end-effector position and gripper status were sent to the policy from the simulated environment instead of the physical environment. The end-effector position was taken from the simulated version because a forward kinematics calculation would be needed to convert the joint position from the physical robot's end-effector position. The gripper status was taken from the simulated environment because it is assumed to be in the same state as the physical gripper after being given identical actions throughout the episode. The gripper status could also have come from the physical gripper.

All the observations available in Robosuite could have been used in our training and physical testing. However, we have prioritized observations that have shown promising results on previous vision-based robotic grasping tasks [36].

6.2. PolicyNode communication

The PolicyNode was the crossroad for all communication in the physical system. It published and subscribed to the different parts of the ROS2 system. In addition, it created the simulated Robosuite environment and collected the commands from the agent in Stable Baselines.

Figure 6.2 shows how the node sequentially communicated with all parts of the system for every agent-step. Firstly, PolicyNode continuously checked if the physical robot and gripper had arrived at their goal state, as shown in fig. 6.2. A loop was implemented to ensure that the offset between the physical and simulated robot the same with an absolute tolerance of 0.0001 radians.

When the physical hardware was in the same state as the simulated environment, a service call was sent to the ZividNode, and an image was sent in response. The

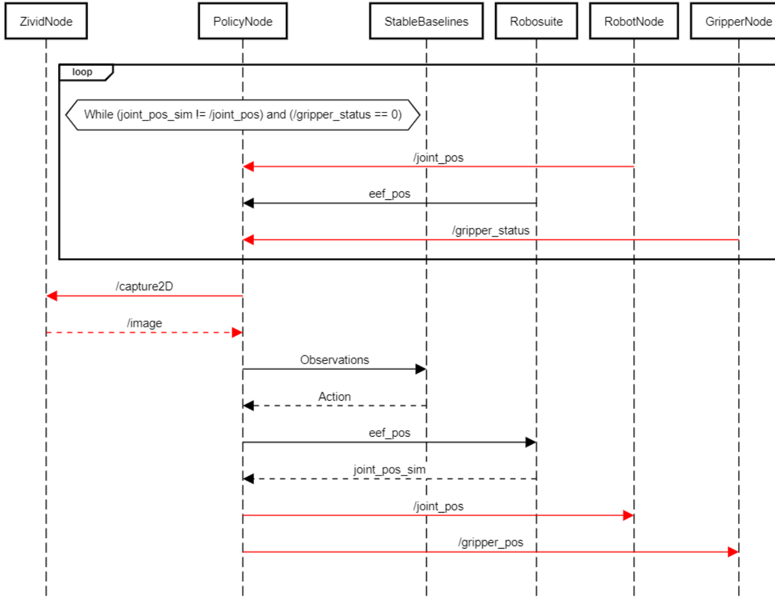


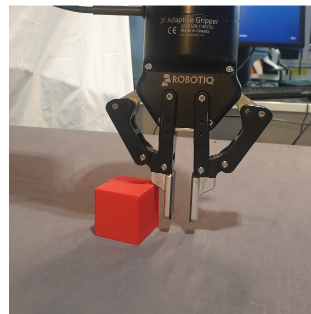
Figure 6.2.: Shows the communication for the policyNode. The red lines illustrates the ROS2 communication.

PolicyNode then sent its observations to the agent in Stable Baselines. The observation consisted of the physical image, the simulated end-effector position, and the gripper position. The gripper position was read directly from the agent action and sent to the GripperNode. When the agent had processed the observations and found an action, the PolicyNode received the actions consisting of end-effector and gripper positions. In turn, the simulated robot moved to its position, and the joint's position was sent back to the PolicyNode. Lastly, the physical robot received the simulated robot's joint position, and the gripper received its action.

6.3. Safety

A height restriction was set in the simulation to prevent the gripper from crashing into the table, as mentioned in section 3.4. This consequently prevented the physical gripper from crashing into the table, but we found other ways the gripper could crash when in contact with the object. Two configurations where the torque sensors detected a crash during testing are shown in fig. 6.3. In addition, the friction between the gripper and the object was the reason for several detected collision during testing. This is discussed in section 9.3.

(a)



(b)

Figure 6.3.: Shows how the gripper can crash into the object despite the hight limit implemented.

Part III.

Experiments

Chapter 7.

Hand-eye Calibration

To be able to place the camera accurately in simulation a transformation matrix was needed. As mentioned in section 5.3.3 the transformation matrix for camera placement can be found through a hand-eye-calibration.

The hand-eye calibration was based on the open-source python code from Zivid [104]. Zivid recommend capturing 10 to 20 images with their corresponding end effector positions. Figure 7.1 shows how a set of image and end effector data can look like. The data is used as input to the Zivid code and a transformation matrix is returned when the calibration is finalized. In addition to the matrix, the code returns rotation and translation residuals for every pair of images and end effector positions. Residuals are the difference between the final transformation matrix and the suggested transformation matrix from each pair of data.

We did not find the Zivid hand-eye-calibration code satisfactory. The reason for this was that it was not possible to remove pairs of images and end effector

Figure 7.1.: Example image form the hand-eye calibration. The robot is in the following pose: x = 602.00 mm, y = -126.24, z = 224.32, a = -77.59, b = 67.03, c = -137.71

positions with bad residuals. The code was modified to be able to use 3D images saved in a folder with the corresponding end effector positions in a .txt file. By doing this we had full control over what pairs of images and end-effector positions that were used in the final calibration. Pairs with high residuals could be removed. This change also made it possible to manually adjust the acquisitions of the images separately to make sure that high quality point clouds was used in the calibration.

In the final calibration 20 images was taken whereof 2 were removed because of bad residuals. The end effector position was read from the KUKA smartPad and written in the .txt file. The same acquisition based of a configuration file from Zivid [25] was used for all the 20 images. All images, corresponding end-effector positions, configuration file, results, and calibration code are to be found in the hand_eye_calibration directory in the digital appendix C.3.3.

7.1. Results

7.1.1. Estimated transformation matrix

The Zivid camera was originally placed based on the arguments discussed in section 5.3.4. An estimated transformation matrix was found by measurement in the physical environment and testing in simulation:

$$T_{custom} = \begin{bmatrix} 0.0000 & 0.5801 & -0.8146 & 1450.0000 \\ 1.0000 & 0.0000 & 0.0000 & 0.0000 \\ 0.0000 & -0.8146 & -0.5801 & 850.0000 \\ 0.0000 & 0.0000 & 0.0000 & 1.0000 \end{bmatrix}$$

When multiplying the rotation part of T_{custom} with the matrix described in fig. 3.7 the custom transformation matrix used in simulation was:

$$T_{custom,sim} = \begin{bmatrix} 0.0000 & -0.5801 & 0.8146 & 1450.0000 \\ 1.0000 & 0.0000 & 0.0000 & 0.0000 \\ 0.0000 & 0.8146 & 0.5801 & 850.0000 \\ 0.0000 & 0.0000 & 0.0000 & 1.0000 \end{bmatrix}$$

7.1.2. Calibrated transformation matrix

The hand-eye calibration gave us a the following transformation matrix:

$$T_{calibrated} = \begin{bmatrix} 0.0074 & 0.5884 & -0.8085 & 1422.5784 \\ 1.0000 & -0.0028 & 0.0072 & -19.3504 \\ 0.0019 & -0.8086 & -0.5884 & 849.8094 \\ 0.0000 & 0.0000 & 0.0000 & 1.0000 \end{bmatrix}$$

Residuals for all the 18 images are to be found in hand_eye_calibration/results.txt in the digital appendix C.3.3. The $T_{calibrated}$ matrix had the following residuals:

Residual for rotation (deg)	Max: 0.210, Average: 0.112, SD: 0.039
Residual for translation (mm)	Max: 0.672, Average: 0.424, SD: 0.110

When multiplying the rotation part of $T_{calibrated}$ with the matrix described in fig. 3.7 the calibrated transformation matrix used in simulation was:

$$T_{calibrated,sim} = \begin{bmatrix} 0.0074 & -0.5884 & 0.8085 & 1422.5784 \\ 1.0000 & 0.0028 & -0.0072 & -19.3504 \\ 0.0019 & 0.8086 & 0.5884 & 849.8094 \\ 0.0000 & 0.0000 & 0.0000 & 1.0000 \end{bmatrix}$$

7.1.3. Comparison

The difference in translation between the $T_{custom,sim}$ and $T_{calibrated,sim}$ is:

$\delta x = -2.74$ mm

$\delta y = -1.94$ mm

$\delta z = 0.19$ mm

By looking at the angular differences between the rotation matrix for the custom camera $R_a = R_{custom,sim}$ and the calibrated camera $R_b = R_{calibrated,sim}$ we can find the angle of rotation θ about an axis $\overrightarrow{n}$ that describe the rotation between the two. First we find the rotation matrix from the custom orientation to calibrated orientation R_{ba}.

$$R_{ba} = R_b^T R_a \tag{7.1}$$

Then we find the angle of rotation θ

$$cos(\theta) = \frac{Tr(R_{ba}) - 1}{2} \tag{7.2}$$

This results in an angular difference between $T_{custom,sim}$ and $T_{calibrated,sim}$:

$\delta\theta = 0.739$ deg

7.2. Discussion

The hand-eye calibration resulted in a transformation matrix $T_{calibrated}$ with residuals in translation lower than a millimeter for all of the 18 sets of images and end-effector positions. When used to place the camera in simulation, it can be argued that the placement got a close to identical perception compared to the physical camera.

Small changes in camera orientation may change the placement of objects in the captured image to a larger degree than translation. The residuals for rotation has an average of 0.112 degrees. This corresponds to a 2 mm difference in perceived placement in a 1000 mm distance. We see this as satisfying results considering we have a low resolution image as agent observation.

By comparing the $T_{calibrated}$ and T_{custom} we can see how the angular difference is higher than the average residuals of the hand-eye calibration. The translation difference of 2.74 mm in the x-direction is also higher than the average residuals. This shows how the hand-eye calibration can be a tool to get a higher quality placement than with our manual measuring.

Chapter 8.

Training in simulation

This chapter presents the results and discussion of the agents trained in simulation. The overall goal of this thesis is to compare the results of trained agents on sim-to-real transfer. A discussion of results is presented but due to the overall goal is not an in deep evaluation and comparison of results done. The final agents presented are the same as the ones tested in the chapter 9. The plots are collected directly from the Weights and Biases framework [91].

8.1. Results

Under follows, a list of the trained agents and their different configurations are presented. The goal of training these agents is to compare which configurations best tackle the sim-to-real transition. This section presents the training results from the agents used in the sim-to-real testing. The agents success rate are measured on 20 attempts every second update of the agent.

For plots of loss, hyperparameters and other things relevant for training with PPO see appendix B and appendix D.

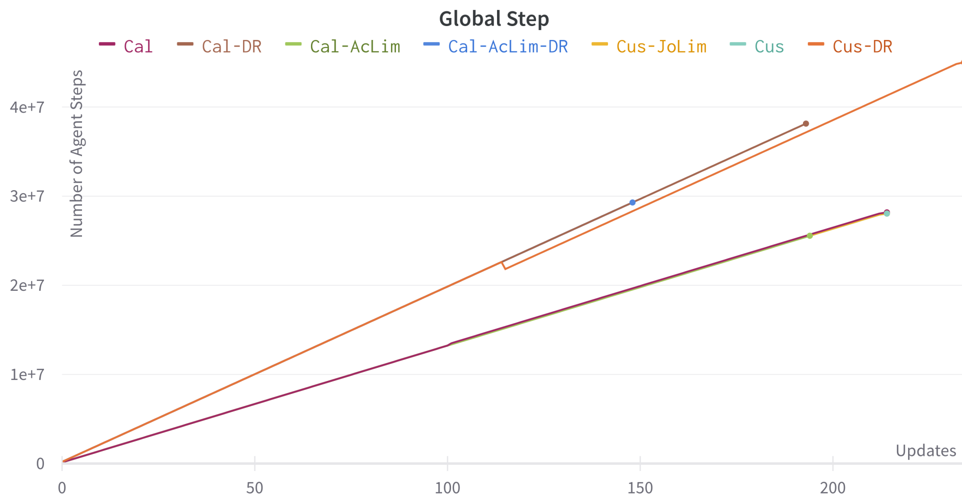


Figure 8.1.: Showing number of agent steps in each environment per update on the PPO agent. It also shows the total number of agent steps used to train each model.

Cus: The camera is placed by manual measurements, trying to put it close to the camera placement in lab.

Cus-DR: Same camera placement as *Cus*, but with visual randomization as described in section 2.4.7.

Cus-JoLim: Same camera placement as *Cus*. A rotational constraint added with $\pm 110°$ on joint 7.

Cal: The camera is placed according to the hand-eye calibration described in chapter 7.

Cal-DR: Same camera placement as *Cal*, but with visual randomization as described in section 2.4.7.

Cal-AcLim: Same camera placement as *Cal*. The action space is limited to only include the cartesian coordinates [x,y,z] and the gripper command [g].

Cal-AcLim-DR: Same camera placement as *Cal*. Same [x,y,z,g] action space limit as *Cal-AcLim*. Visual domain randomization added as described in section 2.4.7.

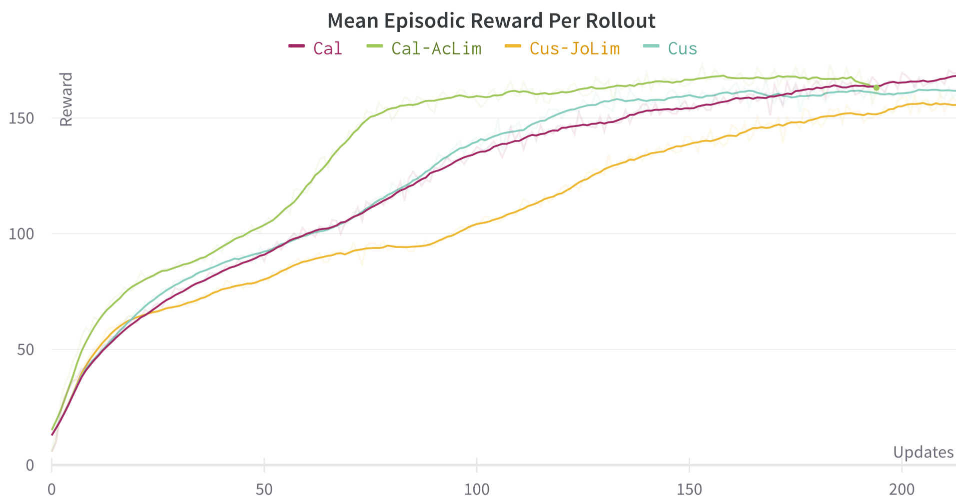


Figure 8.2.: Shows mean episodic reward for *Cal*, *Cal-AcLim*, *Cus-JoLim*, *Cus* during training. This compare the result for agents trained without domain randomization.

