Ludvik Kasbo

Reducing the Sim-To-Real Gap in Reinforcement Learning for Robotic Grasping with Depth Observations

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

Ludvik Kasbo

Reducing the Sim-To-Real Gap in Reinforcement Learning for Robotic Grasping with Depth Observations

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

Norwegian University of Science and Technology
Faculty of Engineering
Department of Manufacturing and Civil Engineering

Preface

This master's thesis is written for the Department of Mechanical and Industrial Engineering at the Norwegian University of Science and Technology in the field of robotics and automation. It was written in the spring semester of 2022 and delivered in august 2022 due to illness, causing delays in the project process.

This thesis and its experiments were written and conducted by Ludvik Kasbo, but the framework for conducting the experiments was created in tight collaboration with Petter Rasmussen and Ole Jørgen Rise. They deserve special thanks for their effort and cooperation.

I want to express my gratitude towards my supervisor Lars Tingelstad and substitute supervisor Eirik Njåstad who have provided answers to my questions, encouragement, and support during this project.

I hope this thesis will contribute toward creating successful algorithms and systems for autonomous robotic grasping tasks in the future.

Summary

Robotic grasping serves as a barrier for many robotic manipulation tasks. Generalized autonomous grasping remains an unsolved challenge, but reinforcement learning has shown promising results within robotic grasping. There are several challenges related to using reinforcement learning for robotic grasping, but sample inefficiency is one of the most notable. Utilizing simulated environments to increase the amount of data available in training is a proposed solution. Previous works [36, 13] show that directly transferring an algorithm from simulation to a real-world setting causes a large loss in performance. This is called the sim-to-real gap and is mainly caused by the inability of simulators to create exact replicas of real-world physics and visuals.

Most grasping algorithms are trained utilizing images. The intuition is that simulated and real images are too different, causing a significant performance loss when the unfamiliar real images are utilized. In this thesis, we conduct experiments to examine how training with depth information affects the sim-to-real gap. The experiments show that using depth information in addition to RGB images reduces the sim-to-real gap by 21.8% compared to using only RGB images. To conduct these experiments, we have created a modern simulation framework consisting of well-known reinforcement learning and robotics frameworks. This framework and the environments created are also replicated with a real-world robotic setup for grasping with the high-performing 3D camera, Zivid Two.

Sammendrag

Griping med roboter er en barriere for mange robotmanipulasjonsoppgaver. Generalisert autonom griping er fortsatt et uløst problem, men bruk av forsterkende læring har vist lovende resultater. Det er flere utfordringer knyttet til bruk av forsterkende læring for griping med roboter, men et av de mest bemerkelsesverdige problemene er data ineffektivitet. Å bruke simulerte miljøer for å øke mengden data som er tilgjengelig i trening er en mulig løsning. Tidligere arbeider [36, 13] viser at direkte overføring av en algoritme fra simulering til virkeligheten medfører et stort ytelsestap. Dette kalles sim-to-real gapet og er hovedsakelig forårsaket av simulatorer sin manglende evne til å lage eksakte modeler av virkelighetens fysikk og visuelle elementer.

De fleste algoritmer for griping bruker bilder som observasjoner. Intuisjonen er at simulerte og ekte bilder er for forskjellige, noe som forårsaker et stort ytelsestap når algoritmen bare er vant til å se simulerte bilder. I denne oppgaven gjennomfører vi eksperimenter for å undersøke hvordan trening med dybdeinformasjon påvirker sim-to-real gapet. Eksperimentene viser at bruk av dybdeinformasjon i tillegg til RGB-bilder reduserer gapet mellom simulator og virkeligheten med $21,8\%$ sammenlignet med å bare bruke RGB-bilder. For å gjennomføre disse eksperimentene laget vi også et moderne simuleringsrammeverk bestående av velkjente forsterkende læring og robotikk rammeverk. Dette simuleringsrammeverket og simulerings-miljøene som ble laget, er også gjenskapt med et virkelig fysisk robotoppsett med det avanserte 3D-kameraet Zivid Two.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Traditionally automation and robotics have required the task and objects to stay constant and not deviate from their original form. Complex tasks involving novel objects and experiences have had little to no potential of being automated. Reinforcement learning provides a framework and set of tools for robotics that can be used to create these sophisticated and hard-to-engineer behaviors that are required to solve complex tasks. One of the hardest tasks in robotics is manipulation tasks where a robot has to grasp one or more objects. Robotic grasping has become one of the largest unsolved challenges in robotic manipulation [11], but it is essential for carrying out many robotic manipulation tasks.

There are several examples of grasping systems trained with reinforcement algorithms that reach grasp success rates of 92-96 % on novel objects [15, 32]. This performance is not adequate for practical use. One common problem for most reinforcement learning algorithms is sample ineffiacy. Robots need hundreds to thousands of running hours to collect enough data to train successful algorithms, which is both impractical and expensive. One solution could be to use synthetic data from training in simulation. Algorithms can be partially or fully trained in simulation, and then the policy can be transferred to its real-world application.

In [13] an algorithm was trained to 98% success rate on novel objects in simulation. When deployed to a real-world grasping scenario, the algorithm achieved only 21%. This performance difference is known as the sim-to-real gap or reality gap. It is mainly caused by the inability of simulators to create exact replicas of real-world physics and images. This causes differences in sensing, actuation, physics, and image observations to occur. Closing the sim-to-real gap would allow algorithms to be trained entirely in simulation and therefore eliminate the problems related to sample inefficiency. This would be a large step towards making reinforcement learning for robotic manipulation sufficient for solving tasks that today require human interaction.

Domain randomization is a known technique to help reduce the sim-to-real gap. The intention behind randomizing different factors in the simulated environment is to force the network to extract only semantically relevant features and therefore extract the same information from the simulation and real-world observations despite the differences between the simulated and real sensory input. The visual modality is most notable as position, velocity, and acceleration measurements are considerably easier to mimic in simulation. By training an algorithm with domain randomization, the real-world grasp success in [13] rose to 37%. Modern physics engines and renders produce high-quality images, but they do not look real enough. Even small changes in lighting texture, color, and capturing method will create very different values in the RGB format even though the images resemble each other to the human eye. The intuition is that even though an algorithm is trained with visual domain randomization, the real image domain will still present a partially novel domain.

In this thesis, we will test how adding depth information will affect the sim-to-real gap. The idea is that depth is less dependent on the variable and random factors that affect images, such as lighting and textures. A high-quality 3D camera is therefore expected to present less of a domain difference and reduce the sim-to-real gap.

This master thesis is partially based on the work done in the specialization project [16] preceding this paper. Some of the background and theory parts are from or are based on this work.

1.1 Objectives

The overarching goal of this thesis is to:

- Investigate how RGB-D image observation will affect the sim-to-real transfer compared to using RGB observations.

The intention is that depth images from a state-of-the-art 3D camera will be less affected by the variable factors causing the sim-to-real gap for RGB images alone, as depth is independent of light, shadows, and textures as opposed to RGB values in images.

In order to conduct experiments for the overarching goal, several sub-goals, but still time-consuming and challenging goals are defined:

- Create a reinforcement learning framework for robotic grasping that allows training with RGB-D images, RGB images, and other observables commonly used in robotic grasping.

- Create a custom simulated environment that facilitates successful sim-to-real transfer.
- Train a high-performing grasping policy utilizing reinforcement learning.
- Process real depth data observations from the camera to resemble simulated depth data.

1.2 Contributions

For this master's thesis, a framework for reinforcement learning in simulation was constructed using the robosuite [37] framework on top of the physics engine MuJoCo [20] with the reinforcement learning framework Stable-Baselines3 [26]. A large part of this system was developed in combination with Petter Rasmussen and Ole Jørgen Rise [28].

The simulated environment was adapted so that it could be replicated in a real-world setting in a robotics lab with the 3D camera Zivid Two.

Training and hyperparameter tuning was conducted to create a high-performing reinforcement learning algorithm utilizing RGB-D images to reduce the sim-to-real gap.

Tests were performed to show that the RGB-D algorithm had a smaller sim-to-real gap than the RGB algorithm.

1.3 Report Structure

This master's thesis consists of the following chapters.
Chapter 2 - Background Provides the reader with fundamental concepts, theory, and related work, enabling the reader to comprehend the work presented in the following chapters.
Chapter 3 - System Configuration Describes the key technologies, design choices, and frameworks built and used to train and test reinforcement learning algorithms in this thesis. This chapter also presents specific parameter choices utilized for training, such as the reward function, action and observation space, normalizing, hyperparameter tuning, and processing of depth information from the real-world camera.
Chapter 4 - Experimental Setup This chapter explains the setup used for conducting the experiments. This includes both the setup of the simulated environment, algorithms used for training, and the real-world testing setup.
Chapter 5 - Results Presents the results from both the simulated training and

the real-world performance of the grasping algorithms.
Chapter 6 - Discussion This chapter presents a discussion of the results and evaluates the performance of the algorithms.
Chapter 7 - Conclusion Concluding remarks and further work.

Chapter 2

Background

This chapter starts with a review of the field of robotic grasping. Then basic theory about reinforcement learning is presented, followed by the key equations forming the PPO algorithm later utilized for training in this thesis. Then, a discussion of the challenges related to using reinforcement learning for solving robotic grasping tasks is presented. This part focuses on how sample efficiency problems can be solved by closing the sim-to-real gap utilizing synthetic data. The chapter is concluded with a review of related work. This chapter is based on and includes parts from the Robotic Grasping chapter from the specialization project [16] preceding this paper.

2.1 Robotic Grasping

This section discusses the fundamentals of robotic grasping. Common approaches used in robotic grasping and relevant theory will be presented. This section is based on and includes parts from the Robotic Grasping chapter from the specialization project [16].

A grasp can be defined as obtaining complete control of an object's motion. Control of the object is achieved by restraining the motion by applying forces at specific contact points of the object.

Robotic grasping is a complex task. There are several challenges related to robotic grasping in general, which makes it a difficult task, but the lack of observables and the large action space are two of the main hindrances for creating successful algorithms. Cameras and different 3D sensors are the only observables available for information about the target objects. Images are challenging for computers to interpret and lead to ambiguity about object pose, material properties, shape, and mass.

The robot's end-effector, in this case, the gripper, can have many dimensions. For example, the Allegro hand (see figure 2.1) has four fingers with three joints each. This makes a total of 12 dimensions. The wrist posture has an extra 6 degrees of freedom due to the position of the end effector. In reinforcement learning, this will create a larger action space, which will take longer to explore and complicate the learning process.

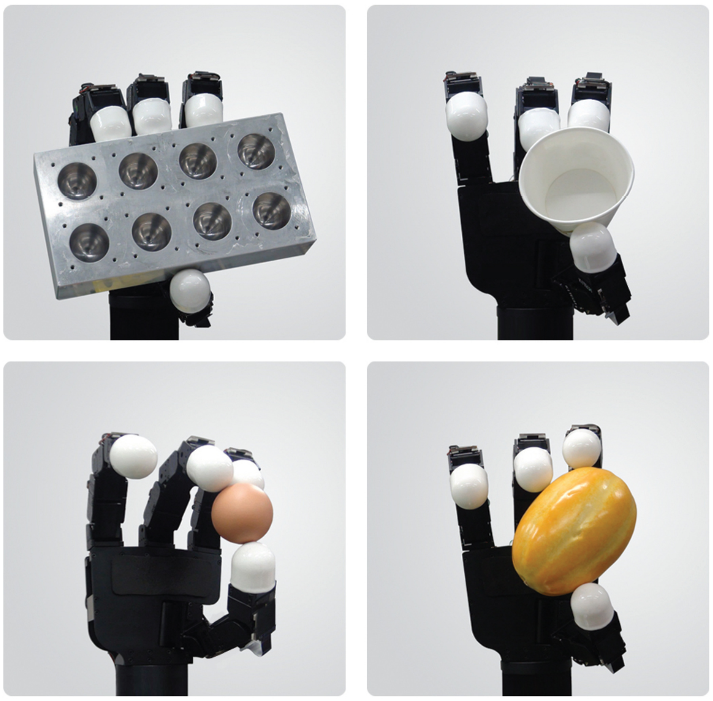


Figure 2.1: The Allegro Hand. (Image is from `wiki.wonikrobotics.com`.)

2.1.1 Generalized Autonomous Grasping

Generalized autonomous grasping is a robot's ability to pick up any given objective within a reasonable size and mass. To achieve this, we need some way for the robot to interact with its environment. I.e., a robot arm with a suitable end-effector mounted. The robot needs some way of sensing its environment. This is often solved with a camera. Either wrist-mounted or over the shoulder. The robot also needs some algorithm for mapping the sensory information to some action that can allow the robot to grasp the object presented. Cameras are the preferable source for sensing the state space. Normal RGB and RGB-D cameras are commonly used when developing autonomous grasping systems.

In addition, approaches can be further divided into model-based and model-free approaches. Model-based approaches use information about the object, for instance, a CAD model of the target object, when evaluating and choosing different grasp strategies. Model-free approaches do not utilize any pre-known information about the objects when grasping.

2.1.2 Model-Free Robotic Grasping

Model-free approaches can generalize to unseen objects very well [30] and have grown into becoming the main direction of research within robotic grasping. The primary difference from model-based approaches is that model-free approaches do not use any prior knowledge about the objects.

