UiT The Arctic University of Norway

Faculty of Engineering Science and Technology
Department of Computer Science and Computational Engineering

Vision-based Robotic Grasping in Simulation

using Deep Reinforcement Learning

Jostein René Danielsen

Master thesis in Master of Science, Computer Science, Narvik, May 2021

“Simplicity is prerequisite for reliability.”
–Edsger Dijkstra

“Beware of bugs in the above code;
I have only proved it correct, not tried it.”
–Donald Knuth

Abstract

This thesis will investigate different robotic manipulation and grasping approaches. It will present an overview of robotic simulation environments, and offer an evaluation of PyBullet, CoppeliaSim, and Gazebo, comparing various features. The thesis further presents a background for current approaches to robotic manipulation and grasping by describing how the robotic movement and grasping can be organized. State-of-the-Art approaches for learning robotic grasping, both using supervised methods and reinforcement learning methods are presented. Two set of experiments will be conducted in PyBullet, illustrating how Deep Reinforcement Learning methods could be applied to train a 7 degrees of freedom robotic arm to grasp objects.

Acknowledgements

A number of people have supported me during this work, which has culminated into this thesis. First of all I would like to thank my family and my fellow students for encouraging me during this period. The work put into this thesis have been overwhelming and I have to admit that it has been very challenging. I am grateful to my supervisors, Bernt Arild Bremdal and Andreas Dyrøy Jansson, who provided advice and support throughout the work in form of constructive feedback and advice.

Contents

Abstract **iii**

Acknowledgements **v**

List of Figures **xi**

1 Introduction **1**
1.1 Limiting the Scope 5
1.2 Thesis Structure 6

2 Background **9**
2.1 Vision-based Robotic Grasping: Four key tasks 9
2.2 Robotic Grasping Approches and Criteria 10
2.2.1 Analytical Approach 12
2.2.2 Empirical Approach 13
2.3 State-of-the-Art 13
2.3.1 Supervised Deep Learning Approaches for Robotic Grasping 13
2.3.2 Deep Reinforcement Learning Approaches for Robotic Grasping 14
2.4 Machine Learning 15
2.5 Machine Learning Paradigms 16

3 Foundations of Reinforcement Learning **19**
3.1 Elements of a Reinforcement Learning System 20
3.2 States 21
3.3 Actions 22
3.4 Policy 23
3.5 Reward 23
3.6 Model of Environment 23
3.7 Exploration vs Exploitation 25
3.8 Markov Chains 25
3.9 Markov Decision Processes 25
3.10 Expected Return 27

3.11 Value Functions 27
3.12 Optimal Value Functions 29
3.13 Classical Reinforcement Learning Methods 30
3.13.1 Dynamic Programming vs Monte Carlo Methods . . . 30
3.13.2 Q-Learning 30
3.14 Deep Reinforcement Learning Methods 31
3.14.1 Policy Gradient and Q-Learning methods 31
3.14.2 DQN 32

4 Simulators 33
4.1 Robot Operating System 34
4.2 URDF 34
4.3 SDF 35
4.4 PyBullet 35
4.5 CoppeliaSim 35
4.6 Gazebo 36
4.7 MuJoCo 37
4.8 Software Libraries and Tools 37
4.8.1 OpenAI Gym 37
4.8.2 Reinforcement Learning Libraries 38
4.9 Comparison 39
4.9.1 Features 40
4.9.2 Robotic Manipulation and Grasping 44
4.9.3 Robotic Learning in Simulation 46
4.10 Discussion 47

5 Experiments 49
5.1 Experimental Setup 49
5.2 Diverse Kuka Environment 50
5.2.1 Goal 50
5.2.2 The Environment 50
5.2.3 Actions 51
5.2.4 Observations 51
5.2.5 Reward 51
5.3 Simple Kuka Environment 52
5.3.1 Actions 52
5.3.2 Observations 52
5.3.3 Reward 52
5.4 DQN 53
5.5 Experiment 1 54
5.5.1 Moving the camera 54
5.5.2 TD3 55
5.5.3 Non-image based observations 56
5.6 Experiment 2 56

5.7 Experiment 1 - Results 57
5.8 Experiment 2 - Results 58
5.9 Discussion 60

6 Future Work 63
6.1 Different Algorithms 63
6.2 Transfer of Model 64
6.3 Simulators 64

7 Concluding remarks 65

Bibliography 69
.1 Appendix - Source Code 73
.2 Appendix - Project Description 73

List of Figures

1.1 The nine technolocigal pillars of Industry 4.0 2

2.1 Typical vision-based robotic grasping pipeline [22]. Top row: Model-based approaches. Bottom row: Model-free approaches. 10
2.2 Classical programming vs machine learning. 16
2.3 The three types of machine learning. 16

3.1 The basic components of a RL system. 21
3.2 Different RL methods: model-based and model-free learning. 24
3.3 An example of a simple MDP from Wikipedia. Green circles are states, orange circles are actions, and red arrows are rewards. 26

4.1 Robotic manipulations. Table information from recent survey [8]. 44
4.2 Figure from [13]. First column: task related to grasping an object rigidly and moving it around. Second column: CPU timing results as thousands of evaluations per second. Third column: Speed-accuracy trade-off with respect to a consistency measure. 45
4.3 Robotic learning. Table information from recent survey [8]. . 46

5.1 KukaDiverseObjEnv 50
5.2 KukaGymEnv 50
5.3 Model architecture of baseline DQN. 53
5.4 Eye-in-hand camera 55
5.5 Mean avg. reward for previous 10 episodes. 57
5.6 Mean avg. reward for previous 10 episodes. 58
5.7 (64, 64) depth image observation before grayscale, cubic filtering and cropping preprocessing. 61

6.1 Imported and configured Nachi MZ07 model in CoppeliaSim. 64

1 Introduction

Humans can see objects and easily determine how to pick them up and move them. Even if the objects are novel, it is generally considered to be a simple task not given much thought. Robots have a much harder time performing this seemingly trivial task, and it remains a critical challenge to design robots and software that can operate at levels of dexterity and control resembling that of a human. There is an interest in robotic grasping in both industry and academia, and it has been researched in literature for quite some time. Despite this, it still remains a very challenging problem and still is considered an unsolved problem in robotics [48].

Over the last decades robots have increased their capability to work in controlled environments, and are efficient at grasping and manipulation in repetitive and familiar environments [3]. In such environments, the physical parameters such as object geometry and environment geometry, material properties like friction, texture and weight of the objects are usually known. One example of this could be picking up parts on a car-assembly line, or working with other specifically designed tasks in industry settings[1].

1. `https://go.nature.com/3svzmRp`

Robotic grasping is interesting, because it is a stepping stone for further steps in a robotics pipeline, such as robotic manipulation and placement. One primary goal of robotics is *dextereous manipulation*. Dextereous manipulation increases the robots capability to perform complex human-like tasks.

Robotics for automation and optimization systems become especially important with regards to the current digital revolution, known as The *Fourth Industrial Revolution* or *Industry 4.0*.

Nayyar and Kumar ([34, p. 2]) refer to Industry 4.0 as:

> ... a planned phase of the industrialization process in accordance with future expectations.

Industry 4.0 is usually characterized by automation in industry and other sectors facilitated by the use of smart technology. One of the important aspects of Industry 4.0 is the notion of the *Smart Factory*, where manifacturing is automated and the different parts of the factory are all interconnected digitally into one system [16]. There are nine pillars, or fundamental technologies that are necessary for Industry 4.0 to become a reality ([34, pp. 6-7]), as shown in fig. 1.1[2].

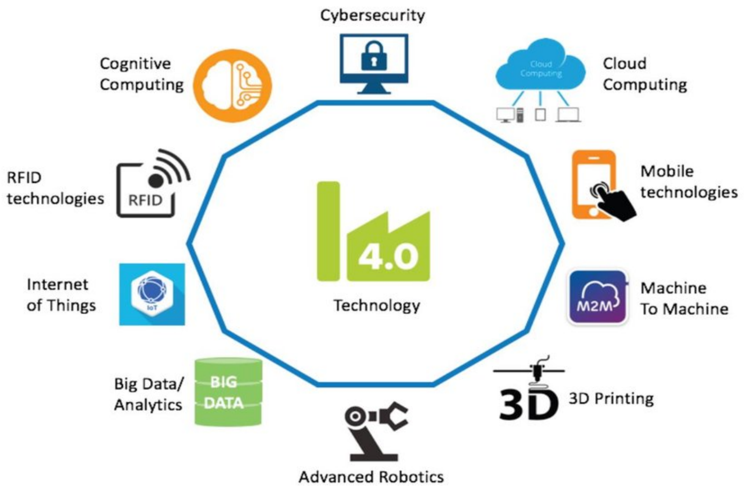


Figure 1.1: The nine technolocigal pillars of Industry 4.0

Robotic technology is one pillar of Industry 4.0, and could help facilitate automation and optimization in industries. It could also help solve tasks that are considered too dangereous or challenging for humans to solve easily.

2. `https://bit.ly/3svUc32`

Human-Robot Collaboration (HRC) technologies, such as the Kuka LBR IIWA robot, is an example of a collaborative robot (*Cobot*) that belongs to a new type of robotic technology. They work in close cooperation with humans, in a safer manner, and don't need the same security restrictions as other industrial robots [16].

Robotic systems have improved a lot since the early days of robotics. This is largely due to the development of optical sensors, which means robots don't have to rely as much on human inputs for grasping objects, and can learn to grasp objects automatically utilizing vision-based systems. The progress of machine learning in the field of computer vision has also greatly facilitated robotic learning based on visual data [15].

Although robotics has seen a lot of progress over the last decades, there are still many challenges associated with this field. Particularly, robust robotic grasping is challenging due to several factors. One of the challenges could be imprecisions in the sensors and the actuators of the robotic system. This often leads to uncertainty about the shape of the object, pose of the object and pose of the actuator, material properties, and object mass [29] [30].

There's also some challenges facing robotic grasping systems utilizing machine learning and deep learning techniques. Deep learning based approaches often require a lot of training data to perform well. Oftentimes it becomes time-consuming, expensive, and impractical to generate the data required for training a particular model. Especially in robotics, the generation of training data from real world settings is problematic due to the abovementioned problems, and also the fact that the generated data are not invariant to hardware changes (changing gripper, robotic hand) and the experimental parameters (table height, camera placement) [22].

Training robotic grasping in simulations are a good alternative to overcome these problems. Simulations can be used to generate vast amounts of training data with annotations. Additionally, they allow faster training, compared to training a model in a real world setting. However, training a model in simulation is challenging due to the mismatch between the properties of the simulation and the real world. Models trained in simulation usually don't transfer well to the real world.

The domain of robotic manipulation and grasping is a complex, interdisciplinary field that requires knowledge from different fields including, but not limited to: robotics, control systems engineering, computer vision, AI and ML, programming, physics and mathematics, hardware and algorithm design. Knowledge of the application domain is also an important aspect. In an industrial setting where real-time and robust systems is a practical necessity, certain elements of the task pipeline may be replaced by more robust and simple methods from computer vision or the control engineering discipline, while other parts that are more difficult to automate may be replaced by more advanced learning-based methods from AI and machine learning.

1.1 Limiting the Scope

The task description is attached in the appendix. Briefly summarized, it includes the following elements:

- Investigate current approaches for robotic manipulation and grasping.
- Determine how to train a data-driven method
- Determine environments used for training
- Hardware
- Look into single view 3D reconstruction using 3D reconstruction from point clouds with an RGB-D sensor mounted on a robot arm.
- Demonstrate pipeline/methodology for reconstruction of objects with an RGB-D sensor mounted on a robot arm.

The task description of the thesis is comprehensive and contain many elements. To limit the scope of the thesis and make it fit into the timespan available, the thesis emphasize the following questions:

- *How are learning-based methods applied in current approaches of robotic manipulation and grasping?*
- *What kind of knowledge and software tools are needed in order to set up learning-based experiments in a robotic simulator?*
- *How can learning-based methods be applied in a robotic simulation environment?*

By focusing on these questions, the thesis offer an interpretation of what level of knowledge individuals assigned to work on robotic simulations should have.

The questions lead to four distinct focus-points:

1. **Literature review**
   Investigate current data-driven/learning-based approaches for robotic manipulation and grasping.

2. **Training**
   Determine how to train the system with reinforcement learning.

3. **Environment**
   Investigate and compare simulation environments used for training of robotic manipulation and grasping, focusing on applicability, flexibility, complexity and availability of models and platforms for learning.

4. **Implementation**
   Choose one particular simulation environment and implement chosen reinforcement learning methods.

A literature review will provide insight into the complex and interdisciplinary nature of the field. It will further add information on how learning-based methods are applied in current approaches for robotic manipulation and grasping.

Exploring how training with reinforcement learning is applied is of great importance in respect to investigating tools and approaches to learning. Training using Reinforcement Learning (RL) is the primary primary focus for learning-based methods in this work. RL is closely related to the field of robotics and the techniques it offers to automatically acquire end-to-end solutions by trial and error, meaning the task doesn't necessarily have to be subdivided into multiple sequential sub-tasks. Furthermore, there is no need for a training set, which often could be time-consuming and impractical to produce.

The third element considers simulators. It is an important point, because simulators function as a test-bed for different algorithms before being deployed on the real platform. It's also important to include a comparison of a selection of simulators, due to the different sub-domains of robotics they are targeted to and the particular features they have that support the field of robotic manipulation and grasping. The last element is more practical, and gives more insight into how an experiment can be set up in a simulator.

1.2 Thesis Structure

Chapter two will provide a backgroud for the thesis and include an investigation of the field of robotic manipulation and grasping with respect to data-driven/learning-based methods and State-of-the-Art methods. Following the literature review, there will be a short overview of the main paradigms of machine learning, and where RL fits in. In chapter three, the foundations of classical reinforcement learning will be presented. Chapter four compares the simulators PyBullet, CoppeliaSim and Gazebo with respect to built-in features, their applicability for robot manipulation and grasping, and their suitability for robotic learning. In chapter five, an experiment is presented and set up in one of the three simulators, the learning methods are described, and the results from the learning methods will be presented. Chapter six will discuss the results from the experiments, and the following chapter will present future work.

2 Background

A robotic grasping system can be divided into three sub-systems [25]: a grasp detection system, a grasp planning system and a control system. The grasp detection system are the first sub-system in the pipeline, and can be further divided into three tasks: target object localization, pose estimation and grasp detection. The grasp detection system together with the grasp planning sub-system makes up a typical robotic grasping pipeline. The control system pertain to the control engineering discipline.

2.1 Vision-based Robotic Grasping: Four key tasks

Vision-based robotic grasping can be decomposed into four key tasks [39]:

1. Target object localization
2. Pose estimation
3. Grasp detection
4. Motion planning

Target object localization finds the location of the target object from the sensor data. Object pose estimation estimates the position and orientation of the target object with respect to a reference frame. Grasp detection finds the grasp points/region on to the target object. Motion planning determines the path from the robot hand to the grasp region on the target object.

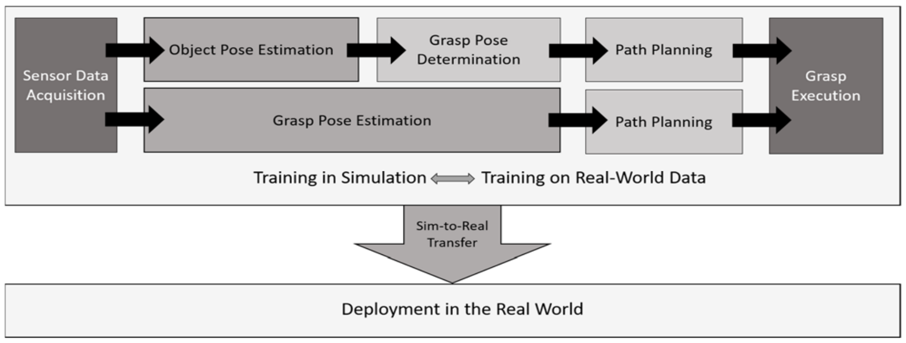


Figure 2.1: Typical vision-based robotic grasping pipeline [22]. Top row: Model-based approaches. Bottom row: Model-free approaches.