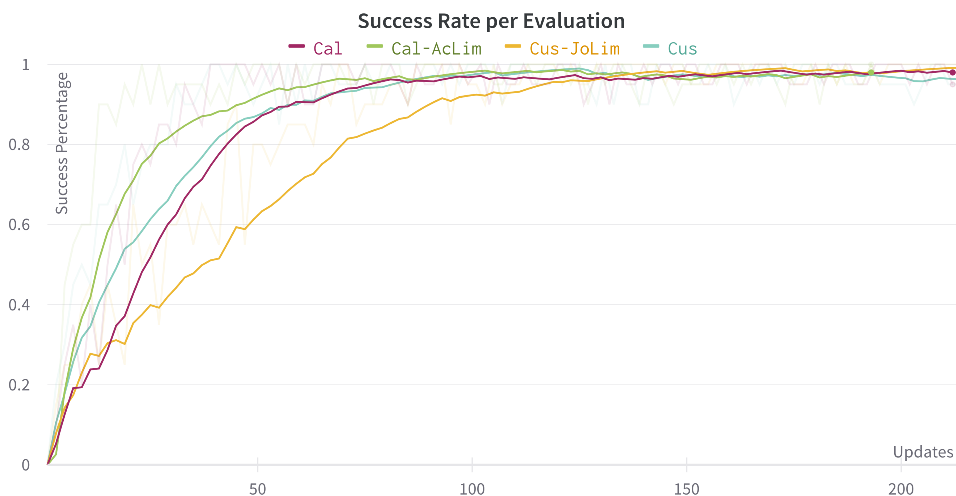


Figure 8.3.: Shows the success rate on lifting the cube for *Cal*, *Cal-AcLim*, *Cus-JoLim*, *Cus*. This compare the result for agents trained without domain randomization.

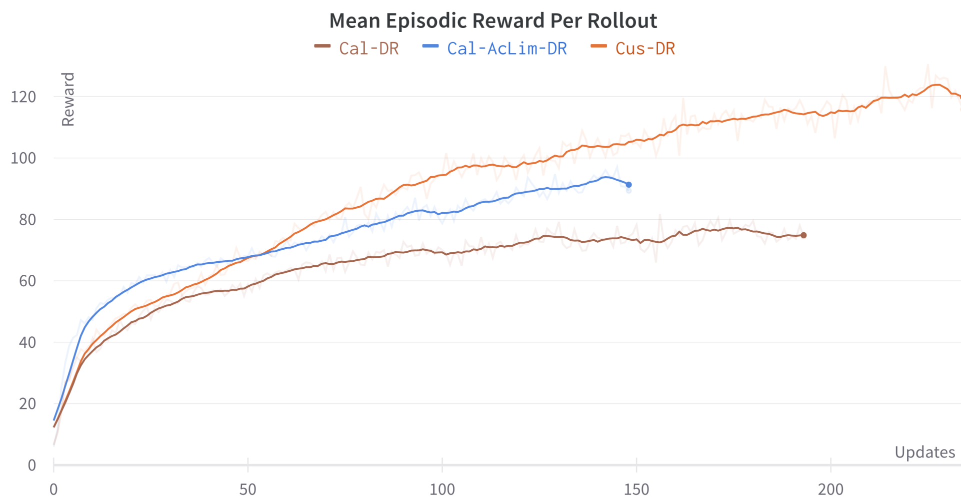


Figure 8.4.: Shows mean episodic reward for *Cal-DR Cal-AcLim-DR Cus-DR*. This compare the result for agents trained with domain randomization.

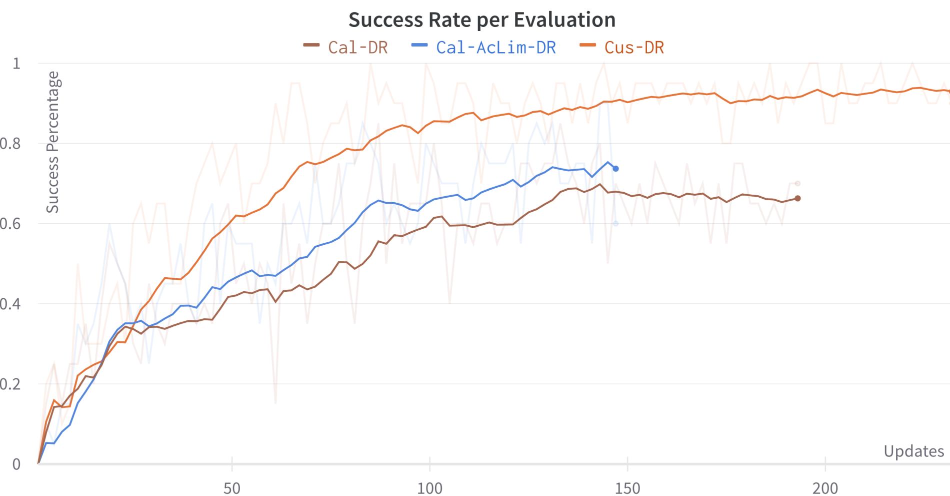


Figure 8.5.: Shows the success rate on lifting the cube for *Cal-DR Cal-AcLim-DR Cus-DR*. This compare the result for agents trained with domain randomization.

Model	Number of Updates
Cus	39
Cus-DR	85
Cus-JoLim	69
Cal	39
Cal-DR	Never, Highest 85%, 83. update
Cal-AcLim	23
Cal-AcLim-DR	Never, highest 90%, 143. update

Table 8.1.: Number of steps to reach 100% success rate for the first time for all agents. This is to compare the speed at which they learn

Table 8.2.: Agents tested outside of training on simulated environment for 20 episodes. The lift success and the average agent step until successful grasp is measured. The video of all agents can be seen in appendix C.3.2 in the simulation folder

Algorithm	Number of lifts/20	Average Lift agent Step of the successful lifts
Cus	100%	14.8
Cus-DR	100%	28.4
Cus-JoLim	95%	22.2
Cal	95%	15.7
Cal-DR	80%	64.2
Cal-AcLim	100%	17.8
Cal-Aclim-DR	75%	33.6

8.2. Discussion

The task of lifting the object has several phases. During the comparison of the different policies these phases were discussed.

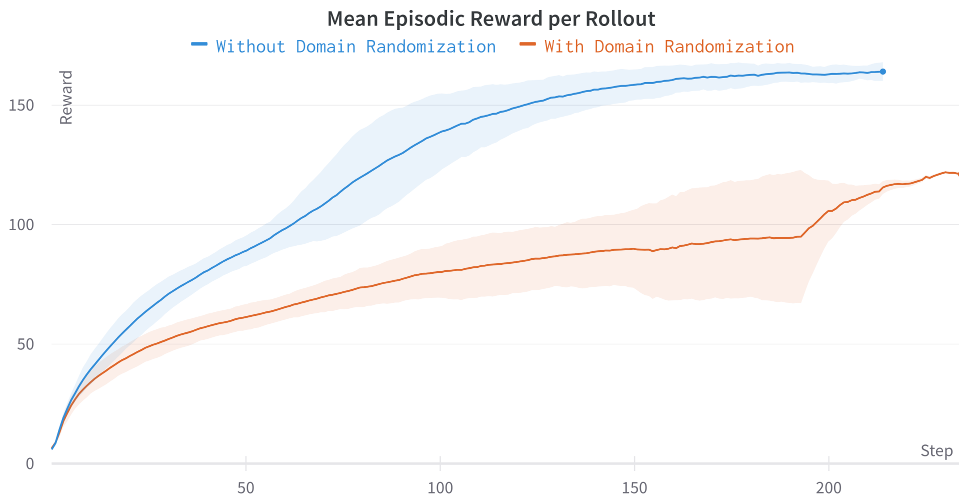


Figure 8.6.: Shows the mean and standard deviation of the agent trained with and without domain randomization.

Reaching:	The object must be placed between the fingers of the gripper.
Grasping:	The fingers must be closed while the gripper position stays unchanged.
Lifting:	The gripper must lift the object while staying closed.

8.2.1. Agent steps during training

Figure 8.1 shows the number of agent steps taken by each environment per update on the agent. All the environments using domain randomization have a higher number of steps per update, this is due to the increased complexity of the environment and therefore a need for more data per update to learn as discussed in section 2.4.7.

8.2.2. The Baseline, Custom Camera *Cus*

The agent *Cus* can be seen as a baseline when measuring performance of the other agents. Its performance on reward and success rate can be seen in fig. 8.2 and

fig. 8.3 with a light blue color. The *Cus* agent shows solid results during training and reaches a success rate of 100% and a final mean reward of 162.

How the three lifting phases introduced above affect the reward/learning curve can be seen in fig. 8.2. Looking at the shape for the *Cus* agent, we see that the agent has a steep learning curve for about the first 25 updates. Considering that the agent has few successful grasps during this period, might one argue that the reaching reward is the main contributor to reward in this early learning phase. The learning tends to slow down when the agent reaches a mean reward between 80 and 110. Remembering that 88 is the max reward an agent can get from reaching during an episode, and 111 is the max it can get from reaching and grasping means that the agent is close to completing the reaching and grasping phase of the task. Looking at fig. 8.3 we can see that *Cus* agent has a success rate of 95% at around update 60, while it still only a reward of around 100. This means that the only phase left to learn is to hold the cube after lifting. Looking back at fig. 8.2 we can see that the learning starts to speed up again around update 70. This is when the agent learns to hold the cube after lifting. The learning curve starts to flatten out when it reaches a reward of around 150. It slowly grows to a max mean reward of 162, with a peak at 166.

This presentation of the learning phase is, of course, a simplification. There is a good chance that there are episodes during the early stages of the learning process where the agent holds the cube for some time. However, due to the clipping upon updating of the PPO policy are significant changes, both positive and negative, when updating clipped because of the noisy advantage function estimate. This is introduced in section 2.5.

8.2.3. Custom Camera with Joint Limits

The *Cus-JoLim* agent has all the same hyperparameters and training configurations as *Cus* except for a joint limit on joint 7 at $\pm 110°$ compared to the original $\pm 175°$ as shown in table 5.1.

The *Cus-JoLim* agent reaches a success rate of 100% although at a quite slower update rate then *Cus* as shown in both fig. 8.3 and table 8.1. It also have a quite slower learning curve compared to *Cus* as shown in fig. 8.2. One possible reason for this is the added complexity of the robot controller when it reaches the new joint limit. We experienced that when the robot reached its limit in joint 7 and still wanted to rotate, the other joints in the robot had to move to continue the rotation. This causes the position of the end effector to move and makes the environment harder to predict for the agent. A video of this is shown in appendix C.3.2 for the video called *Cus_JoLim_vid_phsical.mp4*. In other words, the environment is less predictable.

8.2.4. Calibrated Camera

Looking back to chapter 7 and the transformation matrix for the *Cus* - and *Cal* - agent, we see that there is not too much difference between the two matrices. Because of this, the two agents have very similar observations. Therefore should, a relatively identical learning and success response be expected. This hypothesis is confirmed by comparing the two agents in fig. 8.2 and fig. 8.3 where they perform almost identical. There are, of course, some small differences between the two agents. It is a slight tendency that *Cus* learns a bit faster, but the *Cal* gets a better final score. At update 130, is there a difference in reward of 10 in favor of the *Cus* agent. This is also a difference that could occur when running the same agent twice. So its hard to argue otherwise than that this is caused by differences in optimizations upon updates of the PPO algorithm. Nevertheless, it also shows how sensitive the training process is in general.

To conclude, if there are any essential differences in simulator training between *Cal* and *Cus* should the mean of multiple runs for these two agents have been compared. Because the goal of this thesis was to compare them on the sim-to-real transfer, was this not done.

8.2.5. Limiting the action space

By limiting the action space, the agent has a smaller space to optimize and thus an easier way to find the optimal actions in different states as discussed in section 3.4. This was the goal when training the *Cal-AcLim* agent. By looking at fig. 8.2 and fig. 8.3 we can see that the *Cal-AcLim* agent had the quickest learning curve by a good margin and also is the fastest to reach 100% success rate. This shows that limiting the action space is beneficial for this specific task.

However, it is worth mentioning that taking away a part of the action space might lead to a sub-optimal behavior for the agent, which means that it might encounter situations where the optimal solution for grasping would be to rotate the gripper around the z-axis (in yaw). This could be the case in cluttered environments, but considering that we only have one cube laying flat on the table, does this not seem to be the case here.

8.2.6. Training with Domain Randomization

Figure 8.6 shows the difference in mean episodic reward when the agent is trained with and without domain randomization. Although the domain randomization gets more data to train on per update, is there still a big difference in the final reward. This shows how complicating the environment makes it much harder to train, as discussed in section 2.4.7.

Looking at fig. 8.4 we can see that the agents trained with visual domain randomization also have a steep learning curve in the beginning but seem to struggle more to learn to hold the cube after a lift. We can see that the agents *Cal-AcLim-DR* and *Cus-DR* have a slight improvement over time while *Cal-DR* seem to flatten out after 100 steps and are no longer able to improve. Figure 8.4 shows that there are some differences in results between the custom and calibrated camera placement. Even the *Cal-AcLim-DR* performs worse than the custom camera *Cus*. It is worth mentioning that *Cal-AcLim-DR* still showed potential for improvement when stopped and, if given more training, possibly would have reached a higher reward. It is hard to tell if the difference in results comes from dissimilarities in RGB observations, suboptimal optimization steps, or something else. More tests should have been done to get to the bottom of this.

Not all the domain randomization agents trained reached a final success rate of 100% except for *Cus-DR*. *Cus-DR* scored a 100% success rate both during training and when tested outside of training as shown in table 8.2. *Cal-AcLim-DR* scored a top success rate of 95% during testing in training while only scoring 75% when tested outside. And lastly, *Cal-DR* scored 85% during training and 80% on testing outside training. The amount of training time per agent is different here, and it would have been desirable to have similar for a more accurate comparison.

8.2.7. Notes on results

The agents were not able to reach a max reward of 200, which makes sense. Remembering that fig. 8.2 and fig. 8.3 are plots of mean reward per collection of the rollout buffer means that all episodes during the collection have to be a perfect score of 200. This is not possible due to several reasons.

First of all, does the robot start some distance away from the cube, meaning it needs to take some agent steps to reach, grasp, and lift the cube. This means that some of the first steps per episode cannot get a maximum reward. Another factor is the physics of the interaction between the gripper and the cube. This leads to the gripper and robot sometimes pushing the cube outside of the image frame, making it impossible for the agent to locate the cube. An example of this interaction is shown in fig. 3.8.

For further comparison of results, should multiple runs of the same agent have been executed, where the mean of the runs would be compared against other agents.

Chapter 9.

Physical experiments

This chapter presents the results and discussion related to the zero-shot transfer from simulated training to physical testing. The agents used here score the highest average reward upon evaluation during training. The performed experiments are mainly used as a proof of concept for the system but also compare different DRL configurations.

9.1. Experiment design

Each trained agent was tested in the physical environment with 20 episodes. This was the same amount of episodes used during tests in training. Based on the performance during initial testing, we also considered the number of episodes sufficiently high to separate the agents from each other. We did not conduct further testing because of time limitations. Each episode was set to 200 steps, the same as the simulated training.

In simulation, the objects were placed with a random placement between 60 and 70 cm from the robot and with a 15 cm offset from the orthogonal line from the robot. In the physical tests, the box was placed within the same 60 to 70 cm range with a 10 cm offset from the orthogonal line. Figure A.2 in the appendix show the 9 placements for the cube. The cube was placed in each of these placements with a rotated and a nonrotated configuration. It was placed an extra time in the fig. A.2b and fig. A.2h to get a total of 20 configurations. These fixed initial positions were used instead of a random placements because of the difference in performance depending on the location of the box.