A more traditional approach to model-free robotic grasping has been to divide the process into a series of steps. First, the robot senses the state, defines a suitable grasp, and then plans and executes the grasp [19, 21]. This allows the problem of sensing and deciding on a grasp strategy to be completely separated from the problem of controlling the robot and executing the grasp. This can significantly simplify the problem [14]. Studies have shown that deep neural networks applied to large datasets of pre-labeled grasps can successfully calculate new grasps on novel objects directly from sensory inputs like images and point clouds [12, 25]. Creating these datasets requires either tedious human labeling or many months of training on a physical system. A major drawback of separating the grasp strategy from the control of the robot is that this approach cannot react to changes in the environment or refine its strategy while executing the grasp. These methods don't resemble the grasping behavior seen in animals and humans. The grasping process of a human is a dynamic process where the plan and actions are continually updated based on the latest information obtained by the senses. This approach is more robust to unpredictable information.

2.1.3 Reinforcement Learning for Robotic Grasping

Reinforcement learning has shown very promising results within robotic grasping. By processing raw sensory inputs, such as images, it manages to create control policies by trial and error automatically. A major advantage of reinforcement learning is the ability to learn pre-grasp manipulations. Techniques like pushing and shifting can be very useful when dealing with objects in hard-to-grasp poses, cluttered environments, or objects stacked together. Deep reinforcement learning is reinforcement learning where deep neural networks are used. Deep reinforcement learning has been accelerated by the increased capacity of modern computation tools and resources. Deep neural networks are more capable of dealing with the large state and action spaces within vision-based robotics. High-dimensional sensory inputs can be mapped to control outputs utilizing end-to-end joint training for both perception and control

2.2 Reinforcement Learning

This section presents the fundamentals of reinforcement learning. The theory and equations forming the basis of the popular PPO algorithm utilized for experiments in this thesis will also be discussed. Section 2.2 until 2.2.8 is based on the reinforcement learning chapter from the specialization project written as preparations for this master thesis [16].

Reinforcement learning is a type of machine learning that utilizes an agent that interacts with an environment and learns through trial and error. It uses feedback from the environment to learn whether the chosen actions were good or bad. The environment, robots, tasks, objects, etc. differs greatly from different applications. This makes supervised learning impractical as it would require a labeled data set to be created for every new application. Reinforcement learning creates the data during training and is practical when data collection and labeling are challenging. Recent advances in reinforcement learning like [22] and [31] where reinforcement learning was used to master the games of chess and go have given reinforcement learning a great gain in popularity in recent years.

2.2.1 Reinforcement Learning Fundamentals

In reinforcement learning, we have an *agent* that operates in an *environment*. The agent chooses which actions to conduct within the environment and receives feedback based on the outcome of its actions. This feedback is called *reward* and is given out based on specified parameters of how well the agent is accomplishing the goal of the task. The agent's objective is to maximize its reward.

The environment and agent are always in a state $s \in S$. This state represents all aspects of the current situation that the agent is in. In a robotic setting, this might be the geometry of every object present in the environment and all physical parameters. The actions that the agent can perform are often noted $a \in A$. Where A is the *action space*. The action space defines every possible action that the agent can choose. For example, if the agent is controlling a robot by sending out joint positions, then the action space would be

$$[\text{min angel}, \text{max angel}] * \text{number of joints}$$

The action space can be discrete, continuous, or include both types of variables. The actions allow the agent to change the state s.

The agent uses a policy π in order to choose an action a given a state s. The policy is a mapping from states to actions. The policy chooses the action that gets the highest reward given the state s. The agent gets a reward R after every timestep t. R is a function of the state and the observations. The reinforcement learning structure can be modeled as seen in figure 2.2.

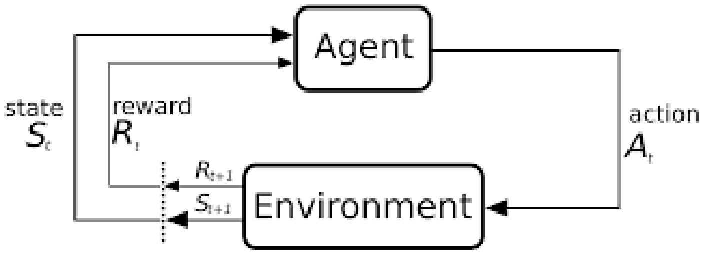


Figure 2.2: The RL structure (credit:https://en.m.wikipedia.org/wiki/File:Markov_diagram_v2.svg).

An episodic setting is when the task is restarted when the task ends. It's important to note that the agent chooses its action not just based on the next reward. The agent's objective is to maximize the cumulative reward for the whole duration of the episode. We want to maximize the total return J.

$$J = R_t + R_{t+1} + R_{t+1} + R_{t+2} + R_{t+3}... \tag{2.1}$$

The agent needs to learn the relationship between rewards, actions, and states. This is done through exploration. Exploration is either embedded directly in the policy, or it can be done separately and only during the learning process. Exploration is challenging to implement since the agent needs to weigh between choosing known actions that will lead to relatively high rewards or exploring and possibly ending up with new strategies that might lead to even higher rewards.

2.2.2 Markov Decision Process

Reinforcement learning approaches are based on the Markov Decision Process. The Markov property is when the next state and the reward only depend on the previous state s and action a [29]. This can be expressed mathematically as:

$$P[S_{t+1}|S_t] = P[S_{t+1}|S_1, ..., S_t] \tag{2.2}$$

The Markov Process builds on the Markov property and is a series of states where the states only depend on the previous state and its action. The environment may change state because of the agent's actions. The mapping between actions and the new state is called the transition probability or function. The transition function can be defined as the probability of ending up in a new state s' after performing action a in state s. This can be denoted as:

$$T(s', a, s) = P(s'|s, a) \tag{2.3}$$

The following equations must be satisfied to define a proper probability distribution over possible next states:

$$0 \leq T(s, a, s') \leq 1 \tag{2.4a}$$
$$\sum_{s \in S} T(s, a, s') = 1 \tag{2.4b}$$

The idea behind the Markov Decision Process is that the current state s gives enough information to make an optimal decision. It is not important which states and actions the system has been in before s.

2.2.3 Optimal Criteria and Policy

As previously stated, the goal is to have a policy that given all possible states $s \in S$ can choose the actions a that maximize the rewards.

In a real-world setting, we will always have stochastic noise. Therefore we can not know the reward exactly given some action. It is therefore normal to use the expected return of the reward function. The goal of reinforcement learning is therefore to find the optimal policy π^* that maps states to actions that maximize the expected rewards J. There are different optimal behaviors depending on how we define J. The finite horizon is an optimal criterion that attempts to maximize the expected reward for the finite time until the horizon T for t steps:

$$J = E\left\{\sum_{t=0}^{H} R_t\right\} \tag{2.5}$$

The discounted finite horizon is a different optimal criterion where a variable γ (with $0 \leq \gamma \leq 1$) acts as a discount factor:

$$J = E\left\{\sum_{t=0}^{H} \gamma^t R_t\right\}. \tag{2.6}$$

The γ factor will decrease the importance of rewards that are further back in time. This increases the value of the rewards earlier in the time, creating a policy that will strive to claim its rewards faster. An example is if we are trying to learn a robot to pick up a square. The reward function is simply 1 for a successful grasp and 0 if no grasp is detected. In this case, the reward will be higher for the agent if it manages to get a successful grasp within 10 seconds than if it is successful within 20 seconds. The agent will try to find a policy that is as fast as possible while still successful.

If the γ is too small the policy will be myopic and greedy and could lead to poor performance [17].

2.2.4 Value Function

Value functions are methods that estimate the value of being in a given state. It tries to estimate the discounted sum of rewards from this point onward. The state-value function links the optimality criteria to policies. The value of a given state can be expressed as the expected value of the reward given that the agent will follow the policy π. With the discounted finite horizon optimal criteria the value function is expressed as:

$$V^{\pi}(s_t) = E_{\pi}\left\{\sum_{t=0}^{H} \gamma^h R_t | s_0 = s\right\}. \tag{2.7}$$

Where the value metric is the expected sum of rewards.

The value function is represented by a neural net and it's frequently updated during training using the data that our agent collects. Because the value estimate is the output of a neural net it is going to be a noisy estimate. There will be some variance because our network will not always predict the exact value of that state.

2.2.5 Q-Function

A similar function to the state-value function is the state-action value function, also known as the Q-function. This function returns the value of a given state and action. It is defined as the expected return when taking action a from state s and following the policy, π from there:

$$Q^{\pi}(s,a) = E_{\pi}\left\{ \sum_{t=0}^{\infty} \gamma^{h} R_t | s_0 = s, a_0 = a \right\}. \tag{2.8}$$

2.2.6 Advantage Function

The advantage function measures how good or bad a decision is given a certain state. It is expressed as the state-value function minus the value function.

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

The advantage function tries to estimate the relative value of the selected action in the current state. It's an estimate of how much better the chosen action is than the expectation of what would normally happen in the state it was.

2.2.7 Model-Based RL

An important difference between different approaches is whether the agent learns or has access to a model of the environment. If so, the RL algorithm is a Model-Based approach. A model of the environment is defined as a function that predicts the state transitions and rewards.

2.2.8 Model-Free RL

A model-free approach is the alternative to model-based learning. Model-free approaches dose not involve a model of the environment. They learn the value functions and optimal policies directly from interacting with the environment. Model-free methods are often easier to implement and for tuning hyperparameters. Due to these advantages, they are often used more than model-based methods [23].

There are two main branches model-free learning, policy-based and value-based. In value-based methods the agent learns the state-action value function (Q-function), $Q(s,a)$ for the optimal $Q^*(s,a)$. This optimization is often performed off-policy. This means that the agent can use data collected at any time during the training when updating the policy.

In policy-based methods, a policy is explicitly represented as $\pi(a|s)$. The parameters θ are optimized either trough gradient descent or an objective function $J(\pi_\theta)$. Policy optimization techniques are almost always on-policy. On-policy means that the agent always uses the latest policy when creating data which is used for updating the policy.

2.2.9 Policy Optimization

Policy gradient algorithms are algorithms that aim to update the policy by gradient ascent 2.10.

$$\theta_{k+1} = \theta_k + \alpha \ \nabla_\theta J(\pi_\theta)|_{\theta_k} \tag{2.10}$$

$\nabla_\theta J(\pi_\theta)$, is called the policy gradient. Given a trajectory $\tau = (s_0, a_0, ..., s_{T+1})$ the probability for the trajectory from π_θ is

$$P(\tau|\theta) = \rho_0(s_0) \prod_{t=0}^{T} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t) \tag{2.11}$$

we can use 2.11 to calculate an expression for the policy gradient:

$$\nabla_\theta J(\pi_\theta) = \underset{\tau\sim\pi_\theta}{\mathrm{E}} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau) \right] \tag{2.12}$$

Since this is an expectation we can estimate it with a sample mean. By collecting a set of trajectories $\mathcal{D} = \{\tau_i\}_{i=1,...,N}$ we can estimate the policy gradient with the expression:

$$\hat{g} = \frac{1}{|\mathcal{D}|} \sum_{\tau\in\mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau) \tag{2.13}$$

where $|\mathcal{D}|$ is the number of trajectories.

The policy gradient can be rewritten in different forms which all have the same expected value but different variances. The policy gradient has the general form

$$\nabla_\theta J(\pi_\theta) = \underset{\tau\sim\pi_\theta}{\mathrm{E}} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \Phi_t \right] \tag{2.14}$$

where Φ_t can be chosen to be equal to the advantage function [1].

$$\Phi_t = Q^{\pi_\theta}(s_t, a_t) \tag{2.15}$$

By subtracting the value function we get

$$\Phi_t = A^{\pi_\theta}(s_t, a_t) \tag{2.16}$$

This gives us a policy gradient widely used in most popular Policy Optimization algorithms. We can see that if the advantage function is positive, meaning that the actions that the agent took in the sample trajectory resulted in better than average return, we will increase the probability of selecting them again in the future when the agent encounters the same state. If the advantage function is negative, the opposite will happen and we'll reduce the likelihood of the selected actions.

2.2.10 PPO

This section will present the key equations and ideas behind the PPO algorithm.

Simply running gradient ascent 2.10 on one batch of collected experience will update the parameters in your network so far outside the range where this data was collected. The advantage function, which is a noisy estimate of the real advantage, will deviate severally from the real advantage, and the training process will not find a satisfying policy.

The PPO and TRPO algorithms were created in order to combat this problem. They are designed to take the biggest possible improvement step on a policy, without stepping so far that we accidentally cause the policy to collapse. TRPO solves this problem by using a complex second-order method and PPO uses first-order methods and some other tricks to accomplish this.

There are two variants of PPO. PPO-Penalty and PPO-Clip, we will focus on PPO-Clip. PPO-Clip has the following function for updating its policy

$$\theta_{k+1} = \arg\max_{\theta} \underset{s,a\sim\pi_{\theta_k}}{\mathrm{E}} \left[L(s,a,\theta_k,\theta)\right] \tag{2.17}$$

where L is given by

$$L(s,a,\theta_k,\theta) = \min\left(r_t(\theta)A^{\pi_{\theta_k}}(s,a),\ \ \mathrm{clip}\Big(r_t(\theta), 1-\epsilon, 1+\epsilon\Big)A^{\pi_{\theta_k}}(s,a)\right), \tag{2.18}$$

and

$$r_t(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} \tag{2.19}$$

Given a sequence of sampled actions and states $r_t(\theta)$ will be larger than 1 if the action is more likely now than it was in the old policy. It will be between 0 and 1 if the action is less likely now than it was before the last gradient step.

The objective function that PPO optimizes is an expectation over batches of trajectories of two terms where the first one is $r_t(\theta)$ times the advantage estimate. The second term is similar to the first one except that it contains a truncated version of $r_t(\theta)$ by applying a clipping operation between $1 - epsilon$ and $1 + epsilon$. Epsilon usually is around 0.2. The *min* operator is applied to the two terms to get the final result.

The min operator over the extra clipping expression acts as a regulator. If the policy tries to change too much, the clipped version will be used. The hyperparameter ϵ corresponds to how far away the new policy is allowed to change from the old.

