NTNU

Norwegian University of
Science and Technology

Deep Reinforcement Learning for Robotic Manipulation

Sindre Benjamin Remman

A project which precedes a Master of Science in Cybernetics and Robotics
Supervisor: Anastasios Lekkas, ITK
Submission date: December 2019

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Preface

This report was written based on my work on the pre-project of my master's thesis at NTNU, during the fall of 2019 under the supervision of Anastasios Lekkas.

One of this thesis's two main goals is to examine the viability of using reinforcement learning to handle some parts of the operation of a robotic manipulator in a safety-critical application. To do this, an agent was trained on a simple task using a simulated version of a robotic manipulator, and then the learning was transferred to a real robotic manipulator. The other main goal of this thesis is to examine the usefulness of the combination of the framework ROS and the simulator Gazebo in the context of robotic reinforcement learning.

The pre-project mainly serves as a learning period to prepare for the master's thesis. What I have learned most about during the work on this thesis is scientific writing and reinforcement learning in the domain of robotic manipulators. It has also been very educational to see how it is to write a single piece of text of the size of this report.

The following equipment was provided by NTNU at the start of this semester:

- The robotic manipulator OpenMANIPULATOR-X from ROBOTIS, which was mounted on a wooden base when received.
- A valve that was used by a previous master student during similar work, but was not used during this thesis.
- Raspberry Pi Camera V2.
- Raspberry Pi B.
- Intel Realsense D435 depth camera.
- ADC0844CCN analog to digital converter.

Most of the physical equipment was only tested to ensure that it works, except for the manipulator itself, which was used for some parts of the results in the thesis.

The ROS meta-operating system was used to communicate with the manipulator, using packages that were supplied by ROBOTIS[1]. ROBOTIS also had a setup for simulation of the manipulator in the simulator Gazebo which also was used. The Deep Deterministic Policy Gradient and Hindsight Experience Replay implementation used in the thesis is based on an implementation by Alishba Imran [2]. The implementation uses the machine learning library PyTorch, which I had much experience with from before this project. This implementation was adapted to enable it to be used with ROS, and the hyperparameters were adjusted to give better results on the tasks at hand. The code written for the Gazebo

and ROS applications was written in Python 2.7 and ran on my personal computer.

All figures in the thesis have been created by the author unless otherwise stated. Some of the plots were created using MATLAB, the remaining were created using matplotlib. Most of the other figures that appear in this thesis were created using draw.io.

A final thank you goes to:

- Anastasios Lekkas for his encouraging support and supervision throughout the semester.
- The department of Engineering Cybernetics's workshop at NTNU, for creating the lever that will be used in the upcoming work on the master's thesis.
- Anders Haver Vagle, who worked on a similar project in the spring of 2019, and who provided some insights which helped me increase the simulation speed in Gazebo.
- Alberto Ezquerro, Miguel Angel Rodriguez and Ricardo Tellez for creating the ROS package openai_ros, which made it easier to set up the reinforcement learning environment for Gazebo [3, 4].
- Karl Ylvisaker who worked on a similar project this fall, and who designed the idea for the lever made by the department of Engineering Cybernetics's workshop.
- Thomas Nakken Larsen, fellow Cybernetics student and my roommate, who helped me with filming and clipping the video demonstration enclosed with this report.
- Friends and family for their continued support throughout this period with a more demanding workload than normal.

Abstract

In robotics, there exist many problems which are hard to engineer an explicit solution for. It may be that the robot has a complex structure, or it may be that the required behavior is difficult to define beforehand. An example of this last case is when strong uncertainties are part of the problem. Based on the recent advances in Artificial Intelligence (AI), and more specifically in Reinforcement Learning (RL), this thesis aims to examine the combination of RL and robotics. RL can be used to make an agent learn how to control the robot, and it can also enable the robot to adapt to unexpected situations. The variant of RL used in this thesis is called Deep Reinforcement Learning (DRL). This variation combines traditional RL with the powerful modeling tool Artificial Neural Networks (ANNs).

To examine the combination of RL and robotics, the robotic manipulator OpenMANIPULATOR-X was used. This manipulator was provided by NTNU at the start of the semester. Using a simulated model of the manipulator, which was simulated in the simulator Gazebo, and Robot Operating System (ROS), a meta-operating system for robotics, an RL agent was trained to do a task. This task involved making the end-effector of the manipulator reach a randomly selected point in the manipulator's workspace.

After the RL agent was trained on the task in the simulator, the agent was transferred to the real world manipulator. The agent had sufficiently good performance, especially considering that it was never trained on the real manipulator. A video demonstration was made of the experiments done with the real world manipulator. This demonstration is enclosed together with this report.

When this project is continued in the spring of 2020 during the work on the master's thesis, a new objective is needed. This objective will be to create an RL agent which can learn how to push and pull a lever using the manipulator mentioned above. Therefore, a lever was made in collaboration with fellow master student Karl Ylvisaker and the workshop of the Department of Engineering Cybernetics during the work on this pre-project.

Sammendrag

Det eksisterer mange problemer innenfor robotikk som er vanskelige å designe en eksplisitt løsning for. Det kan være på grunn av at roboten har en kompleks oppbygning, eller det kan være at robotens oppførsel er vanskelig å definere på forhånd. Et eksempel på sistnevnte er hvis det er mye usikkerhet i problemet. Basert på nylige fremskritt innen kunstig intelligens, og mer spesifikt innen *forsterkende læring* (eng. Reinforcement Learning), vil denne rapporten utforske kombinasjonen av forsterkende læring og robotikk. Forsterkende læring kan brukes for å lage en agent som kan lære seg å styre roboten, og det kan også la roboten tilplasse seg til uventede situasjoner. Varianten av forsterkende læring som brukes i denne rapporten er *dyp forsterkende læring* (eng. Deep Reinforcement Learning). Denne variasjonen kommer fra å kombinere tradisjonell forsterkende læring med det effektive modelleringsverktøyet *kunstige nevrale nettverk*.

For å undersøke kombinasjonen mellom forsterkende læring og robotikk, ble robotmanipulatoren OpenMANIPULATOR-X brukt. Denne manipulatoren ble supplert av NTNU ved starten på semesteret. Ved å bruke en simulert versjon av manipulatoren, som ble simulert i simulatoren Gazebo, og ROS, et meta-operativsystem for robotikk, ble en agent som bruker forsterkende læring opptrent til å utføre en oppgave. Denne oppgaven besto i å få endepunktet til manipulatoren til å nå tilfeldig valgte punkter i manipulatorens arbeidsområde.

Etter at agenten var blitt trent på oppgaven i simulatoren, ble agenten overført til den virkelige manipulatoren. Agenten presterte godt, særlig med betraktning av at den aldri hadde blitt trent på den virkelige manipulatoren. En videodemonstrasjon ble lagd av eksperimentene som ble gjort på den virkelige manipulatoren. Denne demonstrasjonen er vedlagt sammen med denne rapporten.

Når dette prosjektet blir fortsatt våren 2020 i sammenheng med masteroppgaven, trengs det en ny oppgave. Denne oppgaven blir å lage en agent som kan lære seg, ved hjelp av forsterkende læring, å trekke og dytte en spake med manipulatoren nevnt over. I den sammenheng ble en spake lagd i løpet av arbeidet på dette prosjektet. Spaken ble lagd i samarbeid med Karl Ylvisaker, som også er masterstudent, og det tekniske verkstedet til Instituttet for teknisk kybernetikk.