2.2 Robotic Grasping Approches and Criteria

Robotic grasping is a wide field, and it is challenging to organize all of the existing methods into a few categories with respect to different criteria. In the robotics research community, robotic grasping approaches are usually divided into two approaches: analytical approaches and empirical approaches [5]. Methods used in robotic grasping and manipulation can be categorized with respect to several different criteria [22]:

Model-based vs model-free methods
The robotic grasping system has access to knowledge about the target objects. This could be a CAD model or scanned model.

Rigid or flexible objects
The objects to be grasped could be rigid or articulated. Articulated means composed of several rigid parts. They could also be flexible or deformable.

Known or unknown objects
This specifies whether the method is able to generalize to unknown objects not seen during the training phase, or if it only works on a limited set of objects used during training.

Type of machine learning

This criteria specifies the type of machine learning used during training of the system. Usually the system will be trained using supervised learning (SL) or reinforcement learning (RL). Labeling of the dataset (annotations) could either be manually created by humans, or obtained automatically.

Discriminative vs generative approaches

Approaches also differ based on how the machine learning algorithm handles grasp candidates. Discriminative approaches usually sample grasp candidates and give them a score (ranking) using a neural network. Generative approaches directly generate grasp poses from the input data.

Training environment

The environment used for training of the robotic system is another distinction. Robotic systems could be trained solely in simulation or the real world, or the system could be trained in a simulation environment and also the real world. There are some challenges when transferring the learnt knowledge from simulation to the real world (sim-to-real-transfer). The most prominent problem is *the reality gap* - differences between simulation environment and real-world environment. Several methods address this problem, e.g. domain randomization and domain adaptation.

Sensors and sensor data

A robotic grasping system could utilize different sensors for observation of the environment. Commonly used sensor data in robotic grasping are: RGB, depth, RGB-D, point clouds, or a combination of different sensor data.

Open-loop vs closed-loop systems

Another distinction is closed-loop vs open-loop systems for robotic grasping. Open-loop systems are also known as non-feedback systems, whilst closed-loop systems are known as feedback systems. This means that the output of an open-loop system will not be used to correct the system. Contrary to open-loop systems, closed-loop systems are able to self-correct in order to achieve the desired goal state by comparing the goal state with the actual output from the system. Closed-loop systems usually have a higher execution time compared to open-loop systems. A popular technique used for closed-loop robotic grasping systems is *visual servoing*. Visual servoing controls the robot using visual feedback. Visual servoing is, in essence, a method for robotic control where the sensor used is a camera (visual sensor or similar). Servoing consists primarily of two techniques [6]. One involves using information from the image to directly control the degrees of freedom of the robot. The other involves the geometric interpretation of the information extracted from the camera, such as estimating the pose of the target and parameters of the camera (assuming some basic model of the target is known).

Robotic hardware

The robotic hardware used in the robotic system also determines the type of robotic grasping and manipulation that can be performed. Typical robotic hardware used in a vision-based robotic system consist of the robotic arm, the end-effector, and the sensors gathering data for training.

Grasping scenario

Approaches also differ based on the type of grasping scenario that is studied. Different grasping scenarios vary with regards to the objects used in the environment. It also depends on the configuration of the objects in the robotic workspace. The environment could contain single separated objects or densely cluttered objects. In densely cluttered environments, some of the objects to be grasped could be partially or fully occluded by other objects. There's also a distinction between static and dynamic environments. An example of a static environment could be a table with a certain number of objects to be grasped, and an example of a dynamic environment could be a moving conveyor belt. For static environments, both open-loop and closed-loop algorithms could be used. For dynamic environments, it is necessary to use closed-loop algorithms. Advantages of using closed-loop methods are ability of the robot to adapt to changes in the environment, and also the camera calibration and position controls don't have to be as accurate as it would have to be in an open-loop system [17].

2.2.1 Analytical Approach

Analytical approaches, also known as geometrical approaches, usually analyze the shape of the target object to determine the grasp pose [22]. Historically, these approaches were based on the analytical constructions of force-closure grasps [14].

Analytical methods for robotic grasping often suffers from a high time-complexity and poor generalization capabilities to new objects.

2.2.2 Empirical Approach

Empirical approaches, also known as data-driven approaches, were introduced to remedy the difficulties encountered by the analytical approaches. Empirical approaches utilize machine learning and deep learning. Data-driven approaches have gained popularity in recent years due to the increased availability of datasets, better computational resources and better algorithms [22].

2.3 State-of-the-Art

Analytical approaches for robotic grasping, besides being time-consuming, usually offer poor generalization abilities to novel objects, since the features are often hand-crafted by domain experts and the control system are specialized for the specific task [11]. A literature review of current State-of-the-Art methods for learning-based robotic grasping will be briefly discussed in this section. Literature review for learning-based approaches will here be divided into supervised deep learning approaches and deep reinforcement learning approaches.

2.3.1 Supervised Deep Learning Approaches for Robotic Grasping

Supervised Learning methods for robotic grasping can be categorized into discriminative and generative approaches. Discriminative approaches use the grasp configuration as input for the neural network and output a ranking for the current grasp configuration, while generative approaches generate the grasp configuration as output citekleebergerSurvey.

Discriminative Approaches

Discriminative approaches use a neural network to sample and rank grasp configurations. An example of this is the work of Levine et al. [28], where hand-eye coordination for grasping from monocular images (RGB) is learned via a large convolutional neural network (CNN). To train their network they equipped 6-14 robotic manipulators that collected over 800,000 grasp attempts over a two month period. The trained CNN was able to predict the grasp success for a given grasp candidate from the RGB image of the cluttered bin, and navigate the gripper to the bin using visual servoing [28].

In the work of Mahler et al. [29], they suggest a Grasp Quality Convolutional

Neural Network (GQ-CNN) for predicting robust grasps from depth images. The GQ-CNN is trained on a synthetic dataset created in a simulator. The dataset consists of 6.7 million synthetic point clouds and associated grasp candidates. During testing, when the robot tries to pick up an object, the attached depth camera returns a 3D point cloud. From this point cloud antipodal points represent several different grasp candidates. Among these, the GQ-CNN ranks and outputs the most robust grasp.

Contrary to discriminative approaches, for generative approaches the neural network outputs grasp configurations. Robotic grasp detection is a generative approach that is similar to object detection in computer vision [37] [43]. An example of robotic grasp detection is the generative approach suggested by Jiang et al. [20]. They introduced a generative method that learns grasping of objects from RGB-D images and their aligned depth maps. From these images, the goal is to estimate the location, orientation, and opening width of the parallel jaw gripper. These parameters can be represented as an oriented rectangle in the image plane as shown in

Lenz et al. [27] also uses a generative learning-based approach with oriented rectangles to represent grasp configurations. In their work, they proposed a two-step cascaded system with two DNN's. The first DNN has fewer features than the second DNN, and discards unlikely grasp candidates. The second network is slower, but only considers the grasp candidates with highest rank from the first. Kumra et al. [26] utilizes a similar approach, but increase the complexity of the two networks, and obtain a better performance. In their work, two 50-layer deep convolutional residual neural networks were used.

2.3.2 Deep Reinforcement Learning Approaches for Robotic Grasping

A lot of research has been done on robotic grasping. In real-world scenarios the robot cannot obtain a 100% success rate in grasping a variety of objects. This is due to inaccuracies in the potential grasp detection and ultimately leads to the problem of not being able to select the best possible grasp for an object. Recent work has addressed this by converting it into a detection problem which works on visual aspects of the image to infer the location where the robotic gripper needs to be placed, involving robotic manipulation and grasping.

Levine et al. [28] demonstrates how an implementation of hand-eye coordination may be applied using a stereo-vision RGB camera as sensory input to control movement and gripping. By continuously recomputing the most promising motor commands, the approach continuously integrates sensory cues from the environment, allowing it to react to perturbations and adjust the grasp to maximize the probability of success based using an adapted 16-layer convolutional neural network.

Kalashnikov et al. [21] adapted a different approach called QT-Opt by using a scalable self-supervised vision-based reinforcement learning framework based on Q-learning. The framework is based on a general formulation of robotic manipulation as a Markov Decision Process (MDP). At each time step, the policy observes the image from the robot's camera and chooses a gripper command. The grasping task is defined using a sparse reward; a successful grasp results in a reward of 1, and a failed grasp a reward of 0. A grasp is considered successful if the robot holds an object above a certain height at the end of the episode.

Intelligent manipulation benefits from the capacity to flexibly control an end-effector with high degrees of freedom (DoF) and dynamically react to the environment. However, due to the challenges of collecting effective training data and learning efficiently, most grasping algorithms today are limited to top-down movements and open-loop execution. In [40] Song et al. offer an approach they have labelled *Grasping in the Wild*. This approach to robotic grasping deep reinforcement learning exploits a grasping model that have an "action-view" which is used to simulate future states with respect to different possible actions. By evaluating these states using a learned value function (Q-function), the authors claim the method is able to better select corresponding actions that maximize total rewards (i.e., grasping success).

Quillen et al. [36] explored several off-policy, model-free deep reinforcement learning methods for vision-based robotic grasping. The tested algorithms were Q-Learning methods, Path consistency learning (PCL), Deep Deterministic Policy Gradient (DDPG), Monte Carlo Policy Evaluation, and Corrected Monte Carlo. The algorithms were tested and evaluated in a simulated grasping benchmark made in PyBullet containing a Kuka robotic arm with a two-finger parallel gripper, and a tray with diverse objects. They concluded that Deep Q-Learning (DQL) methods performed better on the grasping tasks than the other algorithms in low-data regimes. The Monte Carlo and the corrected version of it, performed betterin high-data regimes.

2.4 Machine Learning

Machine learning differs from classical programming in how the rules or logic of the program is arrived at. In classical programming, a program is written to follow a set of rules. The program takes in the input data, processes the input data according to the rules created by the programmer, and outputs the results. Machine learning automates the process of arriving at the rules by taking in the input data and desired results, and outputting the rules.

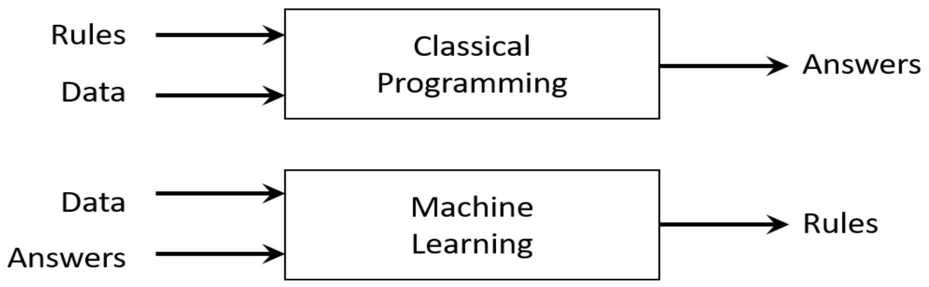


Figure 2.2: Classical programming vs machine learning.

2.5 Machine Learning Paradigms

Machine learning algorithms are often classified into three types of learning: *supervised learning*, *unsupervised learning* and *reinforcement learning*. These learning paradigms are differentiated by the type of feedback the system learns from.

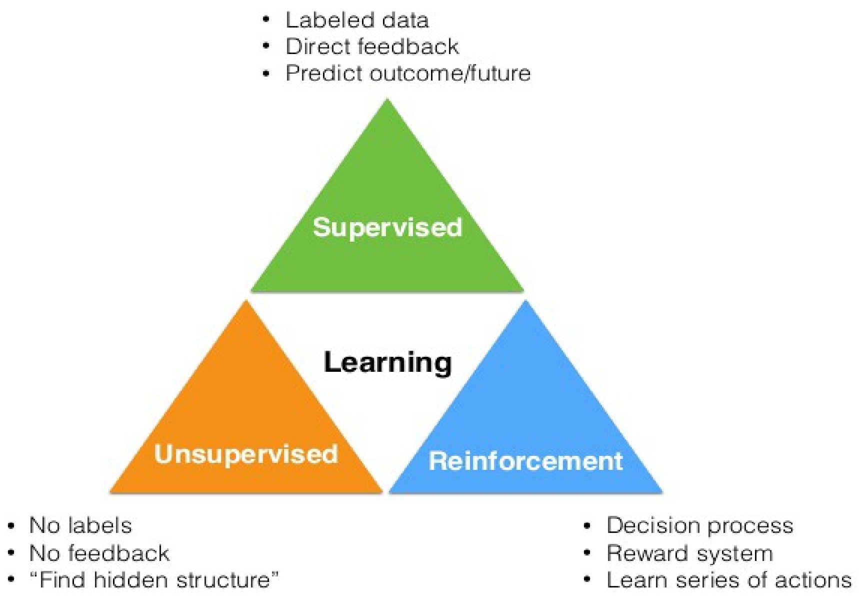


Figure 2.3: The three types of machine learning.

In *supervised learning*, a training set of (input, output) pairs is provided. The input in a training set sample consists of some *features* to learn from, and the output, also known as the *target* or *label* value, which is the correct response for the given input. A supervised learning algorithm trains on the samples in the training set and tries to learn a function that maps from input to output. The goal of supervised learning is generalization to new samples that weren't encountered in the training set. Supervised learning algorithms can be grouped into classification and regression algorithms. Classification algorithms processes the inputs and tries to decide which of N classes the inputs belong to. Regression algorithms try to predict a real, continuous value from the inputs.

Unsupervised learning differs from supervised learning in the sense that there is no feedback supplied to the learning algorithm. The algorithm tries to detect similarities and patterns in the input data. The most common tasks of unsupervised learning are: *clustering* and *dimensionality reduction*. A clustering algorithm groups the input data into different clusters where each cluster contains data that share some common properties. Dimensionality reduction algorithms transforms high-dimensional data to a more compact low-dimensional data representation. The low-dimensional data representation should ideally contain all of the necessary features of the high-dimensional data, but encoded in a lower dimension. Dimensionality reduction techniques could remove noise, simplify the learning set provided to the learning algorithm, and help the learning algorithm converge faster.

Reinforcement learning lies somewhere between supervised and unsupervised learning methods. In reinforcement learning, the learning algorithm receives feedback whether the answer was correct or wrong, but doesn't get any instructions on how to correct the answer. The learning algorithm explores different possibilities through trial and error until it figures out how to achieve the goal.

Self-supervised learning (SSL) is a machine learning method that could be considered to be part of unsupervised learning. In SSL, the labels are generated automatically from the training data, and supervised learning methods are used for training the system.

3 Foundations of Reinforcement Learning

Reinforcement learning (RL) is a machine learning paradigm which draws inspiration from *behavioral learning theory*, and how animals learn and adapt through interactions with the environment Learning through *trial-and-error* and the *Law of Effect* are the underlying concepts that reinforcement learning builds upon. They suggest that animals learn which actions or behaviour to follow in an environment by repeatedly trying out different actions, and the learned behaviour is based on the type of response the environment provides, meaning that responses the animal considers satisfying are reinforced and are more likely to occur in the future.

Reinforcement learning focuses on how an agent can learn from interaction with an environment, in order to maximize a reward signal (Sutton et al. [41, pp. 1-13]). In RL, the decision maker in the environment is called the *agent*, and the world the agent acts in and learns from is called the *environment*. There could be environments with one agent (single-agent systems), or environments with several agents (multi-agent systems). At every step of interaction, the agent observes the state of the environment, and performs an action in the environment. The action performed by the agent yields a feedback to the agent from the environment, in the form of a *reinforcement*. The reinforcement is either a *reward* or a *punishment* that informs the agent how good the current state of the environment is. The action makes the agent transition to a new state in the environment. The goal of the agent is to try and maximize the reward in the long run (the cumulative reward), which is called the *return*.