Most modern implementations of this algorithm include other elements like clipping of value functions, normalization, and other tricks that will not be discussed in detail in this paper.

2.3 Reward Function

Finding a suitable reward function is challenging but very important for creating an environment where the agent can be successful. This section discusses the importance of the reward function, reward shaping, and related work.

In reinforcement learning, the agents try to maximize the accumulated long-term reward. The outcome of a trained policy heavily relies on the reward function itself. Defining a good reward function in robotic reinforcement learning can be very difficult [17]. If the reward signal always is the same, then the agent cannot determine which action is better. It often seems natural to only reward upon task achievement, but the agent might receive such a reward so rarely that it might never succeed. Reinforcement learning algorithms are also notorious for exploiting the reward function in ways that are not anticipated by the designer.

2.3.1 Reward Shaping

Reward shaping is engineering a reward function that guides the agent towards successful episodes [18]. Robotic grasping is a complicated task that requires the agent to achieve several sub-tasks. Locating the target object, moving the gripper close, finding a suitable grasp, and finally executing the grasp. [10] solved this problem with reward shaping. Creating good reward functions in robotics is often hard and demands good domain knowledge and trial and error.

2.3.2 Related Work

Qt-Opt [15] uses a sparse reward where the agent is rewarded only if it manages to execute a successful grasp. A completely random initial policy would have a very low chance of succeeding with an unconstrained action space, and the agent would require unrealistic large amounts of data to learn a successful policy. Qt-Opt utilized a weak scripted exploration policy to bootstrap data collection. This engineered policy was biased toward reasonable grasps.

Training in a simulated environment provides more freedom when engineering the reward function. This is because every metric that is simulated in the environment is available. To check if a grasp was successful, Qt-Opt had to take a picture when the robot had executed a grasp. Open the gripper so the grasped object would fall, close it, take another picture, and compare the images to check if they were different (successful grasp) or identical (not a successful grasp). In a simulated environment, it's possible to simply check the object's height by retrieving the object's location to check if the grasp is successful.

[10] trained a PPO agent in a robosuite environment. They designed a reward function consisting of three different rewards:

$$r = r_1 + r_2 + r_3 \tag{2.20}$$

where $r_1 \in [0, 1]$ is a metric of how far away the gripper is from the target object's position. $r_2 = 0.25$ if both the fingers of the gripper are touching the cube and $r2 = 0$ otherwise. $r_3 = 1$ if the target's center is higher than a certain value, indicating that the robot must be lifting the cube. $r_3 = 0$ otherwise.

The first reward , r_1, encourages behavior that brings the gripper close to the target object. r_2 rewards the agent if it manages to place the gripper around the target and make contact, and finally r_3 rewards a successful grasp. This is an example of reward shaping which is easy to construct in a simulated environment. The agent trained in [10] does not need any demonstration data, imitation learning or engineered policies to train a successful policy.

2.4 Robotic Grasping with Reinforcement Learning

In this section, we will discuss and look into some of the problems and challenges that arise when using reinforcement learning to solve robotic grasping challenges. This section is based on and includes parts of the work done in the specialization project presiding this paper [16].

Reinforcement learning has the last few years increased in popularity amongst researchers in the robotics community [36]. Reinforcement learning provides a framework and tools that allow robots to learn dexterous grasping end-to-end directly from raw sensory information like cameras.

In the context of robotic grasping, the action and state space are often continuous and high-dimensional. This causes some very difficult challenges when utilizing reinforcement learning. Some of the most notable problems in reinforcement learning for robotic grasping that will be discussed briefly are exploration vs exploitation, stable and reliable learning, and latency. Sample inefficiency and sim-to-real will be discussed in greater detail.

Exploration and Exploitation

Due to the high-dimensional action and state space in robotics and often sparse rewards, finding an efficient exploration method is often challenging in robotic grasping.

Adding demonstration data to the data buffer is a common technique for off-policy methods. This exposes the algorithm to high-reward behavior. Starting the training process with "scripted" policies is also a conventional method for initialization.

ϵ-greedy is a very common method for exploration in robotics. It is a simple method where the agent has a ϵ probability of exploring and a $1 - \epsilon$ probability of choosing the action that maximizes the return.

The amount of exploitation when using on-policy algorithms depends on the initial conditions. Further into training, the scale of exploration is reduced, and the policy favors exploitation over exploration. This might trap the policy in local optima.

Deterministic policies add noise to their training actions to create an exploration effect. This approach will not perform sufficiently when dealing with spares and deceptive rewards.

Stable and reliable learning

It is common that the performance of reinforcement learning methods depends on careful settings of the hyperparameters, making them difficult to use in practice. This problem is extra notable for off-policy algorithms, which are useful in robotic grasping due to their sample efficiency. The problem of stable and reliable learning can be categorized into two main challenges. Reducing sensitivity to hyperparameters and reducing issues related to local optima and delayed rewards.

Designing algorithms that automatically tune their own hyperparameters or developing algorithms that are robust to hyperparameter settings by design are desirable solutions, but also very difficult to create. This requires deep knowledge about the reasons behind the sensitivity of current reinforcement learning algorithms. Using automated approaches to hyperparameter tuning is a common technique.

Reducing issues related to local optima and delayed rewards can be difficult as the reinforcement learning objective itself can present a challenging optimization landscape. This means that the usual benefits of over-parameterized networks don't necessarily resolve the issues relating to local optima.

Latency

The state of a real robotic system is continuously changing as it moves. This is contradictory to the MDP as it assumes synchronous execution. Latency is the measurement of the delay from when the state is measured until the new action is calculated and applied. The amount of latency depends on the hardware and complexity of the system. It is possible to account for latency by predicting the changes in the state using a learned dynamic model. Another approach within model-free methods is to include the previous action as a part of the state definition.

2.4.1 Sample Inefficiency

Sample inefficiency is a problem that is very limiting within reinforcement learning for robotic grasping. The root of the problem is that many robotic grasping tasks require new data to be collected for training. Collecting data with a real-life robot is very time-consuming.

Many reinforcement learning algorithms are constructed to learn a task from scratch. An end-to-end reinforcement learning algorithm that uses a camera to grasp might require extensive amounts of data just so that it manages to place its end-effector close to the targeted object. The remaining part of the task, exploring different strategies for performing the actual grasp, will probably require even more data.

Off-policy algorithms can reuse old episodes (data) and exploit useful information, but many SOTA algorithms are not proficient enough at this. On-policy algorithms require new data for every update step.

Data collection can be very time-consuming for robotic grasping. Creating real-world training data can be difficult as it requires a robot set up in a safe environment as the robot's behavior will be unpredictable. Robot hours can often be expensive as the hardware itself is expensive and the robot might need human supervision or even intervention to reset the episode/task. The robot itself cannot run faster than real-time, and a grasping situation can last around 10-30 seconds. Additional reset time might also occur. Even scaleable learning with multiple robots might require months of training in order to acquire a sufficient amount of training data [15].

One approach for overcoming the problem of sample inefficiency could be to simply generate more data. In a real-world setting, this can be done by simply using more robots to collect data simultaneously. Qt-Opt [15] did this successfully, but this approach requires expensive equipment and engineering efforts. See figure 2.3.

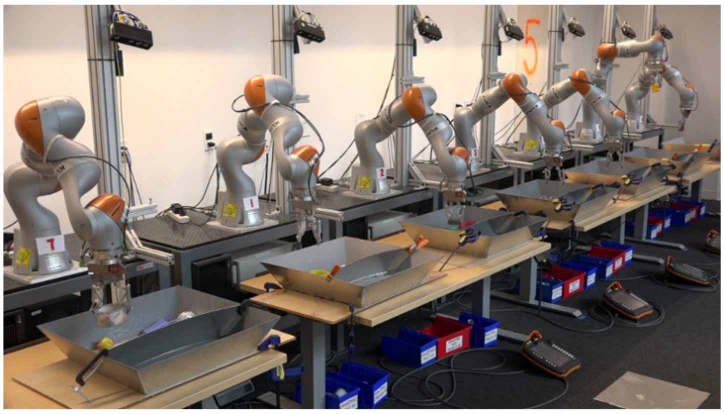


Figure 2.3: Seven robots simultaneously collecting data (credit [15]).

Another way to increase the amount of data is to add or solely use synthetic data. Synthetic data is most commonly created with simulators. As discussed later in 2.5.2 there is a gap between data created in the real world and data created in simulators, often called the Sim-To-Real gap. This gap causes successful policies in simulators to perform poorly in a real-world setting. This gap must be reduced to make synthetic data more useful.

2.5 Simulation

Simulation and different physics engines used for simulation are thoroughly discussed in the specialization project [16] and form the basis for this section. In this section, we explain what a simulator is and argue why simulated environments can drastically improve training efficiency.

As discussed in 2.4.1, one of the major disadvantages of reinforcement learning in robotics is the need for collecting huge amounts of data to train successful policies. This process can be drastically accelerated using simulated environments. Simulation allows multiple agents to train at the same time and to train at a speed faster than real-time. It also reduces the need for human intervention and allows for greater customization in training environments and episodes. Setting up training in simulation often requires less engineering effort and eliminates the safety hazard that follows real-life training with robots.

The domain gap between the simulated environment and the real world is a significant problem holding back the use of simulation. This is further discussed in chapter 2.5.2.

2.5.1 Robotic Simulators

To simulate training for tasks that will be executed in the real world, we need some sort of software that emulates the physical attributes and properties of the real world. We also need some way of recreating the physical environment that the real-world robot will operate in, in the simulated environment.

There are several commercial and open-source physics simulators available that have been created with the intent of being used for robotics. We will call these robotic simulators.

[7] defines a robotics simulator as:

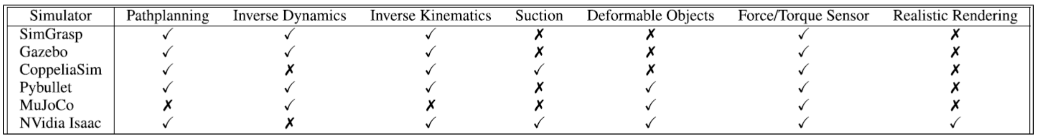
> an end-user software application that includes at least the following functionality:
>
> 1. Physics engine for realistic modeling of physical phenomenon.
> 2. Collision detection and friction models.
> 3. Graphical User Interface (GUI).
> 4. Import capability for scenes and meshes.
> 5. API especially for programming languages used by the robotics community (C++/Python).
> 6. Models for an array of joints, actuators, and sensors are readily available.

Robotic grasping is a precision task requiring the fine movement and contact modeling of rigid bodies. There is also a need for physics that handles contact in a precise and robust manner. This reduces the field of suitable simulators.

[7] also states that:

> To be useful for manipulation, research simulators must have actuator models for position control, velocity control, and torque control, as these are the most commonly used control modes for physical arms. The simulator needs to support torque sensors as well as visual sensors, namely RGB and RGB-D. Finally, built-in features that are relevant especially for manipulators are Inverse and Forward Kinematics solvers, and path planning.

[7] has made a table (see 2.1) comparing features of different simulators used in state-of-the-art research within robotic manipulation.



Simulator	Pathplanning	Inverse Dynamics	Inverse Kinematics	Suction	Deformable Objects	Force/Torque Sensor	Realistic Rendering
SimGrasp	✓	✓	✓	✗	✗	✓	✗
Gazebo	✓	✓	✓	✗	✗	✓	✗
CoppeliaSim	✓	✗	✓	✓	✗	✓	✗
Pybullet	✓	✓	✓	✗	✓	✓	✗
MuJoCo	✗	✓	✗	✗	✓	✓	✗
NVidia Isaac	✓	✗	✓	✓	✓	✓	✓

Table 2.1: Comparison of physics simulators (credit [7]).

2.5.2 Sim-to-Real Gap

The sim-to-real gap, also referred to as the reality gap, is the mismatch between data collected in simulated environments and real-world settings. This gap causes reinforcement learning algorithms trained on synthetic data(in a simulator) to perform worse when deployed in a real-world setting. This mismatch is caused by a series of factors, such as differences in sensing, actuation, physics, and the possibility of the agent being exposed to novel experiences in the real-world setting. Rendered images dose rarely look like their real-world counterparts. These differences are caused by the inability of simulators to create exact replicas of real-world physics and images.

Better and more realistic simulators will contribute to closing the gap for the actuation and account for variability in dynamics and physics. The gap in sensing is a bit more complex as this also involves the problem that the agent can face situations in the real world that haven't appeared in the simulator.

2.5.3 Closing the Sim-to-Real Gap

Closing the sim-to-real gap could change how we approach reinforcement learning for robotic grasping. It would allow algorithms to be trained entirely in simulation and therefore eliminate the problems related to sample inefficiency by allowing reinforcement learning engineers and researchers to create as much data as their data resources allow. Training in simulation would also lower costs related to robotics hardware, and setups and eliminate the risk of accidents related to training robots in the real world. Closing the sim-to-real gap attracts multiple researchers and their efforts. Publications within this field have increased by several orders of magnitude over the last few years [36].

Modern simulators are becoming more complex and realistic, but they are still not a sufficient representation of the real world. It is challenging to create high-quality rendered images in simulations. This is especially problematic within robotics, where the main sensor information comes from cameras.