Contents

Preface i

Abstract iii

Sammendrag iv

Table of Contents vi

List of Tables vii

List of Figures ix

Glossary xi

Acronyms xiii

1 Introduction 1
- 1.1 Background and motivation . . . 1
- 1.2 Objectives and research questions . . . 3
- 1.3 Contributions . . . 4
- 1.4 Outline of the report . . . 4

2 Theory 7
- 2.1 Machine learning . . . 7
  - 2.1.1 Reinforcement learning . . . 8
  - 2.1.2 Artificial Neural Networks . . . 11
  - 2.1.3 Deep reinforcement learning . . . 23
  - 2.1.4 Reward shaping/functions . . . 25
  - 2.1.5 Transfer learning from simulator to real-world . . . 28
- 2.2 Robotic manipulators . . . 29
  - 2.2.1 Forward kinematics . . . 30
  - 2.2.2 Inverse kinematics . . . 30

3 Equipment and setup **31**
3.1 Software . . . 31
3.1.1 Robot Operating System . . . 31
3.1.2 Gazebo . . . 31
3.1.3 PyTorch . . . 32
3.1.4 OpenAI Gym . . . 32
3.1.5 openai_ros package . . . 32
3.2 Hardware . . . 33
3.2.1 OpenMANIPULATOR-X . . . 33
3.2.2 Lever . . . 34

4 Implementation and results **37**
4.1 FetchReach-v1 task in OpenAI Gym . . . 37
4.2 Fetch-reach task in Gazebo . . . 37
4.3 Transferring the learned policy to the real-world manipulator . . . 44

5 Discussion and analysis **47**
5.1 MuJoCo versus Gazebo . . . 47
5.2 Performance of the tasks . . . 48
5.3 Transferring the policy learned in the simulator to the real world manipulator 48

6 Conclusion **51**
6.1 Answering the research questions . . . 51
6.2 Further work . . . 52

Bibliography **55**

List of Tables

4.1 Hyperparameters for FetchReach-v1 and the tasks created for Gazebo . . 42
4.2 The results from transferring the learned policy to the real-world manipulator and comparing with the actual inverse kinematics 45

List of Figures

2.1 Illustration of the Reinforcement Learning (RL) process 9
2.2 Example of a perceptron 13
2.3 Example of a neural network with a single hidden layer 14
2.4 Plot of Sigmoid and Perceptron activation functions 14
2.5 Plot of Tanh and ReLU activation functions 19
2.6 Illustration of the workings of local receptive fields 20
2.7 Illustration of the workings of local receptive fields with stride=2 21
2.8 Example of a convolutional layer 21
2.9 Example of a max-pool layer 22
2.10 Example of an average-pool layer 22

3.1 The structure used in openai_ros, taken from [3] 33
3.2 The old valve with the manipulator 34
3.3 The new lever with the manipulator 35

4.1 How the FetchReach-v1 task looks like in MuJoCo 38
4.2 The success rate for the FetchReach-v1 task 38
4.3 Gazebo task setup 39
4.4 Actor-network architecture 41
4.5 Critic-network architecture 41
4.6 Success rate and average reward when the goal is at a fixed point in front of the manipulator. 42
4.7 Success rate and average reward when the goal is at a random point 180 degrees in front of the manipulator 43
4.8 Success rate and average reward when the goal is at a random point in the work-space of the manipulator (360 degrees around it). 44

Glossary

agent Anything that can observe the environment through sensors and act on it using actuators [5, p.34]. i, iii, 5, 9, 25, 27–29, 37, 39, 48, 51, 52

black box A device where the internal mechanics are not considered, only the relationship between output and input. 29

Markovian Transition Model

$$P(s_{t+1} = s' | s_t, s_{t-1}, \ldots, s_1, s_0, a_t) = P(s_{t+1} = s' | s_t = s, a_t = a)$$

The probability of reaching a state *s'* if action *a* is taken in state *s* depends only on state *s*, and not on earlier states [5, p.646].. 8, 9

Acronyms

AI Artificial Intelligence. iii

ANN Artificial Neural Network. iii, 1, 4, 7, 12, 13, 15, 28

CNN Convolutional Neural Network. 4, 7, 19, 22, 23

DDPG Deep Deterministic Policy Gradient. 3, 4, 24, 25, 52

DOF Degrees of Freedom. 2, 25, 33, 51

DQN Deep Q-Network. 23–25

DRL Deep Reinforcement Learning. iii, 1, 2, 4, 7, 22, 23, 28, 32, 34, 52

HER Hindsight Experience Replay. 3, 4, 27, 28, 39, 52

MDP Markov Decision Process. 9

MSE Mean Squared Error. 15, 18

ReLU Rectified Linear Unit. 18, 40

RL Reinforcement Learning. iii, ix, 1–5, 7, 9, 11, 22, 23, 25, 27, 29, 32, 39, 47, 48, 51–53

ROS Robot Operating System. iii, 3, 31, 33, 51

SGD Stochastic Gradient Descent. 17

Tanh Hyperbolic Tangent. 18, 40

Chapter 1

Introduction

1.1 Background and motivation

> *An autonomous system is one that possesses self-governing characteristics which, ideally, allow it to perform pre-specified tasks/missions without human intervention* [6].

The yearning for autonomous robots has existed since their inception. As it stands today, there is still a long way to go to completely autonomous robots that can adapt to arbitrary situations. Merging human-level adaptability with robots has proven a very difficult task. Nevertheless, in the last few years, researchers have come increasingly closer to achieving this goal.

There are countless situations where it may be very difficult or even impossible to predict what the robot needs to do. There may be elements that are difficult to model, for instance, friction, or there may be elements that have an unpredictable nature, such as uncertainties in the robot's environment. Control theory and robotics provide tools for explicitly designing a solution to robotics problems, but this may in some cases not be enough. Even though these tools exist, it still means that the robot is reacting and acting according to already defined behavior. What is required for improving robot autonomity further, is for the robot to have the ability to learn from its experiences, and improve its behavior based on what it has learned. This is where Reinforcement Learning (RL) comes into the picture as a promising option. Compared to traditional methods, which assume a more closed form world, RL can enable the robot to discover new behavior based on what it experiences.

RL has already had great success in other areas. DeepMind managed in 2013 to combine RL with an Artificial Neural Network (ANN) to create the first deep learning model to learn control policies directly from high-dimensionality sensory input. Specifically, they created a computer program that learns to play Atari 2600 games using just the raw pixels of the screen as input. When RL is combined with ANNs the resulting field is called Deep Reinforcement Learning (DRL) [7]. In the board games chess, shogi and Go, a

system named AlphaZero (also created by DeepMind) managed to beat world-champion computer programs in all three games using DRL[8]. Recently, DRL has also been used successfully on robotic manipulation tasks [9], which provides some of the inspiration for this thesis.

RL provides a framework that can be used to enable a robot "to autonomously discover an optimal behavior through trial-and-error interactions with its environment"[10]. This means that instead of the engineer explicitly designing a solution to a problem, the engineer can facilitate for the robot to discover the solution on its own. As of today, combining RL with robotics has both positive and negative sides. This ability to learn new strategies is one of the more, if not the most, desirable traits of robotic reinforcement learning.

There are certainly also negatives; several roadblocks exist that limit the faster development of more intelligent robots. Kober et al. [10, pp.1249-1252] describe four challenges that robot reinforcement learning has to overcome:

- *The curse* of dimensionality
  - Robots often have high-dimensional state and action spaces because of their many Degrees of Freedom (DOF). This means that evaluating every state and action becomes unfeasible using traditional methods. This challenge, as described later in this thesis, is currently solved by using neural networks.
- *The curse* of real-world samples
  - Robotic reinforcement learning suffers from many real-world problems. Some of these are: depreciation due to overuse of the robot, the dynamics of the robot may change during operation, time-discretization because the algorithms are implemented on a computer, and being unable to speed up the robot's operation during training.
  - This curse is often attempted to be solved by using a simulator such as Gazebo[11] or MuJoCo[12] to simulate the robot and its environment. This leads to the next curse.
- *The curse* of under-modeling and model uncertainty
  - When simulators are used to train an RL program on a robot, this curse emerges. Small errors in the simulator compared to the real-world can accumulate when training, which makes the program unable to perform as well when applied to the real robot.
- *The curse* of goal specification
  - Specifying the goal and how the RL program should get incentivized to reach this goal is one of the most difficult challenges in RL. Specify the goal too detailed, and the engineer is essentially telling the robot what to do; specify it too vague, and the robot may never solve the problem.

To better understand how to use RL with robots, more experiments and research is needed. The motivation of this thesis is therefore to improve the understanding of how to do robotic reinforcement learning successfully.

1.2 Objectives and research questions

Following is the research questions of this thesis:

- Is it feasible to let reinforcement learning handle (at least some parts) of the operation of a robotic manipulator in safety-critical applications?
- How practical is it to use the simulator Gazebo and the framework ROS for reinforcement Learning?

To answer these research questions, a sequence of objectives has been created:

1. Build a theoretical background in reinforcement learning.
    - As the author had little experience with RL before this project, especially policy gradient methods, a significant amount of time was spent at the beginning of the project period to learn about this.
2. Learn how to perform real-time control of the manipulator in Gazebo and the real-world using Robot Operating System (ROS).
    - As the author had no experience with either Gazebo or ROS before this project, a considerable amount of time was spent during the whole semester to learn about these.
3. Decide which reinforcement learning algorithm to use for the experiments.
    - The algorithm that was used for the experiments is Deep Deterministic Policy Gradient (DDPG), which was combined with the technique Hindsight Experience Replay (HER).
4. Do some experiments with the selected reinforcement learning algorithm using an environment provided by OpenAI Gym.
    - The OpenAI Gym environment selected for this was the FetchReach-v1 environment.
5. Set up an environment for reinforcement learning in Gazebo.
    - Using the openai_ros package as a framework, the environment was created.
6. Create a task to test the environment created, and train an agent on this task.
    - Using the openai_ros package as a framework, a reacher task was created. Then an agent was trained on this task and the results from this are shown in Chapter 4.