In simulation, the policy was rewarded when reaching toward the object. Furthermore, higher rewards were given when an object was grasped and lifted over the table. We have tried to find metrics that could be measured to reflect these rewards:

Bottom contacts:	When one or both bottom surfaces of the gripper fingers were in contact with the object.
Inside contacts:	When the inside of a gripper finger was in contact with the object.
Outside contacts:	When one or both of the gripper fingers were in contact with the object, but the interaction can not be classified as a bottom contact or a inside contact.
Grasp:	When the inside of both gripper fingers were in contact with the object.
Lift:	When the box was no longer in contact with the table.
Steps grasped:	Number of agent steps the box was in contact with both fingers during a grasp.
Reach:	When the object was in the center of the gripper for 6 agent steps or more. This corresponds with the end effector position in simulation fig. 3.9. A reach was given from visual evaluation.

A collection of images to illustrate the different conditions are placed in the Appendix fig. A.1.

These are binary metrics, meaning that an episode was either given the metric or not. The amount of contacts or grasps during an episode was not used in the comparison between the agents. These are recorded in the raw data, but were not presented as a part of the results in section 9.2. The metrics were chosen to be binary to make sure that policies that were able to reach towards the box rather than policies that had one episode with several contacts or grasps, was given a higher percentage in fig. 9.1.

The contact metrics was chosen because they were quantifiable without measurements. The contacts was separated into outside, bottom and inside contacts because they indicate different aspects of the problem. They all indicate that the agent was able to locate the box to some extent, despite not being able to grasp the object. Moreover, a inside contact indicated that the agent was only a grasp command away from grasping the object. A bottom contact indicated that the gripper was approaching the object from above, but an offset in the 2D plane prevented it from completing the grasp.

The reaching variable was measured in addition to these contact metrics to quantify if a agent was able to locate the object, but no contact was made. This was also a variable directly related to the reaching reward given in simulation.

The amount of grasps during episodes were also measured. This is used to quantify if the agent has learned the second phase of the lifting task, as described in section 8.2.

The amount of agent steps of a grasp and a lift variable was also counted to see if the policy was able to keep the grasp consequently have solved the third and last phase of the lifting task.

In addition to the metrics presented, the number of steps before a termination was recorded and the reason for the termination was noted.

9.2. Results

The empirical data from the test are summarized in table 9.1 and table 9.2. The raw data used as basis for the results are to be found in the digital appendix, C.3.3. Videos are provided as documentation for the experiments in the attachments. For each agent a video show the general performance during a random episode. These videos can be found in the digital appendix appendix C.3.2. The agents tested in the physical experiments are the same as presented and described in section 8.1.

Table 9.1.: Results from the physical tests where the main performance metrics are highlighted in grey. The numbers are given in percentage of 20 episodes.The raw data from the test are to be found in the digital appendix C.3.3.

Algorithm	Contacts	Outside contacts	Under contacts	Inside contacts	Reach	Number of grasps	Number of lifts	Average lift time
Cus	60 %	20 %	50 %	30 %	10 %	10 %	10 %	6
Cus-DR	60 %	45 %	5 %	40 %	30 %	25 %	15 %	2
Cus-JoLim	65 %	30 %	65 %	5 %	10 %	20 %	5 %	15
Cal	85 %	65 %	15 %	45 %	35 %	30 %	25 %	14
Cal-DR	75 %	45 %	30 %	15 %	5 %	0 %	0 %	0
Cal-AcLim	100 %	75 %	45 %	50 %	50 %	45 %	15 %	8
Cal-AcLim-DR	100 %	50 %	45 %	95 %	85 %	65 %	45 %	11

9.3. Discussion

This section discuss the results related to the physical experiments. During the comparison of the different agents the reaching, grasping and lifting phases were discussed. A description of these phases are presented in section 8.2.

The rating of the performance of the agents were mainly based of the reach, grasp and lift metrics. This is because they relate to the three phases of the picking task,

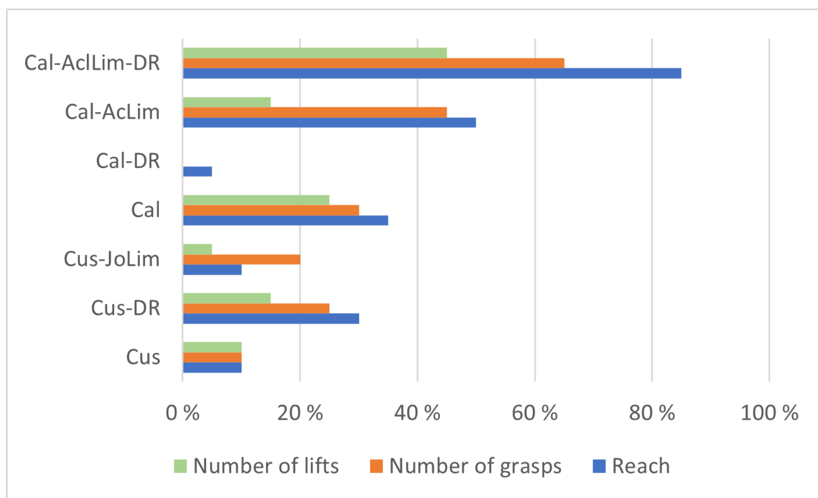


Figure 9.1.: Shows the performance of the trained agents based of the reach, grasp and lift parameters.

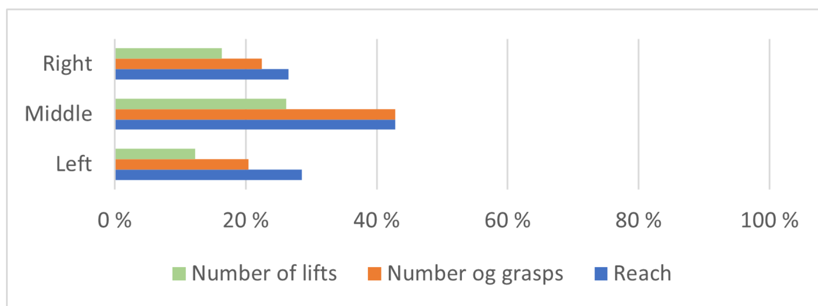


Figure 9.2.: Shows the performance of the trained agents for specific cube placement.

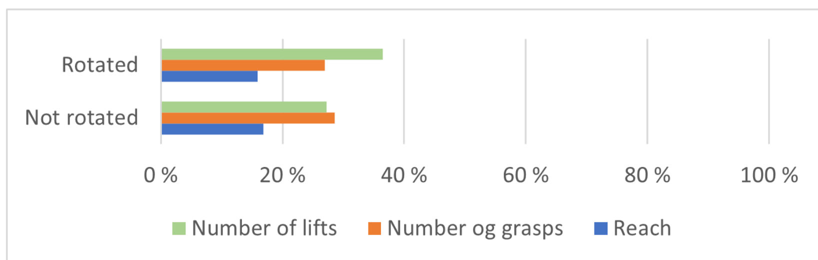


Figure 9.3.: Shows the performance of the trained agents for a specific cube orientations.

Table 9.2.: When and why the agents were terminated during the physical tests. The average termination is the average of the agent step where the system was terminated.

Algorithm	Average termination	Reason for termination
Cus	47	Joint lim 7
Cus-DR	20	Joint lim 7
Cus-JoLim	118	Joint lim 2, 4,5, 6
Cal	22	Joint 7
Cal-DR	39	Joint 7
Cal-AcLim	167	Friction collision or object contact while being closed
Cal-AcLim-DR	136	Friction collision

and thereby describe if the agents has solved these phases or not. The contact metrics are quantified to get a broader picture of how the agent performs.

9.3.1. Joint limits

One of the main challenges when testing the trained agents in a physical environment was the joint limits for the robot. Even though the joint limit for joint 7 was set to ± 175 degrees 5.1, we had problems moving the joint further than ± 110 degrees. The safety configuration uploaded to the Sunrise Controller threw the error message "ESM state violated" if the limit was exceeded. The safety configuration was set ahead of the testing and was not changed due to restrictions from the supervisor. This problem could be solved if the safety configuration was changed. However, if the safety configuration was changed, the robot could still move to the joint limits. Another downside could be the stretching of the gripper cable because of the gripper rotation.

The simulation had less restrictive limits for joint 7, and when a joint position above 110 degrees was sent from simulation to the physical robot during sim-to-real transfer the system came to a halt. The table 9.2 shows how several of the trained policies had an average termination under 50 steps. Most of these terminations was because of restrictive joint limit in the physical setup. It was clear that this prevented the policy from getting better results.

Different solutions were tested to solve this challenge. One of them was to limit the joint 7 movement in simulation during training. The policy could then learn to solve the task and move the robot with these restrictions. Table 9.2 show that the *Cus-JoLim* agent eliminated the problem of errors in joint 7 and reached a higher average termination step of 118 compared to *Cus*s' 47. However, *Cus-JoLim* did

not improve on overall performance compared to *Cus*, as shown in fig. 9.1, even though it got more agent steps to solve the task. It scored higher on the number of grasps, but the number of lifts was lower. The numbers from table 9.1 shows that the policy was about even in having contacts with the object.

Another problem with the *Cus-JoLim* was that it introduced several other joint errors by moving them to their respective limits. Joint 2, 4, 5 and 6 all reached their limits at some point during testing 9.2, and the system came to a halt. From observations illustrated in appendix C.3.2 *Cus_JoLim_vid* did we see that when the limit of joint 7 was reached, the other joints moved to rotate the gripper. This made the system less predictable and consequently made it harder for the agent to maneuver the robot. After reaching joint 7 limit during an episode, it never come close to the cube.

Cus-JoLim is the only agent with a higher grasp than reach percentage table 9.1. This implies that the agent is quick in grasping when it gets close to the cube. We also see that it has the highest average lift time as shown in table 9.1. These observations can be used to argue that the agent has a high performance in the grasping and lifting phase of the picking task. It seems like the low performance is due to poor execution in the reaching phase when it goes from sim-to-real.

Another way of dealing with joint errors was to limit the action space. Using action space in Cartesian coordinates where only x, y and z position changes are given to the robot would drastically restrict the movement of joint 7. This is what is done with *Cal-AcLim* and *Cal-AcLim-DR*. These agents eliminated the problem of joint errors and have the highest average termination as shown in table 9.2. Further presentation of the results and challenges with limiting the action space is presented in section 9.3.4.

9.3.2. Calibrated camera

A hypothesis was that a calibrated camera would outperform a camera placed incorrectly on the sim-to-real transfer. The pixels could correspond more accurately with a better match between the image observation in simulation and the physical system. This could, in turn, help the policy better recognize patterns in the observations.

According to our results the *Cal* agent show a better performance compared to *Cus* agent. It scored better on all the three main measuring metrics, reach, grasp, and lift, by over doubling the success percentage of the *Cus* agent as shown in fig. 9.1. These results are quite reliable, considering that these two agents have almost identical performing results in the simulator, as shown in fig. 8.3 and table 8.2, and that the *Cal* agent has half the number of agent steps before termination

compared to the *Cus*, agent as shown in table 9.2. This means that the *Cal* agent has less time to solve the task than the *Cus* agent and is still able to outperform him. Therefore, it seems like camera calibration shows promising results in helping close the sim-to-real gap.

Despite the promising results of the calibrated camera, does the *Cus-DR* agent outperform the *Cal-DR* agent by a good margin. The *Cal-DR* agent has 0% in both grasp and lifting for the phyiscal testing as shown in fig. 9.1. The reason for this can be a problem with the joint errors and the poor training performance of the *Cal-DR* in simulation. By comparing table 8.2 and table 9.2 we see that the average termination in the physical world for the *Cal-DR* is lower with a number of 39 agent steps compared to the average agent step for the first successful grasp in simulation which is about 64. This is a 25 agent-step difference. This implies that the agent does not have enough time in the physical episode to get a successful grasp because it receives a joint error before it usually would grasp.

It should be mentioned that the *Cus-DR* agent suffers from this same problem but to a lower degree. The difference between the average first successful grasp and the average physical termination step is 8 agent-steps. The number of 8 agent steps seems to be small enough such that the *Cus-DR* do not suffer too much in terms of performance. However 25 agent steps in difference for *Cal-DR*, seem to be too big of a difference and therefore cause a drastic decrease in performance.

It is also worth mentioning that *Cal-DR* are unable to reach satisfactory results in simulation. As seen in fig. 8.4, the reward is the lowest recorded from all the tested agents, and it also only reached a max success rate of 85% as shown in table 8.1. This suggests that this agent has not sufficiently learned to reach and lift the cube. The poor results for the *Cal-DR* agent might be interpreted as a training problem due to the complexity of domain randomization rather than a problem caused by the camera placement.

9.3.3. Domain randomization

Domain randomization for camera position was implemented to make sure that the agent had the physical camera placement within its domain as illustrated in fig. 2.5. The camera was uniformly placed in a 1 cm offset in every direction during training. The rotation was offset by 0.01 rad = 0.57 degrees.

From the residuals of the hand-eye calibration shown in section 7.1.2, we see that the highest translation residual was 0.67, which is within the domain randomization of 1 cm. We also see that the highest residual for rotation is 0.21 degrees which is within the domain randomization of 0.57 degrees. In theory, the agent has been trained with the exact Zivid placement in its domain.

We see from the results of the hand-eye calibration that the uncalibrated camera is offset by $(x, y, z) = (2.74, 1.94, 0.19)$. From the residuals we find that the minimum offset is $(x, y, z) = (-2.07, -1.27, 0.0)$. This is not within the domain randomization of 1 cm. From our calculations, the offset for the uncalibrated transformation matrix related to the calibrated transformation matrix is 0.738 degrees. When subtracting the residual, we get a minimum offset of 0.528 degrees. This is within the domain randomization of 0.57 degrees. This means that the uncalibrated versions do not have the exact physical camera placement within their domain.

The results from implementing domain randomization show an increase in performance on all three performance metrics of the task for two out of three agents. This is true for both *Cus-DR* compared to *Cus* and *Cal-AcLim-DR* compared to *Cal-AcLim* as shown in fig. 9.1. The *Cal-DR* however, scores much worse than its *Cal* counterpart. Why this could be is discussed in section 9.3.2. In short, we think the reason for this may be joint error problems and poor training results rather than the effect of domain randomization itself.

Both *Cus-DR* and *Cal-AcLim-DR* had a lower average termination than their non-randomized counterparts, as shown in table 9.2. They also show worse results in simulation, especially *Cal-AcLim-DR* which only scores a success rate of 75% compared to *Cal-AcLim*'s 100% when tested outside of training. This is shown in table 8.2. Regardless of this, *Cus-DR* and *Cal-AcLim-DR* still score better on the sim-to-real transfer showing that domain randomization is a promising tool for closing the reality gap. It is worth mentioning that *Cal-AcLim-DR* still showed potential for further improvements before it was shut down in simulation, as shown in fig. 8.5. If it were given more time could likely event better results be expected.

The agents trained with domain randomization had more data to train on, meaning more data to optimize upon. This is, of course, beneficial, but considering that the agents trained without domain randomization already scored a 100% success rate on the smaller amount of data, would more data during training probably make no difference in the final result.

To summarize, we argue that the results for the *Cal-DR* were due to other problems than the specific domain randomization. Furthermore, we argue that there are indications that the domain randomization used in this thesis helps in closing the reality gap.