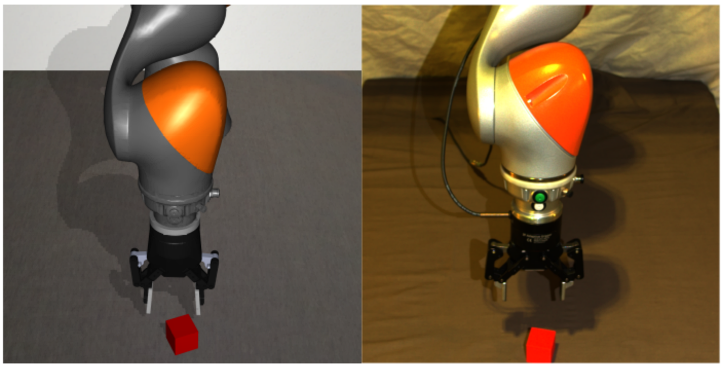


Figure 2.4: Simulated image of the KUKA IIWA R820 compared to a real photo of the robot.

Humidity, temperature, positioning, and wear-and-tear might change the real robot's physical parameters, making it deviate from the simulated version.

2.5.4 Domain Randomization

Domain randomization is well documented to be essential for having a successful sim-to-real transfer. Domain randomization is a technique that aims to narrow the reality gap of the robotics simulation. This method consists of using some random parameters in the simulator instead of trying to match them exactly to the real world. For example, modeling the friction can be very hard, so instead of carefully modeling the friction coefficient of different surfaces, the domain randomization approach would be to use random friction coefficients within a somewhat plausible interval. This will reduce the possibility of the agent relying too much on a bias in the training data, which doesn't exist in the real world. Introducing perturbations in the form of randomization of different factors will make the agent more resilient to the mismatches between reality and simulation.

We can divide domain randomization into two different types. Visual randomization and dynamics randomization.

Visual Randomization

Rendered images in simulators will always have different textures, lighting, and camera positions than images from the real world. This makes creating training data in simulators for vision tasks extra challenging. Visual randomization can therefore be applied so that the agent has trained images with large variability so that it can generalize when operating in a real-world setting.

Dynamics Randomization

Dynamics randomization is a method for closing the sim-to-real gap which involves using some randomization for physical parameters like surface friction, robot joint damping coefficients, actuator force gains, object dimensions, and masses. MuJoCo [20] and other physics simulation platforms must simplify the underlying physics model to achieve acceptable runtime speeds. Many of the physics parameters such as friction, damping, and contact constraints do not fully capture real-world dynamics due to the simplified model. To compensate for this, it is important to utilize randomization for important parameters in the physics model. Open AI [3] managed a successful sim-to-real transfer for dexterous in-hand manipulation tasks with a five-finger hand by applying the above-mentioned techniques.

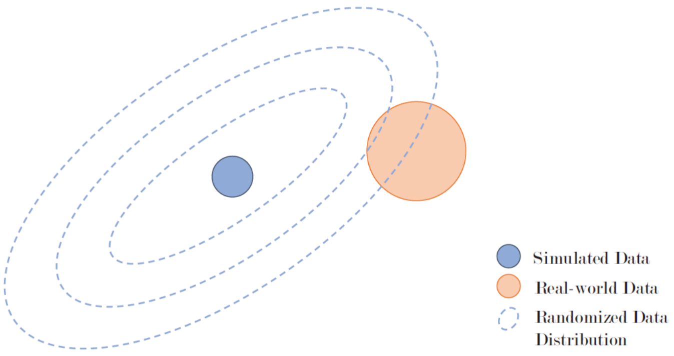


Figure 2.5: A model illustrating the intuition behind domain randomization (credit [36]).

2.6 Related Works

Previous work within the field of reinforcement learning for robotic grasping that is relevant to this thesis is presented in this section. This section is a revised version of the findings from the specialization project presiding this paper [16].

The field of robotic grasping lacks standardized benchmarks for evaluating the performance of robotic grasping systems. This is unfortunate as comparing different methods and approaches becomes more challenging and subject to uncertainty. The grasp success rate is a common measurement of performance. Most researchers use tests where the success rate is measured when grasping objects lying still. This reported grasp success is the metric used for comparing the performance of different known robotic grasping systems in this paper.

2.6.1 Reinforcement Learning Approaches

QT-Opt [15] is a general-purpose reinforcement learning algorithm created to solve grasping tasks. The algorithm is a closed-loop system. Qt-Opt is constantly re-planning its next move depending on the input from the camera. This is because the policy is learned by optimizing the reward across the entire trajectory, which allows the policy to learn complex behaviors. This includes pregrasp manipulation, dealing with cluttered scenes, learning retrial behaviors as well as handling environment disturbance and dynamic objects. Some of these maneuvers are illustrated in figure 2.6

There are several other interesting features of the Qt-opt system. It has a very flexible action space. The gripper is commanded in all degrees of freedom in 3 dimensions. It operates directly on raw RGB observations from an over-the-shoulder camera which doesn't need to be calibrated.

By learning from data collected from previous experiments (offline data), Qt-Opt reached a 86% grasp success. This data consisted of 580K real-world grasp episodes/attempts collected with seven different KUKA LBR IIWA robot arms with two-fingered grippers. The data was collected over the course of four months, with a total of about 800 robot hours. With an additional 28000 grasps collected for fine-tuning the joints of the robot, the algorithm achieved a grasp success of 96%.

Another method that achieved high grasping success is Grasping in the Wild [32]. [32] created a new low-cost handheld device for collecting grasping demonstrations. This device utilized a gripper and an RGB-D gripper-centric camera (wrist mounted). They collected about 12 hours of gripper-centric RGB-D video using this hardware (see figure 2.7). From this data, 6 degrees of freedom grasping trajectories were recovered using classic visual tracking algorithms. This data was

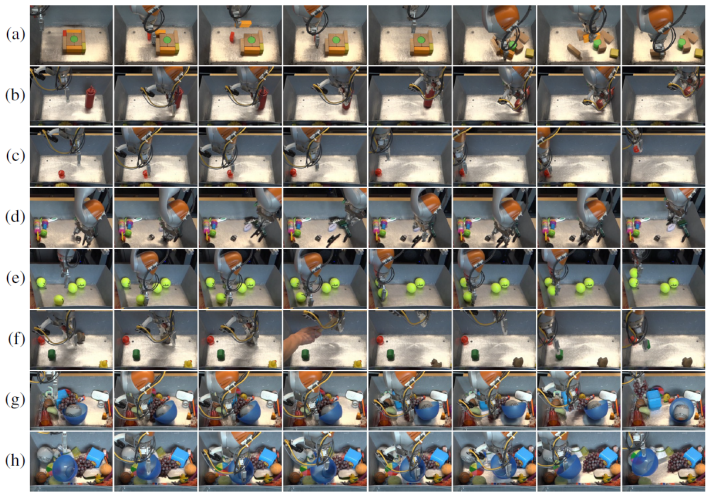


Figure 2.6: Eight grasps from the QT-Opt policy, illustrating some of the strategies discovered by their method: pregrasp manipulation (a, b), grasp readjustment (c, d), grasping dynamic objects, and recovery from perturbations (e, f), and grasping in clutter (g, h). (credit[15])

used to train a robust end-to-end closed-loop grasping model with reinforcement learning. A deep neural network was used to model a value function that maps the images from the camera to the expected rewards in that state. An important aspect of this method is that it utilizes "action-view" based rendering to simulate future states with respect to different possible actions (simulating what the camera would see if it moved forward or sideways). These states are evaluated using the learned value function in a closed loop while executing grasps to predict how the gripper should move in the next step to maximize rewards. The algorithm was fine-tuned on a real robot platform using trial and error with standard off-policy Q-learning to bridge the domain gap between data collected from human demonstrations and data from the real robot. This method achieved a grasp success of 92%.

2.6.2 Training in Simulation

There have been several attempts to develop approaches that train on data collected completely or partially in simulated environments. Trying to close the reality gap when using synthetic data is crucial.

A common approach for closing the reality gap is to use generative models to translate simulated images into realistic images. This type of translation is often task-agnostic, meaning that all features relevant for solving the task at hand might

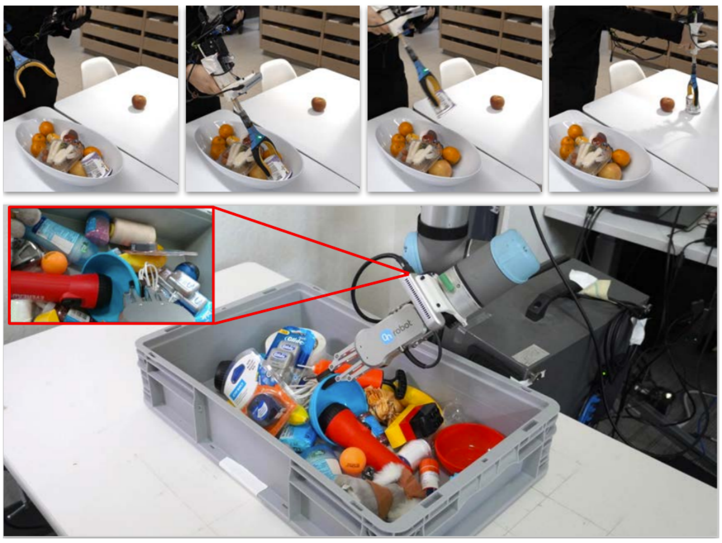


Figure 2.7: Demonstration of the handheld device for collecting grasping demonstrations and the robot trained with demonstration data. (credit[32])

not be intact after the translation. [27] introduces RLscene consistency loss, which ensures that the translation operation doesn't affect the Q-values associated with the image loss. This enables the possibility of creating task-aware translation. An approach for simulation-to-real-world transfer for reinforcement learning, RLCycleGAN, was developed using this loss. The RLCycleGAN translation was tested with Q-learning on a simulated robotic grasping task. The benchmark standard simulator, without any adaptation, resulted in a policy that achieved 21% grasp success in the real world in contrast to 95% in the simulator. See table 2.2 for results compared to some other GAN methods. [27] concludes that:

"RLCycleGAN offers a substantial improvement over a number of prior methods for sim-to-real transfer, attaining excellent real-world performance with only a modest number of real-world observations."

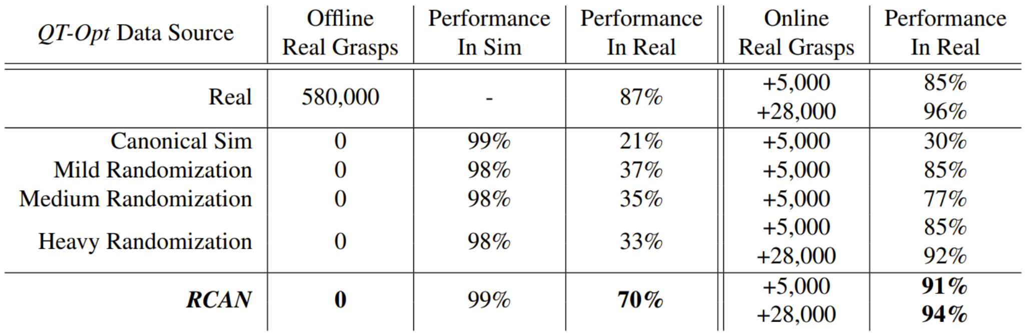
A different method also trying to close the reality gap is the RCAN method [13]. Randomized-to-Canonical Adaptation Networks is an approach trying to close the visual reality gap. This method uses no real-world data. An image-conditioned generative adversarial network learns to translate randomized rendered images into their equivalent non-randomized, canonical versions. A robotic grasping framework can then train using the canonical versions of the images. When deployed in a real-world setting, the RCAN can also translate the real-world images into canonical versions. This allows the algorithm to infer on data that is similar to the data it has been trained on. This approach for closing the reality gap was

Table 2.2: Comparison of achieved grasping success between different translation techniques on a benchmark simulated grasping task (credit[27])

Simulation-to-Real Model	Grasp Success
Sim-Only	21%
Randomized Sim	37%
GAN	29%
CycleGAN	61%
GraspGAN	63%
RL-CycleGAN	70%

deployed with the Qt-Opt algorithm and tested with different amounts of randomization in the training data and with on-policy joint fine-tuning with real-world grasp attempts. See table 2.3 for the results. RCAN achieves a grasping success of 70% in the real world only utilizing simulated training data, which is over double the success compared to not using RCAN.

Table 2.3: Average grasp success rate on test objects after 102 grasp attempts on each of the multiple Kuka IIWA robots. The first four columns of the table highlight the performance after training on a specified number of real-world grasps. Zero grasps imply that all training was done in simulation. The last two columns highlight the results of on-policy joint fine-tuning on a small number of real-world grasps. (credit[13])



QT-Opt Data Source	Offline Real Grasps	Performance In Sim	Performance In Real	Online Real Grasps	Performance In Real
Real	580,000	-	87%	+5,000	85%
				+28,000	96%
Canonical Sim	0	99%	21%	+5,000	30%
Mild Randomization	0	98%	37%	+5,000	85%
Medium Randomization	0	98%	35%	+5,000	77%
Heavy Randomization	0	98%	33%	+5,000	85%
				+28,000	92%
RCAN	**0**	99%	**70%**	+5,000	**91%**
				+28,000	**94%**

SURREAL [10] is an open-source, scalable framework that supports state-of-the-art distributed reinforcement learning algorithms. It also includes the SURREAL Robotics Suite, which is a set of benchmark robotic tasks of varied complexity created in the MuJoCo physics engine. A version of the on-policy algorithm PPO was implemented in SURREAL and tested on a Block lifting task. The task was to lift a block of a table using a two-fingered gripper mounted on a Sawyer robot. The algorithm successfully trained an agent to repeatably manage to solve the task.

[33] tried to learn quadruped locomotion from scratch using simple reward signals. The focus was to close the reality gap. The control policy was learned in a physics simulator and then deployed on real robots. It was found that actuator dynamics and the lack of latency modeling were the main causes of the model error. By implementing simulated latency and developing an accurate actuator model, they managed to significantly narrow the reality gap

Chapter 3

System Configuration

This chapter describes the system and framework created to conduct reinforcement learning experiments for robotic grasping both in simulation and in a real-world setting. It also presents some of the essential elements, like the Zivid camera used for real-world image observations, hyperparameter tuning, the construction of the reward function, normalization, and action and observation space.