1.3 Contributions

A reinforcement learning environment was created for the robotic manipulator OpenMANIPULATOR-X in the OpenAI Gym. This environment uses a simulated version of the manipulator in the simulator Gazebo. A task was created for the environment, where the objective is for the manipulator to place its end-effector in the vicinity of a given random point. When this environment is refined further during work on the upcoming master's thesis, it may be released online.

A physical lever was created, which is better suited for the OpenMANIPULATOR-X than the valve previously made for similar projects. The lever includes a servo motor that can be used to reset the lever back to its original position. This lever was designed in collaboration with fellow master student Karl Ylvisaker and was created by the Department of Engineering Cybernetics's workshop at NTNU. The department of Engineering Cybernetics's workshop also provided the servo motor used in the lever.

The relatively new technique Hindsight Experience Replay (HER), combined with the reinforcement learning algorithm DDPG, was applied to the simulated version of the manipulator on the task mentioned above. HER was introduced by [13] in 2017 and DDPG was introduced by [9] in 2015. This solved the task well, even considering the very sparse rewards provided, and the continuous state and action spaces. The policy was also transferred to the real manipulator, and performed relatively well, even though it had never been trained on the real manipulator.

1.4 Outline of the report

The report is divided into six chapters:

- Chapter 2: Theory
  - It starts by giving a brief definition of the different types of machine learning. After that, an introduction to RL and the concepts used in this is given. Then an explanation of ANNs is provided, which entails both feedforward neural networks and Convolutional Neural Networks (CNNs). After that, a walkthrough of DRL and the algorithm used in this thesis is given. To finish the machine learning section of the theory chapter, some additional concepts such as Hindsight Experience Replay (HER), reward shaping, and transfer learning from simulator to real-world are explained. At the end of the theory chapter, a brief overview of robotic manipulators and some of the main terminology using in combination with these are given.
- Chapter 3: Equipment and setup
  - An overview of the main software and hardware used in this thesis is provided.
- Chapter 4: Implementation and results

  - This chapter explains how the tasks solved by the RL agent was set up, and the results from these tasks are given.

- Chapter 5: Discussion and analysis
  - The results from Chapter 4 are discussed and analyzed.

- Chapter 6: Conclusion
  - Finally, a conclusion to the thesis is given. This conclusion first aims to answer the research questions posed in Section 1.2. In the end, an overview of the work that remains to do in the future on this project is provided. This work will be continued in the author's master's thesis this upcoming spring of 2020.

Chapter 2

Theory

2.1 Machine learning

The idea behind machine learning is to help solve problems that are difficult to engineer a solution to by hand. Some examples of such problems are computer vision tasks, speech- and text recognition. There are three main variants of machine learning:

- Supervised learning
  - Concerns learning from a training set of labeled examples [14, p.2]. Often used in, for instance, image recognition.
- Unsupervised learning
  - Concerns finding structures in collections of unlabeled data [14, p.2].
- Reinforcement learning
  - It is more similar to how humans learn compared to the other variants. Mainly concerns how a decision-making agent can obtain information about its environment by exploring, and then exploiting this information to maximize some feedback metric [14, pp. 1-3].

This chapter and thesis, mainly considers Reinforcement Learning (RL), although some examples from supervised learning are used when considering Artificial Neural Networks (ANNs) and Convolutional Neural Networks (CNNs). Specifically, the variant of RL that is used is called Deep Reinforcement Learning (DRL). DRL combines traditional RL with the modeling power of ANNs. In this section, the different parts that make up DRL are discussed, in addition to the different variants of DRL that are relevant for this thesis. Some problems and solutions that are more specific for robotic RL are also explored.

2.1.1 Reinforcement learning

First, a brief overview of some of the terms used in this section is given:

- Agent:
  - An agent is here defined as something that can observe the environment (through sensors in the real-world) and act on it (using actuators in the real-world) [5, p. 34].
- State:
  - A state, s, represents all the information needed to describe what is important in the problem that is being modeled at the given time [15, p. 10]. The observation/state space is denoted by $\mathcal{S}$. This is the space that contains all the states that are available to the agent.
- Action
  - An action, a, is what is used by the agent to act on the environment. Not all actions can be performed in all states; the actions that can be performed in a given state is denoted by $A(s)$. The action space is denoted by $\mathcal{A}$. This is the space that contains all the actions that are available to the agent.
- Transition model
  - A transition model $T(s, a, s') = P(s'|s, a)$[1] is the probability that the next state is state s' given that the agent takes action a in state s. In a Markovian Transition Model, the assumption is that the next state is not influenced by what the previous states were, only the current state and the action taken in that state. The transition model is a proper probability distribution over the possible next states. This means that [15, p. 11]

$$0 \leq T(s, a, s') \leq 1, \forall s, s' \in \mathcal{S}, \forall a \in \mathcal{A}$$

$$\sum_{s' \in \mathcal{S}} T(s, a, s') = 1, \forall s \in \mathcal{S}, \forall a \in \mathcal{A}.$$

- Reward function
  - A function that specifies how the reward is given. The reward function can, for instance, be dependent on the current state, $R(s)$; current state and action, $R(s, a)$; or the transition, $R(s', s, a)$.
- Off-policy vs on-policy algorithms
  - An off-policy algorithm uses a different policy to act in the environment than what it uses to decide how the value function and/or policy should change. On the other hand, an on-policy algorithm uses the same policy for both purposes.

[1] The notation $P(x|y, z)$ means the probability of x given y and z

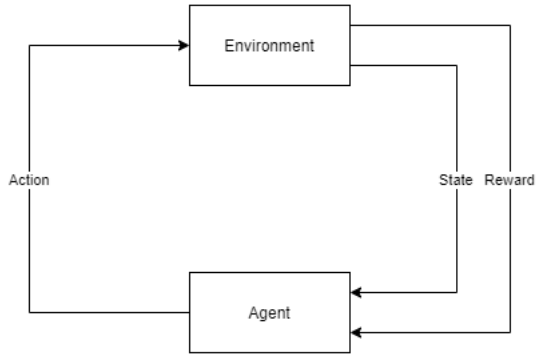


Figure 2.1: Illustration of the RL process

In RL there is an agent that performs actions on an environment and then receives feedback by a scalar reward signal that is a measure of its performance. This form of feedback is called *reward* or *reinforcement* [5, p. 830]. The basic RL operation is a process where the agent receives a state from the environment, performs an action based on this state, and then receives a reward. This process is then repeated continuously for the entire operation. The RL process is illustrated in Figure 2.1.

Markov Decision Process

In RL the problem is modeled as a Markov Decision Process (MDP). In an MDP a Markovian Transition Model and additive rewards are assumed. This means that an MDP consist of a set of states, $\{s^1, \ldots, s^n\} \in \mathcal{S}$; a set of actions, $\{a^1, \ldots a^n\} \in \mathcal{A}$; a transition model, $T(s, a, s')$; and a reward function $R(s, a, s')$ [5, pp.646-647].

To decide what action is to be performed in a given state, a *policy* is needed. The policy is denoted by π, and it can either be deterministic or stochastic. In the case of a deterministic policy, the policy is a direct mapping from states to actions $\pi(s) = a$; in the case of a stochastic policy, the policy is a probability distribution over all actions in a given state, $\pi(s, a) = P(a|s)$.

The agent's purpose in an MDP is to maximize the reward. To do this, it aims to discover the optimal policy π^*, the policy that optimizes the expected reward. There are mainly three ways of defining what it means to optimize the reward [15, pp.13-15]:

- The finite horizon model
  - $E[\sum_{t=0}^{h} r_t]$, where the notation $E[\ldots]$ means the expected value.
  - This model indicates that the expected reward should be optimized over the next h transitions. A problem with this model is that the optimal value for h is difficult to discover.
- The discounted infinite horizon model
  - $E[\sum_{t=0}^{\infty} \gamma^t r_t]$

- This model indicates that the expected reward should be optimized over the entire future. Where $\gamma \in \mathbb{R} : \gamma \in [0, 1]$ is called the discount factor. The discount factor describes how much future rewards should be weighted compared to more immediate rewards. If $\gamma \approx 0$, this means that only immediate rewards are considered; if $\gamma = 1$, rewards in the distant future are weighed just as much as immediate rewards. It is common to have γ close to, but not equal to, 1.

- The average reward

  - $\lim_{h\to\infty} E[\frac{1}{h}\sum_{t=0}^{h} r_t]$

  - This model indicates that the average reward should be maximized over time.