9.3.4. Limiting the action space

Based on our results, limiting the action space outperforms the other policies, as seen in fig. 9.1. The *Cal-AcLim* agent scores 50% on the reach metric compared to the 35% of the *Cal* agent. It also shows a good score on number of grasps with 45% compared to 30% for the *Cal* agent. It, however, is outperformed by the *Cal* agent on the number of lifts by 25% compared to its own 15%. As we can see from both table 9.1 and fig. 9.1, the *Cal-AcLim* agent is the agent with the highest difference percentage of successful grasps compared to successful lifts. A possible reason for this is discussed further down.

The reason why limiting the action space is beneficial is discussed in section 3.4. It is worth mentioning that the agents trained with an action limit have the highest number of steps before termination and, therefore, a longer period to solve the task, as shown in table 9.2. Because of this, is it hard to tell if the better performance comes from a more suitable configuration or just more time to solve the task compared to the *Cal* agent. The reason for the better performance may be a combination of both. One might argue that limiting the action space makes the task simpler and requires fewer decisions during training.

There can be visual arguments for why the limited action space is a better technique. Firstly, the cube is more visible during the reach phase if the gripper is over the cube. This is because the fingers of the gripper are on each side of the cube, regardless of the configuration. The agents trained without action limitation, however, might rotate the gripper in yaw (about the z-axis) and therefore occlude the visibility of the cube. Secondly, the image of a cube between the two fingers can also be a more recognizable pattern for the policy, with a red cube between two black gripper fingers. Finally, an agent with limited action space also has fewer robot configurations, making it easier to "remember" the states for the policy during training.

We see from table 9.2 that the agents with a limited action space had a high average termination. This was because they eliminated the joint 7 error as discussed in section 9.3.1. They, however, introduced two new collision errors regarding friction and physical contact between the gripper and the cube. The first of them is *friction collision*. This error occurred when the gripper had grasped the cube, and the agent gave an action to go further down in the z-direction. Because the friction between the gripper and the cube was higher than the KUKA safety configuration allowed, the robot reported a collision error. The second error came when the gripper made contact with the cube when it had a closed configuration or when *bottom contacts* were made. Examples of these last errors are shown in fig. 6.3.

The *friction collision* can be solved by reducing the friction between the gripper

and the object. The pinching force was set to the minimum force of 20 N, so this cannot be reduced further. A possible solution could be to change the rubber surface of the gripper fingers, consequently reducing the friction coefficient. However, this could reduce the probability of successful grasps due to objects slipping out of the fingers. It was discovered that a possible reason for the *friction collision* could be the re-grasp function of the Robotiq gripper [66]. This function allows the gripper to apply a higher pinching force if it detects movement of the object that is grasping. This re-grasp increases the force in which the object is grasped, which again leads to more friction between gripper and cube. By removing this function, friction collisions may be avoided.

The collision errors presented in fig. 6.3, is a problem caused by the gripper's design in regards to the object wanted to grasp. The error shown in fig. 6.3a is an internal collision error in the gripper, while collision with a closed gripper is caused by sharp edges on the outside of the gripper fingers, as seen in fig. 6.3b.

The reason for the big difference in the percentage of successful grasps compared to successful lifts can be somewhat explained by the *friction collision* error. Because of this, there were several occasions when the gripper had grasped the cube but didn't manage to lift it. Fixing the *friction collision* error would most likely have increased the lift percentage.

9.3.5. Placement and orientation of the cube

During testing, the agents performance for different cube placements and orientations was recorded. The placement of the cube and the orientation is described in section 9.1.

Looking at fig. 9.2 which shows the average performance of all the agents for the cubes' left, right, and middle placement, can we see an equal performance for the left side compared to the right side. This shows that the agent did not have a preferred side when grasping the objects and that the randomization in the cube placement during training made the agent equally good on each side. However, the performance in the middle shows better results on all three performance metrics compared to the two outer sides. A possible reason for this is that the robot starts closer to the cube with its initial configuration compared to the cases where the cube is placed on the outer sides. It, therefore, has a more straightforward and shorter route to the reaching phase of the task. Consequently, the agent is more likely to score higher on all three performance metrics.

For results on the agent's performance on a cube rotation compared to no rotation, can we look at fig. 9.3. Here we can see that the agent scores about equal in the two cases. A possible reason for this is that the agent has learned to locate the

cubes red color and therefore does not give too much attention to the rotation configuration of the cube. It is also worth mentioning that the RGB images used as observation have a low resolution of 84x84. It is, therefore, hard to differentiate the cube's rotation, even for the human eye.

9.3.6. Sources of error

The difference in the measurements of performance in the physical environment compared to the simulation can be problematic. The results are not directly comparable because the reach variable is not measured but is instead a visual observation. A lift in simulation is precisely defined, but we do not have the exact position of the cube available during the physical tests.

The agents perform differently during training in simulation. Figure 8.6 shows how all the agents trained with domain randomization have a lower performance during training than those trained without. If parameters and network structure were improved to better the domain randomization performance, could the comparison of the different agents have been more reliable. However, if an agent performing worse in the simulator shows better results on the sim-to-real transfer, is it still a strong indication that the configurations for this agent works.

The initial positioning of the robot sometimes made it harder for the robot to grasp the cube when placed in the middle fig. A.2e for some episodes over others. This is because the gripper sometimes started directly above the cube, while other times, one of the fingers started above. This might lead to some agents having less ideal initial positions, which could affect the agents' comparison. Later tests could have been run with a fixed initial position to eliminate this. This would prevent some agents from getting an advantageous initial position. The initial position of the end-effector could also be higher to prevent a grasp from being only a few agent-steps away in some initial configurations.

It could be beneficial to record other metrics to verify the performance of the policies. The number of agent-steps every *reach* lasts could be interesting to discuss. A long-lasting *reach* suggests that the policy is great at the first phase of picking the object. In addition, the number of agent-steps the gripper is closed and open during an episode can help differentiate what state the policy thinks it is in. If a policy never has a closed gripper, it may be better at the first phase of the grasping process. If the policy always has the gripper closed, it may assume that it has grasped the cube and is collecting a reward for a grasp when in reality, the cube is still lying on the table.

The test from this experiment requires a higher number of episodes tested to get more scientifically satisfactory data. We have chosen a few tests of many agents

rather than multiple tests of a few agents.

The cube slipped from the gripper's fingers when the agent tried to grasp the cube on adjacent sides. From testing, we saw that the agent could perform these types of grapes in simulation. A better simulation imitating this behavior or new physical gripper fingers to handle these types of grasps could be implemented to better the dynamic differences between the simulation and the physical environment.

Part IV.

Discussion and Conclusion

Chapter 10.

Discussion

This section discusses some overall topics regarding this thesis and highlights some choices and obstacles encountered along the way.

10.1. Zivid Two

Only 2D images are used in training and physical testing in this project. The 3D image abilities have not been implemented in the system, and the quality has not been tested. The Zivid camera was mainly chosen to make a communications system that can be used in further research. We have tested the communication and used the data from the camera in our testing. If other researchers decide to exploit the high-quality point clouds, steps towards an implementation are already taken.

An argument for the use of a 3D camera is the possibility to perform a high-quality hand-eye-calibration. We have used this, which has helped us achieve better results on the sim-to-real transfer. A hand-eye-calibrated camera is also necessary if it is a need for a pixel to world coordinates conversion. This is the case if object detection is used during training. The use of object detection was not used in this thesis, but our hand-eye-calibrated solution allows for training regimes that take advantage of this. This could be interesting to test to see if it outperforms our current solution.

The quality of the 2D images from the Zivid camera was not ideal. The projector used during the structured light capturing process made a light frame in the image that was hard to find a good solution to mimic in the simulations. In addition, the projector cast light directly on the robot, which made for a strong specular reflection on the robot. This reflection resulted in some very bright regions on the robot, which led to considerable differences in the color of the robot in the simulator compared to the physical world.

The Zivid intrinsic parameters were approximated in simulation to mimic the physical camera. This is a simplification of the Zivid camera. The Zivid camera has a high degree of complexity with a specific focus distance that is hard to mimic in simulation. If the 3D images are to be used with the Zivid camera, work must be done to implement a satisfactory simulated Zivid camera.

10.2. Sim-to-real solution

With our sim-to-real solution introduced in chapter 6 is it possible to transfer a policy trained with one controller over to the physical world where another controller is used. This is the case if the right inputs to the physical controller are in place in the simulated environment. The benefit of this solution is that it removes the need to build a similar controller in either simulation or the physical world that is identical to the other. This can be a tedious and error-prone procedure. Another benefit of our sim-to-real solution is that the controller allowing for best learning performance can be used in simulation, while the controller allowing for best safety can be used in the physical world. This allows for a better learning process as well as a safer sim-to-real transfer.

10.3. Errors during physical testing

During physical testing did, several joint- , friction-, and collision- errors occur. This occluded the comparison of the performance of the different agents. Ideally should, all agents compared against each other have equally many agent-steps. This excludes the possibility that a better performing agent is caused by more agent-steps rather than its configuration. This might be the reason for some of our results, especially when comparing a limited action space against other agents.

10.4. Robotiq 2F-85

The interaction between the gripper and the objects is an essential part of grasping. With mismatches in the interaction between both table and objects, it is hard for an agent to learn how to act in a physical environment after training in simulation. We see this as one of the most significant drawbacks of our system. As mentioned earlier, the fingertips of the gripper were made longer to make up for a difference in the total height of the gripper, but it was not made an effort to improve the models from Robosuite further.

A new and updated model should be implemented with an underactuated structure to improve the gripper. It must also be done an effort to improve the sim-

ilarity of the friction in simulation and the physical environment. The re-grasp function of the physical gripper should be disabled to prevent *friction collision* error between the object and the table during a grasp. Another solution is to change the gripper used to one that does not cause this error.

10.5. General discussion

This thesis argues for the performance of different sim-to-real transfer techniques. However, due to errors upon physical testing is it likely that some of the results fall short. Nevertheless, can the thesis be considered as a proof of concept to see if the frameworks and implementations made can make a viable setup for sim-to-real transfer.

Considering that our best agent reaches a success rate of 45% on the same object trained upon during training, does it show that it still is a long step before a zero-shot transferred agent can be used in everyday grasping tasks. In [34] that uses sim-to-real via sim-to-sim technique score a 70% zero-shot grasp success on unseen objects. This is a solid result but shows that even the state-of-the-art techniques out there have some more improvement to make to get to a full 100% success rate. It should be mentioned that by finetuning in the real world with only 5000 real-world grasps, did their method achieved a 91% success rate. So the field shows some very promising results.

One of the downsides of using the A100 80 GB GPU to train agents together with Stable Baselines is the allocation of memory on the MANULAB computer when loading the trained agent. The loading function required an allocation of 80 GB to be able to load the agent. This was not possible on the MANULAB computer due to missing computer space. The solution was to instantiates an array full of zeros when allocating the 80 GB memory. The NumPy array of zeros did not use the whole memory needed for the array, only the non-zero elements. Since we only used one agent when running on the physical setup instead of the 64 agents in simulator, we never actually use the 80 GB allocated and therefore did not encounter any problems with this solution.

Upon early stages of the development of the training phase, MuJoCo returned errors if the configuration of the environment was physically impossible to calculate. This resulted in the training being shut down and was an obstacle on trying to optimize the agent further. To solve this was a *try* and *exception* code snipped made in Robosuite in the *base.py* file in the environment folder. The code is shown in appendix F. If MuJoCo gave the same error, an exception would be made, which allowed for further training. The downside of this is that a copy of the previous tuple would be gathered in the rollout buffer instead of the tuple

leading to the error. This means that the same tuple was gathered twice in a row in the rollout buffer. The error occurred very rarely, at most once or twice between each update. This means that only 2 or 4 out of the over 130 000 tuples gathered before an update section 4.2.1 were duplicated. We argue that this is sufficiently small amount to make any difference during training.

All agents used in the sim-to-real transfer were tested through 20 episodes. This low number of repetitions limits the possibility of comparing the agents with high confidence. We also found that more data collected through the experiments would be helpful in the comparison of agents. The number of agent-steps with open/closed gripper would be easy to record and could help us in comparing which agents that were restrictive in closing their gripper and not. The trajectory of the agents throughout the episodes would also be interesting to compare because of the different approaches the agents had. These different approaches is not possible to understand from the data we collected.

During testing, some agents moved within a small range of robotic configuration. It seemed like the robot had learned what range to be in. A test was conducted to see how the robot would react if it had a black image as the observation. We saw from several tests that the robot started drifting to the left when given a black image, and no contacts were made. This implies that the images was used to make choices in what actions to take.

Chapter 11.

Conclusion and Further Work

11.1. Conclusion

The main objective of this thesis has been to test the sim-to-real transfer of a trained Deep Reinforcement Learning (DRL) agent, with the goal of controlling a robot arm to grasp an object. To do this, we divided our task into three parts. Firstly, build a realistic simulator environment for the desired task. Secondly, train the agent to a satisfactory behavior and finally implement the physical setup to be ready for the sim-to-real transfer.

Upon building a realistic simulated environment, a model of KUKA LBR iiwa 14 R820 has been implemented. The light conditions, the texture of materials, dimensions, and physical parameters of objects in the physical lab have been imitated in an attempt to narrow the reality gap. The ability to place the camera according to a transformation matrix has been implemented.

In the training implementation, a Singularity Image File (SIF) file has been made to make it possible to train on the Idun HPC cluster. A wrapper that enables training with RGB and proprioceptive observations and the ability to limit the action space has been implemented. Domain randomization has been used in an effort to narrow the reality gap. Reward shaping has been used to speed up learning. The observation space was chosen based on state-of-the-art papers on the DRL field.

To implement the physical setup Robot Operating System 2 (ROS2) communication has been developed to make the hardware communicate with each other. A solution where the simulation runs parallel to the physical environment is used to enable the sim-to-real transfer. This solution allowed for a sim-to-real transfer with two different robot controllers.

Experiments were conducted as a proof of concept for the training structure and

communication and in an effort to distinguish between different sim-to-real transfer solutions. Different agents have been trained and tested where all except for one (*Cal-DR*) had successful lifts on the physical environment. The most successful agent was trained with domain randomization, a limited action space and a calibrated camera. The agent was tested 20 times in a physical environment and got a 45% lift success rate. The experiments conducted in this suggest that calibrating the camera, limiting the action space, and using domain randomization help in closing the reality gap. However, due to termination errors upon physical testing and a small number of physical runs per agent, does our results suffer from a different number of agent-steps to solve the task per agent and too little data to give a strong conclusion of our results. These two last arguments weaken our conclusion regarding configurations that help solve the reality gap.

Regardless, our results imply that the sim-to-real techniques used in this thesis is successful in a sim-to-real transfer and can be used for further research in limiting the sim-to-real gap.

11.2. Further work

In this section, suggestions for improving the simulation, training and sim-to-real transfer are presented.

11.2.1. Further work in simulation

One of the main drawbacks of the simulated environment used in this thesis was the gripper model in Robosuite. The simulated gripper had loose joints and dynamic differences compared to the physical gripper. These differences may have interfered with the agent performance in the sim-to-real transfer. An updated model of the gripper parts and the under-actuated structure should be implemented to improve the similarity between the simulated and physical environment as discussed in section 10.4.