Sutton et al. [41] mention three important characteristics that could be considered to be the most distinguishing features of a RL problem:

1. RL problems could be considered to be *closed-loop* problems, since the actions an agent takes influence the input it receives later. The essential closed-loop behaviour of an RL system is illustrated in fig. 3.1.

2. The agent is not informed what actions to take, but discovers the actions that gives the agent the best long-term reward through trial-and-error. It discovers a mapping between the states and the actions that results in a good policy.

3. In an environment, the actions of an agent might influence what rewards the agent obtains in the future, and not only the immediate reward.

The third characteristic is closely related to the *credit assignment problem*. When an agent gets a reward it is often difficult for the agent to know what action should get the credit for it, and similarly what action to blame when the agent gets a bad reward. More generally, it could be described as: the problem of determining the sequence of actions that yielded in a particular outcome [32]. The credit assignment problem usually occurs when the rewards are *sparse* and *delayed*, meaning that the rewards provided to the agent is limited, and often provided at the end of an episode.

An example of this could happen in robotic grasping systems, when the rewards are sparse and delayed. The reward could be set to 1 for a successful grasp and 0 for a failed grasp. The problem or difficulty relating to this reward setup becomes evident when the robot must perform a large sequence of actions before it gets the final reward of 1. In the start of the training, the agent will explore the statespace and try out random actions, and it might be difficult for the agent to even reach the final reward.

3.1 Elements of a Reinforcement Learning System

Apart from the agent and the environment, there are four main elements of a RL system: a reward function, a value function, a policy, and a model of the environment.

The interaction between the agent and the environment could be divided into a sequence of discrete time-steps, $t = 0, 1, 2, \ldots$. The time-steps are usually restricted to be discrete to keep things simple, but could be extended to continuous time-steps. An *episode* constitutes a sequence of time-steps of interaction between the agent and the environment. The agent starts in an initial state, and perform steps until it reaches a *terminal* state, which ends the episode and starts a new episode. Episodes are also called *trajectories* or *rollouts* in RL. Mathematically, a trajectory could be represented as $\tau = (s_0, a_0, s_1, a_1, \ldots)$. The first state in the environment s_0 is often randomly sampled from a start-state distribution ρ_0: $s_0 \sim \rho_0(\cdot)$ [46].

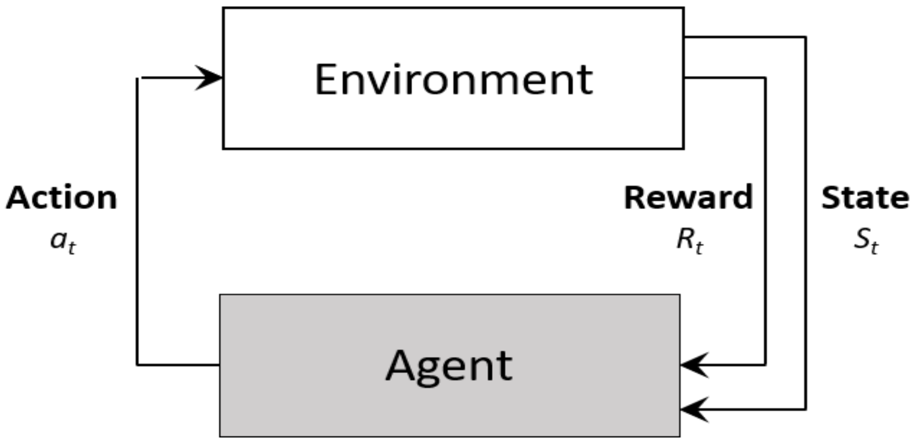


Figure 3.1: The basic components of a RL system.

In the next subsections, terminology related to reinforcement learning and the fundamental ideas which it builds upon will be introduced. Examples of how it relates to the problem of robotic grasping will be given.

3.2 States

A state completely describes the state of the environment, whereas an observation contains only partial information of the state. At every time-step t, the agent obtains a representation of the state of the environment $S_t \in \mathcal{S}$. The set $\mathcal{S}$ denotes the state-space, which is the set of all states in the environment the agent can visit.

Environments where the agent can perceive the full state of the environment are called *fully observed environments*, as opposed to *partially observed environments*.

States in RL problems could be represented in a variety of ways, and there's a distinction to be made between discrete and continous state-spaces. In discrete state-spaces it is possible to enumerate all of the states in the environment. An example of this is in chess, where the state-space are the 64 squares of the board. Traditional reinforcement learning methods typically considers the case of discrete state-spaces, and tabular RL methods are a way to solve them. For continuous state-spaces, a state could be represented as a vector of features, where each feature lies in a continuous range of real numbers. In a robotic grasping scenario, the feature vector could for instance contain the joint angles of the robotic arm as one feature, and the data from a sensor mounted on the arm as the other feature. Each joint angle will lie in a continous range.

In robotics, the true state of the environment is rarely completely observable and noise-free. It will often be difficult for the agent to know which state it is in, and differentiate between states that appears to be similar [23]. Because of this, environments for robotics applications are often represented via partially observable environments. It is common to use filters in robotics applications to get an estimate of the true state [23]. An example of this is a RL system where a robot arm learns to play table-tennis. Two features of the state will be the position and velocity of the ball. In order to track the ball, a Kalman filter might be used to get an estimate of these features along with the uncertainty of the estimate.

3.3 Actions

The set of possible actions $\mathcal{A}$ the agent can perform in an environment is called the *action-space*. At each time-step t in an episode, the agent is in a state S_t and can choose an action $A_t \in \mathcal{A}(S_t)$, where $\mathcal{A}(S_t)$ denotes the available actions in state S_t.

The action-space differs between various environments. Action-spaces could be either discrete or continuous. In the game of PacMan, the action-space is discrete, because the agent can choose to either go in one of four directions, wait in a cell, or eat a pellet. Continuous state-spaces are common in robotics, and actions are usually represented as real-valued vectors. For robotic grasping, an action could be the feature vector representing the amount of torque to apply to the joint motors of the arm.

3.4 Policy

A policy helps an agent decide which action to choose in a particular state. The agent tries to find the policy that maximizes the cumulative reward with respect to the goal of the reinforcement learning task. There are three types of policies in RL: discrete, stochastic, and parametrized policies. Parametrized policies will be discussed later as they are essential for Deep RL (DRL).

Discrete Policy
A discrete policy is a function that maps from a state to a given action: $\pi(s)$.

Stochastic Policy
A stochastic policy is the more general case. It is a function that maps from a given state s to the probabilities for choosing each action: $\pi(a|s)$.

A policy π is considered to be equal to or better than another policy π' if the expected return of π is equal or greater than π' for all states in the state-space:

$$\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s) \quad \forall s \in \mathcal{S}$$

The policy *pi* that satisfies the above inequality is referred to as the *optimal policy*. There might be several optimal policies in a RL problem, and the set of all optimal policies is usually denoted π^*.

3.5 Reward

The reward signal/function is closely related to the *goal* of the RL problem, and the reward signal affects the policy the agent chooses. As mentioned earlier, the agent tries to maximize the reward in the long run, over a trajectory, with respect to a certain goal. The reward is dependent on the state, action taken by the agent, and the next state: $R_t = R(S_{t-1}, A_{t-1}, S_t)$.

3.6 Model of Environment

A model of the environment makes it possible to make inferences of how the environment will behave. The model predicts the next state and reward from the current state and action. *Models* are often used for *planning* in RL, which entails that it's possible to decide on the future actions the agent should take before the agent experiences them.

RL methods that use models for planning are called *Model-based* methods. Their counterpart are referred to as *Model-free* methods and uses trial-and-error for learning the policy.

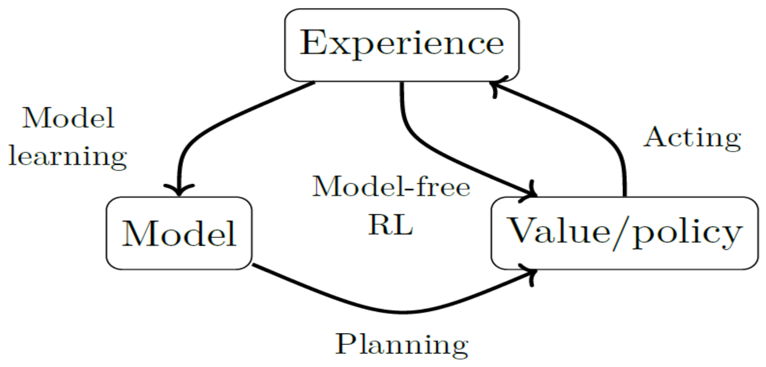


Figure 3.2: Different RL methods: model-based and model-free learning.

Model-free vs Model-based methods

The advantage of having a model of the environment is the ability it gives the agent to plan ahead. It makes it possible for the agent to form a policy based on what it will experience in the future. However, the agent is not always given a model of the environment, but will in many cases have to learn the model from interacting with the environment. The challenge of learning the model from experiences is that the learned model could be biased. The agent could exploit the bias in the model and learn to behave well in this particular environment, but perform badly in the real world scenario [46]. A relevant example of this is an agent learning in a simulated robotic environment. The agent could try to learn the model of the environment, but the model it learns will be biased towards a certain configuration of the real world. The light conditions and dynamics of the simulation might match only a subset of the real world scenarios.

Model-free methods on the other hand often have lower *sample-efficiency* than model-based methods [12]. The sample-efficiency of an algorithm refers to the amount of data or experiences that needs to be gathered before the algorithm reaches a certain performance during training. In other words, an algorithm has low sample-efficiency if it needs to sample many experiences before it learns something useful related to the goal.

3.7 Exploration vs Exploitation

On-line decision making involves a fundamental choice; exploration, where we gather more information that might lead us to better decisions in the future or exploitation, where we make the best decision given current information. Exploration is inherently costly in terms of resource, time and opportunity. It is of great interest to address the dichotomy between exploration of uncharted territory and exploitation of existing knowledge. Such question exists in both the stateless RL settings and in multi-state RL settings. The agent need to balance between greedily exploiting what has been learned so far to choose actions that yield near-term higher rewards, and continuously exploring the environment to acquire more information to potentially achieve long-term benefits [7].

3.8 Markov Chains

Markov chains were initially studied by the mathematician Andrey Markov. A Markov chain is a stochastic process with no memory, which randomly evolves from one state to the next state at each time-step. Markov chains have a finite number of states, and the probability for a state transition from s to s' remains fixed for every time-step. The no-memory condition means that the probability for a state transition solely depends on the current state and next state, and no prior states in the episode. This condition is known as the *Markov property*.

3.9 Markov Decision Processes

The main components and the sequential decision making aspect of a RL problem are often modeled mathematically in terms of a *Markov Decision Process* (MDP). MDP's were described by Richard Bellman in the 1950s. MDP's extend Markov chains to also include actions and rewards, and similarly to Markov chains, MDP's also satisfy the Markov property.

The Markov Property

The Markov property states that the next state the agent transitions to and the reward it gets will only depend on the current state and action, and not the prior states and actions taken by the agent. An example of this is in the game of chess, where the current board position state and move, is enough to evaluate the next board position.

$$P(S_{t+1}|S_t) = P(S_{t+1}|S_1, \ldots, S_t)$$

One of the main differences between Markov chains and MDP's, is that the transition probability for moving from a state s to the next s' doesn't remain fixed, but will depend on the action a chosen by the agent.

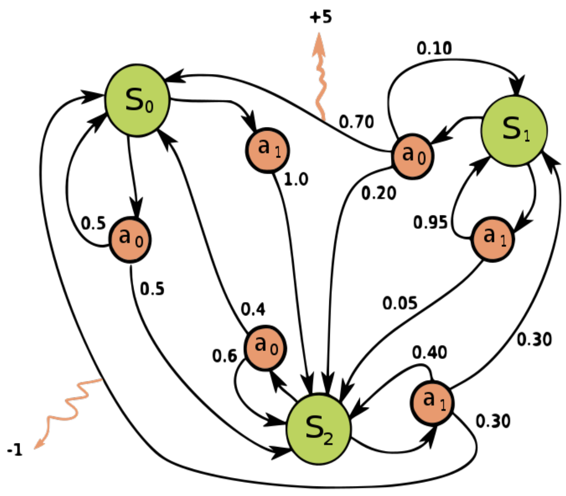


Figure 3.3: An example of a simple MDP from Wikipedia. Green circles are states, orange circles are actions, and red arrows are rewards.

A MDP can be represented as the tuple (S, A, R, P, ρ_0) where:

- $\mathcal{S}$ is the state space
- $\mathcal{A}$ is the action space
- $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$ is the reward function
- $P : \mathcal{S} \times \mathcal{A} \mapsto \mathcal{S}$ is the transition function that maps from a (state, action) tuple to the probability of ending up in a next state s'. $\mathcal{P}(s'|s, a)$ is the transition probability of going to state s' from state s, if action a is chosen. This function can be deterministic or stochastic.
- ρ_0 is the starting state distribution.

The dynamics of a MDP are specified with respect to the *state-transition function*:

$$p(s'|s,a) = P(S_{t+1} = s'|S_t = s, A_t = a) = \sum_{r \in \mathcal{R}} p(s', r|s,a) \tag{3.1}$$

where $p(s', r|s, a)$ specifies the probability of next state and reward s', r given the state-action tuple (s, a):

$$p(s', r|s,a) = P(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a) \tag{3.2}$$

3.10 Expected Return

The objective of the agent is to maximize the expected reward it gets over a trajectory. The first type of return the agent could consider, is the sum of rewards over a certain amount of steps in the environment:

$$G_t = R_{t+1} + R_{t+2} + \cdots + R_T$$

where T denotes the final step in the episode. This return is called the *finite-horizon undiscounted return*, and is used for *episodic tasks*.

The other type of return is called the *finite-horizon discounted return*, and is used for *continuing tasks*:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

where $0 \leq \gamma \leq 1$. The parameter γ is called the *discount rate*.

3.11 Value Functions

Contrary to the reward signal, which is an immediate response to the agent, the value function tries to estimate the rewards the agent can expect to obtain in the future. More specifically, the *value* of a state s gives an estimate of the expected return the agent can expect to obtain in the future, taking into consideration the states that are likely to follow from state s [42, p. 70].

A state could for instance return a low reward, but the value of that state could be good, since the trajectory that follows the state contains good future rewards. The future rewards depend on the actions taken, so the value function necessarily depends on the policy of the agent.

There are two types of fundamental value functions in RL.

The value function mentioned above, considers the value of a state s with respect to a policy π, and is referred to as the *state-value function*: $v_\pi(s)$.

It is defined mathematically, in terms of a MDP, as:

$$v_\pi(s) = \mathbb{E}_\pi\left(G_t \mid S_t = s\right) = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right] \tag{3.3}$$

where $\mathbb{E}_\pi\left(G_t \mid S_t = s\right)$ is the expected value of the return, given the state s at time t.

The other type of value function takes into consideration the action a taken in the state s, and following a policy π thereafter: $q_\pi(s, a)$. It is known as the *action-value function*.

It is defined as:

$$q_\pi(s, a) = \mathbb{E}_\pi\left(G_t \mid S_t = s, A_t = a\right) = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right] \tag{3.4}$$

Equation 3.3 can be formulated recursively:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a)\left[r + \gamma v_\pi(s')\right] \tag{3.5}$$

Similarly, for the action-function 3.4, there is a recursive formulation:

$$q_\pi(s, a) = \sum_{s'} \sum_{r} p(s', r|s, a)\left[r + \gamma v_\pi(s')\right] \tag{3.6}$$

The recursive formulations are a useful way to express the value functions, and it is known as the *Bellman equation for v_π* and the *Bellman equation for q_π*, respectively. It says that the value of the current state is the immediate reward, plus the expected reward from the next state. The Bellman equations for updating and estimating the value of a state is central to many algorithms encountered in RL.