To optimize the reward based on any of these models, and uncover the optimal policy π^*, value functions and the Bellman Equation have to be introduced.

Value functions and Bellman Equations

This section takes inspiration from [14, ch. 1].
Value functions are a way to link the optimal criteria to policies. A value function represents how valuable it is to be in a given state according to a policy, and is denoted by $V^\pi(s)$. In other words, the value function evaluated in a specific state is the expected return when starting in state s and following the policy π after that. The value function using the discounted infinite horizon model described above is:

$$V^\pi(s) = E_\pi\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k}|s_t = s\right\}, \tag{2.1}$$

where E_π is the expected value when using policy π.
It can also be useful to use a variant of the value function, a function that describes both how valuable a state s is, and the value of taking action a in that state. This type of function is called a *Q-function*:

$$Q^\pi(s, a) = E_\pi\left\{\sum_{k=0}^{\infty}\gamma^k r_{t+k}|s_t = s, a_t = a\right\}.$$

Value functions have a recursive relationship that makes deriving them possible. This recursive relationship comes from the value function's dependency on the reward in the

next state as seen in Equation (2.1).

$$\begin{aligned}
V^{\pi}(s) &= E_{\pi}\left\{\sum_{k=0}^{\infty}\gamma^{k}r_{t+k}|s_t = s\right\} \\
&= E_{\pi}\left\{r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{\infty} r_{t+\infty}|s_t = s\right\} \\
&= E_{\pi}\left\{r_t + \gamma\sum_{k=0}^{\infty}\gamma^{k}r_{t+k+1}|s_t = s\right\} \\
&= E_{\pi}\left\{r_t + \gamma V^{\pi}(s_{t+1})|s_t = s\right\} \\
&= \sum_{s'} T(s,\pi(s),s')(R(s,\pi(s),s') + \gamma V^{\pi}(s')), \text{ for } s = s_t,\ s' = s_{t+1}
\end{aligned}$$

This last equation is what is called the *Bellman Equation*.
Similarly, the Q-function can also be expressed recursively:

$$Q(s,a) = r(s,a) + \gamma \max_{a'} Q(s',a').$$

For the optimal policy mentioned in the previous section, the value function $V^{\pi *}$ has the following property:

$$V^{\pi *}(s) \geq V^{\pi}(s), \text{ for all } s \in \mathcal{S} \text{ and all policies } \pi.$$

This means that the optimal policy is the policy that maximizes the value function for all states. The optimal action in a given state s is then:

$$\pi^*(s) = \arg\max_{a} \sum_{s' \in \mathcal{S}} T(s,a,s')(R(s,a,s') + \gamma V^{\pi *}(s')).$$

Expressed with an optimal Q-function, this becomes

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

which includes neither the reward function nor the transition model. Therefore, when a model-free[2] approach is used, Q-functions are used instead of value functions. It happens frequently that the transition model is not known in RL, which means that Q-functions are useful and necessary in many situations.

2.1.2 Artificial Neural Networks

This section takes inspiration from chapters 1 and 2 in [16] but is written in the author's own words.

[2] A model-free approach is one where the reward function and transition model are unknown

A brief note: A hyperparameter is a parameter whose value is set before learning starts. This is different from the parameters of the network (weights and biases), which are continuously updated throughout the learning process.

Artificial Neural Networks (ANNs) are, as the name implies, networks consisting of artificial neurons. An artificial neuron has five main components: inputs, outputs, weights, biases, and an activation function. The relationship between these are

$$y = f(w^T x + b),$$

where y is the output vector, $f(\dots)$ is the activation function, w is the weight matrix, x is the input vector, and b is the bias vector. For an example consider Figure 2.2 where the activation function is

$$f(w^T x + b) = \begin{cases} 1, & \text{if } w^T x + b > 0 \\ 0, & \text{if } w^T x + b \leq 0 \end{cases}$$

An artificial neuron with this activation function is called a *perceptron* and is one of the most basic types. For Figure 2.2 the variables x, w, b, y are

$$x = \begin{bmatrix} 4 \\ 2 \\ -1 \end{bmatrix}$$

$$w = \begin{bmatrix} 2 \\ -1 \\ 3 \end{bmatrix}$$

$$b = 2$$

$$y = f(w^T x + b) = f(\begin{bmatrix} 2 & -1 & 3 \end{bmatrix} \begin{bmatrix} 4 \\ 2 \\ -1 \end{bmatrix} + 2) = f(5) = 1$$

The perceptron's activation function is not commonly used today, other activation functions are used instead. One of the properties needed from an activation function is that a small change in the bias and weights will result in a small change in the output from the neuron. This is not true for perceptrons, as a small change in either of these parameters can make the perceptron output a 0 instead of a 1, which is as radically different a change as can be had in this case. Even if perceptrons are not commonly used today, it can be shown that a network of perceptrons can approximate any function. This is because a perceptron can act as a NAND gate, which is universal for computation [16, ch. 1].

For artificial neurons to be able to approximate complex functions, they have to be put together into a network, an ANN. An ANN is structured into layers. Every ANN has at least an input layer and an output layer, and may also have any number of hidden layers. For an example of an ANN with a single hidden layer consider Figure 2.3. In this example, there are 3 neurons in the input layer, 4 neurons in the hidden layer, and 2 neurons in the output layer. This type of network is called a feedforward network, which means that all

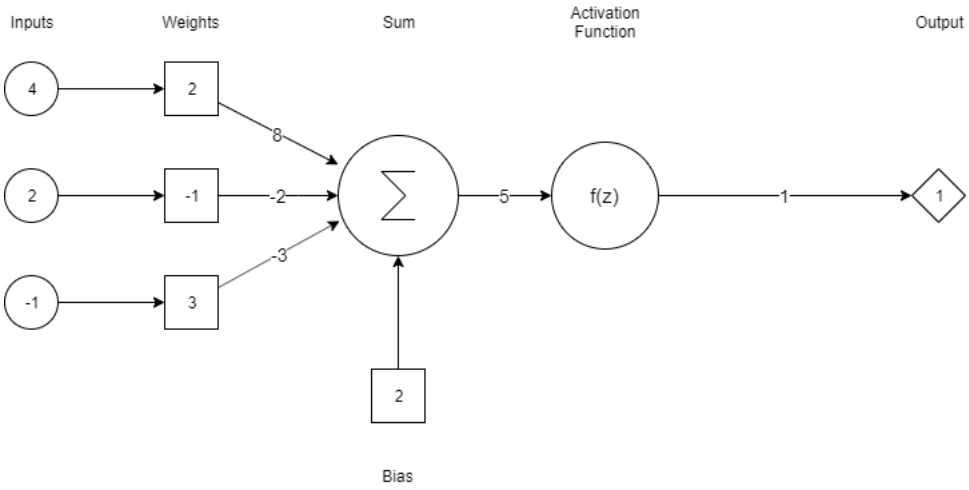


Figure 2.2: Example of a perceptron

neurons are only connected to the neurons in the preceding layer, without a cycle occurring. The number of neurons in each layer and the number of layers are hyperparameters.

The ability to act as a function approximator is the main advantage that an ANN can provide. What makes ANNs special compared to other function approximators, is that they are very good for data-driven approaches. The function that an ANN approximates is often quite complex, and often near impossible for humans to create on an analytical form. For the neural network to approximate this type of complex function, a set of algorithms that gradually adjusts the weights and biases of the network is needed. These are the learning algorithms. To examine the learning algorithms, another artificial neuron, the *sigmoid neuron* has been introduced.

The difference between the perceptron and the sigmoid neuron is the activation function which for the sigmoid neuron is

$$f(z) = \sigma(z) = \frac{1}{1 + e^{-z}}.$$

The relation between output and input for a single sigmoid neuron is then

$$y = \sigma(w^T x + b) = \frac{1}{1 + e^{-(w^T x + b)}}.$$

As mentioned above, a necessary property of an activation function is that a small change in the parameters (weights and biases) of the network, results in a small change in the output of the neuron. Figure 2.4 shows that this is true for the sigmoid neuron, and not true for the perceptron.