3.1 Simulation Framework

The simulation framework constructed for this master's thesis consists of a robosuite module, Stable-baselines3 module, and the MuJoCo physics engine. An overview of the architecture is presented in 3.1.

3.1.1 MuJoCo

This section is from the specialization project [16]. MuJoCo [20] is a general-purpose physics engine that is made to be used in research and development, including the fields of robotics and machine learning. MuJoCo is a library written in C/C++ with an API written in C. The runtime simulation module operates on low-level data structures which are preallocated by the built-in XML parser and compiler. The MuJoCo library includes interactive visualizations rendered in OpenGL.

MuJoCo is a popular simulator in robotic research [7]. This is due to its contact stability. MuJoCo has been used to train policies for robotic manipulators in simulation, both for proof of concepts and for transfer to real-world systems. MuJoCo has many of the features that a simulation should have but doesn't have support for path planning and inverse kinematics.

MuJoCo and Pybullet are the most common physics engines in DRL research [35]. An advantage for MuJoCo is that it is known to be more efficient than Pybullet [9].

MuJoCo has the built-in ability to randomize the textures of rendered objects in simulation. It also has built-in functions for randomizing the characteristics of the camera. MuJoCo does not support multiple back-end physics engines.

It is worth noting that DeepMind, the well-known British artificial intelligence subsidiary of Alphabet Inc, bought the MuJoCo physics engine in 2021 for usage in robotics research and development. DeepMind is currently working to make MuJoCo open source and free for everyone to use [8].

3.1.2 Robosuite

Robosuite is a framework for simulating robot learning. Robosuite is powered by the MuJoCo physics engine. The motivation behind creating robosuite was the challenges of reproducibility and the limited accessibility of robot hardware. Robsuite was created to be:

- A standardized benchmark tasks for rigorous evaluation and algorithm development.
- S modular design that offers great flexibility to design new robot simulation environments.
- A high-quality implementation of robot controllers and off-the-shelf learning algorithms to lower the barriers to entry. [37]

Robosuite provides two main categories of APIs:

1. Modeling APIs for defining simulation environments in a modular and programmatic fashion.
2. Simulation APIs for interfacing with external inputs such as from a Policy or an I/O Device. [37]

3.1.3 Stable-Baselines3

Stable-Baselines3 (SB3) [26] is a library of different deep reinforcement learning algorithms implemented in Python [24]. SB3 provides a simple API that makes it easy to apply to novel tasks and environments. The algorithms are created following a consistent interface and accompanied by comprehensive documentation. The algorithms have been tested and verified against other codebases and published results by comparing the learning curves [26].

The SB3 implementations allow for multi-processing with 12/13 of the algorithms allowing for faster data collection, which is important due to the sample inefficiency problems related to using reinforcement learning for robotic grasping tasks.

The SB3 implementation is made to work with environments following the gym [6] interface.

3.1.4 Environment Wrapper

The robosuite environments must be wrapped to match the gym interface to function with SB3. To solve this, a custom wrapper inspired by robosuite's GymWrapper was created. Most of the wrapper was completely rewritten to utilize the observation and action spaces needed to create a simulated environment that matches the real-world setup.

3.1.5 Framework Overview

The Modeling API is used to specify a simulation model that the MuJoCo Engine instantiates. This creates a simulation runtime called the environment. The environment is then wrapped with the custom wrapper. A reinforcement learning model using the algorithm of choice is created with SB3.

The reinforcement learning model is used to start the data collection process. An episode consists of many steps. A step starts with the policy from SB3, using the observations to calculate the next action. The action command is then sent to the robot's controller, which is passed on to MuJoCo as torques. MuJoCo then simulates the step, and observations are generated through the sensors and sent back to the SB3 policy along with the rewards. This is illustrated in 3.1. This process is repeated until the length of the episode is reached.

The reinforcement learning model then utilizes all the actions, observations, and rewards collected from the episodes to update its policy. SB3 allows for running with multiple agents at once, meaning that the data collection process can be run in parallel on different cores allowing several agents to collect data simultaneously in separate environments.

3.2 Algorithm

The PPO algorithm was chosen as the preferred algorithm for creating a high-performing grasping policy that could be used for testing the sim-to-real gap on a robotic grasping task. PPO isn't designed specifically for sample efficiency but is relatively easy to tune compared to other high-performing algorithms. PPO still

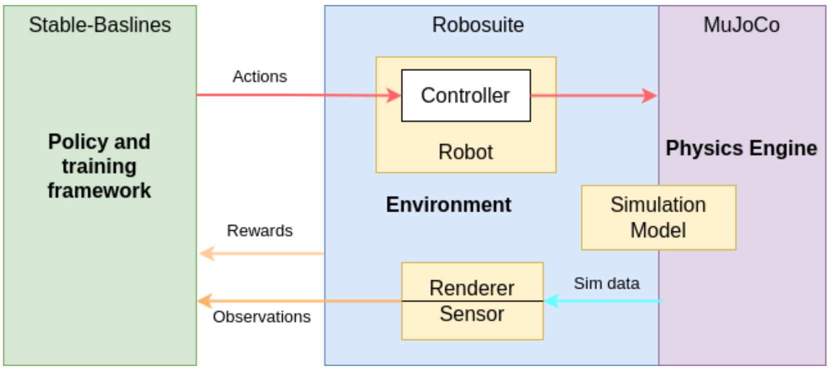


Figure 3.1: An overview of the framework architecture used for training in simulation.

achieves close to or equal to state-of-the-art performance on different benchmarks in deep reinforcement learning.

There is no specific strategy for exploring with PPO except for the policy being stochastic. The amount of exploring and randomness depends on initial conditions and training procedures [2]. The policy will typically become less and less random during training and exploit actions that have already been found to be rewarding. The SB3 implementation has an entropy term that helps to prevent premature convergence towards one action probability that would dominate the policy and prevent exploration, possibly causing the policy to get trapped in local optima.

3.2.1 Hyperparameter Tuning

A time-consuming task when setting up the experiments was hyperparameter tuning. Between 40-60 runs ranging from 1 to 72 hours in length were conducted to find hyperparameters that trained successful policies given the environment setup. The basis was the default hyperparameters used in [10]. Hyperparameters that were crucial for training successful policies when domain randomization was used were Buffersize and Bacthsize. There was a significant need for more data per learning step as the complexity of the environment rose with the added domain randomization.

Finding a suitable size and complexity for the neural network utilized, given the different sizes of the image observable, was also a demanding task. Training with a larger observation space by increasing the resolution of the images was not successful.

3.3 Observation Space

The observation space consists of 3 different inputs

- RGB-D or RGB image. The baseline was a [100,100,4/3] Matrix consisting of integers $\in [0, 255]$
- Robot end-effector pose. A vector consisting of three elements [x,y,z], Describing the end-effector position.
- Gripper Status. A single value. 0 or 1 describing if the gripper is closed or not.

3.4 Action Space

The first series of algorithms were trained with an action space consisting of the four following actions:

$$\text{Action space} = [\text{dx, dy, dz, (rotation around z-axis)}]$$

The angle of the end effector is fixed to zero around the x and y-axis. Allowing the agent only to control the rotation of the last joint, rotating the gripper around the z-axis. The rotation around the x and y axis is fixed to reduce the action space's size. Dx, dy, and dz is the amount of movement in the x, y, and z - direction.

These algorithms failed when deployed on the real-world system as the end effector rotation around the z-axis seemed to deviate too much from the simulated system and caused the system to stop due to conservative joint limits.

The algorithms, therefore, had to be trained with an action space consisting only of dx, dy, and dz and fixed rotation around the x, y, and z-axis.

3.5 Reward Function

As described in 2.3, the design of the reward function is crucial for the agent to succeed with training a successful policy.

The reward function designed for this grasping problem consists of three different rewards.

- R_1, a reward based on how close the gripper is to the target object. Range [0,0.5].
- R_2, "Detected grasp", a binary reward for grasping the target object. Either 0 or 0.25.

- R_3, "Successful grasp", a binary reward for lifting the target object. Either 0 or 2.25.

R_1 rewards the agent for how close it can bring the gripper to the target object. The function used for calculating this reward is:

$$R_1 = \frac{1}{2}(1 - tanh(d_{go}) * 0.5 \tag{3.1}$$

where d_{go} is the distance between the gripper and the target object. R_1 is close to 0.5 when the gripper is as close as it gets to the target object.

There is also a reward, R_2, which is equal to 0.25 if both the fingers of the gripper are in contact with the target object and zero otherwise.

The main reward is R_3, which is equal to 2.25 if the target object is lifted more than 0.04 meters above the table. A central detail is that the function used to check if there is a grip also has to be verified in order for this reward to be returned. This was incorporated due to some unwanted behavior in the earlier stages of this project. The reward function rewarded the agent every time the target object was higher than 0.04 meters from the table, resulting in the agent trying to hit the target object so that it would bounce off the table, and the agent would receive the "Successful Grasp" reward.

The R_1 reward is crucial to encourage movement close to the target objective in the early stages of training. A reward from R_2 and R_3 would be very rare, and the agent might therefore never learn a successful policy. The R_2 reward has the same effect as the R_1 reward and rewards the agent for placing the gripper around the target objective. This is necessary as "Successful grasps" would still be very rare even though the agent is encouraged to stay close to the object by the R_1 reward.

Another important aspect is that the episode does not terminate when the task is completed and the agent has achieved a successful grasp. If the episode was ended immediately after the grasp, the agent would learn to grasp the object, but not lift it off the table before the last timestep as this would increase the accumulated reward over the episode. By not ending the episode, we encourage the agent to grasp the target objective as quickly as possible and then continue holding it until the end of the episode.

3.6 Normalizing Observations and Rewards

The rewards and the observations, except for the images, are normalized with a moving average. The Images are simply normalized between 0 and 1 by dividing the input by 255.

Robosuite has a built-in wrapper called VecNormalize, which takes care of normalizing both the rewards and the observations for all the parallelized environments.

3.6.1 Observations

The observations are normalized with the following formula 3.2

$$obs_{norm} = clip\Big(\frac{obs - obs_{mean}}{\sqrt{obs_{var} + \epsilon}}, -10, 10\Big) \tag{3.2}$$

where clip() represents a clipping action, clipping the normalized observations between -10 and 10. Obs is the raw observation values, obs_{mean} is the running mean of the observations, obs_{var} is the running standard deviation and ϵ is a small constant to avoid dividing by zero.

3.6.2 Rewards

The rewards are normalized with the following formula 3.3

$$rew_{norm} = clip(\frac{rew}{\sqrt{rew_{var} + \epsilon}}, -10, 10) \tag{3.3}$$

where rew is the raw reward value and rew_{var} is the running standard deviation of the rewards.

3.7 Real-World Robot Control System

The real-world robotic system consists of a KUKA IIWA R820 robot with a Robotiq 2F 85 gripper. The control system was built with ROS2 and includes nodes for the camera, gripper(open/close), and the robot joints. The robot was directly controlled using KUKA Sunrise Workbench. More implementation details of this system can be viewed in [28].

3.7.1 Robot Control System

In order to control the robot and the camera, we have chosen to use Ros2. Ros2 is an open-source robotics framework for software developers. Our system consists of the main node, which subscribes to all the observations. It chooses an action by running the policy with real-world observations as input and finally publishes the chosen action to the robot.

The controller used to control the robot is not the same as the controller used in robosuite. Utilizing end-effector position coordinates might lead to the real-world robot choosing different joint positions to achieve the end-effector position, as the robot has 7 DoF. This enforces that the input to the real-world controller has to be joint positions as these are not ambiguous. The policy uses end-effector positions as action space as this reduces the complexity of the problem. An extra simulation step is therefore added to make the real-world system work. The chosen end-effector position is passed to a robosuite environment, and the time step is simulated. The final joint positions after executing the action are observed and fed to the real-world robot. This causes extra runtime, but the ambiguity of using different controllers in the real-world setup and simulation is alleviated.

The code was written to be simple, understandable, and suited for prototyping. Runtime was not considered an important factor. This resulted in a real-world system that runs considerably slower than the simulated system. A cycle in the real-world system takes around 2-4 seconds compared to 0.1 seconds in simulation.

3.7.2 Added Constraints

The real-world robot has very conservative joint limits in the current setup, which we could not change. This meant that the robot could achieve joint configuration where the robot had to be reset. Episodes where the robot reaches these configurations are terminated as the robot has to be manually reset.

3.8 Real World Camera Observables

This section presents the Zivid Two camera used when testing the system in the real-world. It also discusses the image quality and presents the data processing steps created to use the depth information.

3.8.1 Technology

Zivid Two is a state-of-the-art 3D camera that provides high-resolution and precis point clouds of even very small, densely packed, and highly detailed objects.

Zivid Two uses structured light to create its perception of depth. Structured light is a technique where a known pattern is projected onto the scene. The deformation of the light pattern when striking the surfaces allows a vision system to calculate the depth in the image. The Zivid camera consists of a projector that projects the light pattern onto the scene and a camera capturing images of the scene both with and without the pattern projected onto it. Some of the patterns projected onto the scene by the Zivid Two Camera can be seen in figure 3.2.