3.12 Optimal Value Functions

Optimal policies have the same state-value function. The state-value function is called the *optimal state-value function* in this case, and denoted v^*:

$$v^*(s) = \max_{\pi} v_{\pi}(s) \quad \forall s \in \mathcal{S} \tag{3.7}$$

Similarly, the action-value function is the same for optimal policies, and called the *optimal action-value function*:

$$q^*(s,a) = \max_{\pi} q_{\pi}(s,a) \quad \forall s \in \mathcal{S}, a \in \mathcal{A}$$

The optimal action-value function can also be written as:

$$q^*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma v^*(S_{t+1}) \mid S_t = s,\ A_t = a\right] \tag{3.8}$$

Equations 3.7 and 3.8 can be written as Bellman equations in the following form ([41, p. 76]):

$$v^*(s) = \max_{\pi} v_{\pi}(s) \quad \forall s \in \mathcal{S} \tag{3.9}$$

$$v^*(s) = \max_{a \in \mathcal{A}(s)} \sum_{s',r} p(s',r|s,a)\left[r + \gamma v^*(s')\right] \tag{3.10}$$

$$q^*(s,a) = \sum_{s',r} p(s',r|s,a)\left[r + \gamma \max_{a'} q^*(s',a')\right] \tag{3.11}$$

and they are refered to as *Bellman optimality equations*.

Since the value functions store a value for each state in the MDP, the memory considerations are of importance when applying RL algorithms. MDP's with huge state and action spaces require a lot of memory to store the approximations of the value functions, policies and the models [42, p. 80]. For MDP's with finite, small state and action spaces it is common to store these approximations in tables. These methods are refered to as *tabular methods*.

However, in many real world applications of RL, the state-spaces are high-dimensional and it is impractical to use the tabular approaches encountered in traditional RL. The value functions need to be approximated in this case. Apart from the memory problem, there's also a problem relating to the time needed to build up the tables and the issue of generalization to new states ([41, p. 255]).

This especially holds true for state and action spaces usually encountered in robotics applications. Such state or action spaces are usually continuous and in the form of sensory inputs. In such tasks, a lot of the states will not have been encountered before by the agent. Therefore, the agent needs to learn by generalizing from previous visited states to the the new states.

Combining traditional RL methods with *function approximation* methods from supervised learning, is one way of getting around this problem. This method forms the basis of *Deep reinforcement learning* (DRL), and will be talked about in the section on DRL.

3.13 Classical Reinforcement Learning Methods

The purpose of reinforcement learning is for the agent to learn an optimal, or nearly-optimal, policy that maximizes the "reward function" or other user-provided reinforcement signal that accumulates from the immediate rewards. This section elaborates on classical reinforcement learning methods.

3.13.1 Dynamic Programming vs Monte Carlo Methods

Reinforcement learning is typically defined in the form of a Markov decision process (MDP). Many reinforcement learning algorithms for this context use dynamic programming techniques. The main difference between the classical dynamic programming methods and reinforcement learning algorithms is that the latter do not assume knowledge of an exact mathematical model of the MDP and they target large MDPs where exact methods become infeasible. Dynamic Programming (DP) requires a complete knowledge of the environment or all possible transitions, whereas Monte Carlo methods (MC) work on a sampled state-action trajectory on one episode. DP includes only one-step transition, whereas MC goes all the way to the end of the episode to the terminal node.

3.13.2 Q-Learning

The Q-learning algorithm was introduced by Chris Watkins in 1989 and was an important breakthrough in RL. It is an off-policy temporal difference (TD) algorithm [42, p. 157]. The simplest form of the algorithm could be written as:

$$q^{\text{new}}(s_t, a_t) \leftarrow q^{\text{old}}(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a q(s_{t+1}, a) - q^{\text{old}}(s_t, a_t) \right] \quad (3.12)$$

where $\alpha \in (0, 1]$ is the learning rate, $\gamma \in [0, 1]$ is the discount factor, $s_t \in \mathcal{S}$, and $a_t \in \mathcal{A}$.

The action-value for each state-action pair is updated using the update rule in 3.12. It has been shown that the learned action-value function directly approximates the optimal action-value function q^*, irrespective of what policy is being followed [42, p. 157].

3.14 Deep Reinforcement Learning Methods

Deep reinforcement learning algorithms (DRL) can be divided into two classes, namely model-free algorithms and model-based algorithms. This work concerns model-free algorithms. For model-free algorithms there are two fundamental types of algorithms: Policy gradient/optimization and Q-learning methods. Additionally, there are methods that combine elements from both. Actor-Critic methods is an example of this.

In DRL, the policy depends on the parameters of the neural network (weights and biases), and are called *parameterized policies*: $\pi_\theta(s)$ for discrete policies and $\pi_\theta(a|s)$ for continuous policies, where θ represents the network parameters.

3.14.1 Policy Gradient and Q-Learning methods

Policy optimization methods learns the policy directly, contrary to Q-learning methods that learns it indirectly from the action-value function. Policy optimization methods optimize the parameters of the network θ directly via gradient ascent/descent on the expected return function, or by maximizing the local approximations of it [46].

Q-learning methods learn the policy by approximating the action-value function $q_\theta(s, a)$. The optimization is often performed off-policy, which facilitates the use of previously collected data for the update. A common technique is to use experience replay, that stores previous experiences in a replay buffer. During training the algorithm samples from it randomly, making the experiences decorrelated. This has been shown to stabilize training [46].

3.14.2 DQN

A Deep Q-Network (DQN) is a DRL method that contrary to the tabular Q-learning method, approximates the action-value function $q_\pi(s, a)$ with a neural network. The network will take the state as input, and output a prediction of the expected return for each action. DQN is an off-policy algorithm, and utilizes a replay buffer to store experiences, and samples from it randomly during training.

4 Simulators

The first part of this chapter concerns simulators used in robotics. It will introduce four robotic simulators that could be used for robotic tasks, particularly for robotic manipulation and grasping scenarios. Additionally it will introduce relevant tools often used in robotic simulators . It will compare the simulators PyBullet, CoppeliaSim and Gazebo with regard to several factors. In the last section, a discussion related to the choice of simulator will be presented.

Learning-based methods for robotic grasping requires data for training. One way of generating this data is through experimental trial and error. The process of learning through trial and error on a real robotic platform is costly, time-consuming, and may result in robotic damage. Therefore, it is often desirable to obtain the training data for the robot in a simulator. After training in simulation, it is possible to specalize the training on the real robotic platform.

However, training in a simulator imposes some challenges. It is necessary that the simulator is able to accurately simulate the robotic task to be performed. Another concern is the transfer of the learned model from the simulater to the real world, called "Sim-to-Real Transfer". This is often challenging due to the "reality gap" (Bousmalis et al., 2018). That is, simulator trained models generally do not work as well in the real world compared to in simulation. Some techniques, such as *domain randomization* and *domain adaptation* have been suggested to overcome these difficulties.

In choosing a simulator for the robotic task to be performed, several factors need to be accounted for. In this project, factors such as active community support, documented usage in research, usability and possible integration with other tools and libraries for machine learning and deep learning have been emphasized in choosing the simulator.

There are a lot of simulators used for robotic grasping and manipulation. Some of the more well-known simulators are: MoJuCo, Gazebo, CoppeliaSim (formerly known as V-REP), Webots, RoboDK, PyBullet, Nvidia Isaac and Unity.

4.1 Robot Operating System

The Robot Operating System (ROS), was initially started back in 2007. Efforts of creating an ongoing collaborative framework for robotics was the initial goal that lead to its creation. It aims at facilitating the creation of

> complex and robust robot behaviour across a wide variety of robotic platforms[1].

Creating general and robust robot software is difficult, and therefore the idea was to fuse together the expertise from different groups working on robotics to make collaboration and building upon existing work easier, and functions like a middleware for robotics.

ROS is an open-source framework that supports Unix based operating systems. It includes several software libraries and tools for building robot applications and writing robot software.

4.2 URDF

Unified Robot Description Format (URDF) is a XML format which is used to describe a robot model. URDF includes descriptions for the kinematics and dynamics of the robot, a visual representation, and a collision representation of the robot. The format can only represent robots in a tree structure, ruling out parallel robots. Another assumption is that the robot consists of rigid links that are connected by joints.

URDF's are the standard format for importing robots in ROS, and can cover a vast majority of robot models, including most articulated robot arms. However,

1. `https://www.ros.org/about-ros/`

they can only specify the properties of a robot separate from the rest of the simulation (e.g. pose of a robot with respect to the world frame of the simulation can not be specified.)

4.3 SDF

Simulation Description Format (SDF) is a new XML format which tries to remedy the deficiencies of the URDF format. SDF is a more complete format that includes descriptions of the robot and the simulation world. It was initially created for the Gazebo simulator. In addition to the URDF format, SDF can define robot sensors, surface properties of a robot, surface textures, friction between joints and links, environment lights, terrain, and many more.

4.4 PyBullet

PyBullet [10] is a Python module for the physics engine Bullet. The Bullet engine was initially developed by Erwin Coumans who won a Scientific and Technical Academy Award for the work he did on Bullet. The physics engine has been used in computer games, for visual effects in animation movies, and in a plethora of research projects. Bullet is a free and open-source project and is currently hosted on GitHub[2]. It supports real-time collision detection, soft body dynamics and rigid body dynamics.

Bullet was initially targeted for game development, visual effects in movies and robotics. With the introduction of PyBullet, it has improved the support for robotics, reinforcement learning and virtual reality (VR) applications. It includes robotic examples for the simulation of the Minitaur qaudruped, running humanoids, and robotic grasping setups.

4.5 CoppeliaSim

CoppeliaSim, formerly called V-REP, is a robot simulator for rigid body dynamics, that was initially developed at Toshiba R&D. It is currently maintained by the company Coppelia Robotics AG, located in Zurich, Switzerland. There are three software versions of CoppeliaSim: player, edu and pro.

2. `https://github.com/bulletphysics/bullet3`

	player	**edu**	**pro**
Full simulation functionality	Yes	Yes	Yes
Full editing capabilities	No	Yes	Yes
Commercial usage	Yes	No	Yes

The pro version is not free, but the player version is free to use for everyone, and the edu version is free to use for students, teachers and people in academia.

CoppeliaSim supports an integrated development environment and is based on a distributed control architecture[3]. The control arhitecture was designed with flexibility, portability and scalability in mind, in the sense that it facilitates easier adjustment of the control code of a specific object/model in the simulation [38].

The user has the ability to choose between or combine different programming techniques in CoppeliaSim: embedded scripts, add-ons, plug-ins, remote API clients and ROS nodes [38].

CoppeliaSim supports several physics engines for the rigid body dynamics calculations: Bullet, ODE, Vortex and Newton. Additionally, there are a lot of builtin modules that could help facilitate robotic research and prototyping, such as sensor and actuator models, motion planning, and forward and inverse kinematics [8].

A recent toolkit named PyRep was introduced by James et al. [19]. It is a toolkit built on top of CoppeliaSim that facilitates rapid prototyping in RL, imitation learning (IL), state estimation, mapping, and computer vision [19].

4.6 Gazebo

Gazebo [24] is a robotics simulator initially developed at University of Southern California in 2002. At the time of its creation, many of the robotic simulators were restricted to 2D worlds. Gazebo was designed as a freely available, open-source simulator to cater to the need for a 3D dynamic multi-robot environment that could recreate and simulate complex environments. Its development was partly motivated by the, at the time, increasing use of robotic vehicles for outdoor applications [24].

By default, Gazebo uses the Open Dynamics Engine (ODE) as the underlying

3. `https://www.coppeliarobotics.com/`

physics engine, to simulate dynamics and kinematics for articulated rigid bodies. However, Gazebo is designed such that it is possible to replace ODE with other physics engines. It has the ability to support Bullet, Simbody and DART physics engines.

It is possible to integrate Gazebo with ROS. The combination of Gazebo as the 3D robotics simulator and ROS as the interface for the robot is a popular setup.

4.7 MuJoCo

MuJoCo [45], short for Multi-Joint dynamics with Contact, was released back in 2015. It was designed for research and development in robotics, biomechanics, graphics and animation, and other application domains that require fast and accurate simulation[4].

MuJoCo has an API for programming in C, and is compatible with Windows, Linux and Mac. The open-source library mujoco-py[5], released by OpenAI, makes it possible to program MuJoCo from Python 3.

MJCF is MuJoCo's native format for loading XML models.

4.8 Software Libraries and Tools

4.8.1 OpenAI Gym

OpenAI Gym is an open-source library for reinforcement learning research made by OpenAI. It was created to provide a variety of benchmark problems for comparing RL algorithms [4]. OpenAI Gym includes many different environments with a common interface, and the environment makes no assumption about how the code for the agent is structured. The gym framework assumes that the environment can be modeled as a MDP, and focuses on episodic RL tasks.

Following Python code illustrates the standard interface an agent follows for interacting with the environment.

Listing 4.1: Gym interface in Python.

4. `http://www.mujoco.org/`
5. `https://github.com/openai/mujoco-py`

```
import gym
env = gym.make("EnvName-v1")
observation = env.reset()
for _ in range(episodes):
  env.render()
  # agent taking random actions
  action = env.action_space.sample()
  # A transition in the MDP
  observation, reward, done, info = env.step(action)

  if done:
    observation = env.reset()
env.close()
```

The main methods that need to be implemented are:

step(self, action)
The step method transitions the environment by the action. It returns a new observation, reward signal, a done flag that indicates if the new state is a terminal state, and an info ...

reset(self)
The reset method resets the state of the environment and returns an observation. It needs to be called at the beginning of every episode.

render(self, mode='human')
The render method renders one frame of the environment.

close()
The close method closes the environment.

seed()
The seed method is used to specify a seed for providing reproducability when doing experiments.

4.8.2 Reinforcement Learning Libraries

OpenAIGym provides a standard framework for the environment. There are a handful of open-source RL algorithm libraries that provide code for the agent side, such as: Stable Baselines, OpenAI Spinning Up and RLlib.

4.9 Comparison

In this section, the robotic simulators PyBullet, CoppeliaSim and Gazebo will be compared. A comparison of features offered by the simulators, their suitability with respect to robotic manipulation tasks, and their application for robotic learning will be discussed. All of the mentioned simulators are being actively developed, used in research and have a large user community, which means it's unlikely that they will be deprecated in the near future. Free and open-source software simulators with a large user community also have an advantage compared to licensed, closed-source software, as they benefit from the contributions from different robotic research groups and, more generally, people working in different robotic fields.

Collins et al. included a citation count from Google Scholar of 28 reviewed simulators in the period from 2016 to 2020, where the citations were gathered from one or more research papers related to the simulator, reference manual or other citation type and the search was filtered with the keyword robotics [8]. Out of all the simulators, the top three cited simulators were Gazebo as number one, MuJoCo and CoppeliaSim coming in second and third, respectively. PyBullet was the seventh most cited simulator.

The citation count for PyBullet, CoppeliaSim and PyBullet suggest their active usage in research, and why it is reasonable to choose these simulators for comparison.

Furthermore, Collins et al. [8] define a robotic simulator as a software that includes at least the following basic functionality:

i Physics engine for realistic modelling

ii Collision detection and friction models

iii Graphical User Interface (GUI)

iv Possibility to import scenes, models and meshes

v API facilitating programming language(s) used by robotics community

vi Builtin models for joint arrays, actuators and sensors ready to use

4.9.1 Features

Both CoppeliaSim and Gazebo support multiple physics engines. According to Collins et al., the support of multiple physics engines is a domain randomization technique that could prevent a learned policy to overfit for the current simulation environment [8]. When it comes to scene editor, PyBullet is the only simulator that doesn't have one. Models and properties need to be edited and configured programmatically. Both CoppeliaSim and Gazebo offers a GUI for scene editing, which can increase usability and facilitate rapid prototyping. Though Gazebo offers a scene editor, it has been stated to be quite limited according to [31]. Pitonakova et al. states similar concerns related to poor general usability, difficulty altering and optimizing models, various issues with the interface, and difficulty installing dependencies for Gazebo and third-party models [35].

Out of all the simulators, CoppeliaSim is the only one that offers mesh editing capabilities and the support for mesh manipulation during runtime (e.g. milling operations). PyBullet is the only simulator that supports soft-body contacts. This could be an asset when trying to simulate more realistic scenarios.