To know how to change the parameters of the ANN, a function that describes how good the current output of the ANN is is required. This is called a *cost function*. The lower the

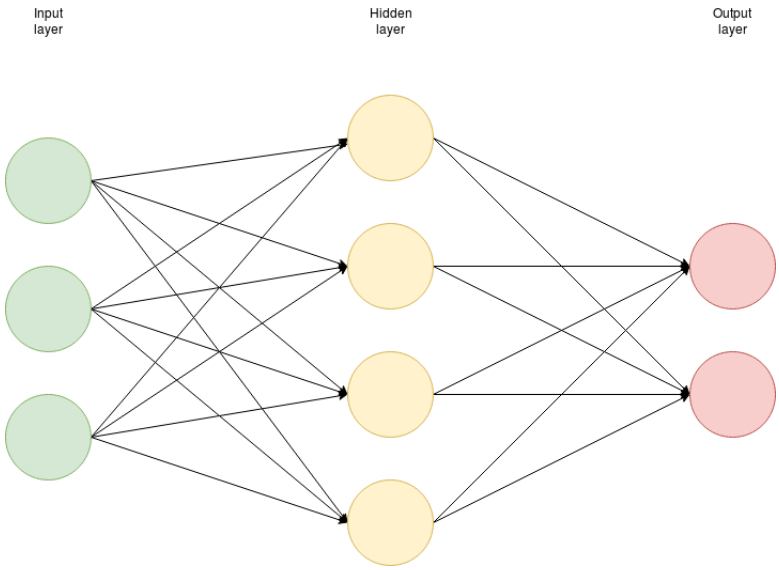


Figure 2.3: Example of a neural network with a single hidden layer

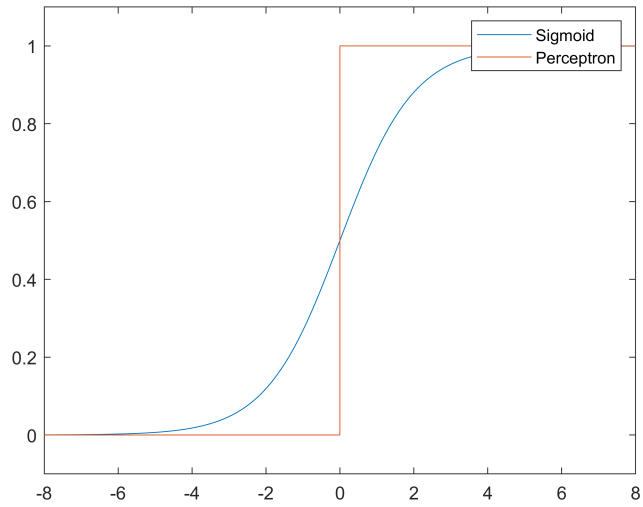


Figure 2.4: Plot of Sigmoid and Perceptron activation functions

cost function is for a given example, the better. For an example of a cost function, consider

$$C(w,b) = \frac{1}{2n}\sum_x ||y(x) - f(w^T x + b)||^2. \tag{2.2}$$

Here $y(x)$ is the target output given input x. w and b are all the weights and biases in the ANN and n is the number of training examples. In supervised learning, the target is the ground truth label which the network tries to predict given input x. The cost function in Equation (2.2) is called the Mean Squared Error (MSE) and is one of the most fundamental cost functions.

The goal of updating the parameters of the ANN to optimize performance can now be stated more explicitly: the parameters should be changed such that the cost function is minimized. The most commonly used method to do this is through the use of gradient descent. In gradient descent, the parameters of the ANN are changed according to

$$\theta \leftarrow \theta - \alpha \nabla C(\theta),$$

where θ is a vector that contains the parameters, that is the weights and biases, α is a hyperparameter called the learning rate, and ∇C is the gradient of the cost function with regards to the parameters. Calculating the gradient of the cost function in a single operation can be challenging. But since the network is divided into layers, it is possible to calculate the gradient of each layer and use *backpropagation* to propagate the gradient backward through the layers. Before explaining backpropagation, a brief overview of the notation is given:

- w^l_{jk}: the weight for the connection from neuron k in layer (l-1) to neuron j in layer l
- b^l_j: the bias of neuron j in layer l
- $z^l_j = (\sum_k w^l_{jk} a^{l-1}_k) + b^l_j$: the preactivation of neuron j in layer l (before being passed through the activation function
- $a^l_j = f(z^l_j)$: the activation of neuron j in layer l (activation is the result after being passed through the activation function)
- δ^l_j: The error in neuron j in layer j

The objective of backpropagation is to calculate $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ for all weights and biases in the neural network. Backpropagation is essentially applying the chain rule from calculus to every layer. For a function

$$y = f(u), u = g(x)$$

the chain rule says that the derivative of y with respect to x is

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}.$$

This means that for a single neuron j in the output layer L, the error is

$$\delta^L_j = \frac{\partial C}{\partial z^L_j} = \frac{\partial C}{\partial a^L_j}\frac{\partial a^L_j}{\partial z^L_j} = \frac{\partial C}{\partial a^L_j} f'(z^L_j)$$

Then the error in the entire output layer is

$$\nabla_{z^L} C = \delta^L = \nabla_{a^L} C \odot f'(z^L), \tag{2.3}$$

where z^L is a vector of the preactivations of the neurons in the output layer, δ^L is a vector of the errors of the neurons in the output layer and a^L is a vector of the activations of the neurons in the output layer. The symbol $\odot$ means element-wise multiplication, for example

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \odot \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} x_1 y_1 \\ x_2 y_2 \\ x_3 y_3 \end{bmatrix}.$$

To calculate the error in any layer other than the output layer, another equation is needed. For the layer right before the output layer, the error can be derived as follows for a single neuron j in layer $L-1$

$$\begin{aligned} \delta_j^{L-1} &= \frac{\partial C}{\partial z_j^{L-1}} \\ &= \sum_k \frac{\partial C}{\partial z_k^L} \frac{\partial z_k^L}{\partial z_j^{L-1}} \\ &= \sum_k \frac{\partial z_k^L}{\partial z_j^{L-1}} \delta_k^L \\ &= \sum_k w_{kj}^L \delta_k^L f'(z_j^{L-1}) \end{aligned}$$

which in matrix form is

$$\delta^{L-1} = ((w^L)^T \delta^L) \odot f'(z^{L-1})$$

It turns out that this equation applies to any layer except the output layer, so for all layers l the error is

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot f'(z^l) \tag{2.4}$$

The motivation for finding the errors is to make it easier to change the biases and weights of the network, in such a way that the cost function decreases. This means that $\frac{\partial C}{\partial b_j^l}$ and $\frac{\partial C}{\partial w_{jk}^l}$ needs to be found for all layers l and neurons j and k. Using the error in each layer, these partial derivatives are short and concise:

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l \tag{2.5}$$

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \tag{2.6}$$

since

$$\frac{\partial z_j^l}{\partial b_j^l} = \frac{\partial}{\partial b_j^l} (\sum_k w_{jk}^l a_k^{l-1} + b_j^l) = 1$$

$$\frac{\partial z_j^l}{\partial w_{jk}^l} = \frac{\partial}{\partial w_{jk}^l}(\sum_k w_{jk}^l a_k^{l-1} + b_j^l) = a_k^{l-1}$$

The backpropagation algorithm can now be introduced, using Equation (2.3), Equation (2.4), Equation (2.5), and Equation (2.6) that was just derived [16, ch. 2].
The backpropagation algorithm:

1. **Input:** Set activation a^1 for input layer
2. **Feedforward:** For each 2, 3,..., L compute $z^l = (w^l)^T a^{l-1} + b^l$ and $a^l = f(z^l)$
3. **Error in output layer, δ^L:** Compute vector $\delta^L = \nabla_{a^L} C \odot f'(z^L)$
4. **Backpropagate the error:** For each l = L-1, L-2, ..., 2 computer $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot f'(z^l)$
5. **Output:** The gradient of the cost function is given by $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$ and $\frac{\partial C}{\partial b_j^l} = \delta_j^l$

The backpropagation algorithm can be used to find the gradient of the cost function for a single example. It is common to compute the gradient of multiple training examples, which is then used to update the parameters of all the neurons. This collection of training examples is called a *minibatch*. A learning algorithm that is commonly used together with backpropagation is Stochastic Gradient Descent (SGD), and is as follows:

1. **Input a set of training examples, a minibatch of size** m
2. **For each training example** x: Set the corresponding input activation $a^{x,1}$, and perform the following steps:
   - **Feedforward:** For each $l = 2, 3, \ldots, L$ compute $z^{x,l} = w^l a^{x,l-1} + b^l$ and $a^{x,l} = f(z^{x,l})$.
   - **Output error** $\delta^{x,L}$: Compute the vector $\delta^{x,K} = \nabla_a C_x \odot f'(z`x, l)$
   - **Backpropagate the error:** For each $l = L-1, K-2, \ldots, 2$ compute $\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1} \odot f'(z^{x,l})$
3. **Gradient descent:** For each $l = L, L-1, \ldots, 2$ update the weights according to the rule $w^l \leftarrow w^l - \frac{\eta}{m}\sum_x \sigma^{x,l}(a^{x,l-1})^T$, and the biases according to the rule $b^l \leftarrow b^l - \frac{\eta}{m}\sum_x \delta^{x,l}$

To use SGD in practice, an outer loop that generates minibatches is also needed. A loop outside that again, which goes through multiple *epochs* of training is also commonly used. An epoch is in supervised learning defined as one complete run-through of all the training examples. After an epoch, a set of examples that were not included in the training set, called the *test set*, is run through the neural network. This test set is used as a way to evaluate the performance of the neural network between epochs. It is common to randomize between epochs which examples are in the training set and which are in the test set. After all the epochs are done, the best practice is then to have another unseen set of examples called the *evaluation set* passed through the neural network. The evaluation set is never used for training and is a way to see how well the neural network performs on new data after completing the training.