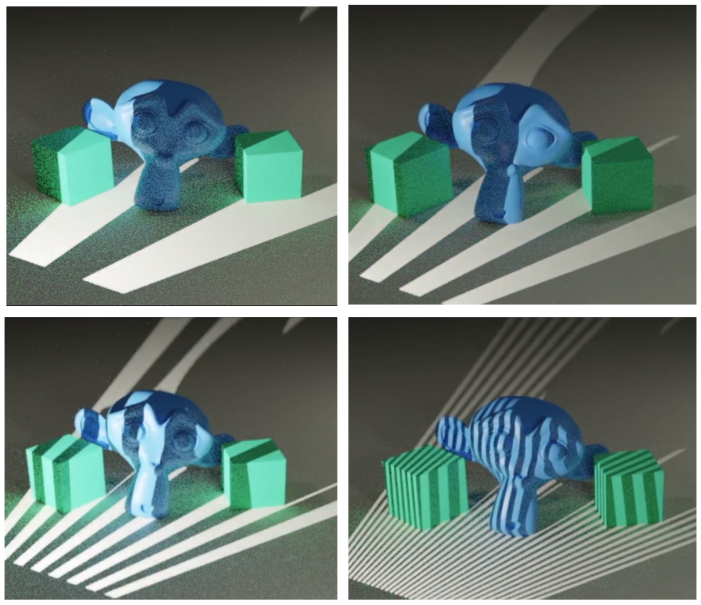


Figure 3.2: Examples of the structured light patterns used by Zivid Two

The strip engine uses the displacement of the patterns to calculate the depth of every pixel captured by the camera.

A common challenge especially in bin picking applications is that the other objects, the bin's walls, and corners will create interreflections from the light projected onto the objects you want to capture. This causes the decoded signal to be garbled and the point cloud becomes distorted [38]. New patterns in Zivid two enable the stripe engine to filter out the interreflections and recovers the unreflected signal. The stripe engine also aims to reduce the distortion which can occur with reflective and shiny objects like reflective cylinders or chrome plated parts.

3.8.2 Specifications

The Zivid Two camera has a capture time of 80 ms to 1 s. This includes the time from initializing the capture until the point cloud is ready to copy. The actual acquisition time can be shorter. A capture time of one second will create a relatively long time step in the Algorithm. A typical frequency in continuous control problems is 10 to 20 Hz.

Zivid reports the following general specifications about image quality and attributes.

Imaging	1944 x 1200 (2.3 MP)
Point cloud output	3D (XYZ) + Color (RGB) + SNR
Aperture (A)	f/1.8 to f/32
Gain (G)	63%
Projector Brightness (B)	0.25x to 1.8x
	1x = 360 lumens

Table 3.1: General specifications of the Zivid Two Camera.

The Zivid Two camera has also been carefully tested. Providing optimal working distance and precision.

Focus distance (mm)	700
Optimal working distance (mm)	500 to 1100
Field of view (mm)	754 x 449 at 700
Spatial resolution (mm)	0.39 at 700
	6.6 x 10^{-4} per distance (z) in mm

Table 3.2: Operating distance and field of view of the Zivid Two Camera.

Zivid also reports some measurements for point precision, polarity and trueness.

Point precision	1 σ Euclidian distance variation for a point between consecutive measurements at focus distance, 700 mm. [2]
Local planarity precision	1 σ Euclidian distance variation from a plane for a set of points within a smaller local region at focus distance 700 mm
Global planarity trueness	Average deviation from a plane in field of view at fous distance 700 mm.
Dimension trueness	70-percentile dimension error in field of view within optimal working distance and typical temperature range $< 0.30\%$

Table 3.3: Typical specifications for the Zivid Two Camera .
[2] Measured with Gaussian filter disabled.

3.8.3 Depth-Image Processing

The depth image captured by Zivid Two has large areas where depth information is lacking due to limitations caused by the structured light technique and several random points where depth information is lacking due to noise and reflections. An exmaple can be seen in figure 3.3.

The larger white areas to the left of the robot arm and at the left side of the gripper and gripper fingers are caused by the structured lighting being projected from the right side of the camera lens. These areas are therefor not to be hit by the structured lighting, causing the camera to have no measurement of depth. These areas and points are filtered out by their value and filled in by utilizing an inpainting method developed in [34] and implemented in cv2 [4].

The resulting image looks like 3.4

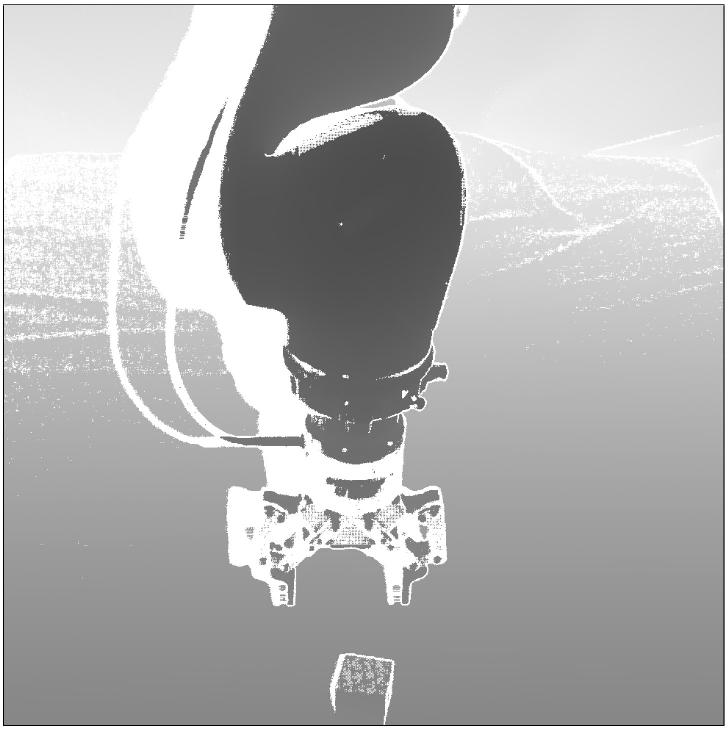


Figure 3.3: Raw depth image from Zivid Two.

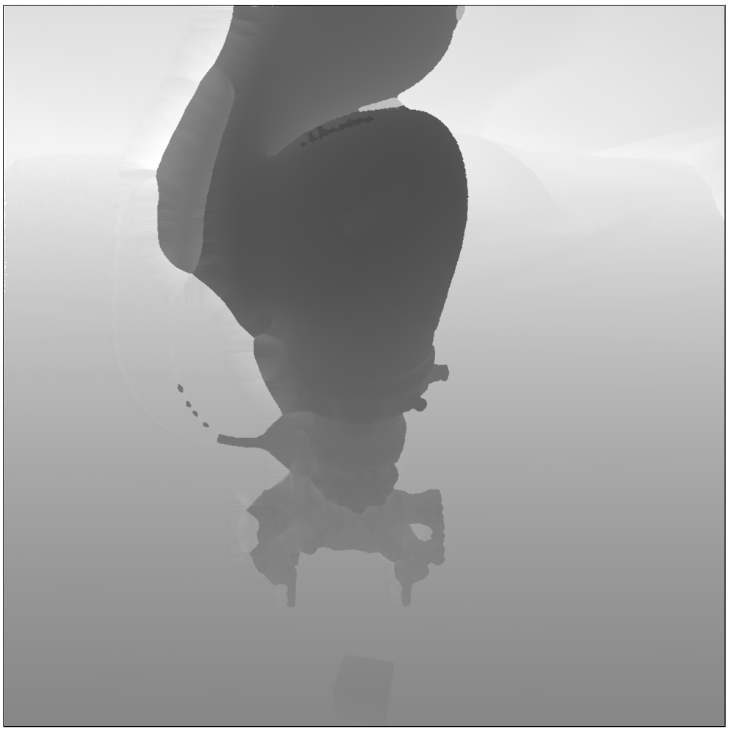


Figure 3.4: Inpainted depth observation.

The depth image was resized to 100 x 100 with bicubic interpolation over a 4×4 pixel neighborhood. The result is compared to the depth image from simulation in 3.5.

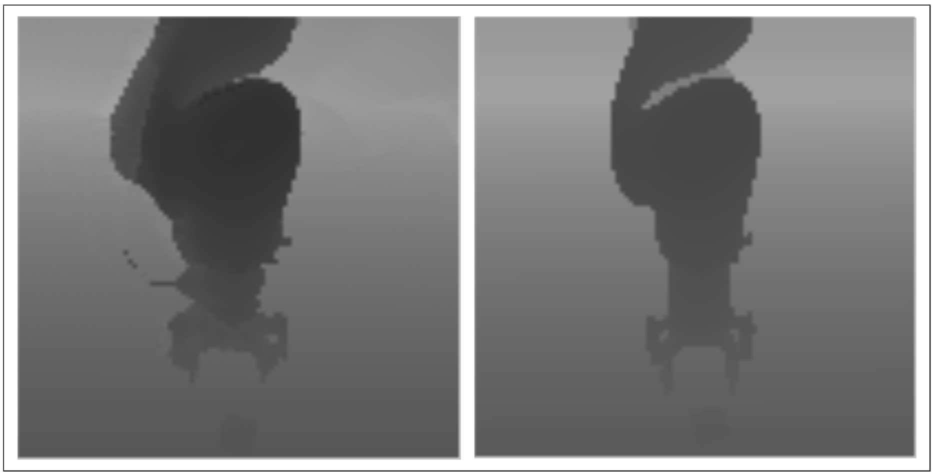


Figure 3.5: Comparison of Real depth Image **(Left)** and simulated image **(right)**.

The depth values are integers between 0 and 255. These intervals are so large that the reported precision and trueness from the Zivid Two camera have little to no effect on the observations. There is therefore no need to add extra noise in the simulated data.

Chapter 4

Experimental Setup

This section describes the setup for the experiments conducted in this thesis. First, the task and important parameters related to the algorithm are presented. Then follows a description of the simulated environment and the environment utilized for real-world testing.

To answer the overarching goal of how RGB-D image observation will affect the sim-to-real transfer compared to using RGB observations, we conducted an experiment comparing the sim-to-real gap for two algorithms. One was trained with RGB image observations, and one was trained with RGB-D image observations. The sim-to-real gap was measured by finding the average grasp success in simulation and comparing it to the grasp success rate when deploying the algorithm in the real-world setup. A sub-goal was to create a baseline policy (RGB) that achieved a high grasp success rate, $90 - 100\%$, in simulation to make the results relevant for other applications trying to achieve state-of-the-art success rates. In addition to the image observations, we utilized common observations such as gripper status(open/closed) and end effector position. The modern algorithm, PPO, which has proven to be successful and easy to use, was used to conduct the training. We tuned the hyperparameters empirically and trained the algorithms for 50 million time steps.

To review the sub-goals of this thesis, we will measure and evaluate the performance of the algorithms and provide a discussion on the performance of the simulation framework.

4.1 Task

The task that the agent is going to learn is a simple block-lifting task. The robot starts the episode by hovering the gripper above the table, as shown in figure 4.2. A cube is placed with a random rotation within a rectangle of 10 by 30 cm beneath the gripper. In simulation, the task is counted as successful if both the gripper fingers are placed on the cube simultaneously as the cube has been elevated from the table. In the real-world setup, the task is counted as successful if the human supervisor can see that the gripper has performed a grasp of the cube and the cube has been elevated off the table.

4.2 Algorithm and Training

The algorithm used for all experiments is the PPO algorithm implemented in stable-baselines3 [26].

4.2.1 Neural Network Size

We used a custom feature extractor for the image observations consisting of the following layers:

- Convolutional layer (n input channels,32, kernel size=7, stride=2, padding=3, bias = False)
- Batch Normalization 2d
- ReLU
- MaxPool2d (kernel size = 3, stride = 2)
- Convolutional layer (32, 32, kernel size=3, stride=1, padding=1, bias = False),
- Batch Normalization 2d
- ReLU
- MaxPool2d (kernel size = 3, stride = 2),
- Linear (in features=3872, out features=256, bias=True)
- ReLU

These parameters are from the RGB-D algorithm, but the layer types are identical to the RGB algorithm. The end-effector and gripper observations had no feature extractor and were fed directly into the first and only shared layer.

We used one shared layer (250,512) between the policy and value network. The rest of the policy and value network was identical except for the final output layer. We used one hidden layer of size 512 and one layer of size 256. The architecture of the neural networks can be seen in figure 4.1.

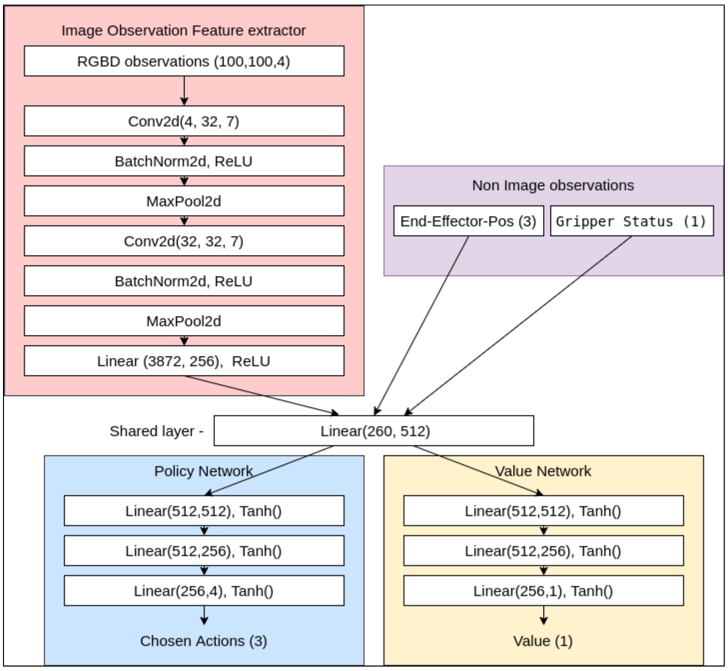


Figure 4.1: Neural network architecture

4.2.2 Computer Resources Utilized for Training

The Idun cluster was used to conduct all training for this master's thesis. The Idun cluster is a high-availability and professionally administrated computing platform for NTNU. Different nodes and GPUs were used to train depending on availability. The main bottleneck when training is the number of CPU cores available. Most of the training was done using 64 CPU cores and one NVIDIA A100 80Gb GPU.