The next three tabels is a summarization of key features provided by the three simulators in terms of general features, models and mesh features, and programming support. The information presented in the tables are based on experience gathered during thesis work and recent work done comparing and surveying robotic simulators [8, 35].

Simulator	CoppeliaSim	PyBullet	Gazebo
		Features	
Platform	Available for Windows, Linux and MacOS. Binaries available for all.	Available for Windows, Linux and MacOS.	Available for Windows, Linux and MacOS. Binaries available for Linux Debian.
Physics engines	Bullet 2.78, Bullet 2.83, ODE, Vortex, Newton	By default, PyBullet uses Bullet 2.x API.	ODE available by default. Possible to change engine to: GazeboBullet, Simbody and DART (Need to build from source).
Code/scene editor	Code and scene editor included.	No scene editor. Possible to use Blender as scene editor. Blender has Bullet integrated.	Code and scene editor included.
Meshes	Possible to manipulate meshes during runtime (e.g., milling operations).	Not supported.	Not supported.
Scene objects	Scene objects can be manipulated (moved, added) during simulation by user. Possible to reset simulation to original state, which facilitates its use for RL.	Scene objects can be manipulated (moved) during simulation by user. Possible to reset simulation to original state, which facilitates its use for RL.	Scene objects can be manipulated (moved, added) during simulation by user. Environment doesn't go back to original state when simulation is reset.
Multi-body import	Yes	Yes	Yes
Soft-body contacts	No	Yes	No
Discrete Element Method (DEM) simulation	Yes	Yes	Fluidix
Fluid Mechanics	No	No	Fluidix
Particles	Includes particle systems.	Not supported.	Not supported.
Headless mode	Yes	Yes	Yes
Teleoperation	Yes	Yes	Yes
Realistic rendering	No	No	No
IK, FK	Yes	Yes	Yes
RGBD, LiDAR sensors	Yes	Yes	Yes
Force sensor	Yes	Yes	Yes
Linear actuator, Cable actuator	Linear actuator	Linear actuator	Linear actuator

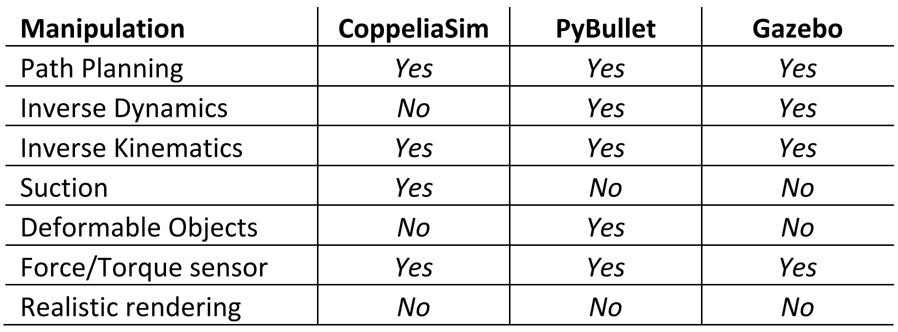
Simulator	CoppeliaSim	PyBullet	Gazebo
Models and mesh capability			
Robot types	Provides various types of default mobile and non-mobile robots such as: humanoids, hexapods, wheeled robots, quadcopter, snake robots and a good selection of robotic arms.	Includes examples for mobile and non-mobile robots such as quadruped robots, humanoid robots, robot arms, etc.	Model library is not as diverse. Includes mostly mobile and aerial robots. Not many robot arms are included in the model library.
Meshes	Meshes are imported as a collection of subcomponents. It facilitates modification of the individual parts of the model. Simulator also supports import of URDF and SDF.	Meshes are imported programmatically as a collision/visual shape from Wavefront OBJ files, and can be combined to form a multibody. The recommended way according to the PyBullet user manual, is to import meshes as URDF, SDF, or MJCF.	Meshes are imported as single objects. Models consisting of several components have to be assembled from DAE files in Gazebo. Simulator also supports import of URDF and SDF.
Mesh modifiability	Possible to modify mesh geometry directly in simulator (mesh simplification, split meshes, combine meshes). Mesh simplification interface allows user to optimize the triangle count of heavy models for faster and more reliable simulation.	Not possible to modify mesh geometry. Needs to be done in third-party software (e.g. Blender).	Not possible to modify mesh geometry in simulator. Needs to be done in third-party software (e.g. Blender).

Simulator	CoppeliaSim	PyBullet	Gazebo
		Programming	
Scene	Scene saved natively (*.ttt format or *.simscene.xml format).	Scene editing done programmatically via Python.	Scene is saved in XML format.
Programming	6 ways to program: embedded scripts, add-ons, plug-ins, remote API clients, ROS nodes and TCP/IP, ZeroMQ nodes.	Python. No built-in support for ROS.	Programming in Gazebo is done via Gazebo C++ plug-ins, ROS nodes or python bindings for Gazebo (pygazebo).
Scripts	Various types of scripts are supported, such as: the sandbox script, add-ons, and embedded scripts. Embedded LUA scripts are part of a scene or a model and are saved/loaded together with the scene.	N/A	Supports scripting via GazeboJs, which provides a scripting interface as a javascript client to the Gazebo simulator. Pitonakova et al. states the difficulty of recognizing how a third-party robot model works and its plug-in dependencies.
API documentation	API documentation, user manual, tutorials and code examples exist. Support forum with a large user	API documentation, quickstart guide/user manual and example tutorials/demos exist. Large support forum	API documentation, guided tutorials for beginners to advanced users, example codes and large user

4.9.2 Robotic Manipulation and Grasping

Robotic manipulation is a large and varied field within robotic research, investigating tasks such as the design of robotic arms and grippers, motion and grasp planning, collaborative manipulation, and specific tasks related to manipulation, such as pick-and-place, assembly, peg-in-hole, deformable objects and shelf picking [9, 8].

The fundamental types of actuators and sensors supported by a robotic simulator partially determines it usability for robotic manipulation and grasping. In [8], the authors state that actuators for position, velocity, and torque control are the most commonly used actuators when controlling a robot arm, and must be supported by the simulator. They also state the need for force/torque sensors, and lists usefull built-in features such as: inverse and forward kinematics algorithms, path planning, and inverse dynamics algorithms.



Manipulation	**CoppeliaSim**	**PyBullet**	**Gazebo**
Path Planning	*Yes*	*Yes*	*Yes*
Inverse Dynamics	*No*	*Yes*	*Yes*
Inverse Kinematics	*Yes*	*Yes*	*Yes*
Suction	*Yes*	*No*	*No*
Deformable Objects	*No*	*Yes*	*No*
Force/Torque sensor	*Yes*	*Yes*	*Yes*
Realistic rendering	*No*	*No*	*No*

Figure 4.1: Robotic manipulations. Table information from recent survey [8].

Another concern when designing robotic grasping and manipulation experiments in simulators, is the accuracy of the rigid/soft body contact dynamics the underlying physics engine provides. One aspect of the simulation is the kinematics and visualization, which help achieve convincing visual effects. However, in robotics applications, the physical accuracy of the simulation is very important. Robotic grasping and manipulation is one domain of robotics that is characterized by heavily-constrained systems, and where the need for accurate modelling of contact dynamics holds true [13].

In [13], the authors compare and evaluate the physics engines: Bullet, Havok, MuJoCo, ODE and PhysX on four model systems with different measures of accuracy. The model systems were simulation instances created in a specific state. They were advanced one time step forward given joint torques input, and the output is a new state in a uniform format. The performance metrics they user are consistency measure, energy and momentum conservation. Additionally, they recorded the CPU timing results for all the model systems and analyzed grasp stability.

The consistency measure was related to integration errors. They first obtained a reference trajectory using a small timestep for the given model system, and gradually increased the timestep where they measured the average deviation from the reference trajectory.

One of the model systems was a grasping scenario with a 35-DOF robotic arm, which is a heavily-constrained system with many simultaneous contacts.

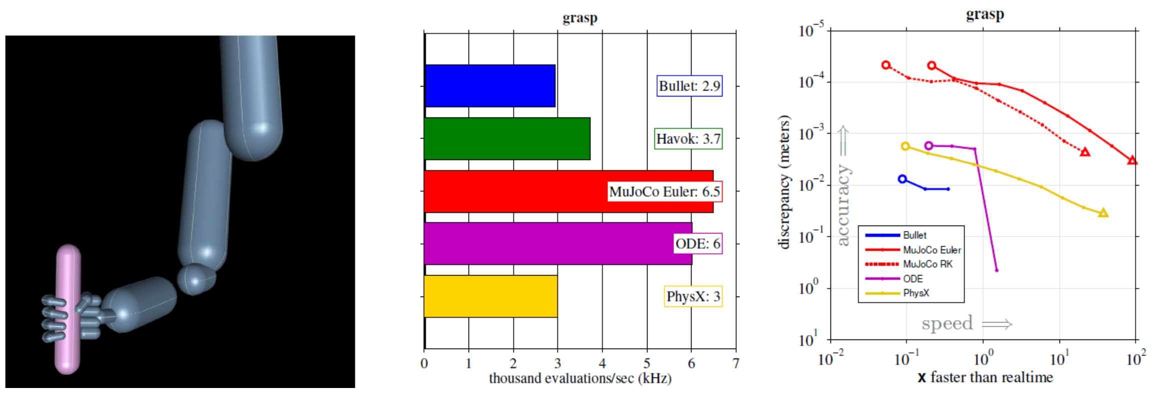


Figure 4.2: Figure from [13]. First column: task related to grasping an object rigidly and moving it around. Second column: CPU timing results as thousands of evaluations per second. Third column: Speed-accuracy trade-off with respect to a consistency measure.

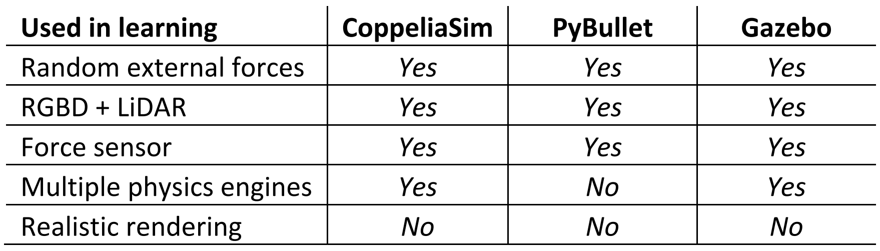
Figure 4.2 shows the model system, cpu timing results, and the speed-accuracy trade-off plot where x is simulation speed and y represents the consistensy measure. The optimal speed-accuracy plot should resemble a Pareto front. Tha authors points out that the reason Bullet and ODE have partial speed-accuracy curves is because for larger timesteps they go unstable on the system.

For grasp stability, they ran the simulation for increasing timestep values and recorded the largest timestep with the object still in the hand after moving it around. Their results show that MuJoCo obtained the best grasp stability, whilst ODE and Bullet obtained the worst.

4.9.3 Robotic Learning in Simulation

There are many advantages of robot learning in simulation. One advantage is that the simulation environment can be used to benchmark and test different algorithms from different areas of machine learning before it is deployed on the real platform. Due to the sample inefficiency of many DRL algorithms, it is often impractical to train the algorithms directly on the real platform, as it requires many episodes before the robot starts to learn good policies, and the exploration of the state-action space can also cause the robot to fail or lead to damage during training [8].

As mentioned in the section above, the accuracy or fidelity of the contact dynamics is important [13, 8], but besides the need for robust contact dynamics, sensor support is also relevant. All of the three simulators have support for RGB-D, LiDAR and force-torque sensors. Out of all the three simulators, CoppeliaSim and Gazebo are the simulators that offers the most diverse set of sensors. In Gazebo, the user also have the ability to add noise to the sensor via a noise model, which may yield a more accurate description of the real sensor. In PyBullet it's more difficult to set up a sensor. There exist functionality for returning RGB image, depth buffer and segmentation mask.



Used in learning	**CoppeliaSim**	**PyBullet**	**Gazebo**
Random external forces	*Yes*	*Yes*	*Yes*
RGBD + LiDAR	*Yes*	*Yes*	*Yes*
Force sensor	*Yes*	*Yes*	*Yes*
Multiple physics engines	*Yes*	*No*	*Yes*
Realistic rendering	*No*	*No*	*No*

Figure 4.3: Robotic learning. Table information from recent survey [8].

In [8], the authors remark upon the importance of a simulators capability of facilitating environment changes between episodes and being able to reset a simulation without shutting down the simulator. Furthermore, they state that overcoming the reality gap is one of the most important consideration researchers need to address when choosing a simulator for robotic learning. Domain randomization is one such mean for overcoming the reality gap when transfering a learned policy to the real platform [44].

Various techniques could include changing the initial position and orientation of the scene objects, randomize textures, light conditions, the characteristics of the camera, and randomizing mass, inertia and friction properties [44, 8]. Small random external forces applied to the robot is another domain randomization technique, and could prevent the policy overfitting to the particular simulation environment [8]. All of the simulators provide this.

All of the simulators have the ability to support domain randomization techniques. The support for randomizing textures of rendered objects and camera characteristics is not supported in Gazebo by default [8]. However, a plugin was created to support domain randomization in Gazebo[6].

None of the simulators provide realistic rendering. This could be a problem when learning vision-based policies for manipulation and grasping in the simulator, and transfering the model to the real platform. However, Tobin et al. showed that it was possible to succesfully transfer a deep neural network trained only on simulated RGB images to the real world where it was used for grasping in clutter [44].

4.10 Discussion

All of the three simulators have decent API documentation, tutorials and support forums with large user bases. Collins et al. [8] provided a citation count that shows the three simulators documented usage in research. It shows that CoppeliaSim, Gazebo, and PyBullet have good documented usage in research compared to the other simulators included in the citation count. Ivaldi et al. performed a survey on robotic simulators based on user feedback [18]. The user ratings with respect to documentation, support, installation, tutorials, advanced usage, active project and community, and API showed that CoppeliaSim had the overall best rating. As mentioned earlier, concerns related to the general usability of Gazebo and various issues with the interface was documented by some authors [31, 35].

Taking into consideration all of these factors, the reviewed surveys, and the different feature comparisons, CoppeliaSim was initially investigated as the robotic simulator for the experiments.

CoppeliaSim offers a lot of useful features for designing robotic experiments including sensors and actuators, robot models, motion planning, forward and inverse kinematics. Out of the three simulators it offers the most diverse set of features. I opted for the combination of CoppeliaSim and PyRep for programming.

6. `https://github.com/jsbruglie/gap`

However, being such a feature-rich simulation enviroment, CoppeliaSim becomes complex for simpler and non-professional tasks. The intended use was to create a simulation that was transferable, utilizing the ROS integration provided by CoppeliaSim, to the real-life Nachi MZ07 six DOF robotic arm contained at the Machine lab at UiT Narvik. However, this led to additional challenges getting the joints of the robotic model to perform according to specifications, controlling the robot during simulations, movement and gripping turned out to be difficult to control.

Eventually, CoppeliaSim was abandoned in favor of PyBullet. PyBullet has released resources related to RL[7] and different RL environments[8] on their Github page. This helps facilitate quick experimentation with various RL environments, and was the primary reason for switching to PyBullet.

7. `https://bit.ly/3y6evrf`
8. `https://bit.ly/3hoy73S`

5 Experiments

In this chapter, the experimental setup will be described, the DRL methods will be described, and the results will be presented.

5.1 Experimental Setup

The robotic grasping experiment in this work is based on a PyBullet OpenAI Gym environment[1] contributed to PyBullet by the authors of [36]. In their work, they used it as a benchmark environment for comparing and evaluating different DRL off-policy algorithms for vision-based robotic grasping.

The PyBullet environment consists of a 7 DOF Kuka robotic arm with a two-finger parallel gripper trying to grasp objects from a bin with random objects. For each episode, the bin spawns a number of random objects from a dataset of 900 objects. The dataset is split into 900 training objects and 100 test objects. The Kuka arm has a fixed number of timesteps per episode to try find a grasp (default is 8). When the max steps has been reached, the episode ends.