Regularization

A major problem with the training of neural networks is what is called *overfitting*. This means that the neural network no longer generalizes to the test data. The problem comes from training too much on the training data. This leads to the network becoming better at recognizing the training data but becoming worse at recognizing data that it has not seen during training. This is generally an undesirable situation. The point of a neural network that is created, for instance, for image recognition is not to recognize images that it has already seen, but to recognize images that it never has seen before. There are several ways to reduce the amount of overfitting, for instance, by increasing the available training data. Another way to reduce overfitting is to employ so-called *regularization techniques*. One of the more common regularization techniques, which is used in the experiments in this thesis, is known as *L2 regularization*. L2 regularization is also called weight decay, and the idea is to introduce another term to the cost function. This term is called the regularization term. For the MSE cost function described above, the cost function with L2 regularization added is

$$C(w,b) = \frac{1}{2n}\sum_x ||y(x) - f(w^T x + b)||^2 + \frac{\lambda}{2n}\sum_w w^2.$$

The term added here is the sum of the squares of all the weights in the network. This sum is scaled by $\frac{\lambda}{2n}$, where $\lambda > 0$ is a hyperparameter known as the *regularization parameter*. n is also here the number of training examples. The regularization term does not include the biases. The effect of the regularization is that the network prefers to learn small weights. To see why smaller weights lead to better generalization see [16, ch. 3].

Other activation functions

The sigmoid neuron is not used much today. It can be shown that as the network gets deeper, using the backpropagation algorithm with sigmoid neurons makes the gradient decay to zero fast [17]. This is called the vanishing gradient problem, and this leads to the learning slowing down, and in the worst case stopping completely up. Some more used activation functions today are the *Hyperbolic Tangent (Tanh)* and the *Rectified Linear Unit (ReLU)*. These are shown in Figure 2.5. ReLU is defined as

$$f(z) = \max(0, z),$$

and Tanh is defined as

$$\tanh(z) = \frac{\sinh(z)}{\cosh(z)} = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Tanh still suffers from the vanishing gradient problem, although the gradient is stronger for Tanh than for sigmoid (since the derivatives are steeper). Tanh is also zero-centered, which is a valued property for an activation function [17]. The activation function primarily used in this thesis is ReLU, as this function has been shown empirically to give good results in general [16, ch. 6].

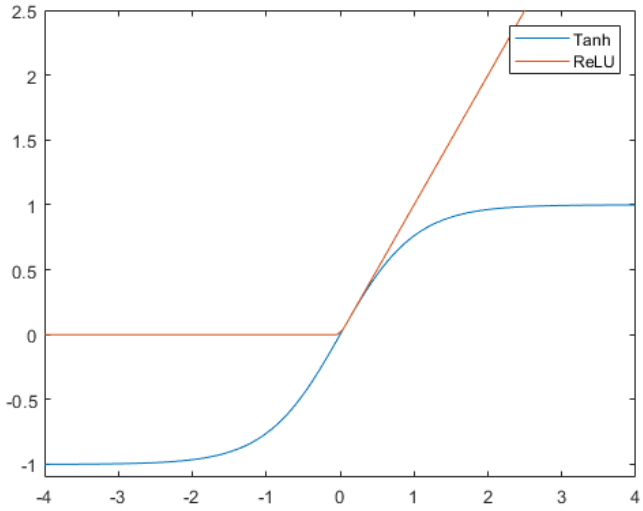


Figure 2.5: Plot of Tanh and ReLU activation functions

Convolutional neural networks

Convolutional Neural Networks (CNNs) are especially good at recognizing and taking advantage of spatial structures, which are highly present in for instance visual data. Three basic concepts are especially important for CNNs: local receptive fields, shared weights, and pooling [16, ch. 6].

Local receptive fields: How a local receptive field works is illustrated in Figure 2.6. The input to the layer is a 7×7 matrix of $7 \times 7 = 49$ neurons, which can, for instance, correspond to an image that has 49 pixels. The local receptive field is marked in green and can be seen as a 3×3 window that slides gradually over the pixels in the input matrix. The output from this is a 5×5 matrix, which consists of $5 \times 5 = 25$ neurons. In this example, a stride length of 1 is used, which means that the local receptive field moves one pixel at the time. Different stride lengths can be used, as well as different window sizes. This means that the output from the layer will have a different size than the input. A stride length of two is illustrated in Figure 2.7. Each entry in the local receptive field learns a weight, and the whole receptive field has a common bias as well. The size and number of local receptive fields and the stride in the CNN are hyperparameters that the user needs to define beforehand.

Shared weights and biases: What is meant by shared weights and biases is that the local receptive field uses the same weights and bias as it moves over the entire matrix. This means that all neurons in the output of the layer detect the same feature, only in different areas of the input matrix. This is useful because the same feature can occur in different

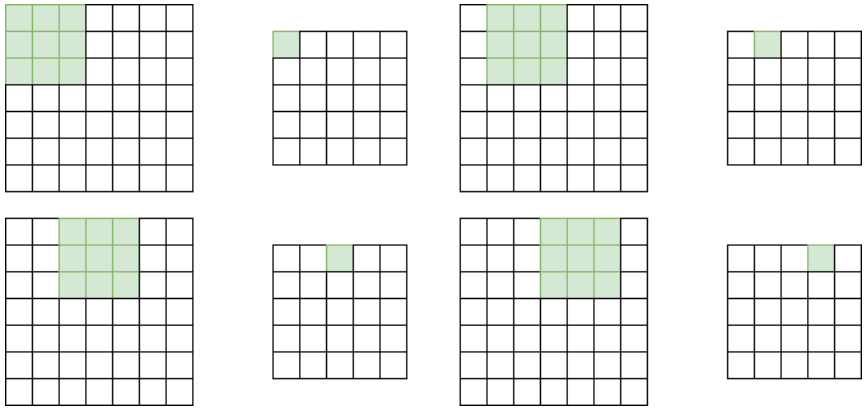


Figure 2.6: Illustration of the workings of local receptive fields

places, for instance, in an image. For the *j*, *k*th output neuron, its value is

$$f(b + \sum_{l=0}^{g-1} \sum_{m=0}^{h-1} w_{l,m} a_{j+l,k+m}).$$

Here $f(\dots)$ is the activation function, the receptive field is of $g \times h$ size, and $a_{x,y}$ is the input activation at position *x,y*. Since each local receptive field detects one feature, it is common to call them *feature maps*. The shared weights and bias is often called a *filter*. Using only one feature map means that the layer only detects one type of feature. It is therefore common to use many feature maps on the same input. Consider Figure 2.8 where the input is a 7×7 RGB-image. This means that there are 3 channels in the input, each corresponding to either the value for red, green or blue in that pixel in the image. Each filter, in this case, has $3 \times 3 \times 3$ (height, width, and the number of channels) weights, and one bias. There are 9 filters in total, which correspond to the 9 channels in the output. This means that the total number of parameters in this layer is $3 \times 3 \times 3 \times 9 + 9 = 252$.

It is common to put convolutional layers in the first layers of the network, and then have fully connected layers near the output. This is because the convolutional layers can then act as feature extractors, and then the fully connected layers can combine these features into an output.