Further work should be done to make the interaction between the gripper and cube more similar in simulation and the physical world. Some tests where done, but tests comparing grasping of the cube on two adjacent sides in simulator and the physical world should be made. While the robot could grasp the cube in the adjacent configuration in the simulator, did the cube slip from the fingers in the physical world. Combining the improved physical parameters with dynamic domain randomization on the cube friction and mass could further reduce this problem.

11.2.2. Further work in training

There are many different tools to limit the reality gap, as discussed in section 2.4. Domain adaptation and data augmentation could be tested to limit the gap further. [99] introduce interesting solutions on the topic.

The physical robot could crash in the table, which resulted in a collision error when testing the sim-to-real transfer on trained agents. A limit in the z-direction was put on the robot during the simulation to prevent this collision error. Another solution to prevent this collision error could have been to further develop the reward shaping by giving a negative reward for contact with the table. This would have enabled the robot to move in the entire operation space. It is not certain that this would have given a better performance considering that our z limit is almost as close to the table as possible without crashing, meaning there is not much more space for the robot to operate in when removing this.

An environment consisting of multiple objects was made during this thesis as an effort to tackle cluttered environments [9]. Because we only wanted proof of concept of our solution and a single object was sufficient enough for our wanted experiments was this but on ice. A natural the next step in this thesis, together with closing the reality gap, is to look at grasping of several objects. Robosuite has support for this, and there are multiple different object-libraries out there with objects that can be placed in the simulation simultaneously.

The agents trained with domain randomization performed worse than the agents trained without. This is expected considering that the environment is less familiar for the agent. The agents trained with domain randomization were given more data to counteract this problem. Some of the agents perform worse than their non-randomized counterparts when tested in simulator. Further work should be done to make the final results of the agents more identical in the simulator. A solution suggested by [27] is to make the network deeper.

The observation space used in this thesis is chosen due to strong results in other relevant papers. However, no in-depth tests were done to measure which observations in our observation space led to the best performance. Further testing on which observations best contribute to better performance would have been interesting.

Further testing for training parameters, network structure, and domain randomization parameters should be tested to improve the agent's performance.

11.2.3. Further work in physical environment

The physical communication, hardware and camera placement support 3D images through either point clouds or depth images. In addition, both Robosuite and Stable Baselines support the use of 3D data as observations. In later iterations of the proposed system, the 3D data can be used as observations for the agent. A challenge could be to mimic the Zivid intrinsic parameters in Robosuite to get similar data from the simulated and physical environment.

Termination due to joint limits was problematic during the testing of the agents. The safety configuration should be updated to make a more robust test setup in the physical environment. To further improve the physical testing, measures should be implemented in the simulated training to prevent the robot from reaching the joint limits. A possibility would be to limit the delta value that tells the agent which angle the simulated end-effector could rotate for every agent step. The size of the delta steps taken now are in section 3.6.1.

A zero-shot transfer to a physical environment can be challenging. By using physical training as a way of updating the agent on real-world data, the transfer can reach a higher performance [34]. To do this, a viable closed-loop training setup must be constructed. It would be beneficial to improve the system's speed so that more agent steps can be done in a shorter time. In addition, measures could be taken to let the environment reset itself and train over several episodes.

Even though parts of the physical system were mimicked in the simulation, it can be further improved. The lighting condition could be improved to mimic the uniform lighting in simulation. The cable from the Robotiq gripper can be removed by making use of the power and communication available through the media flange. The gripper fingers can be changed to mimic the simulated gripper's friction to prevent the *friction collision* error while grasping the object. A part of the solution could be to disable the re-grasp function of the gripper, as further discussed in section 10.4 and section 9.3.4.

References

[1] URL: https://github.com/bulletphysics/bullet3/issues (visited on 12/14/2021).

[2] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. "Hindsight Experience Replay". In: *CoRR* abs/1707.01495 (2017). arXiv: 1707.01495. URL: http://arxiv.org/abs/1707.01495.

[3] Marc G. Bellemare, Will Dabney, and Rémi Munos. "A Distributional Perspective on Reinforcement Learning". In: *CoRR* abs/1707.06887 (2017). arXiv: 1707.06887. URL: http://arxiv.org/abs/1707.06887.

[4] Konstantinos Bousmalis, Alex Irpan, Paul Wohlhart, Yunfei Bai, Matthew Kelcey, Mrinal Kalakrishnan, Laura Downs, Julian Ibarz, Peter Pastor, Kurt Konolige, Sergey Levine, and Vincent Vanhoucke. "Using Simulation and Domain Adaptation to Improve Efficiency of Deep Robotic Grasping". In: *CoRR* abs/1709.07857 (2017). arXiv: 1709.07857. URL: http://arxiv.org/abs/1709.07857.

[5] Michel Breyer, Fadri Furrer, Tonci Novkovic, Roland Siegwart, and Juan Nieto. "Flexible Robotic Grasping with Sim-to-Real Transfer based Reinforcement Learning". In: (Mar. 2018).

[6] Michel Breyer, Fadri Furrer, Tonci Novkovic, Roland Siegwart, and Juan Nieto. "Flexible Robotic Grasping with Sim-to-Real Transfer based Reinforcement Learning". In: (Mar. 2018).

[7] Dian Chen, Brady Zhou, Vladlen Koltun, and Philipp Krähenbühl. "Learning by Cheating". In: *CoRR* abs/1912.12294 (2019). arXiv: 1912.12294. URL: http://arxiv.org/abs/1912.12294.

[8] Paul F. Christiano, Zain Shah, Igor Mordatch, Jonas Schneider, Trevor Blackwell, Joshua Tobin, Pieter Abbeel, and Wojciech Zaremba. "Transfer from Simulation to Real World through Learning Deep Inverse Dynamics Model". In: *CoRR* abs/1610.03518 (2016). arXiv: 1610.03518. URL: http://arxiv.org/abs/1610.03518.

[9] *Cluttered environments.* URL: https://github.com/ojrise/Robot_Learning_master/blob/main/code/src/environments/kuka_env.py.

[10] *Controllers.* 2022. URL: https://robosuite.ai/docs/modules/controllers.html.

[11] Erwin Coumans and Yunfei Bai. *PyBullet, a Python module for physics simulation for games, robotics and machine learning.* http://pybullet.org. 2016–2019.

[12] *CS231n Convolutional Neural Networks for Visual Recognition.* https://cs231n.github.io/convolutional-networks/. (Accessed on 05/15/2022).

[13] *Dart sim.* URL: https://dartsim.github.io/ (visited on 12/14/2021).

[14] *Default hyperparameter values for ppo algorithm.* URL: https://github.com/DLR-RM/stable-baselines3/blob/378d197b00938579a6a5e04c739f41fec23fd8 stable_baselines3/ppo/ppo.py#L68.

[15] Tom Erez, Yuval Tassa, and Emanuel Todorov. "Simulation tools for model-based robotics: Comparison of Bullet, Havok, MuJoCo, ODE and PhysX". In: *2015 IEEE International Conference on Robotics and Automation (ICRA).* 2015, pp. 4397–4404. DOI: 10.1109/ICRA.2015.7139807.

[16] Linxi Fan, Yuke Zhu, Jiren Zhu, Zihua Liu, Orien Zeng, Anchit Gupta, Joan Creus-Costa, Silvio Savarese, and Li Fei-Fei. "SURREAL: Open-Source Reinforcement Learning Framework and Robot Manipulation Benchmark". In: *Conference on Robot Learning.* 2018.

[17] D. Ferigo, S. Traversaro, G. Metta, and D. Pucci. "Gym-Ignition: Reproducible Robotic Simulations for Reinforcement Learning". In: *2020 IEEE/SICE International Symposium on System Integration (SII).* 2020, pp. 885–890. DOI: 10.1109/SII46433.2020.9025951.

[18] Diego Ferigo, Silvio Traversaro, and Daniele Pucci. "Gym-Ignition: Reproducible Robotic Simulations for Reinforcement Learning". In: *CoRR* abs/1911.01715 (2019). arXiv: 1911.01715. URL: http://arxiv.org/abs/1911.01715.

[19] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau. "An Introduction to Deep Reinforcement Learning". In: *CoRR* abs/1811.12560 (2018). arXiv: 1811.12560. URL: http://arxiv.org/abs/1811.12560.

[20] *Gazebo simulator.* URL: http://gazebosim.org/ (visited on 12/14/2021).

[21] Shixiang Gu, Timothy P. Lillicrap, Zoubin Ghahramani, Richard E. Turner, Bernhard Schölkopf, and Sergey Levine. "Interpolated Policy Gradient: Merging On-Policy and Off-Policy Gradient Estimation for Deep Reinforcement Learning". In: *CoRR* abs/1706.00387 (2017). arXiv: 1706.00387. URL: http://arxiv.org/abs/1706.00387.

[22] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor". In: *CoRR* abs/1801.01290 (2018). arXiv: 1801.01290. URL: http://arxiv.org/abs/1801.01290.

[23] *Hand-Eye Calibration Process — ZIVID KNOWLEDGE BASE documentation*. https://support.zivid.com/latest/academy/applications/hand-eye/hand-eye-calibration-process.html. (Accessed on 03/21/2022).

[24] Todd Hester, Matej Vecerık, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Andrew Sendonaris, Gabriel Dulac-Arnold, Ian Osband, John P. Agapiou, Joel Z. Leibo, and Audrunas Gruslys. "Learning from Demonstrations for Real World Reinforcement Learning". In: *CoRR* abs/1704.03732 (2017). arXiv: 1704.03732. URL: http://arxiv.org/abs/1704.03732.

[25] *How To Get Good Quality Data On Zivid Calibration Board — ZIVID KNOWLEDGE BASE documentation*. https://support.zivid.com/latest/academy/applications/hand-eye/how-to-get-good-quality-data-on-zivid-calibration-board.html. (Accessed on 06/08/2022).

[26] Wenbin Hu, Chuanyu Yang, Kai Yuan, and Zhibin Li. "Reaching, Grasping and Re-grasping: Learning Fine Coordinated Motor Skills". In: *CoRR* abs/2002.04498 (2020). arXiv: 2002.04498. URL: https://arxiv.org/abs/2002.04498.

[27] Julian Ibarz, Jie Tan, Chelsea Finn, Mrinal Kalakrishnan, Peter Pastor, and Sergey Levine. "How to train your robot with deep reinforcement learning: lessons we have learned". In: *The International Journal of Robotics Research* 40.4-5 (Jan. 2021), pp. 698–721. DOI: 10.1177/0278364920987859. URL: https://doi.org/10.1177%2F0278364920987859.

[28] *Idun – High Performance Computing Group*. https://www.hpc.ntnu.no/idun/. (Accessed on 02/21/2022).

[29] Robotiq inc. *Robotiq 2F-85 & 2F-140*. English. Robotiq inc. 158 pp.

[30] *Isaac Gym*. URL: https://developer.nvidia.com/isaac-gym.

[31] *Isaac simulator*. URL: https://developer.nvidia.com/isaac-sdk.

[32] Stephen James, Andrew J. Davison, and Edward Johns. "Transferring End-to-End Visuomotor Control from Simulation to Real World for a Multi-Stage Task". In: *Proceedings of the 1st Annual Conference on Robot Learning*. Ed. by Sergey Levine, Vincent Vanhoucke, and Ken Goldberg. Vol. 78. Proceedings of Machine Learning Research. PMLR, 13–15 Nov 2017, pp. 334–343. URL: https://proceedings.mlr.press/v78/james17a.html.

[33] Stephen James, Paul Wohlhart, Mrinal Kalakrishnan, Dmitry Kalashnikov, Alex Irpan, Julian Ibarz, Sergey Levine, Raia Hadsell, and Konstantinos Bousmalis. "Sim-to-Real via Sim-to-Sim: Data-efficient Robotic Grasping via Randomized-to-Canonical Adaptation Networks". In: *CoRR* abs/1812.07252 (2018). arXiv: 1812.07252. URL: http://arxiv.org/abs/1812.07252.

[34] Stephen James, Paul Wohlhart, Mrinal Kalakrishnan, Dmitry Kalashnikov, Alex Irpan, Julian Ibarz, Sergey Levine, Raia Hadsell, and Konstantinos Bousmalis. "Sim-to-Real via Sim-to-Sim: Data-efficient Robotic Grasping via Randomized-to-Canonical Adaptation Networks". In: *CoRR* abs/1812.07252 (2018). arXiv: 1812.07252. URL: http://arxiv.org/abs/1812.07252.

[35] Scott Jordan, Yash Chandak, Daniel Cohen, Mengxue Zhang, and Philip Thomas. "Evaluating the Performance of Reinforcement Learning Algorithms". In: *Proceedings of the 37th International Conference on Machine Learning*. Ed. by Hal Daumé III and Aarti Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, 13–18 Jul 2020, pp. 4962–4973. URL: https://proceedings.mlr.press/v119/jordan20a.html.

[36] Dmitry Kalashnikov, Alex Irpan, Peter Pastor, Julian Ibarz, Alexander Herzog, Eric Jang, Deirdre Quillen, Ethan Holly, Mrinal Kalakrishnan, Vincent Vanhoucke, and Sergey Levine. "QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation". In: *CoRR* abs/1806.10293 (2018). arXiv: 1806.10293. URL: http://arxiv.org/abs/1806.10293.

[37] *Key Concepts in RL*. 2018. URL: https://spinningup.openai.com/en/latest/spinningup/rl_intro.html.

[38] O. Khatib. "A unified approach for motion and force control of robot manipulators: The operational space formulation". In: *IEEE Journal on Robotics and Automation* 3.1 (1987), pp. 43–53. DOI: 10.1109/JRA.1987.1087068.

[39] 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) (IEEE Cat. No.04CH37566)*. Vol. 3. 2004, 2149–2154 vol.3. DOI: 10.1109/IROS.2004.1389727.

[40] Marian Körber, Johann Lange, Stephan Rediske, Simon Steinmann, and Roland Glück. "Comparing Popular Simulation Environments in the Scope of Robotics and Reinforcement Learning". In: *CoRR* abs/2103.04616 (2021). arXiv: 2103.04616. URL: https://arxiv.org/abs/2103.04616.

[41] KUKA. *System Software, KUKA Sunrise.OS 1.17, KUKA Sunrise.Workbench 1.17*. English. KUKA. 667 pp.

[42] *kuka_lbr_iiwa_brochure_en.pdf*. https://www.kuka.com/-/media/kuka-downloads/imported/9cb8e311bfd744b4b0eab25ca883f6d3/kuka_lbr_iiwa_brochure_en.pdf?rev=435b38a9b2dc4cbf9c4fb38a8a9707ef&hash=20D50DAC5646B03E68B4D473B287812A. (Accessed on 02/08/2022).

[43] Swagat Kumar, Hayden Sampson, and Ardhendu Behera. "Benchmarking Deep Reinforcement Learning Algorithms for Vision-based Robotics". In: *CoRR* abs/2201.04224 (2022). arXiv: 2201.04224. URL: https://arxiv.org/abs/2201.04224.

[44] Guillaume Lample and Devendra Singh Chaplot. "Playing FPS Games with Deep Reinforcement Learning". In: *CoRR* abs/1609.05521 (2016). arXiv: 1609.05521. URL: http://arxiv.org/abs/1609.05521.