This environment (KukaDiverseObjEnv) builds on a simpler environment (KukaGymEnv). The simple environment doesn't support visual observations, and the Kuka arm trains on a single block object.

1. `https://bit.ly/3uhVYpB`

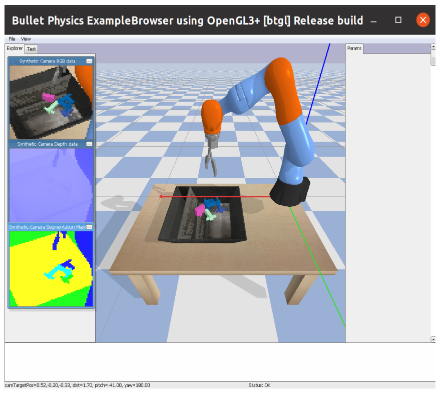

Bullet Physics ExampleBrowser using OpenGL3+ [btgl] Release build

Figure 5.1: KukaDiverseObjEnv

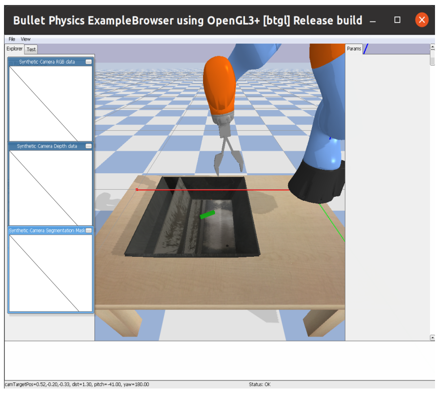

Bullet Physics ExampleBrowser using OpenGL3+ [btgl] Release build

Figure 5.2: KukaGymEnv

5.2 Diverse Kuka Environment

5.2.1 Goal

The goal of the environment is to grasp an object from the tray, and lift it up to a predefined threshold height of 0.2 units. A broader goal is the generalization to the unseen objects in the test set.

5.2.2 The Environment

The environment inherits the Gym environment class, which means it conforms to the OpenAI Gym API and defines the methods mentioned in the OpenAI Gym section. One benefit of the environment using the Gym interface is that it can facilitate the integration with various open-source RL libraries that provide robust implementations of State-of-the-Art RL methods. Stable Baselines, OpenAI Spinning Up and RLlib are some examples of libraries that can provide code for the agent side.

There are two different versions of the PyBullet environment. The first version is a simplified version with a height-hack. This means that for every action taken in an epsiode, the gripper automatically moves downward to the tray a fixed amount. For the second version, the downward z movement is part of the action-space. The second version, is a much harder version of the environment.

5.2.3 Actions

The action-space of the environment depends on the version of the environment and whether it accepts discrete or continuous actions.

Simplified version

- For discrete actions, the agent can choose between seven actions: moving the gripper along the x-axis(2), y-axis(2), yaw rotation of the gripper(2), and one action for not moving(1).
- For continous actions, each of the x, y and yaw actions are limited to the interval $[-1, 1]$.

Harder version

- Same actions as the simplified version with the addition of two actions for moving along the z-axis.
- Same actions as the simplified version with the addition of z-axis in the interval $[-1, 1]$.

5.2.4 Observations

The agent receives visual observations in the form of $(width, height, 3)$ images representing the state. The environment has the ability to provide RGB, RGB-D, and a segmentation mask as image-based observations to the agent.

5.2.5 Reward

For both versions of the environment there is provided a sparse reward. The agent gets a reward of 1 for lifting up an object above the threshold height, and 0 if not.

5.3 Simple Kuka Environment

The goal of the environment is the same goal as in the Diverse Kuka Environment. The most prominent differences between the two environments are: the training set which is reduced to a single block object, the change from image based observation to non-image based observation, and the change of the reward function.

5.3.1 Actions

The action-space is the same as for the Diverse Kuka Environment, where the action-space is discrete and the height-hack is enabled.

5.3.2 Observations

For discrete state-space, the observation is a 9-dimensional vector:

$$(x, y, z, \alpha, \beta, \gamma, x_{\text{gripper}}, y_{\text{gripper}}, \alpha_{\text{gripper}})$$

where x, y, z is the position of the gripper, α, β, γ the orientation of the gripper, and $x_{\text{gripper}}, y_{\text{gripper}}, \alpha_{\text{gripper}}$ is the x, y position and yaw-rotation of the block object relative to the gripper coordinate frame.

5.3.3 Reward

For the Diverse Kuka Environment the reward function was defined as a sparse reward. The reward function for the simple environment is based on the euclidean distance between the gripper and the block object. If the object is grasped and lifted above a height of 0.2, there is an additional reward of 10000.

5.4 DQN

The Q-network used for training, is a Convolutional Neural Network (CNN) that takes a state as input and outputs Q values for each action. The input to the network is a stack of consecutive $(64, 64, 3)$ RGB frames from the camera. One reason for using consecutive frames as input is to try and capture the environment dynamics. Before being sent to the network, the input is converted to grayscale and resized.

Experience replay is used for storing transitions in a cyclic replay buffer, and during training it samples batches from it randomly.

The loss function used for minimizing the error in the network weights is the *Huber Loss*:

$$\mathcal{L}(\delta) = \begin{cases} \frac{1}{2}\delta^2 & \text{for } |\delta| \leq 1 \\ |\delta| - \frac{1}{2} & \text{otherwise} \end{cases}$$

where δ is the temporal difference error:

$$\delta = q(s, a) - (r + \gamma max_a q(s', a))$$

The Huber loss behaves like the mean squared error for small values of the error, and behaves like the mean absolute error for larger values of the error.

Epsilon decay is used to encourage exploration in the beginning of the training and exploitation later on.

The model architecture of the DQN is shown in the figure below.

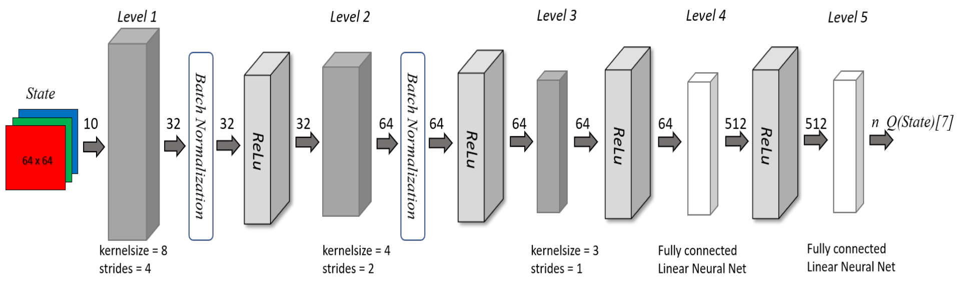


Figure 5.3: Model architecture of baseline DQN.

The input to the model is a stack of frames, and the output are the predicted Q-values $q(s, a_i)$ for each action a_i. In this case, the size of the action-space is 7, so the output of the Q-network are 7 Q-values corresponding to the predicted expected return for each action.

5.5 Experiment 1

Using the Diverse Kuka Environment as described earlier in this chapter, the experiment is using the code offered by Mahyar Abdeetedal in [1]. This code represents the baseline for three differentiated approaches to inspect whether they can improve upon the baseline results, investigating the following three approaches:

1. Changing the static RGB camera to a camera mounted at the gripper (eye-in-hand), making the camera observation dynamic.

2. Changing the vanilla DQN algorithm to the policy gradient algorithm TD3.

3. Switching to the Simple Kuka Environment, and adapt the DQN to handle non-image based observations. Additionally, the reward function is no longer sparse, but takes into account the distance between gripper and object.

The first experiment is based on single-frame observations and a replay buffer of size 100.000.

5.5.1 Moving the camera

The idea behind moving the camera is to find out how dynamically changing images influences the learning. Consequently, the changes in the baseline code are minimal to implement.

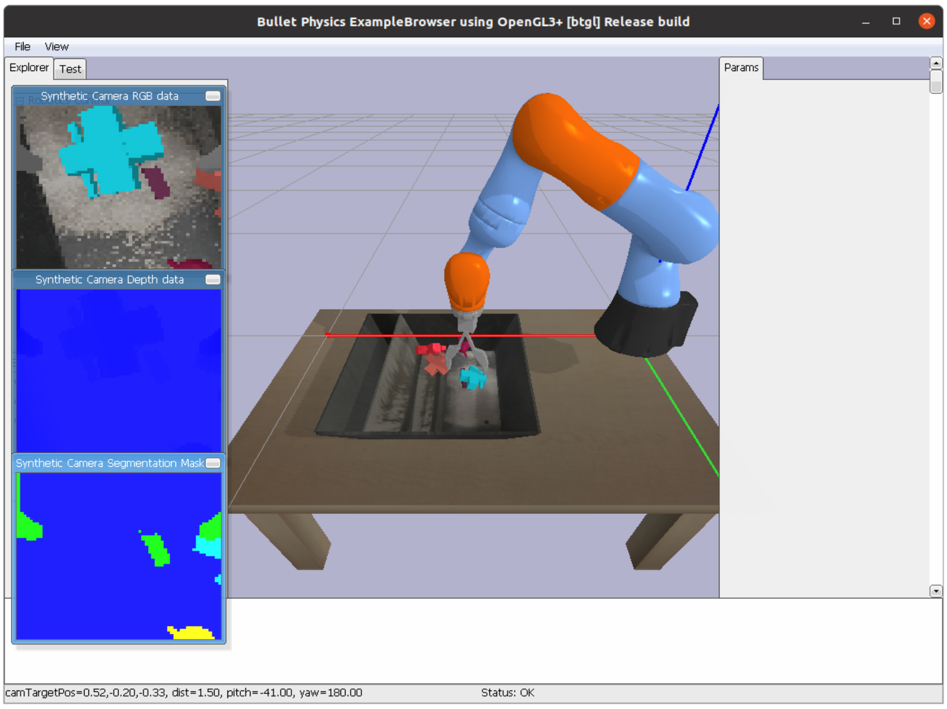

Bullet Physics ExampleBrowser using OpenGL3+ [btgl] Release build
File
View
Explorer
Test
Params
Synthetic Camera RGB data
Synthetic Camera Depth data
Synthetic Camera Segmentation Mask
camTargetPos=0.52,-0.20,-0.33, dist=1.50, pitch=-41.00, yaw=180.00
Status: OK

Figure 5.4: Eye-in-hand camera

5.5.2 TD3

The idea is to replace the Q-learning algorithm DQN with the algorithm Twin Delayed Deep Deterministic Policy Gradient (TD3). TD3 belongs to the class of model-free RL algorithms, and uses both policy optimization and Q-learning, which means it is able to trade-off between the weaknesses and strengths encountered by policy gradient methods and Q-learning methods. Q-Learning methods tend to be less stable, while simultaneously having the advantage of being more sample efficient if tuned correctly, due to their ability to reuse previous experiences [46]. Policy gradient methods tend to be more robust as they arrive at the optimal policy directly, and not indirectly via the action-value function as seen in Q-learning [46]. TD3 only works for continuous action-spaces, while DQN only works for discrete action-spaces.

With this in mind, it is useful to see if TD3 can improve upon the DQN, combining the benefits of both policy gradient methods and Q-learning methods. In this experiment the camera will be dynamically positioned making the observed results comparable to the results acquired, as described in the previous subsection.

A TD3 implementation was developed for this purpose, utilizing the network architecture and replay buffer from [47].

5.5.3 Non-image based observations

The dimensionality of images are fairly large compared to the observation type used in the Simple Kuka Environment. Reducing the data complexity may be an apporach to improve upon baseline results. In this experiment, the baseline DQN is adapted to use the 9-dimensional observation used in the Simple Kuka Environment.

The reward function is also changed in this environment, from a sparse reward to a reward that takes into account the distance between the gripper and block object. It is relevant to see if a reward shaping of the original sparse reward can guide the training better.

5.6 Experiment 2

In the first experiment single frame observations were used. This new experiment takes as input a stack of 10 frames, still using a replay buffer of 100.000. Investigating this setup will give information on whether using a stack of frames will capture the environemnt dynamics better. The experiment will require more time, based on complexity of data and data amount needed to be processed by the algorithms.

The experiment will compare three different approaches:

1. The baseline DQN approach as found in [1], using a static RGB camera.

2. Changing the static RGB camera to a camera mounted at the gripper (eye-in-hand), making the camera observation dynamic.

3. Change the type of input to the network from RGB to greyscale depth images.

Finally, the trained models will be tested on the 100 objects in the testset for 1000 test iterations/episodes. The 100 objects in the test set are new objects, and have not been seen during training. It could hint at the generalization capabilities of the models.

5.7 Experiment 1 - Results

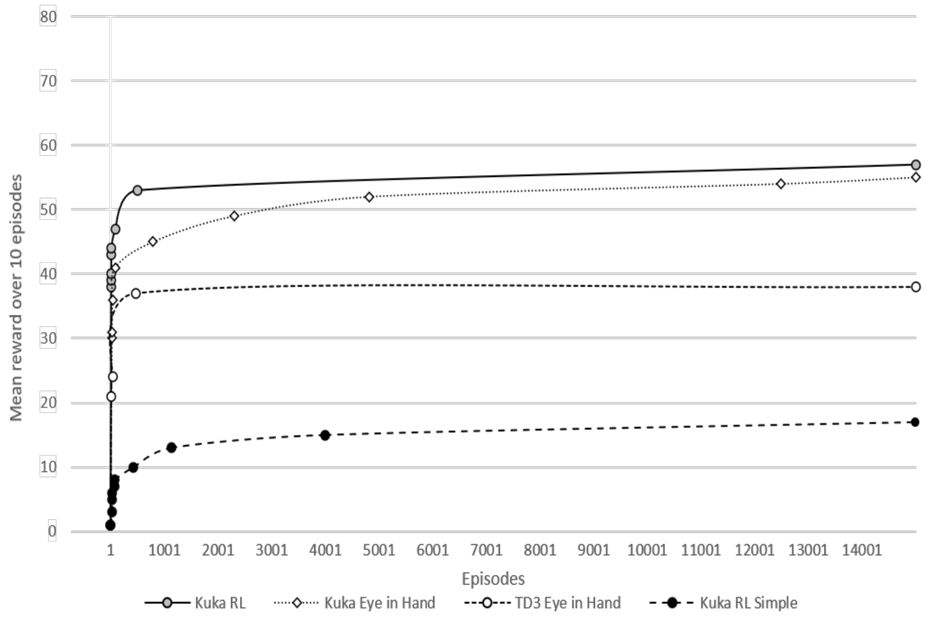


Figure 5.5: Mean avg. reward for previous 10 episodes.

The figure illustrates the mean reward over 10 episodes during training for the DQN approach, labeled *Kuka RL*, the DQN where the camera used as sensory input is moved into the gripper (*Kuka Eye in Hand*, the TD3 approach with camera in gripper (*TD3 Eye in Hand*), and finally the version *Kuka RL Simple* where the DQN is adapted to handle non-image based observations.

The experiment was executed based on single-observation input, that is either a single RGB camera image of 64x64 RGB data except for the *Kuka RL Simple* approach which used an observation consisting of position and orientation of the gripper, and the relative x, y and yaw rotation of the target object with respect to the gripper coordinate frame.

The number of episodes during training was limited to 15.000 due to processing time and for creating comparable result-sets. During training identical datasets were used.

The *Kuka RL Simple* approach did not perform very well. The learning started very slowly and started to level out at around 400 episodes. At the last episode, the approach only leveled out at a 17 mean reward over the 10 last episodes.

The *TD3 Eye in Hand* approach did get better results than the *Kuka RL Simple*, but low mean reward compared to the other approaches. The learning started with fairly the same speed as for *Kuka Eye in Hand* and *Kuka RL*, but leveled out much faster. The mean reward did not increase after episode 700, and ended on a mean reward of 38.