4.3 Simulation Environment

This section describes the setup, variables, and parameters used for the training done in simulation.

4.3.1 Visual and Physical Description of the Environment

The environment created in robosuite consists of a table with legs and a back wall. A KUKA IIWA R820 with a Robotiq 2F 85 is placed on the table with the gripper hovering above the table. The target object is randomly spawned with a random rotation within a rectangle of 10 by 30 cm beneath the gripper.

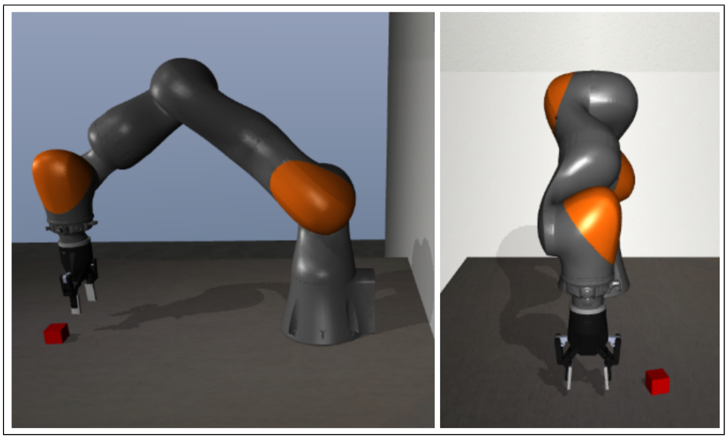


Figure 4.2: Images from the simulation environment.

4.3.2 Image Observations

The Image observations in simulation are a [100,100,4] matrix representing an RGB-D image. The Image observations in simulation are captured using a custom camera created through robosuite's interface. The camera is set to capture an [100,100] image. The camera placement in the real-world setting was calibrated, and the simulated camera is placed with the same extrinsic calibration found in the real-world setting.

Depth Image

The RGB-D image is created by concatenating an RGB image and a depth map recovered from the same camera.

By default, MuJoCo returns a depth map normalized in [0, 1]. Therefore, the original depth map must be recovered with a utility function from robosuite. Lengths up to 3 meters are then normalized between 0 and 255 and rounded off to integers. All lengths over 3 meters are set to 255.

Field of View

The Zivid Two camera has a field of view vertically (fovy) of 36°, but the light from the projector does not cover the whole area within the field of view. This can be seen in figure 4.3 where the large white areas at the top and bottom of the image mark areas where the real camera lacks depth information from the image.

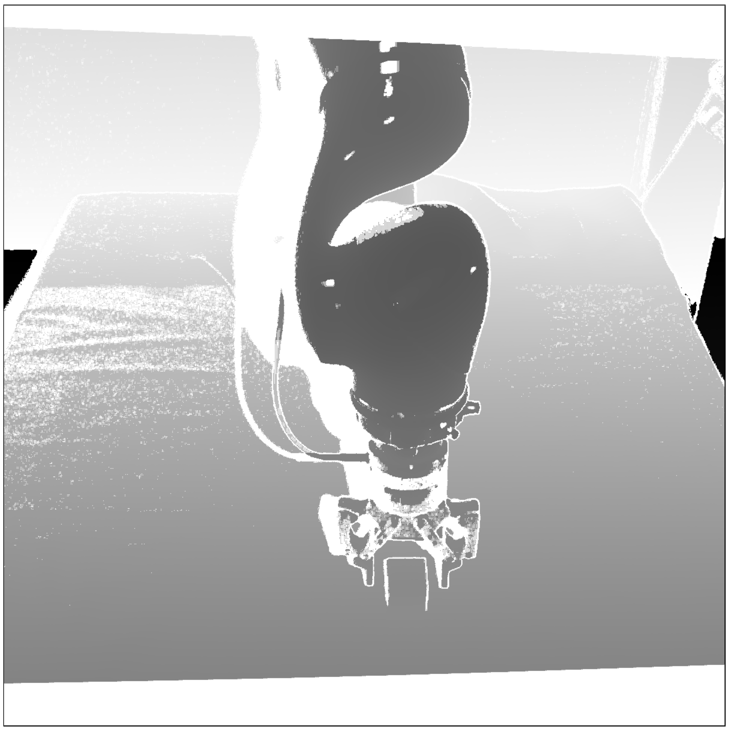


Figure 4.3: Raw depth image from Zivid Two.

We crop the images by a factor of $\frac{980}{1200}$ to remove the areas by the edges that lack depth information. Reducing the fovy to 29.4°.

4.3.3 Domain Randomization

Robosuite has a built-in domain randomization wrapper that wraps the environment and provides the possibility of passing custom arguments to different modder classes that can change environment parameters while training.

Visual randomization

Robosuite has three different Modder classes to control different aspects of the visual environment from which we have used different effects.

The CameraModder is important because there will be errors in the calibration of the camera in the real world. By adding some random values to the camera's placement, we can make the policy less prone to imperfections between the real and simulated placement of the camera. We used the following settings for the CameraModder:

- Randomized position, random value between +-0.01 m in [x,y,z] dimension.
- Randomized rotation, around 1 degree in all three axes.
- Randomized Fovy, random value around 1-2 degrees.

The TextureModder is used for randomizing visual objects' appearances. This includes texture, color, and material properties. We used the following settings for the TextureModder:

- Color of the target object was randomized.
- The material and texture of the target object was randomized.

The LightingModder is used for controlling lighting parameters. This includes the light source properties and pose. We used the following settings for the Lighting-Modder:

- The position of the light source was randomized.
- The specular attribute (degree of reflection) of the lighting was randomized.
- The diffuse attribute (light strength) of lighting was randomized.
- The active attribute of lighting was randomized.
- The ambient attribute of lighting was randomized.

Dynamic Randomization

The DynamicsModder is used for randomizing physical parameters related to the underlying physics model and contact modeling.

The following parameters for the target object was randomized.

- Friction. Perturbation Ratio: 0.1.
- Solref. Perturbation Ratio: 0.1.
- Solimp. Perturbation Ratio: 0.1.

The following parameters for all joints in the simulation was randomized.

- Stiffness: Perturbation ratio: 0.1.
- Friction loss: Perturbation size: 0.05.
- Damping: Perturbation size: 0.01.

- Armature: Perturbation size: 0.01.

4.4 Real World Setup

The real-world setup consists of a Kuka LBR iiwa R820 and the Robotiq 2F 85 gripper mounted on a table. A grey sheet covers the table, and the back wall is also covered by a white sheet to resemble the environment created in the simulation. The lab has a powerful light shining almost directly above the robot. Three smaller lamps were used in addition from the side to reduce shadows and have a more even distribution of light.

4.4.1 Real-World Observations

The end-effector position is sent to the master node by ROS from the Robot Controller and the gripper node. These observations are identical to the observations recovered in simulation.

4.4.2 Image Observations

The Zivid Two camera was used to capture the images in the real-world setup. The camera was mounted in front of the robot looking down at the gripper and target object, as shown in figure 2.5.3. A normal RGB image is captured and posted at the *zivid/color/image_color* topic and a depth image is captured and posted at the */zivid/depth/image* topic. The depth image undergoes the processing steps described in 3.8.3. The RGB image and depth image are combined into an RGB-D Image and cut to have the same fovy as the image observations used in simulation.

Chapter 5

Results

This chapter first presents the performance of the models trained and tested in simulation. Then the success rate of these models when transferred and tested in a real-world setting is presented.

5.1 Simulation Results

Two algorithms were trained in simulation, one using RGB-image observations, end-effector position, and gripper status as observation space. The other used RGB-D-image observations, end-effector position, and gripper status as observation space. They were trained for 50 million time steps each. The algorithms are evaluated and measured on two key factors, success rate and episodic reward mean. The success rate is the percentage of episodes where a successful grasp is detected. The episodic reward mean is based on the reward function described in 2.3 and is a measurement of both success and how fast successful grasps are achieved and maintained.

The success rate converges after around 10-20 million steps as seen in 5.1. The RGB-D algorithm averages a slightly higher success rate from around 15 million steps until the end of training.

The RGB-D algorithm performs better than the RGB algorithm in terms of reward mean per episode for the whole duration of the training process. This can also be seen in the final results, where the RGB-D algorithm achieves a successful grasp on average 7.9% faster than the RGB algorithm.

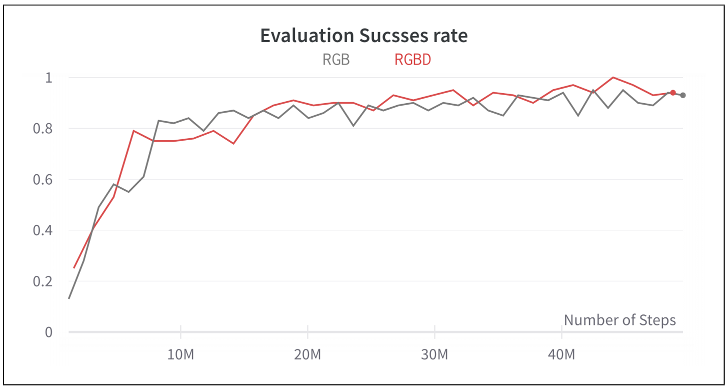


Figure 5.1: Plot of the evaluation success rate during training.

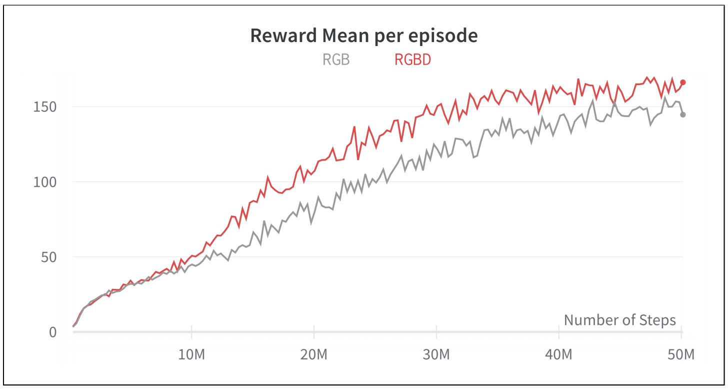


Figure 5.2: Plot of the reward mean per episode during training.

The algorithms performed pretty similar and achieved the following grasping success with domain randomization:

Algorithm	Success rate	Average number of time steps	average time
RGB-D	94.4 %	19.67 timesteps	1.97 s
RGB	92.6 %	21.43 timesteps	2.14 s

Table 5.1: Comparison of RGB and RGB-D tested in simulation.

5.2 Real World Testing

Both algorithms were tested under the same conditions in the lab. Due to the slow run time of the real-world system, the episodes were ended after 50 time steps. The algorithms were tested for a total of 30 episodes.

Algorithm	Success rate
RGB-D	53.3 %
RGB	40.0 %

Table 5.2: Comparison of RGB and RGB-D tested in a real-world setting.

The RGB-D algorithm achieved a grasp success of 53.3% and performed better than the RBD algorithm, which achieved a success rate of 40% 5.3

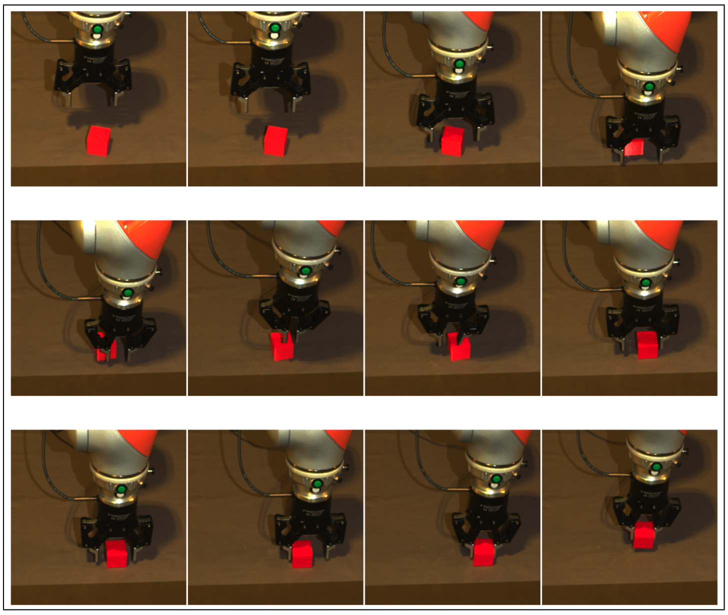


Figure 5.3: A image series from a successful grasp attempt in the real-world setup.

Empirical assessments of the grasping strategies show no specific malfunction in the system. Both failed, and successful attempts seem to involve large amounts of seemingly random movements. Successful grasps occurred seemingly random, following no clear pattern. For example, the agent can move around aimlessly for 30 time steps and suddenly start moving towards the target and achieve a successful grasp. A successful grasp is captured in the image series seen in figure 5.3

5.3 Sim-To-Real Results

By simply subtracting the real-world grasp success from the simulation grasp success, we get a measurement of how large the sim-to-real gap is:

Algorithm	Sim-to-real gap
RGB-D	41.1 %
RGB	52.6 %

Table 5.3: Comparison of how large the RGB and RGB-D sim-to-real gaps are.

The sim-to-real gap is 21.8% smaller for the RGB-D algorithm.

5.4 Training Speed

The training speed is measured in time steps per second. Depending on how many CPU cores are available, this number fluctuates severally. When utilizing 68 cores, a training speed of 280-300 frames per second was achieved.