[45] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. "End-to-End Training of Deep Visuomotor Policies". In: *CoRR* abs/1504.00702 (2015). arXiv: 1504.00702. URL: http://arxiv.org/abs/1504.00702.

[46] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. "Continuous control with deep reinforcement learning." In: *ICLR*. Ed. by Yoshua Bengio and Yann LeCun. 2016. URL: http://dblp.uni-trier.de/db/conf/iclr/iclr2016.html#LillicrapHPHETS15.

[47] Jeffrey Mahler and Ken Goldberg. "Learning Deep Policies for Robot Bin Picking by Simulating Robust Grasping Sequences". In: *Proceedings of the 1st Annual Conference on Robot Learning*. Ed. by Sergey Levine, Vincent Vanhoucke, and Ken Goldberg. Vol. 78. Proceedings of Machine Learning Research. PMLR, 13–15 Nov 2017, pp. 515–524. URL: https://proceedings.mlr.press/v78/mahler17a.html.

[48] A. Rupam Mahmood, Dmytro Korenkevych, Gautham Vasan, William Ma, and James Bergstra. "Benchmarking Reinforcement Learning Algorithms on Real-World Robots". In: *CoRR* abs/1809.07731 (2018). arXiv: 1809.07731. URL: http://arxiv.org/abs/1809.07731.

[49] Ajay Mandlekar, Danfei Xu, Josiah Wong, Soroush Nasiriany, Chen Wang, Rohun Kulkarni, Li Fei-Fei, Silvio Savarese, Yuke Zhu, and Roberto Martín-Martín. "What Matters in Learning from Offline Human Demonstrations for Robot Manipulation". In: *arXiv preprint arXiv:2108.03298*. 2021.

[50] Jan Matas, Stephen James, and Andrew J. Davison. "Sim-to-Real Reinforcement Learning for Deformable Object Manipulation". In: *Proceedings of The 2nd Conference on Robot Learning*. Ed. by Aude Billard, Anca Dragan, Jan Peters, and Jun Morimoto. Vol. 87. Proceedings of Machine Learning Research. PMLR, 29–31 Oct 2018, pp. 734–743. URL: https://proceedings.mlr.press/v87/matas18a.html.

[51] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. "Asynchronous Methods for Deep Reinforcement Learning". In: *CoRR* abs/1602.01783 (2016). arXiv: 1602.01783. URL: http://arxiv.org/abs/1602.01783.

[52] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

[53] *Montech - conveyor belts, transport systems and profile system*. https://montech.com/ch/en/. (Accessed on 05/16/2022).

[54] *ODE*. URL: http://www.ode.org/ (visited on 12/14/2021).

[55] *Ogre3D*. URL: https://www.ogre3d.org/ (visited on 12/14/2021).

[56] *ojrise/Robot_Learning_master: Master repository for robot learning with Petter Rasmussen and Ole Jørgen Rise*. https://github.com/ojrise/Robot_Learning_master. (Accessed on 05/16/2022).

[57] Andrej Orsula. "Deep Reinforcement Learning for Robotic Grasping from Octrees". In: (2021). URL: https://projekter.aau.dk/projekter/files/421582447/Deep_Reinforcement_Learning_for_Robotic_Grasping_from_Octrees.pdf.

[58] *OSRF*. URL: https://www.openrobotics.org/ (visited on 12/14/2021).

[59] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *CoRR* abs/1912.01703 (2019). arXiv: 1912.01703. URL: http://arxiv.org/abs/1912.01703.

[60] Deepak Pathak, Dhiraj Gandhi, and Abhinav Gupta. "Self-Supervised Exploration via Disagreement". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR,

Sept. 2019, pp. 5062–5071. URL: https://proceedings.mlr.press/v97/pathak19a.html.

[61] *PhysX*. URL: https://www.nvidia.com/en-us/drivers/physx/physx-9-19-0218-driver/.

[62] *ra-mtp-ntnu/zivid-ros2: Unofficial ROS2 driver for Zivid 3D cameras*. https://github.com/ra-mtp-ntnu/zivid-ros2. (Accessed on 03/21/2022).

[63] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. "Stable-Baselines3: Reliable Reinforcement Learning Implementations". In: *Journal of Machine Learning Research* 22.268 (2021), pp. 1–8. URL: http://jmlr.org/papers/v22/20-1364.html.

[64] Antonin Raffin and Freek Stulp. "Generalized State-Dependent Exploration for Deep Reinforcement Learning in Robotics". In: *CoRR* abs/2005.05719 (2020). arXiv: 2005.05719. URL: https://arxiv.org/abs/2005.05719.

[65] Aravind Rajeswaran, Vikash Kumar, Abhishek Gupta, John Schulman, Emanuel Todorov, and Sergey Levine. "Learning Complex Dexterous Manipulation with Deep Reinforcement Learning and Demonstrations". In: *CoRR* abs/1709.10087 (2017). arXiv: 1709.10087. URL: http://arxiv.org/abs/1709.10087.

[66] *Robotiq 2F-85, 2F-140 Instruction Manual*. https://assets.robotiq.com/website-assets/support_documents/document/2F-85_2F-140_Instruction_Manual_CB-Series_PDF_20190227.pdf. (Accessed on 12/14/2021).

[67] *ROS on DDS*. https://design.ros2.org/articles/ros_on_dds.html. (Accessed on 12/16/2021).

[68] *ros-industrial/kuka_experimental: Experimental packages for KUKA manipulators within ROS-Industrial (http://wiki.ros.org/kuka_experimental)*. https://github.com/ros-industrial/kuka_experimental. (Accessed on 03/21/2022).

[69] Andrei A. Rusu, Matej Večerík, Thomas Rothörl, Nicolas Heess, Razvan Pascanu, and Raia Hadsell. "Sim-to-Real Robot Learning from Pixels with Progressive Nets". In: *Proceedings of the 1st Annual Conference on Robot Learning*. Ed. by Sergey Levine, Vincent Vanhoucke, and Ken Goldberg. Vol. 78. Proceedings of Machine Learning Research. PMLR, 13–15 Nov 2017, pp. 262–270. URL: https://proceedings.mlr.press/v78/rusu17a.html.

[70] Gerrit Schoettler, Ashvin Nair, Jianlan Luo, Shikhar Bahl, Juan Aparicio Ojea, Eugen Solowjow, and Sergey Levine. "Deep Reinforcement Learning for Industrial Insertion Tasks with Visual Inputs and Natural Rewards". In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2020, pp. 5548–5555. DOI: 10.1109/IROS45743.2020.9341714.

[71] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. "Trust Region Policy Optimization". In: *CoRR* abs/1502.05477 (2015). arXiv: 1502.05477. URL: http://arxiv.org/abs/1502.05477.

[72] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. "Proximal Policy Optimization Algorithms". In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[73] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529 (2016), pp. 484–503. URL: http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html.

[74] *Simbody*. URL: https://simtk.org/projects/simbody (visited on 12/14/2021).

[75] *Singularity container*. https://sylabs.io/guides/3.5/user-guide/index.html. (Accessed on 05/20/2022).

[76] *SNR Value — ZIVID KNOWLEDGE BASE documentation*. https://support.zivid.com/v2.4/reference-articles/settings/processing-settings/snr-value.html. (Accessed on 12/16/2021).

[77] J. Sola and J. Sevilla. "Importance of input data normalization for the application of neural networks to complex industrial problems". In: *IEEE Transactions on Nuclear Science* 44.3 (1997), pp. 1464–1468. DOI: 10.1109/23.589532.

[78] *Start Production Faster - Robotiq*. https://robotiq.com/products/2f85-140-adaptive-robot-gripper. (Accessed on 12/14/2021).

[79] *Step() function in base.py in robosuite*. URL: https://github.com/ARISE-Initiative/robosuite/blob/master/robosuite/environments/base.py.

[80] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[81] Csaba Szepesvári. *Algorithms for Reinforcement Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan Claypool Publishers, 2010. URL: http://dx.doi.org/10.2200/S00268ED1V01Y201005AIM009.

[82] Jie Tan, Tingnan Zhang, Erwin Coumans, Atil Iscen, Yunfei Bai, Danijar Hafner, Steven Bohez, and Vincent Vanhoucke. "Sim-to-Real: Learning Agile Locomotion For Quadruped Robots". In: *CoRR* abs/1804.10332 (2018). arXiv: 1804.10332. URL: http://arxiv.org/abs/1804.10332.

[83] Yuval Tassa, Saran Tunyasuvunakool, Alistair Muldal, Yotam Doron, Siqi Liu, Steven Bohez, Josh Merel, Tom Erez, Timothy Lillicrap, and Nicolas Heess. $dm_control : SoftwareandTasksforContinuousControl$. 2020. arXiv: 2006.12983 [cs.RO].

[84] *tingelst/sunrisedds: CycloneDDS on KUKA Sunrise.OS for integration with ROS2*. https://github.com/tingelst/sunrisedds. (Accessed on 05/15/2022).

[85] *Tips and Tricks when creating a custom environment*. URL: https://stable-baselines3.readthedocs.io/en/master/guide/rl_tips.html#tips-and-tricks-when-creating-a-custom-environment.

[86] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. "Domain randomization for transferring deep neural networks from simulation to the real world". In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2017, pp. 23–30. DOI: 10.1109/IROS.2017.8202133.

[87] Joshua Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. "Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World". In: *CoRR* abs/1703.06907 (2017). arXiv: 1703.06907. URL: http://arxiv.org/abs/1703.06907.

[88] Emanuel Todorov, Tom Erez, and Yuval Tassa. "MuJoCo: A physics engine for model-based control". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2012, pp. 5026–5033. DOI: 10.1109/IROS.2012.6386109.

[89] *Understanding the importance of 3D hand-eye calibration*. https://blog.zivid.com/importance-of-3d-hand-eye-calibration. (Accessed on 12/15/2021).

[90] Matej Vecerık, Todd Hester, Jonathan Scholz, Fumin Wang, Olivier Pietquin, Bilal Piot, Nicolas Heess, Thomas Rothörl, Thomas Lampe, and Martin A. Riedmiller. "Leveraging Demonstrations for Deep Reinforcement Learning on Robotics Problems with Sparse Rewards". In: *CoRR* abs/1707.08817 (2017). arXiv: 1707.08817. URL: http://arxiv.org/abs/1707.08817.

[91] *W&B Company*. https://wandb.ai/site/company. (Accessed on 03/14/2022).

[92] *Webots*. URL: https://www.cyberbotics.com/ (visited on 12/14/2021).

[93] *webots*. URL: https://cyberbotics.com/doc/reference/pbrappearance.

[94] *Weights Biases*. URL: https://wandb.ai/site.

[95] *XML Reference — MuJoCo documentation*. https://mujoco.readthedocs.io/en/latest/XMLreference.html. (Accessed on 05/28/2022).

[96] Andy Zeng, Shuran Song, Johnny Lee, Alberto Rodriguez, and Thomas Funkhouser. "TossingBot: Learning to Throw Arbitrary Objects With Residual Physics". In: *IEEE Transactions on Robotics* 36.4 (2020), pp. 1307–1319. DOI: 10.1109/TRO.2020.2988642.

[97] Fangyi Zhang, Jürgen Leitner, Michael Milford, Ben Upcroft, and Peter I. Corke. "Towards Vision-Based Deep Reinforcement Learning for Robotic Motion Control". In: *CoRR* abs/1511.03791 (2015). arXiv: 1511.03791. URL: http://arxiv.org/abs/1511.03791.

[98] Wenshuai Zhao, Jorge Peña Queralta, Li Qingqing, and Tomi Westerlund. "Towards Closing the Sim-to-Real Gap in Collaborative Multi-Robot Deep Reinforcement Learning". In: *CoRR* abs/2008.07875 (2020). arXiv: 2008.07875. URL: https://arxiv.org/abs/2008.07875.

[99] Wenshuai Zhao, Jorge Peña Queralta, and Tomi Westerlund. "Sim-to-Real Transfer in Deep Reinforcement Learning for Robotics: a Survey". In: *CoRR* abs/2009.13303 (2020). arXiv: 2009.13303. URL: https://arxiv.org/abs/2009.13303.

[100] Yuke Zhu, Josiah Wong, Ajay Mandlekar, and Roberto Martín-Martín. "robosuite: A Modular Simulation Framework and Benchmark for Robot Learning". In: *arXiv preprint arXiv:2009.12293*. 2020.

[101] Yuke Zhu, Josiah Wong, Ajay Mandlekar, and Roberto Martín-Martín. "robosuite: A Modular Simulation Framework and Benchmark for Robot Learning". In: *arXiv preprint arXiv:2009.12293*. 2020.

[102] *Zivid Two Datasheet.pdf*. https://www.zivid.com/hubfs/files/SPEC/Zivid%20Two%20Datasheet.pdf. (Accessed on 12/16/2021).

[103] *Zivid's Knowledge Base — ZIVID KNOWLEDGE BASE documentation*. https://support.zivid.com/latest/index.html. (Accessed on 05/16/2022).

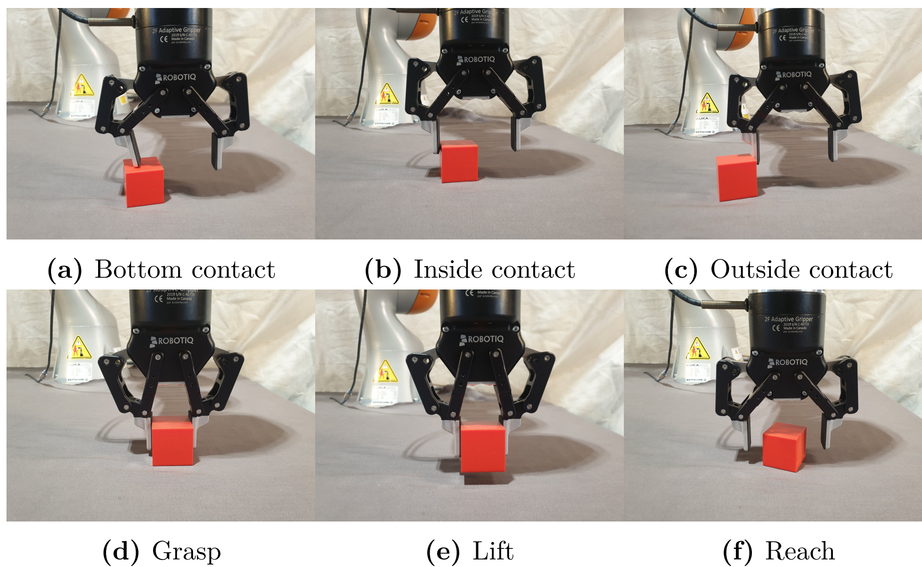
[104] *zivid/zivid-python: Official Python package for Zivid 3D cameras*. https://github.com/zivid/zivid-python. (Accessed on 05/15/2022).

[105] *zivid/zivid-ros: Official ROS driver for Zivid 3D cameras*. https://github.com/zivid/zivid-ros. (Accessed on 05/16/2022).

Appendix A.

Test setup

A.1. Contacts and box placement



(a) Bottom contact **(b)** Inside contact **(c)** Outside contact

(d) Grasp **(e)** Lift **(f)** Reach

Figure A.1.: Shows the different contacts quantified during the experiments.