Kuka Eye in Hand and *Kuka RL* had a very close race all the way. Two distinct observations may be commented in this respect. Firstly, *Kuka RL* had a steep increase of mean reward until episode 700 where it levelled out and had a very slow increase culminating at around 14970 reaching a value of 57. Secondly, *Kuka Eye in Hand* had a slower increase of mean reward compared to *Kuka RL* for the first 1.000 episodes, but after this the *Kuka Eye in Hand* is approaching *Kuka RL* and ends up reaching a value of 55 at 15.000 episodes.

5.8 Experiment 2 - Results

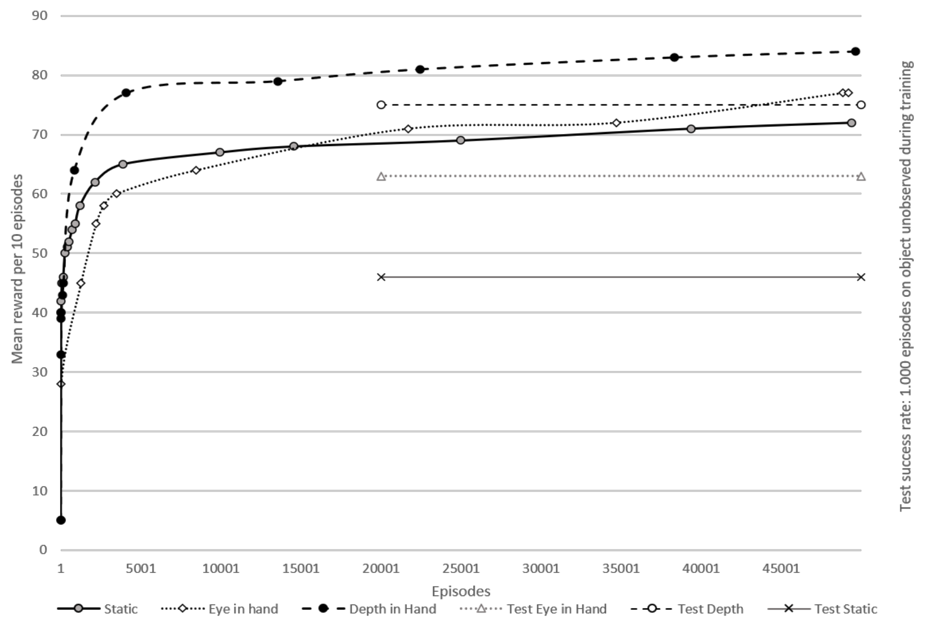


Figure 5.6: Mean avg. reward for previous 10 episodes.

The figure illustrates the mean reward over 10 episodes during training for the DQN approach, labelled *Static*, the DQN with the eye-in-hand camera (*Eye in Hand*), and finally a version substituting the RGB camera with a Depth image camera mounted in the gripper (*Depth in Hand*). The experiment was executed based on a stack of 10 frames used as sensory input to the model. Additionally, after having trained for approximately 50.000 episodes, the approaches were tested for 1.000 test iterations/episodes, where the test set containing 100 previously unseen objects were spawned in the tray. The results are labelled *Test Eye in Hand* for the test of *Eye in Hand*, *Test Depth* for the test of *Depth in Hand*, and *Test Static* for the test of *Static* training. Training and tests used identical reproducible learning- and test-data.

Using a framebuffer of 10 as opposed to a single frame as input to the model could help increase learning, and this is also what can be observed. The *Static* is more or less identical to *Kuka RL* in Experiment 1, and it performs relatively good reaching a mean average of 72 at 50.000 episodes. The testing, *Test Static*, illustrated however that the training did not prepare the network to handle objects not previously seen during training, giving a mean reward over 1.000 episodes of only 46. Rotations of the gripper were barely visible and could be interpreted as random.

Eye in Hand showed a similar behaviour as was observed in Experiment 1, with a more slow increase of mean reward compared to *Static* for the first episodes. But, around episode 15.000 the *Eye in Hand* approach passes the *Static* approach and continues to increase the mean reward over the last 10 episodes until at around episode 49.300 reaching 77. The testing, *Test Eye in Hand*, illustrated that the training to some extent succeeded in preparing the network for objects not previously seen during training, giving a mean reward over 1.000 episodes of 63. Rotations of the gripper became more prominent and oriented the gripper better for grasping objects.

Depth in Hand had a more successful training compared to the two other approaches. *Depth in Hand* used a depth image and converted it to greyscale prior to sending the image into the adapted DQN version. The result showed a steep increase of mean reward until episode 4100 where some degree of leveling occur. The training seems to give better and better results and ended up at 84 mean reward. The testing, illustrated by *Test Depth*, showed that during 1.000 test episodes, an average of 75% of grasp attempts were successful. Rotations of the gripper became more prominent and oriented the gripper better for grasping objects.

5.9 Discussion

The results from the first experiment with single frames as input and a replay buffer of size 100.000 seems to indicate that changing the camera from static to an eye-in-hand camera didn't have a significant effect. In the second experiment, a stack of 10 frames was used as input to the network and a replay buffer of the same size. Notably, the second experiment trained for a longer amount of time than the first experiment. The second experiment shows that changing the camera had an effect at around 15.000 episodes, where the mean reward of the eye-in-hand test surpassed the static camera test. The first experiment trained for approximately 15.000 episodes, and might have shown similar results to experiment two if it had been trained for the same amount of time.

One observed difference between the the static camera and the eye-in-hand camera during testing, was the rotation of the gripper. During testing, the model trained with the eye-in-hand camera (*Eye in Hand*) accounts more for rotation compared to the model trained with the static camera (*Static*). This might be the reason why the *Eye in Hand* surpasses the *Static* in experiment two. The combination of consecutive frames as input and the eye-in-hand camera in experiment two, might result in the model learning to predict a higher expected return from the image observations to the yaw rotation action.

Both DQN and TD3 belong to the class of model-free RL algorithms. This means they sacrifice the potential increase in sample efficiency compared to their counterpart, model-based methods, which often have better sample efficiency [46]. Another concern when training DRL algorithms, is the hyperparameter sensitivity. The results from the model trained with the TD3 algorithm (*TD3 Eye in Hand*) in experiment one, yielded lower mean reward compared to the other models. One reason for this might be the hyperparameters of the TD3 model. DRL methods are often sensitive to hyperparameters, and although TD3 was created to overcome the hyperparameter sensitivity encountered by its predecessor algorithm Deep Deterministic Policy Gradient (DDPG) [46], it still could require some tuning to obtain better results.

The non-image based observation experiment *Kuka RL Simple* have a 9-dimensional observation input consisting of the position of the gripper (x, y, z), the orientation of the gripper (α, β, γ), and the relative x, y position and euler angle of the target object with respect to the gripper coordinate system. The reward function was designed to account for the distance between gripper and target object. The reward function design might be one of the major causes of the low mean reward as it does not take into account the yaw orientation of the gripper. By including the orientation distance between the gripper orientation and the target object orientation in the reward design, it will guide the robot arm to also account for the right grasp orientation.

In the second experiment, the change of input from RGB to depth observation showed a significant increase in mean reward. The depth image input highlights the necessary features, which is the target objects and the height information. Additionally, it removes the noise due to the texture of the tray, as can be seen in figure 5.7. This makes it easier for the CNN net in the DQN algorithm to detect the necessary features, and is perhaps the main reason why depth input had a better result compared to the RGB image input.

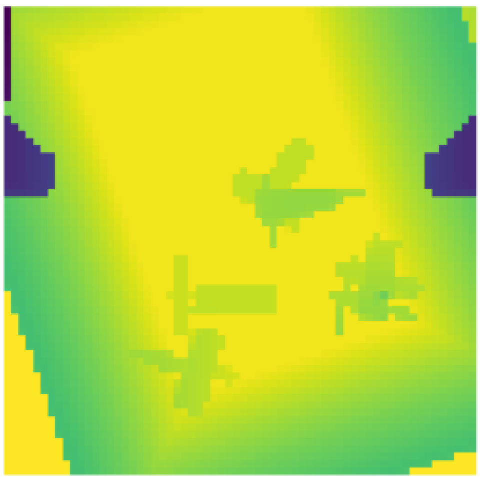


Figure 5.7: (64, 64) depth image observation before grayscale, cubic filtering and cropping preprocessing.

6 Future Work

6.1 Different Algorithms

Future work related to the algorithm part could be to extend the DQN to some of the variants of it, such as: double DQN (DDQN), duelling DQN, and DQN with prioritized experience replay (PER).

Apart from Q-learning methods, the Soft Actor Critic (SAC) algorithm seems like an interesting approach, as it optimizes for expected return and entropy. According to [46], optimizing for a trade-off between entropy and expected return can result in more exploration later on in training, preventing the policy from stagnating at a suboptimal local policy. This might be the reason the DQN leveled out during training, as it settled for a suboptimal policy.

Approaches to help reduce the sample-inefficiency could also be relevant to consider. Curriculum Learning (CL) is one approach that breaks the problem down into a curriculum of tasks gradually increasing in complexity, and utilizes experience from the simpler tasks to solve the original problem [33].

Hindsight Experience Replay (HER) is another technique which could help increase the sample-efficiency when dealing with sparse rewards [2].

6.2 Transfer of Model

Transfering the model to the real Nachi MZ07 robot arm contained at the Machine Lab at UiT Narvik platform requires a preliminary step of changing the Kuka robot arm in the PyBullet environment. Furthermore, the workspace according to the specifications of the Nachi MZ07 model should be accounted for when setting up the digital twin. Additional steps, but not limited to, would be to account for sensor noise, calibration of sensor, and intregration with ROS. Domain Randomization techniques might be necessary to prevent the policy from degrading when transferred to the real platform.

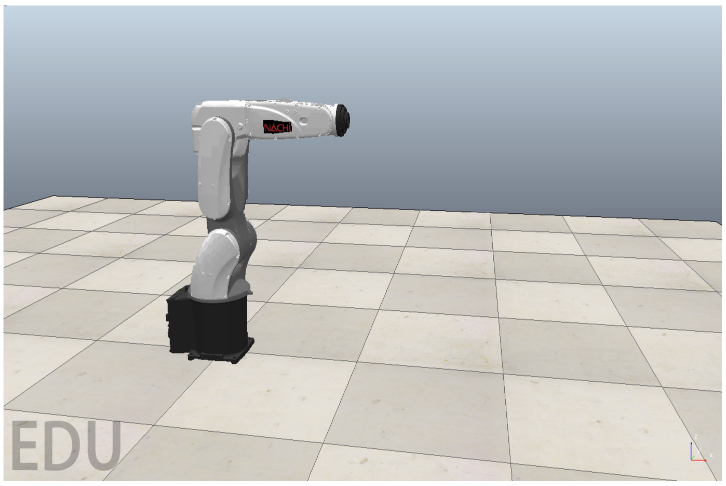


Figure 6.1: Imported and configured Nachi MZ07 model in CoppeliaSim.

6.3 Simulators

Investigating other simulators in more depth could also be listed as future work, especially Gazebo, as it provides good integration with ROS and is one of the most used simulators for robotics.

7 Concluding remarks

The task description of this thesis was very comprehensive and had to be narrowed to fit into the timespan available. Another aspect of the thesis work was that during the lecturing at UiT, robotic simulation environments was only touched upon briefly in the curriculum. Consequently, sufficient knowledge and experience in robotic simulation environments and integration of AI into these environments had to be accomplished during the thesis work.

Limiting the scope of the thesis was necessary to be able to complete the thesis work. By limiting the scope of the thesis, it was decided to try to focus on three specific issues and offer an answer. The questions were:

- *How are learning-based methods applied in current approaches of robotic manipulation and grasping?*
- *What kind of knowledge and software tools are needed in order to set up learning-based experiments in a robotic simulator?*
- *How can learning-based methods be applied in a robotic simulation environment?*

Chapter 2 gave the background for current approaches to robotic manipulation and grasping by describing how the robotic movement and grasping usually is organized. It further presented State-of-the-Art approaches for learning robotic grasping both using supervised methods and reinforcement learning methods.

Chapter 3 focused on reinforcement learning and presented the theoretical foundation approaches for learning in terms of robotic grasping. These two chapters in combination offer an answer to the first question that this thesis try to elaborate on; How are learning-based methods applied in current approaches of robotic manipulation and grasping?

Chapter 4 presented an overview on a subset of robotic simulator environments and an evaluation of three such simulators that are available to the academic community free-of-charge. Chapter 5 offered two distinct set of experiments that focused on elaborating on how simulations can be applied in a robotic simulation environment, and a discussion on the results from the experiments. As such, these chapter offer insight into how learning-based methods can be applied in a robotic simulation environment, thereby an answer to the third question.

The second question, what kind of knowledge and software tools are needed in order to set up learning-based experiments in a robotic simulator, is directly related to elements of Industry 4.0. Robotic technology is one pillar of Industry 4.0, and hold promises to facilitate automation and optimization in industries. But, what are the actual knowledge needed to be able to be able to use these software tools, produce data in a simulation environment, prepare data in the simulation environment for learning, execute the learning in the simulator environment, transfer the learning from the simulator environment into the real-world, and finally deploy this in the real-world robotic environment? The thesis only offers answer to some of these questions. As far as this thesis goes, this may be summarized as follows:

- To be able to operate a robotic simulation environment, knowledge and understanding of the robot in question and the application domain is crucial along with the ability to utilize the features offered by the robotic simulation environment. This is no small or easy task. It requires intimate understanding on how the robot is built, mechanics and control system engineering skills, and finally experience on how to use the robotic simulation environment.

- Programming the simulator and create relevant simulation data for training and testing requires extensive knowledge on sensory capabilities, including computer vision and image processing. Further, knowledge on how to prepare data for training and intimate understanding of how sensory input can be pre-processed and utilized by different AI algorithms, including but not limited to machine learning. Again, this is no simple task. Numerous frameworks offer capabilities in this context, but they also might lack features that could be preferable with the application domain in question.

As far as this thesis goes, the questions stated in the start of the thesis have been addressed. Keep in mind that the domain of robotic manipulation and grasping is a complex field. Adding simulator environments into the equation adds additional complexity.

Bibliography

[1] Mahyar Abdeetedal. Kuka reinforcement learning (dqn), 2020.

[2] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *CoRR*, abs/1707.01495, 2017.

[3] Aude Billard and Danica Kragic. Trends and challenges in robot manipulation. *Science*, 364(6446), 2019.

[4] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[5] Shehan Caldera, Alexander Rassau, and Douglas Chai. Review of deep learning methods in robotic grasp detection. *Multimodal Technologies and Interaction*, 2(3), 2018.

[6] F. Chaumette and S. Hutchinson. Visual servo control, part i: Basic approaches. *IEEE Robotics and Automation Magazine*, 13(4):82–90, December 2006.

[7] Melanie Coggan. Exploration and exploitation in reinforcement learning. 2004.

[8] Jack Collins, Shelvin Chand, Anthony Vanderkop, and David Howard. A review of physics simulators for robotic applications. *IEEE Access*, 9:51416–51431, 2021.

[9] Jack Collins, Jessie McVicar, David Wedlock, Ross Brown, David Howard, and Jurgen Leitner. Benchmarking simulated robotic manipulation through a real world dataset. *IEEE Robotics and Automation Letters*, 5(1):250–257, Jan 2020.

[10] Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. `http://pybullet.`

org, 2016–2019.

[11] Mark R. Cutkosky and Robert D." Howe. *Human Grasp Choice and Robotic Grasp Analysis*, pages 5–31. Springer New York, New York, NY, 1990.

[12] Marc Peter Deisenroth, Gerhard Neumann, and Jan Peters. A survey on policy search for robotics. *Foundations and TrendsÂ® in Robotics*, 2(2):1–142, 2013.

[13] Tom Erez, Yuval Tassa, and Emanuel Todorov. Simulation tools for model-based robotics: Comparison of bullet, havok, mujoco, ode and physx. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4397–4404, 2015.

[14] C. Ferrari and J. Canny. Planning optimal grasps. In *Proceedings 1992 IEEE International Conference on Robotics and Automation*, pages 2290–2295 vol.3, 1992.