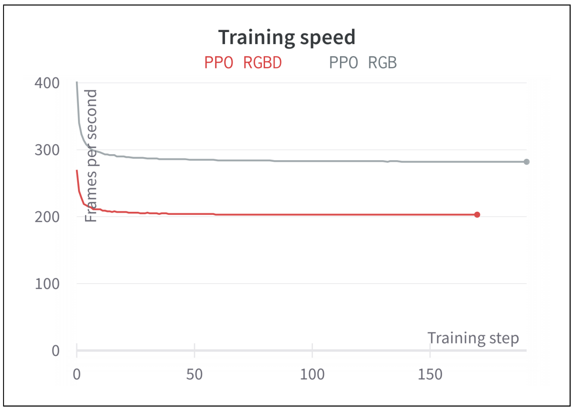


Figure 5.4: Plot of the training speed:
PPO RGB uses 48 cores and one NVIDIA A100 40Gb GPU.
PPO RGB-D uses 64 cores and one NVIDIA A100 80Gb GPU.

Training the final algorithms required 50 hours of running time.

Chapter 6

Discussion and Future Work

This chapter presents a discussion of the results and demonstrates how they relate to the objectives of this thesis. First, the algorithm's performance is discussed, then the effect of adding depth information for reducing the sim-to-real gap. Finally, the performance of the simulation framework is presented.

6.1 Performance in Simulation

The PPO algorithm trained with RGB-image observations, end-effector position, and gripper status as observation space achieved an average grasp rate of 92.6 %. This is a slightly less than average success rate compared to other state-of-the-art results like [13] which achieved 98%. One important factor to account for here is the difference in tasks. [13] was trained on a set of different objects and tested on a set of novel objects. Our environment was trained on a relatively simple geometric form, a cube, and tested on the same object. Although our target object had random color and placement for every episode, the task performed in [13] is considerably harder. The performance of our algorithm is therefore worse than expected. [5] used a similar training environment and achieved a grasp success rate of 92.5%, training and testing on 69 different objects. This also indicates that our algorithm performs slightly weaker than other state-of-the-art methods.

The PPO algorithm, trained with RGB-D image observations instead of RGB, achieved a grasp success of 94.4%. This is a slight improvement. An improvement was expected because the depth channel adds important and precise geometric information. Compared to the results from [5] and [13], judging by the fact that both tasks were considerably harder, the grasp success from the RGB-D algorithm is also lower than expected.

Comparing results from different reinforcement learning papers and projects is notably hard due to the lack of good standardized benchmark tests. Small changes in the environment like light, texture, and friction could impact the agent's interactions with the environment. The robot controllers differ from different frameworks, and their precision and response to actions are important for the algorithm's ability to train successful policies. The control frequency of the robot is another important factor in the environment affecting how well an algorithm can be trained.

Tuning hyperparameters are notably hard in reinforcement learning, and suboptimal hyperparameter choices could be the reason behind the slightly lower than expected performance. There might also be theoretical limitations with the use of PPO for these types of complex grasping tasks. [10] only accomplished simple block lifting tasks and not more complex bin picking tasks using their implementation of PPO.

6.2 Sim-To-Real Gap

Previous work has shown how big the sim-to-real gap can be [15]. Our system found a sim-to-real gap of 92.6% to 40% for the algorithm trained with RGB images. This shows the severity of the sim-to-real gap. The performance of the policy is reduced by 56.8%. This is coherent with related works [13] and the conclusions from [36]. Our method has a significantly smaller sim-to-real gap than baseline tests in [13]. This is likely due to the task being less complex.

Our system found a sim-to-real gap of 94.4% to 53.3% for the algorithm trained with RGB-D images. The gap between simulation and real-world application is reduced by 21.8% when introducing the extra depth dimension in the image observations. The performance improvement is significantly greater for the sim-to-real gap than it is for simulation performance.

By applying some statistics, we can see that the sample size of the real-world testing is small and provides a modest confidence interval. A larger sample size would have given a greater confidence interval for claiming that the RGB-D algorithm performed better than the RGB algorithm in the real-world setup. The large difference in score combined with the sample size still gives us reason to believe that RGB-D can improve the sim-to-real gap. The small sample size was caused by delays due to sick leave.

There is reason to believe that the enhanced sim-to-real performance is due to the depth data being more coherent in the simulation and real-world setup. This can be caused by the RGB values fluctuating too much under the influence of different lights and textures. The depth is a fixed value, which is perfect in simulation but is subject to noise in the real-world setup. The noise effect in the Zivid camera is less than the intervals caused by using unit8 as datatype. A significant source of error is the interpolated depth image caused by the lack of depth information in large areas of the depth image.

6.3 Simulation Framework

The simulation framework used for the experiment functioned well. It was intuitive to use and the modular framework allowed for large amounts of customization. There were some issues that needs to be addressed.

The framework lacked inverse kinematics functionality for the OSC pose controller used for this thesis. This lack of functionality caused the extra simulation step in the real-world setup in order to provide the real robot controller joint angles as input.

The domain randomization wrapper had some instability issues when trying to create environments with multiple random objects.

6.3.1 Training Speed

It is not possible to use more than one GPU to speed up training with stable-baselines3. 64 CPU cores were the maximum available for this thesis. Speeding up the training more would require more CPU cores. Training the final algorithms required 50 hours of running time. Fifty hours is an acceptable amount of time compared to manually programming robots. Fifty hours becomes very long when performing 50 runs for different hyperparameter settings during training.

Chapter 7

Conclusion

This chapter presents some concluding remarks and suggestions for potential further work and exciting possibilities.

7.1 Conclusion

This thesis aimed to test whether using depth information would improve the sim-to-real gap when utilizing reinforcement learning for robotic grasping. This also involved the process of creating a simulating framework that facilitates the training of high-performing reinforcement learning algorithms. The results achieved in simulation underperformed slightly compared to other state-of-the-art results, but we still managed to train algorithms to a high grasp success rate. The performance gap between our system and other state-of-the-art systems is probably due to the complicated process of hyperparameter tuning. Given more time to tune hyperparameters and experiment with environment variables and different algorithms, it is likely that the algorithm could be trained to achieve a higher grasp success rate on the current task.

The results from real-world testing imply that the added depth information reduces the sim-to-real gap. Further work should include a test with a larger sample size to confirm these results. This is an exciting result as it implies that future attempts at closing the sim-to-real gap should involve depth information in the observation space. Adding depth information does not close the sim-to-real gap fully, but a reduction of 21.8% is a significant contribution to closing the sim-to-real gap. Due to error sources such as hyperparameter tuning, the processing of the real-world depth data, and differences between the real-world and simulated controller, there might also be possible to reduce the sim-to-real gap even further with this method.

7.2 Recommendations for Further Work

The simulation and real-world system created to conduct the experiments in this thesis is a good foundation for different types of reinforcement learning research for robotic grasping. In future work, time should be devoted to reducing the time step length in the real-world setup. Comparing the precision and functionality of the simulated and real robot controllers and how this affects the sim-to-real gap is an interesting approach for understanding more about the sim-to-real gap.

Conducting similar sim-to-real experiments for generalized grasping of novel objects is an interesting approach that would require the creation of new objects in the simulation environment and a domain randomization method that is capable of handling multiple random objects.

7.2.1 Using Depth Information

An interesting aspect of research would be to see what type of depth information improves simulation and sim-to-real results the most. For example, running tests with a depth image as a separate feature and testing with point clouds. Utilizing networks created to handle point clouds of various sizes and resolutions would alleviate the need for interpolating and processing real-world depth information.

References

[1] Joshua Achiam. “Spinning Up in Deep Reinforcement Learning”. In: (2018). URL: https://spinningup.openai.com/en/latest/spinningup/extra_pg_proof2.html.

[2] Joshua Achiam. “Spinning Up in Deep Reinforcement Learning”. In: (2018). URL: https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html#other-forms-of-the-policy-gradient.

[3] OpenAI: Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Jozefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, and Alex Ray. “Learning dexterous in-hand manipulation”. In: *The International Journal of Robotics Research* 39 (2020).

[4] G. Bradski. “The OpenCV Library”. In: *Dr. Dobb's Journal of Software Tools* (2000).

[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] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. *OpenAI Gym*. 2016. eprint: arXiv:1606.01540.

[7] Jack Collins, Shelvin Chand, Anthony Vanderkop, and David Howard. “A Review of Physics Simulators for Robotic Applications”. In: *IEEE Access* 9 (2021), pp. 51416–51431.

[8] *DeepMind acquires MuJoCo physics engine for robotics RD*. https://www.therobotreport.com/deepmind-acquires-open-source-mujoco-physics-engine/. Accessed: 2021-12-29.

[9] Tom Erez, Yuval Tassa, and Emanuel Todorov. “Simulation tools for model-based robotics: Comparison of Bullet, Havok, MuJoCo, ODE and PhysX”. In: (2015).

[10] 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.

[11] Hodson R. “A gripping problem: designing machines that can grasp and manipulate objects with anything approaching human levels of dexterity is first on the to-do list for robotics”. In: (2018).

[12] Honglak Lee Ian Lenz and Ashutosh Saxena. “Deep learning for detecting robotic grasps”. In: (2015).

[13] 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: (June 2019), pp. 12619–12629. DOI: 10.1109/CVPR.2019.01291.

[14] Juilan Ibraz, Jie Tan, Chelsea Finn, Mrinal Kalakrishnan, Peter Pastor, Sergey Levine. “How to Train Your Robot with Deep Reinforcement Learning – Lessons We’ve Learned”. In: (2021).

[15] 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: (June 2018).

[16] Ludvik Kasbo. “Understanding the Challenges and Potential of Generalized Robotic Grasping using Deep Reinforcement Learning and Simulation”. In: (2022).

[17] Jens Kober, J. Bagnell, and Jan Peters. “Reinforcement Learning in Robotics: A Survey”. In: *The International Journal of Robotics Research* 32 (Sept. 2013), pp. 1238–1274. DOI: 10.1177/0278364913495721.

[18] Adam Daniel Laud. “Theory and Application of Reward Shaping in Reinforcement Learning”. AAI3130966. PhD thesis. USA, 2004.

[19] Sergey Levine, Peter Pastor, Alex Krizhevsky, and Deirdre Quillen. “Learning Hand-Eye Coordination for Robotic Grasping with Deep Learning and Large-Scale Data Collection”. In: *The International Journal of Robotics Research* 37 (Mar. 2016). DOI: 10.1177/0278364917710318.

[20] DeepMind Technologies Limited. “Mujoco documentation”. In: (). Accessed: 2021-12-08.

[21] Jeffrey Mahler, Jacky Liang, Sherdil Niyaz, Michael Laskey, Richard Doan, Xinyu Liu, Juan Aparicio Ojeay, and Ken Goldberg. “Supersizing self-supervision: Learning to grasp from 50k tries and 700 robot hours”. In: (2017).

[22] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. “Playing Atari with Deep Reinforcement Learning”. In: (Dec. 2013).

[23] Hai Nguyen and Hung La. “Review of Deep Reinforcement Learning for Robot Manipulation”. In: Feb. 2019, pp. 590–595. DOI: 10.1109/IRC.2019.00120.

[24] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary 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: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[25] Lerrel Pinto and Abhinav Gupta. “Supersizing self-supervision: Learning to grasp from 50k tries and 700 robot hours”. In: (2016).

[26] 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.

[27] Kanishka Rao, Chris Harris, Alex Irpan, Sergey Levine, Julian Ibarz, and Mohi Khansari. “RL-CycleGAN: Reinforcement Learning Aware Simulation-To-Real”. In: (June 2020).

[28] Petter Rasmussen and Ole Jørgen Rise. “Sim-to-Real Transfer in Deep Reinforcement Learning for Vision-Based Robotic Grasping (The Implementation and Configuration of Simulation Training and Physical Testing)”. In: 2022.

[29] Sutton RS and Barto AG. *Reinforcement Learning*. Boston, MA: MIT Press., 1998.

[30] Ashutosh Saxena, Justin Driemeyer, and Andrew Ng. “Robotic Grasping of Novel Objects using Vision”. In: *I. J. Robotic Res.* 27 (Feb. 2008), pp. 157–173. DOI: 10.1177/0278364907087172.

[31] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George Driessche, Thore Graepel, and Demis Hassabis. "Mastering the game of Go without human knowledge". In: *Nature* 550 (Oct. 2017), pp. 354–359. DOI: 10.1038/nature24270.

[32] Shuran Song, Andy Zeng, Johnny Lee, and Thomas Funkhouser. "Grasping in the Wild: Learning 6DoF Closed-Loop Grasping From Low-Cost Demonstrations". In: *IEEE Robotics and Automation Letters* PP (June 2020), pp. 1–1. DOI: 10.1109/LRA.2020.3004787.

[33] 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: (June 2018). DOI: 10.15607/RSS.2018.XIV.010.

[34] Alexandru Telea. "An Image Inpainting Technique Based on the Fast Marching Method". In: *Journal of Graphics Tools* 9 (Jan. 2004). DOI: 10.1080/10867651.2004.10487596.

[35] Xintong Yang, Ze Ji, Jing Wu, and Yu-Kun Lai. "An Open-Source Multi-Goal Reinforcement Learning Environment for Robotic Manipulation with Pybullet". In: (2021). arXiv: 2105.05985 [cs.RO].

[36] Wenshuai Zhao, Jorge Peña Queralta, and Tomi Westerlund. "Sim-to-Real Transfer in Deep Reinforcement Learning for Robotics: a Survey". In: (Sept. 2020).

[37] 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.

[38] Zivid. *Zivid Stripe Vision engine.* [Online]. Available from: https://www.zivid.com/videos?wchannelid=lazso2emjc&wmediaid=7fbph3zuxl. [Accessed 11rd June 2022].