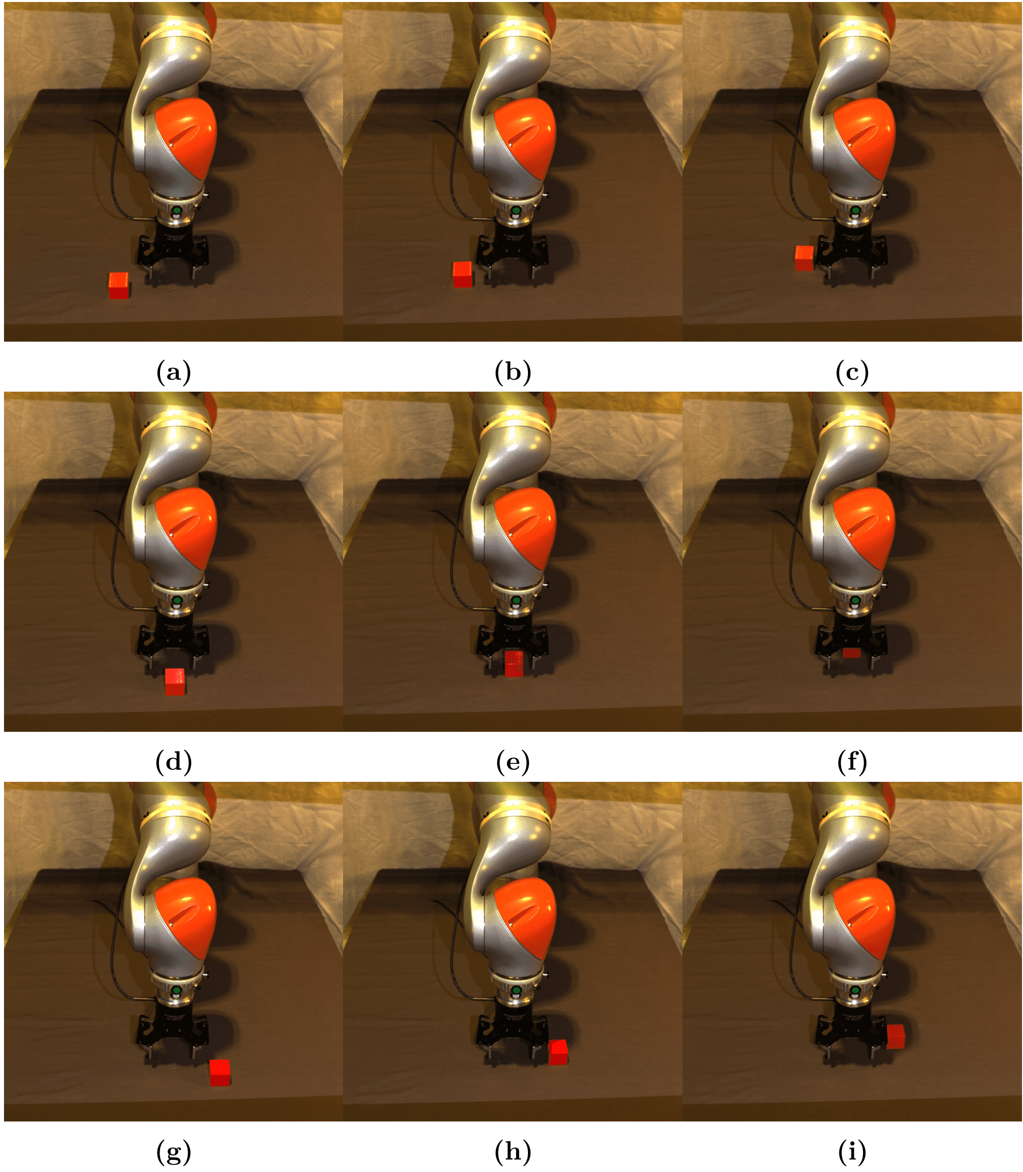
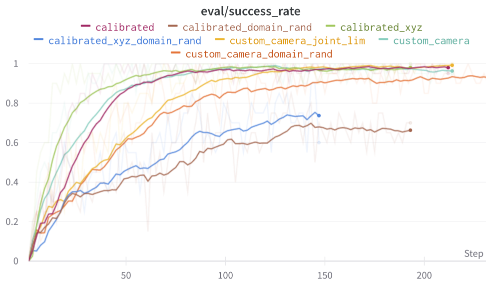
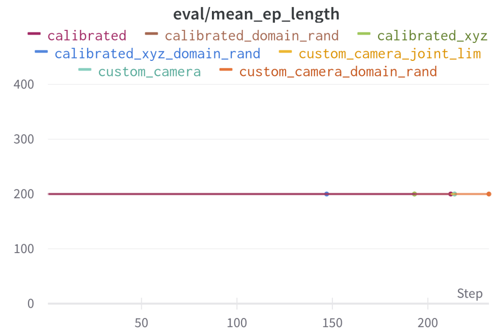


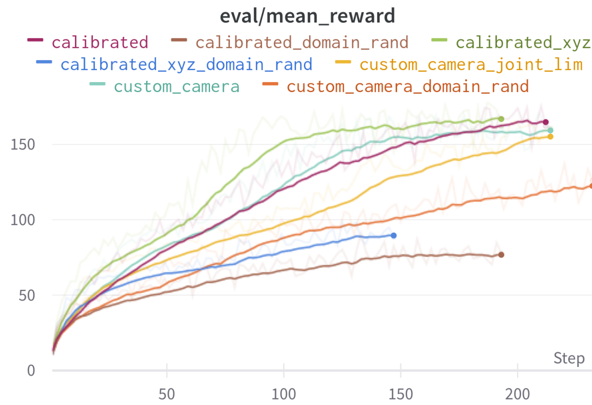
Figure A.2.: Shows nine of the cube positions during experiments.

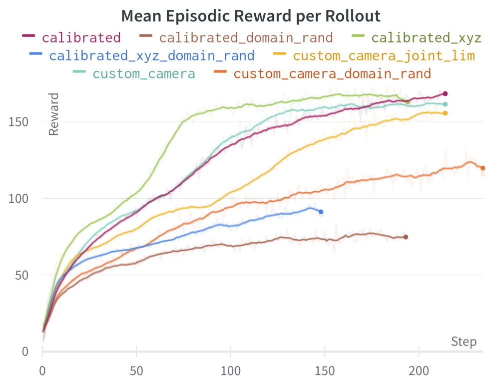
Appendix B.

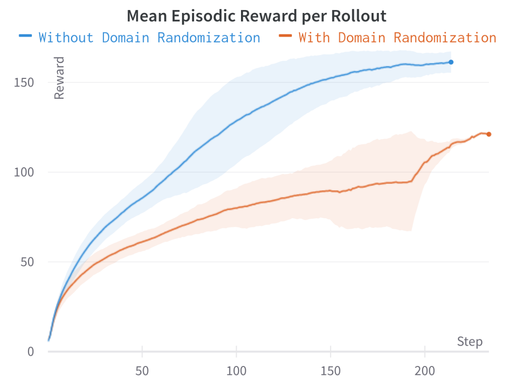
Training Plots

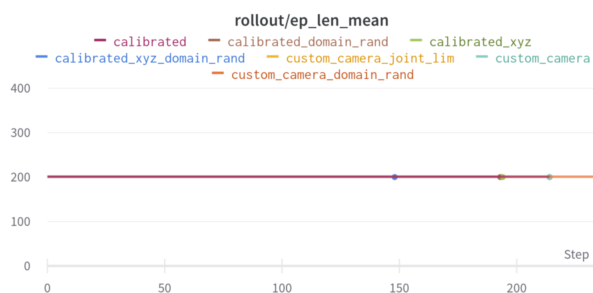

eval/success_rate
calibrated
calibrated_domain_rand
calibrated_xyz
calibrated_xyz_domain_rand
custom_camera_joint_lim
custom_camera
custom_camera_domain_rand
Step

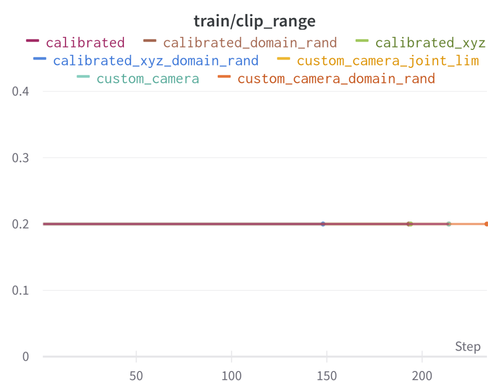

eval/mean_ep_length
calibrated
calibrated_domain_rand
calibrated_xyz
calibrated_xyz_domain_rand
custom_camera_joint_lim
custom_camera
custom_camera_domain_rand
Step

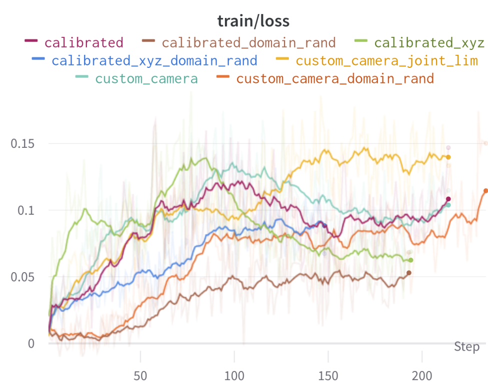

eval/mean_reward
calibrated
calibrated_domain_rand
calibrated_xyz
calibrated_xyz_domain_rand
custom_camera_joint_lim
custom_camera
custom_camera_domain_rand
Step

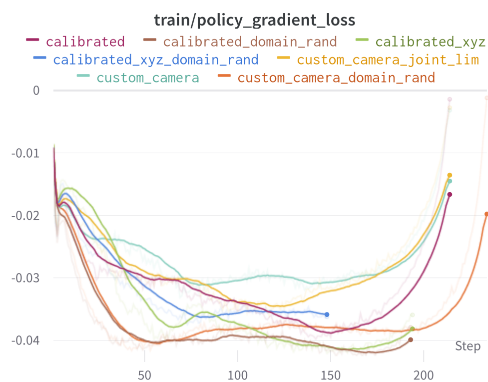

Mean Episodic Reward per Rollout
calibrated
calibrated_domain_rand
calibrated_xyz
calibrated_xyz_domain_rand
custom_camera_joint_lim
custom_camera
custom_camera_domain_rand
Reward
Step

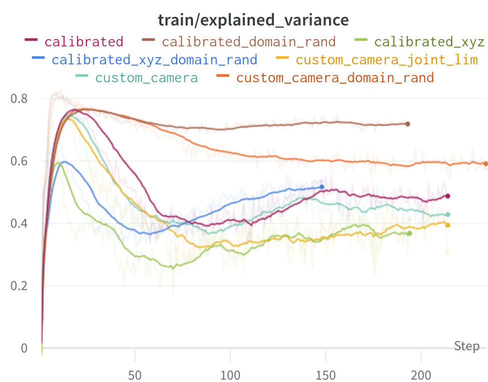

Mean Episodic Reward per Rollout
Without Domain Randomization
With Domain Randomization
Reward
Step

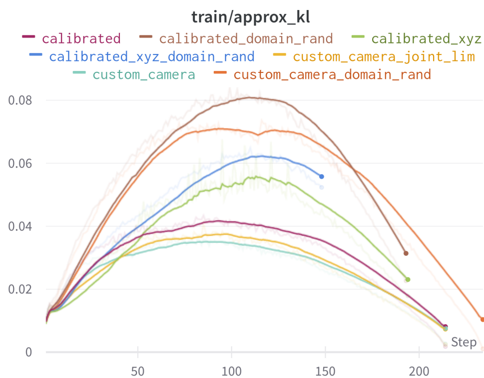

rollout/ep_len_mean
calibrated
calibrated_domain_rand
calibrated_xyz
calibrated_xyz_domain_rand
custom_camera_joint_lim
custom_camera
custom_camera_domain_rand
Step

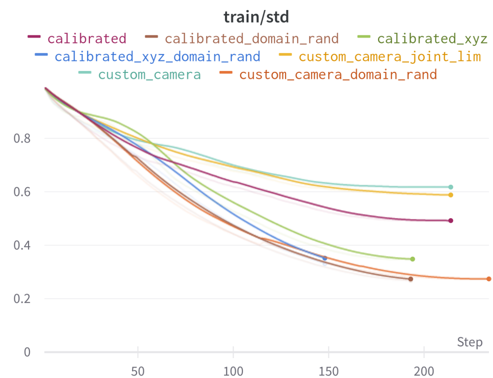

train/clip_range
calibrated
calibrated_domain_rand
calibrated_xyz
calibrated_xyz_domain_rand
custom_camera_joint_lim
custom_camera
custom_camera_domain_rand
Step

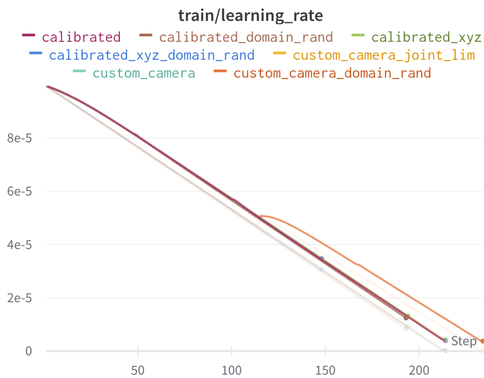

train/loss
calibrated
calibrated_domain_rand
calibrated_xyz
calibrated_xyz_domain_rand
custom_camera_joint_lim
custom_camera
custom_camera_domain_rand
Step

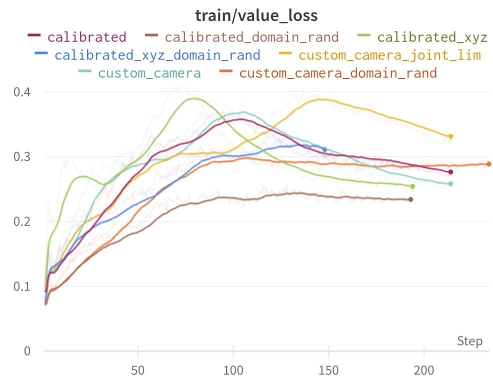

train/policy_gradient_loss
calibrated
calibrated_domain_rand
calibrated_xyz
calibrated_xyz_domain_rand
custom_camera_joint_lim
custom_camera
custom_camera_domain_rand
Step

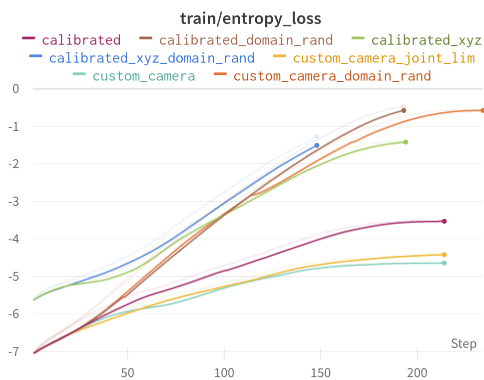

train/explained_variance
calibrated
calibrated_domain_rand
calibrated_xyz
calibrated_xyz_domain_rand
custom_camera_joint_lim
custom_camera
custom_camera_domain_rand
Step

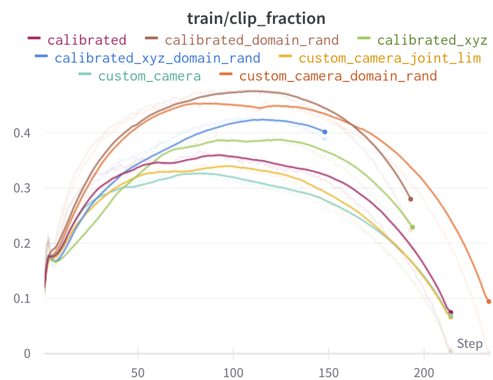

train/approx_kl
calibrated
calibrated_domain_rand
calibrated_xyz
calibrated_xyz_domain_rand
custom_camera_joint_lim
custom_camera
custom_camera_domain_rand
0.08
0.06
0.04
0.02
0
Step
50
100
150
200


train/std
calibrated
calibrated_domain_rand
calibrated_xyz
calibrated_xyz_domain_rand
custom_camera_joint_lim
custom_camera
custom_camera_domain_rand
0.8
0.6
0.4
0.2
0
Step
50
100
150
200


train/learning_rate
calibrated
calibrated_domain_rand
calibrated_xyz
calibrated_xyz_domain_rand
custom_camera_joint_lim
custom_camera
custom_camera_domain_rand
8e-5
6e-5
4e-5
2e-5
0
Step
50
100
150
200


train/value_loss
calibrated
calibrated_domain_rand
calibrated_xyz
calibrated_xyz_domain_rand
custom_camera_joint_lim
custom_camera
custom_camera_domain_rand
0.4
0.3
0.2
0.1
0
Step
50
100
150
200


train/entropy_loss
calibrated
calibrated_domain_rand
calibrated_xyz
calibrated_xyz_domain_rand
custom_camera_joint_lim
custom_camera
custom_camera_domain_rand
0
-1
-2
-3
-4
-5
-6
-7
Step
50
100
150
200


train/clip_fraction
calibrated
calibrated_domain_rand
calibrated_xyz
calibrated_xyz_domain_rand
custom_camera_joint_lim
custom_camera
custom_camera_domain_rand
0.4
0.3
0.2
0.1
0
Step
50
100
150
200

Appendix C.