Pooling: Pooling refers to another type of layer called *pooling layers*. These layers are used to summarize the information in an input matrix in various ways. This means that they take in an input matrix, and then output a matrix that has a smaller size than the input matrix. Pooling layers are often used right after convolutional layers. This is because the output from a convolutional layer is the degree to which the feature that a filter is trying to detect is in that area of the input matrix. The pooling layer then summarizes that information, but still gives approximately the area that the feature occurred. It is most of the time not important to know exactly where the feature occurred, and an estimate is often enough. More specifically, each unit in a pooling layer may summarize a region of $n \times n$ neurons in the previous layer. For an example of a pooling layer, consider Figure 2.9, which shows a pooling layer called max-pool. In this figure, the pooling layer summarizes the max of

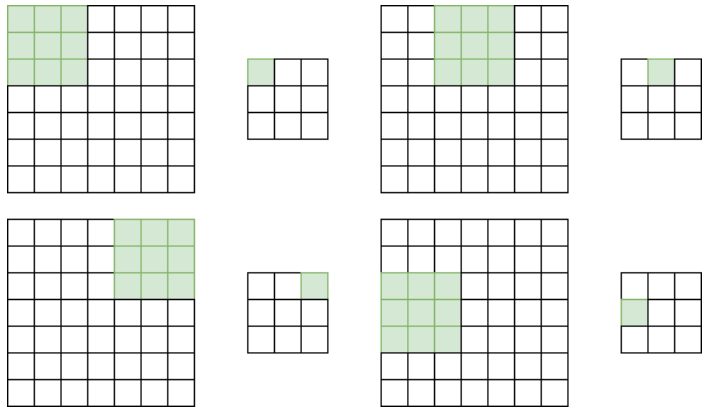


Figure 2.7: Illustration of the workings of local receptive fields with stride=2

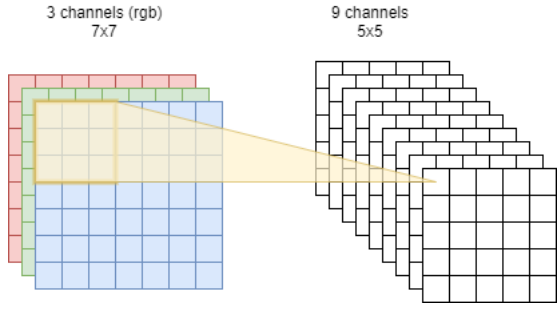


Figure 2.8: Example of a convolutional layer

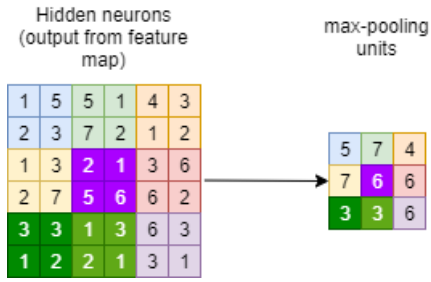


Figure 2.9: Example of a max-pool layer

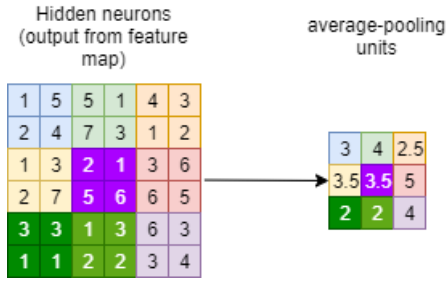


Figure 2.10: Example of an average-pool layer

each 2×2 region of the input matrix. The colored neurons in the max-pooling units summarize the correspondingly colored regions of the hidden neurons. For another example, consider Figure 2.10, where another type of pooling, average pooling, has been used. This layer, similarly to max-pooling, summarizes a region of an input matrix, but instead of taking the max of that area, it summarizes by taking the average. Pooling layers are useful since they reduce the dimensionality, which leads to fewer parameters, and faster training.

CNNs are useful not only for their ability to detect spatial structure but also since they require much fewer parameters than a fully-connected layer. As the number of parameters increases, the training speed decreases.

There are now five different types of neural network layers that have been introduced in this theory chapter:

- Input layer
  - In RL this consists of the current state of the environment. In supervised learning this may, for instance, be an image that needs to be categorized.
- Output layer
  - Can, for instance, correspond to the categories that the input image should be categorized under. Or in the case of DRL in the next subsection, what action the agent should take.

- Fully-connected layer
  - A layer where all neurons are connected to all the neurons of the previous layer. As described in Section 2.1.2.
- Convolutional layer
  - As described above, a convolutional layer slides filters over the previous layer. Then it outputs to what degree the feature that the filter is looking for is in the particular regions of the previous layer.
- Pooling layer
  - As described above, a pooling layer summarizes the information in the previous layer and reduces the dimension size.

2.1.3 Deep reinforcement learning

The *curse* of dimensionality briefly mentioned in Section 1.1 has been a long-standing challenge in RL. Richard E. Bellman who discovered the Bellman equation used in Section 2.1.1 coined this term himself. When he worked with optimal control in discrete high-dimensional spaces, he found an exponential increase in the states and actions. This means that with high dimensionality, it becomes infeasible to evaluate every state (Bellman 1957, cited in [10]). The first RL method to successfully break this curse, by learning control policies directly from a high-dimensionality sensory input was done by DeepMind in [7]. They called this method Deep Q-Network (DQN), and it is based on the Q-learning algorithm. The fundamental idea is that a nonlinear function approximator can map both state and action onto a value. As discussed in Section 2.1.2, neural networks are good for this [6, lecture 3]. DeepMind used a CNN to extract the features from the screen pixels, and then fully connected layers to map these features onto actions. In addition to this neural network, they used two other inventions to make the network be able to learn, namely a *replay buffer* and a separate *target network* to stabilize learning.

Replay buffer: One of the problems that DRL faces is that most optimization algorithms assume that all samples are independent of each other. If the samples are just generated by exploring, they are not independent, since the next state is dependent on the current state, and so on. This is what the replay buffer is there for. Every transition, $\{s_t, a_t, r_t, s_{t+1}\}$, is stored in the replay buffer, and when the neural network(s) are to be updated, a minibatch is randomly sampled from the replay buffer. This means that the samples are independent, and this improves the generalization of the function approximator. Additionally, the replay buffer makes it so that previous transitions are not forgotten, and can be used multiple times.

Target network: The next Q-value in the traditional form of Q-learning is calculated by [14, p. 107]

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a)].$$

DQN similarly uses a cost function based on the Bellman equation:

$$L(\theta_i) = E[(y_i - Q(s, a; \theta_i))^2],$$

where the target y_i is

$$y_i = \begin{cases} r_i, & \text{If } s_i \text{ is a terminal state} \\ r_i + \gamma \max_{a'} \hat{Q}(s', a'; \hat{\theta}_i), & \text{If } s_i \text{ is not a terminal state} \end{cases}$$

Where $\hat{Q}$ represents the Q-value from the target network. The target network is simply a copy of the original network. During operation, the target network either gets slowly changed such that it becomes more similar to the original network or gets its weights copied from the original network occasionally. The reason that a target network is used, and not just the original network, is to avoid having a target that changes all the time. A target function that changes all the time makes training quite difficult since this becomes similar to trying to catch up to a moving target [7].

Policy gradient and actor-critic methods

A policy gradient method is a method that can learn a parameterized policy that maps states directly to actions. This means that a value function is not needed to choose which action to take. A value function may still be used to choose how to update the parameters of the policy, but value functions are not a part of choosing what action to take. The parameter vector of a policy is denoted by $\theta \in \mathbb{R}^{d'}$, and the policy described in Section 2.1.1 is then changed to

$$\pi(s, a, \theta) = \pi(a|s, \theta) = P(A_t = a|S_t = s, \theta_t = \theta)$$

which is the probability that action a is taken in timestep t, given that the environment is in state s, and the parameters of the policy are θ. In the case of an actor-critic method, a value function is learned in addition to the policy, and its weight vector is denoted by $w \in \mathbb{R}^d$.
Policy gradient methods update the parameters with regards to the gradient of a performance measure J. This means that

$$\theta_{t+1} = \theta_t + \alpha \widehat{\nabla J(\theta_t)},$$

where α is the learning rate and $\widehat{\nabla J(\theta_t)}$ is a stochastic estimate which is expected to be close to $\nabla_{\theta_t} J(\theta_t)$ which is the gradient of J with regards to the weight vector θ at timestep t [14, p. 265].

Deep Deterministic Policy Gradient (DDPG)

The inspiration for Deep Deterministic Policy Gradient (DDPG) comes from the success of DQN mentioned above. DQN solves the *curse* of dimensionality from a large observation space point of view, but it does not provide a good solution for continuous action spaces. If a problem with a continuous action space is going to be solved with DQN, a discretization of the action space is needed. This solution is not satisfactory, because the dimensionality

of the action space would then increase exponentially with the Degrees of Freedom (DOF) in the system. If the number of points in the discretization is d, then the dimension of the action space would be d^{DOF}. For even a discretization of just three points, for instance, $a_i \in \{-k, 0, k\}$, this means that for a 7-DOF system, the dimension of the action space is $3^7 = 2187$. For a less rough, more realistic discretization, the dimension size would make solving the problem infeasible, because of the difficulty in exploring this action space [9].