[15] Chelsea Finn, Xin Yu Tan, Yan Duan, Trevor Darrell, Sergey Levine, and Pieter Abbeel. Deep spatial autoencoders for visuomotor learning, 2016.

[16] Ruchand Goel and Pooja Gupta. Robotics and industry 4.0. *A Roadmap to Industry 4.0: Smart Production, Sharp Business and Sustainable Development*, pages 157–169, 2020.

[17] Seth Hutchinson, Gregory D. Hager, and Peter I. Corke. A tutorial on visual servo control. *IEEE Transactions on Robotics and Automation*, 12(5):651–670, October 1996.

[18] Serena Ivaldi, Vincent Padois, and Francesco Nori. Tools for dynamics simulation of robots: a survey based on user feedback. *CoRR*, abs/1402.7050, 2014.

[19] Stephen James, Marc Freese, and Andrew J. Davison. Pyrep: Bringing V-REP to deep robot learning. *CoRR*, abs/1906.11176, 2019.

[20] Yun Jiang, Stephen Moseson, and Ashutosh Saxena. Efficient grasping from rgbd images: Learning using a new rectangle representation. In *2011 IEEE International Conference on Robotics and Automation*, pages 3304–3311, 2011.

[21] 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, 2018.

[22] Kilian Kleeberger, Richard Bormann, Werner Kraus, and Marco Huber. A survey on learning-based robotic grasping. *Current Robotics Reports*, 1:239–249, 12 2020.

[23] Jens Kober, J. Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32:1238–1274, 09 2013.

[24] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pages 2149–2154 vol.3, 2004.

[25] Sulabh Kumra and Christopher Kanan. Robotic grasp detection using deep convolutional neural networks. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 769–776, 2017.

[26] Sulabh Kumra and Christopher Kanan. Robotic grasp detection using deep convolutional neural networks, 2017.

[27] Ian Lenz, Honglak Lee, and Ashutosh Saxena. Deep learning for detecting robotic grasps, 2014.

[28] Sergey Levine, Peter Pastor, Alex Krizhevsky, and Deirdre Quillen. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection, 2016.

[29] Jeffrey Mahler and Ken Goldberg. Learning deep policies for robot bin picking by simulating robust grasping sequences. In Sergey Levine, Vincent Vanhoucke, and Ken Goldberg, editors, *Proceedings of the 1st Annual Conference on Robot Learning*, volume 78 of *Proceedings of Machine Learning Research*, pages 515–524. PMLR, 13–15 Nov 2017.

[30] Jeffrey Mahler, Jacky Liang, Sherdil Niyaz, Michael Laskey, Richard Doan, Xinyu Liu, Juan Aparicio Ojea, and Ken Goldberg. Dex-net 2.0: Deep learning to plan robust grasps with synthetic point clouds and analytic grasp metrics, 2017.

[31] Mirella Melo, Jose Neto, Pedro Silva, João Marcelo Teixeira, and Veronica Teichrieb. Analysis and comparison of robotics 3d simulators. pages 242–251, 10 2019.

[32] Marvin Minsky. Steps toward artificial intelligence. *Proceedings of the IRE*, 49(1):8–30, 1961.

[33] Sanmit Narvekar, Bei Peng, Matteo Leonetti, Jivko Sinapov, Matthew E. Taylor, and Peter Stone. Curriculum learning for reinforcement learning domains: A framework and survey. *CoRR*, abs/2003.04960, 2020.

[34] Anand Nayyar and Akshi Kumar. *A Roadmap to Industry 4.0: Smart Production, Sharp Business and Sustainable Development*. Advances in Science, Technology and Innovation. Springer International Publishing, 2020.

[35] Lenka Pitonakova, Manuel Giuliani, Anthony Pipe, and Alan Winfield. Feature and performance comparison of the v-rep, gazebo and argos robot simulators. 02 2018.

[36] Deirdre Quillen, Eric Jang, Ofir Nachum, Chelsea Finn, Julian Ibarz, and Sergey Levine. Deep reinforcement learning for vision-based robotic grasping: A simulated comparative evaluation of off-policy methods. 02 2018.

[37] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection, 2016.

[38] E. Rohmer, S. P. N. Singh, and M. Freese. Coppeliasim (formerly v-rep): a versatile and scalable robot simulation framework. In *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*, 2013. www.coppeliarobotics.com.

[39] A. Sahbani, S. El-Khoury, and P. Bidaud. An overview of 3d object grasp synthesis algorithms. *Robotics and Autonomous Systems*, 60(3):326–336, 2012. Autonomous Grasping.

[40] Shuran Song, Andy Zeng, Johnny Lee, and Thomas Funkhouser. Grasping in the wild:learning 6dof closed-loop grasping from low-cost demonstrations, 2020.

[41] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

[42] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

[43] Christian Szegedy, Alexander Toshev, and Dumitru Erhan. Deep neural

networks for object detection. In *Proceedings of the 26th International Conference on Neural Information Processing Systems*, volume 2 of *NIPS'13*, pages 2553–2561, Red Hook, NY, USA, 2013. Curran Associates Inc.

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

[45] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012.

[46] Spinning Up. Spinning up, 2020.

[47] Matthew Veres. deepq-grasping, 2020.

[48] Manuel Wüthrich, Felix Widmaier, Felix Grimminger, Joel Akpo, Shruti Joshi, Vaibhav Agrawal, Bilal Hammoud, Majid Khadiv, Miroslav Bogdanovic, Vincent Berenz, Julian Viereck, Maximilien Naveau, Ludovic Righetti, Bernhard Schölkopf, and Stefan Bauer. Trifinger: An open-source robot for learning dexterity, 2021.

.1 Appendix - Source Code

The source code could be found at: `https://source.coderefinery.org/JosteinDanielsen/msc-thesis`

.2 Appendix - Project Description

Faculty of Engineering Science and Technology
Department of Computer Science and Computational Engineering
UiT - The Arctic University of Norway

Robotic grasping of 3D objects

Jostein Rene Danielsen

Thesis for Master of Science in Technology / Sivilingeniør

Problem description

This is a thesis regarding robotic grasping of 3D objects and possible application domains, including: pick-and-place (sorting) and/or assembly of 3D objects. Robotic grasping is a complex operation that tries to imitate the way humans grasp objects. Robotic grasping is usually divided into analytic and data-driven methods. Analytic/geometric methods try to analyze the shape of the object to identify a suitable grasp pose. Data-driven methods are based on machine learning techniques. For robotic grasping to work, it is necessary for the robot to know/infer the shape of the object and the 6DoF pose of the object (x,y,z + orientation of object). Path planning and image recognition are also crucial steps in a robotic grasping pipeline.

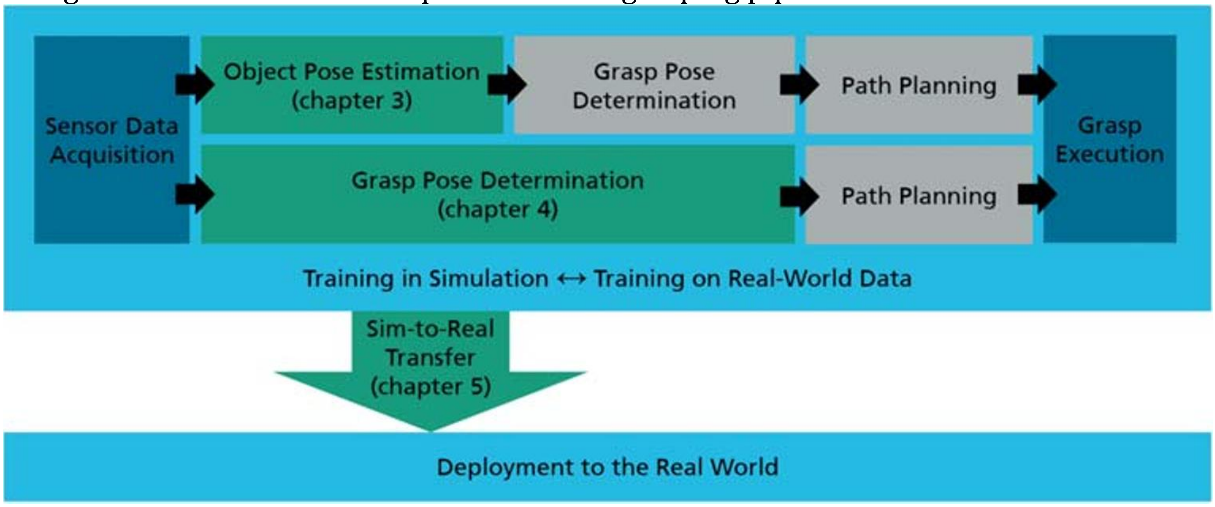

Sensor Data Acquisition
Object Pose Estimation (chapter 3)
Grasp Pose Determination
Path Planning
Grasp Execution
Grasp Pose Determination (chapter 4)
Path Planning
Training in Simulation ↔ Training on Real-World Data
Sim-to-Real Transfer (chapter 5)
Deployment to the Real World

Figure 1: Typical pipeline for robotic grasping. A Survey on Learning-Based Robotic Grasping [Ref. 1].

The suggested thesis should:

- Investigate current approaches (literature review) for robotic grasping and manipulation of 3D objects:
  - Analytic/geometric vs data-driven methods
  - Model-free vs model-based approaches
- For data-driven method:
  - Determine how to train the system. Using supervised learning (SL) or reinforcement learning (RL)
  - Find suitable datasets for training:
    - Annotated datasets
    - Synthetic datasets
    - Augmented datasets
- Determine environments used for training:
  - Furthermore, approaches differ on whether they are trained in a simulation environment, in the real world, or both and utilize various kinds of sensor data (RGB image, depth image, RGB-D image, point cloud, potentially multiple sensors,...) [Ref.1]
  - sim-to-real transfer
- Hardware:
  - *«Du vil derfor jobbe med Raspberry-Pi som utviklingsplattform for robotarmen basert på Makeblock ultimate 2.0 kit (Lenker til en ekstern side.), alternativ en allerede eksisterende robotarm»* [Fra Samlebåndsortering]

    - RGB-D image sensor

- Look into single view 3D reconstruction using 3D reconstruction from point clouds with an RGB-D sensor mounted on a robot arm.
    - Inference and Shape Completion using Keras or PyTorch for fast 3D reconstruction of objects which are free and/or occluded.

- Demonstrate pipeline/methodology for reconstruction of objects with an RGB-D sensor mounted on a robot arm

Keywords:
Robotic grasping (learning-based, physics/geometry based), grasp synthesis, antipodal robotic grasping, robotic pick-and-place (bin-picking), robotic assembly, robotic path planning, sense-plan-act paradigm, robotic 3D reconstruction from point clouds using RGB-D sensor, CNN, reinforcement learning, image processing, computational geometry, YCB Benchmarks – Object and Model Set, VR and simulation environment, synthetic datasets, annotated datasets, augmented datasets, iterative closest point (ICP), Perspective-n-Point (PnP), semantic segmentation, occupancy grid mapping, fast marching cubes algorithm, sim-to-real transfer

References:

- https://link.springer.com/article/10.1007/s43154-020-00021-6 (A survey on Learning-Based Robotic Grasping)
- https://arxiv.org/abs/1809.10790 (Deep object pose estimation NVIDIA)
- https://paperswithcode.com/task/single-view-3d-reconstruction
    - https://ieeexplore.ieee.org/document/8460875 (Using Simulation and Domain Adaptation to Improve Efficiency of Deep Robotic Grasping)
    - https://www.researchgate.net/publication/327808660_Learning_6-DOF_Grasping_Interaction_via_Deep_Geometry-Aware_3D_Representations
- https://arxiv.org/abs/1909.04810 (Antipodal Robotic Grasping using Generative Residual CNN)
- https://arxiv.org/abs/1804.05172 (Closing the Loop for Robotic Grasping: A Real-time, Generative Grasp Synthesis Approach)
- https://arxiv.org/abs/1603.02199 (Learning Hand-Eye Coordination for Robotic Grasping with Deep Learning and Large-Scale Data Collection)
- https://arxiv.org/abs/1609.08546 (Shape Completion Enabled Robotic Grasping)
    - http://shapecompletiongrasping.cs.columbia.edu/
- https://www.researchgate.net/publication/339824179_Robotic_Grasping_Using_Semantic_Segmentation_and_Primitive_Geometric_Model_Based_3D_Pose_Estimation (Robotic Grasping Using Semantic Segmentation and Primitive Geometric Model Based 3D Pose Estimation)
- https://ieeexplore.ieee.org/abstract/document/9212163 (An adaptive robotic grasping with a 2-finger gripper based on deep learning network)

The student should spend the first weeks of his project to investigate state-of-the-art and focus his work. Based on this he should present the specific objectives that his work should pursue and attempt to achieve these objectives in the remaining part of his project. To build on existing UiT research within the field will be considered an asset.

Dates

Date of distributing the task:	<11.01.2021>
Date for submission (deadline):	<15.05.2021>

Contact information

Candidate	Jostein Rene Danielsen
Supervisor at UiT-IVT, IDBI	Bernt Bremdal
Advisor at UiT-IVT, IDBI	Andreas Dyrøy Jansson

General information

This master thesis should include:

- ❄ Preliminary work/literature study related to actual topic
    - A state-of-the-art investigation
    - An analysis of requirement specifications, definitions, design requirements, given standards or norms, guidelines and practical experience etc.
    - Description concerning limitations and size of the task/project
    - Estimated time schedule for the project/ thesis
- ❄ Selection & investigation of actual materials
- ❄ Development (creating a model or model concept)
- ❄ Experimental work (planned in the preliminary work/literature study part)
- ❄ Suggestion for future work/development

Preliminary work/literature study

After the task description has been distributed to the candidate a preliminary study should be completed within 3 weeks. It should include bullet points 1 and 2 in “The work shall include”, and a plan of the progress. The preliminary study may be submitted as a separate report or “natural” incorporated in the main thesis report. A plan of progress and a deviation report (gap report) can be added as an appendix to the thesis.

In any case the preliminary study report/part must be accepted by the supervisor before the student can continue with the rest of the master thesis. In the evaluation of this thesis, emphasis will be placed on the thorough documentation of the work performed.

Reporting requirements

The thesis should be submitted as a research report and could include the following parts; Abstract, Introduction, Material & Methods, Results & Discussion, Conclusions, Acknowledgements, Bibliography, References and Appendices. Choices should be well documented with evidence, references, or logical arguments.

The candidate should in this thesis strive to make the report survey-able, testable, accessible, well written, and documented.

Materials which are developed during the project (thesis) such as software / source code or physical equipment are considered to be a part of this paper (thesis). Documentation for correct use of such information should be added, as far as possible, to this paper (thesis).

The text for this task should be added as an appendix to the report (thesis).

General project requirements

If the tasks or the problems are performed in close cooperation with an external company, the candidate should follow the guidelines or other directives given by the management of the company.

The candidate does not have the authority to enter or access external companies' information system, production equipment or likewise. If such should be necessary for solving the task in a satisfactory way a detailed permission should be given by the management in the company before any action are made.

Any travel cost, printing and phone cost must be covered by the candidate themselves, if and only if, this is not covered by an agreement between the candidate and the management in the enterprises.

If the candidate enters some unexpected problems or challenges during the work with the tasks and these will cause changes to the work plan, it should be addressed to the supervisor at the UiT or the person which is responsible, without any delay in time.

Submission requirements

This thesis should result in a final report with an electronic copy of the report including appendices and necessary software, source code, simulations and calculations. The final report with its appendices will be the basis for the evaluation and grading of the thesis. The report with all materials should be delivered according to the current faculty regulation. If there is an external company that needs a copy of the thesis, the candidate must arrange this. A standard front page, which can be found on the UiT internet site, should be used. Otherwise, refer to the "General guidelines for thesis" and the subject description for master thesis.

The supervisor(s) should receive a copy of the the thesis prior to submission of the final report. The final report with its appendices should be submitted no later than the decided final date.