Digital appendix

C.1. Github code

The links below gives acces to the code used for training of agents and communication of hardware on the pjysical setup.

C.1.1. Training code

https://github.com/ojrise/Robot_Learning_master

C.2. ROS2

C.2.1. ROS2 package

Package for communication with gripper and robot. https://github.com/pettras/policy-communication-ros2

C.3. Attached .zip file

```
digital_appendix
├─ hand_eye_calibration
├─ videos
└─ raw_data
```

C.3.1. hand_eye_calibration

```
digital_appendix
└─ hand_eye_calibration
   ├─ hand_eye_calibration.py
   ├─ results.txt
   ├─ settings.yml
   └─ data
      ├─ eef_pos.txt
      ├─ img01.zdf
      ├─ .
      └─ img20.zdf
```

The hand_eye_calibration folder contains the code for hand-eye calibration based of the Zivid code, the data used in the calibration and the result from the performed hand-eye calibration. In addition, the configuration file used for getting high performing acquisitions in Zivid studio is added. The configuration file comes from the Zivid Knowledge Base [25].

C.3.2. Videos

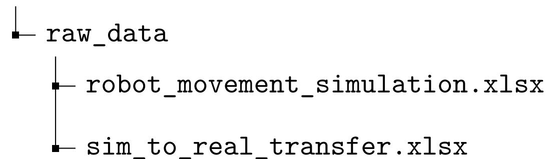
```
digital_appendix
└─ videos
   ├─ physical
   │  ├─ Cal_AcLim_DR_vid_physical.mp4
   │  ├─ Cal_AcLim_vid_physical.mp4
   │  ├─ Cal_DR_vid_physical.mp4
   │  ├─ Cal_vid_physical.mp4
   │  ├─ Cus_DR_vid_physical.mp4
   │  ├─ Cus_JoLim_vid_physical.mp4
   │  └─ Cus_vid_physical.mp4
   └─ simulation
      ├─ Cal_AcLim_DR_vid.mp4
      ├─ Cal_AcLim_vid.mp4
      ├─ Cal_DR_vid.mp4
      ├─ Cal_vid.mp4
      ├─ Cus_DR_vid.mp4
      ├─ Cus_JoLim_vid.mp4
      └─ Cus_vid.mp4
```

The video folder in the digital appendix contain Videos from both simulation and the physical environment for all agents mentioned in this thesis. These are videos of a random episode and is not seen as a result, but rather a visualization of the agents actions in the physical environment. Table C.1 shows the corresponding temination step for the physical videos.

Table C.1.: List of the algorithms added as video attachment and the termination step.

Algorithm	Termination step
Cus	52
Cus-DR	9
Cus-JoLim	50
Cal	12
Cal-DR	179
Cal-AcLim	106
Cal-Aclim-DR	76

C.3.3. Raw data



The raw data from the sim-to-real transfer testing and raw data used to find the simulated robot movement for every time-step of the simulation.

Appendix D.

Hyperparameters

Hyperparameter	Cus	Cus-DR	Cus-JoLim	Cal
Learning Rate Schedule	Linear, $1.0 \cdot 10^{-4} \rightarrow 0$			
Batch size	512			
Number of agent steps	2048	3072	2048	2048
Number of Actors	64			
Number of epochs	10			
Discount Factor γ	0.99			
GAE parameter λ	0.95			
Clip range ϵ	0.2			
Clip range vf	No Clipping			
Normalize advantage	True			
Entropy coeff. c_2	0.0			
VF coeff. c_1	0.5			
Max Gradient Clipping	0.5			
Action noise exploration	True			
Target KL	No Limit			

Hyperparameter	**Cal-DR**	**Cal-AcLim**	**Cal-AcLim-DR**
Learning Rate Schedule	Linear, $1.0 \cdot 10^{-4} \rightarrow 0$		
Batch size	512		
Number of agent steps	3072	2048	3072
Number of Actors	64		
Number of epochs	10		
Discount Factor γ	0.99		
GAE parameter λ	0.95		
Clip range ϵ	0.2		
Clip range vf	No Clipping		
Normalize advantage	True		
Entropy coeff. c_2	0.0		
VF coeff. c_1	0.5		
Max Gradient Clipping	0.5		
Action noise exploration	True		
Target KL	No Limit		

Appendix E.

Hardware communication

This part of the appendix is added as documentation for the communication in MANULAB.

E.1. Communication with the KUKA iiwa 14

E.1.1. General info

Robot IP: 172.32.1

Connection: Ethernet

Python version needed: Python 3.8

ROS2 application: https://github.com/tingelst/sunrisedds

E.1.2. Procedure for turning it on

1. Connect to power (The fuses in MANULAB can only handle one robot per circuit)

2. Turn on the green switch, wait.

E.1.3. Network configuration with Linux

Robot

IPv4 Method = manual

Address = 172.32.1.67

Netmask = 255.255.255.0

ROS

IPv4 Method = manual

Address = 172.32.1.34

Netmask = 255.255.255.0

E.2. Communication with the Zivid camera

E.2.1. General info

Connection: USB3

E.2.2. How to connect a Zivid Two with Linux or Windows

1. Download the lates SDK
2. Run Zivid studio

ROS2 driver for Zivid

System distributions: ROS2 Galactic and Ubuntu 20.04

Link: https://github.com/ra-mtp-ntnu/zivid-ros2

E.2.3. Network configuration with Linux

IPv4 Method = manual

Adress = 172.28.60.1

Netmask = 255.255.255.0

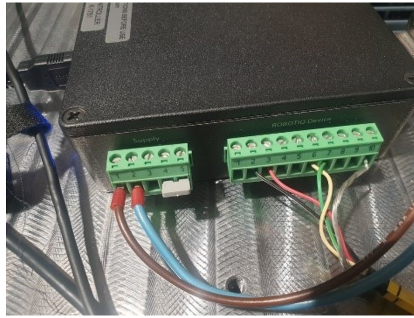


Figure E.1.: Wire configuration for Robotiq controller.

E.3. Communication with the Robotiq 2F-85 gripper

E.3.1. General info

Connection: USB2

Python version: 3.8

E.3.2. Procedure for setting it up

Connect controller to power supply by connecting to Siemens SITOP PSU100S Power supply.

Connect gripper to Robotiq controller with the configuration of the cables illustrated in E.1.

E.3.3. How to connect with python package

```
# Download the package and dependencies
git clone https://github.com/tingelst/robotiq_modbus
    _controller
cd robotiq_modbus_controller
python -m pip install -e .
pip install pymodbus
Pip install serial --upgrade
```

Linux

```
#To find the USB connection
ls /dev
#Give permision to communicate over the port
sudo chmod 777 /dev/ttyUSB1
```

Change the code in the modbus_rtu to the communication (Eg: device = “ttyUSB1”)

E.3.4. Use the python package

```
python -i .\modbus_rtu.py
driver.status()
driver.move(pos=100, speed = 10, force = 10)
driver.status().position.po #current pose
driver.status().position.pr #requested pose
```

E.4. Run the ROS2 system

In .bashrc

```
#Source the ros2 distro
source /opt/ros/galactic/setup.bash
#Enable robot communication
export CYCLONEDDS
#To source the latest ROS2 build
source dev_ws/install/setup.bash
```

How to start the system

Check that the Ethernet and cable configuration, the .bashrc commands and the internet configuration

```
colcon build --symlink-install
```

On smartPad

Turn the key, check that you are in T1 mode and turn the key back

Run Applications > Ros2RobotApplication to stat the robotNode on the Sunrise.Cabinet

In a terminal to check if the /state and /command topics are up:

```
ros2 topic list
```

Launch the Zivid camera node

Terminal1:

```
ros2 launch zivid_camera zivid_camera_standalone.launch.py
```

Terminal2:

```
ros2 run rviz2 rviz2 -d /home/kukauser/dev_ws/src/
    zivid-ros2/zivid_camera/rviz/camera_view.rviz
```

Launch the policyNode node

```
ros2 run master policyNode
```

Launch the gripperNode node

```
sudo chmod 777 /dev/ttyUSB1
ros2 run master gripperNode
```

Test the gripper communication

```
ros2 topic pub --once /gripper_pos master_interfaces/
    msg/GripperPos "pos: 1"
```

Appendix F.

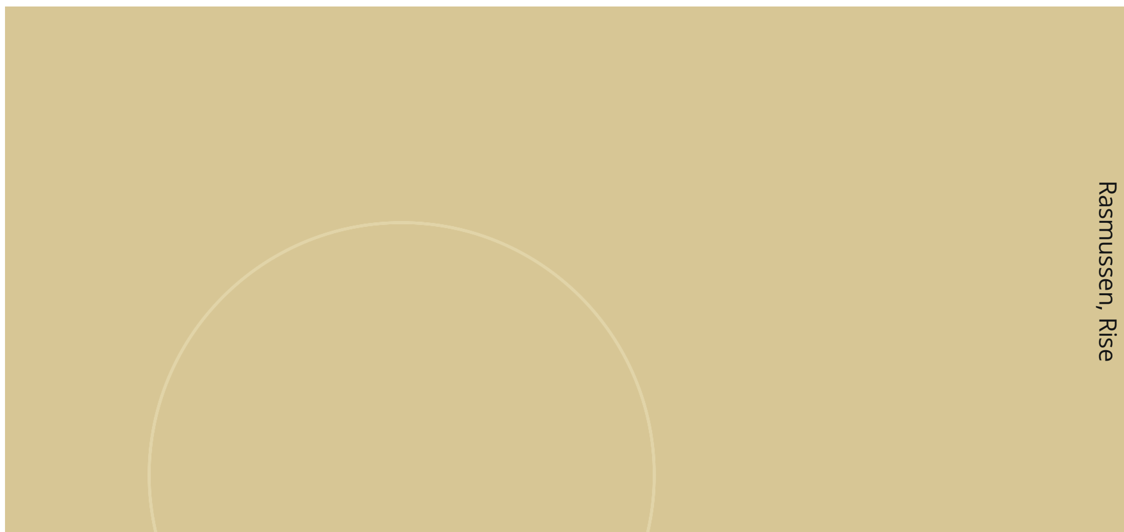
Edited Robosuite code

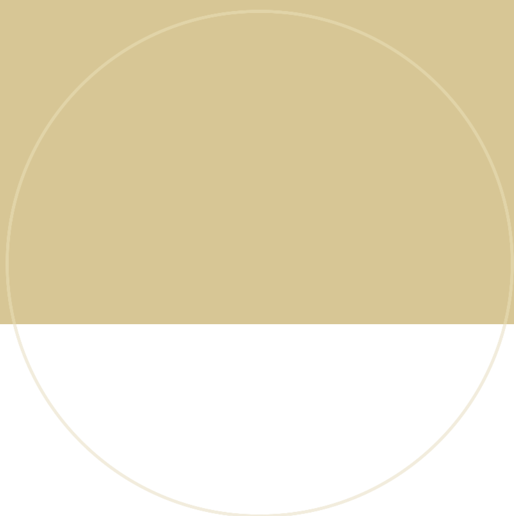
This code presented below is changed in the Robosuite repository. The function changed is the *def step(self, action)* located on line 377 in the file *base.py* in the folder robosuite/environments [79].

```
def step(self, action):
        """
        Takes a step in simulation with control command
            @action.
        Args:
            action (np.array): Action to execute within
                the environment
        Returns:
            4-tuple:
                - (OrderedDict) observations from the
                    environment
                - (float) reward from the environment
                - (bool) whether the current episode is
                    completed or not
                - (dict) misc information
        Raises:
            ValueError: [Steps past episode termination]
        """
        if self.done:
            raise ValueError("executing action in
                terminated episode")

        self.timestep += 1

        # Since the env.step frequency is slower than the
```

```
        mjsim timestep frequency, the internal
        controller will output
    # multiple torque commands in between new high
        level action commands. Therefore, we need to
        denote via
    # 'policy_step' whether the current step we're
        taking is simply an internal update of the
        controller,
    # or an actual policy update
    policy_step = True

    # Loop through the simulation at the model
        timestep rate until we're ready to take the
        next policy step
    # (as defined by the control frequency specified
        at the environment level)
    mujoco_calc_error = False
    for i in range(int(self.control_timestep / self.
        model_timestep)):
        self.sim.forward()
        self._pre_action(action, policy_step)
        try:
            self.sim.step()
        except:
            print("mujoco_calc_error")
            mujoco_calc_error = True
            break
        self._update_observables()
        policy_step = False

    if not mujoco_calc_error:
        # Note: this is done all at once to avoid
            floating point inaccuracies
        self.cur_time += self.control_timestep

        reward, done, info = self._post_action(action)

        if self.viewer is not None and self.renderer
            != 'mujoco':
            self.viewer.update()

        observations = self.viewer._get_observations()
```

```
                if self.viewer_get_obs else self.
                _get_observations()
        else:
            self.cur_time += self.control_timestep

            reward, done, info = self._post_action(action)

            if self.viewer is not None and self.renderer
                != 'mujoco':
                self.viewer.update()

            observations = self.viewer._get_observations()
                if self.viewer_get_obs else self.
                _get_observations()
            done = True

        return observations, reward, done, info
```





Rasmussen, Rise

Master's Thesis 2022

NTNU
Norwegian University of
Science and Technology