DDPG is a model-free and off-policy, actor-critic algorithm that aims to merge the Deterministic Policy Gradient[18] with DQN. Since DDPG is an actor-critic algorithm, this means that it uses two different neural networks. The actor-network decides what action to take during operation; the critic-network evaluates the transitions, and this evaluation is used to update the actor-network. The main elements from DQN that are included in DDPG are the replay buffer and a separate target network as described above. The DDPG algorithm is shown in Algorithm 1. The noise process $\mathcal{N}$ can be chosen to suit the environment and is added to make the agent explore. Because DDPG has a deterministic policy, it would never explore on its own, therefore the noise process is needed.
Since the algorithm chooses what action to take according to the policy

$$a_t = \mu_t(s_t|\theta^\mu) + \mathcal{N},$$

and updates the actor-network using the policy

$$\mu_t(s_t|\theta^\mu),$$

this means that DDPG is an off-policy algorithm.
DDPG performs soft target updates instead of completely copying the weights:

$$\begin{aligned} \theta^{Q'} &\leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'} \\ \theta^{\mu'} &\leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'} \\ 0 &< \tau \ll 1 \end{aligned}$$

Where τ is a hyperparameter defined by the user [9].

2.1.4 Reward shaping/functions

To engineer a reward function for robotic reinforcement learning (or any kind of reinforcement learning) one has to walk a fine line. If the reward function gives rewards too sparse, the agent may never even get a single reward; if the reward appears too frequently, the agent may never discover any innovative solutions to the problem. Consider if one is to create an RL agent that can learn how to use a robotic manipulator to pull a lever. If the reward is given only when the lever is pulled, the agent may most likely never learn to pull the lever. The chance that it would pull the lever through random exploring is very low, given the large state and action spaces. There is therefore never an indication of what actions are better than others, so it cannot possibly learn how to pull the lever. On the other hand, the engineer can find a detailed sequence of states that makes the manipulator pull the lever. Then the reward function can give a reward whenever the environment is in any

Algorithm 1 DDPG algorithm, taken from [9]

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q, θ^μ
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial observation state s_1
    for t=1, T **do**
        Select action $a_t = \mu_t(s_t|\theta^\mu) + \mathcal{N}$ according to the current policy and exploration noise
        Execute action a_t and observe reward r_t and observe new state s_{t+1}
        Store transition $(s_t, a_t, r_t, s_{t+1}$ in R
        Sample a random minibatch of N transitions $(s_i, a_i, r_i, s_{i+1}$ from R
        Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
        Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
        Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

        Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

    end for
end for

of these states. This will likely just make the agent follow the path given by this sequence, without the possibility of discovering a better way to pull the lever.

Hindsight Experience Replay

To make the design of reward functions for robotic manipulation tasks such as the ones in this thesis simpler, a technique called Hindsight Experience Replay (HER) can be used. The idea behind HER is that even if the agent has not achieved the goal that it was supposed to, it still has achieved a goal. In HER, even if the end result was not successful, the end result is treated as a *substituted goal*. Then the reward that the agent would have received if the substituted goal was the real goal is calculated, and the transitions are put into the experience replay buffer. This means that the agent gets some feedback to learn from even though it did not reach the real goal. The intuition behind doing this is that when the agent has learned how to reach enough of these substituted goals, it will learn how to reach an arbitrary goal [19, 13].

When using HER it is possible to use reward functions that give rewards very sparsely. As discussed above, too sparse reward functions do not usually work great with RL algorithms. Although, when HER is used, sparse reward functions have given better results than dense ones [13]. HER has also been shown to work great with binary rewards. This means that when using HER, the reward function can be so simple as to give 0 when the goal is reached, and -1 when the goal is not reached.

When using HER, two new hyperparameters need to be chosen. The first of these hyperparameters is the *replay strategy*. The replay strategy describes how to choose the substituted goals. Andrychowicz et al.[13] describes four different replay strategies for HER:

- final
  - Replay with k substituted goals, which come from the same episode as the transition being replayed, and correspond to the final state in that episode.
- future
  - Replay with k random states which come from the same episode as the transition being replayed and were observed after it.
- episode
  - The same as future, but the random states do not have to be observed after the transition that is being replayed.
- random
  - The same as future and episode, but the random states are simply selected from the entire period of learning.

The second new hyperparameter is the k mentioned above. This is the number of substituted goals that should be replayed for each ordinary transition that is replayed. One of the downsides with HER is that it can only be used with off-policy methods. This is because of the highly off-policy action of substituting goals [19]. DDPG, which as mentioned in Section 2.1.3 is an off-policy algorithm, and can therefore be used together with HER in this thesis.

2.1.5 Transfer learning from simulator to real-world

Choosing what environment to train the agent for robotic reinforcement learning is not a trivial task. The apparently obvious answer is to train the physical robotic manipulator in the environment that it is going to operate in. This approach has several advantages; the most relevant environment to train the agent in is certainly the one that it is going to end up in in the end. There are nevertheless many difficulties with training the agent on the actual robot, as this approach suffers from the *curse* of real-world samples mentioned in Section 1.1. Some of the main problems of training in the real-world are [10, pp. 1250-1251]:

- Degradation of the manipulator due to overuse can be very costly and may lead to needing to replace the manipulator altogether.
- At the beginning of the training, when the ANN may be completely randomized, there is no way to know what kind of motions the manipulator may perform. If there are people or expensive equipment in the area around the robot, the manipulator may damage these.
- Training in the real world cannot be sped up, so if one is going to train the agent in the working environment, the algorithm used needs to be quite sample efficient.
- There is also the possible need for human labor to reset the task, which makes the learning even slower and even more expensive.

Another approach to this problem is to use a similar environment, one that avoids the *curse* of real-world samples altogether, namely a simulator. In a simulator, there is no risk of damaging either the manipulator itself or the environment around it. A simulator can also in many cases run much faster than the real-world speed, which can decrease the time it takes to train a DRL agent. There is also no need for human labor to reset the environment. There are nevertheless many difficulties in training the agent in a simulator. This is mainly because of the *reality gap* between the simulator and the real-world. This reality gap comes from modeling errors, and an agent trained solely on a simulator may develop strategies that are specific to the simulator, and that do not transfer well to the real-world.

To make the transition from the simulator to real-world easier, [20] proposes what they call *Dynamics Randomization*. This means that between each episode of training, the dynamics of the manipulator and environment are randomized. The idea is that since the agent then can learn how to control an arbitrary version of the same manipulator, the discrepancies between the simulator and the real-world may be overcome. The parameters that are randomized in [20] are:

- Mass of each link in the robot's body
- Damping of each joint
- Mass, friction, and damping of the puck. The puck was part of the goal of the environment.
- Height of the table
- Gains for the position controller
- Time-step between actions
- Observation noise

Using dynamics randomization, Peng et al. got good results and managed to transfer successfully to the real-world without any training in the real environment.

2.2 Robotic manipulators

This section takes inspiration from the book "Robot Modeling and Control" [21].
In robotic RL the manipulator is viewed as a black box. This means that the information used when creating a traditional control method for a manipulator is not used in robotic RL. Nevertheless, the terminology can still be used to describe, for instance, what the agent needs to learn to control the manipulator. It can also be useful to use domain knowledge to, for example, shape a reward function for a robotics problem.

In the coarsest partition, there are two types of robotic manipulators: open-chain manipulators and closed-chain manipulators. In this thesis, the manipulator that is used is an open-chain manipulator (details of which can be found in Section 3.2.1), therefore the workings of open-chain manipulators are described in this section.
First, a brief overview of some concepts used in this thesis is given:

- End-effector
  - The end-effector is the tool at the end of the manipulator's arm which is what is ultimately used to perform the manipulator's task. The end-effector can, for instance, be a gripper or a welding tool.
- Joint/configuration space
  - The joint space is the set of all values the joints of a manipulator can have. For a manipulator with n joints, the joint space is n-dimensional.
- Task space
  - The task space is a description of the position and orientation of the end-effector given in Cartesian coordinates. Often this is a 6-dimensional space. The three first dimensions correspond to the position of the end-effector (X-Y-Z) and the three last dimensions correspond to the orientation of the end-effector given in rotations about the coordinate axes.

2.2.1 Forward kinematics

Forward kinematics concerns deriving the position and orientation of the end-effector, given the values of the joint variables. This means that the forward kinematics describes how to transfer coordinates given in the joint space to coordinates in the task space. For open-chain manipulators, this is generally one of the simplest problems. The forward kinematics of an open-chain manipulator is described using a homogeneous transformation matrix H:

$$H(q_1, \ldots, q_n) = \begin{bmatrix} R_n^0 & o_n^0 \\ 0 & 1 \end{bmatrix} = T_n^0 = A_1(q_1) \ldots A_n(q_n),$$

where $q_1 \ldots q_n$ are the joint variables. R_n^0 and o_n^0 describe the rotation and translation of the end-effector frame relative to the base frame of the manipulator.

2.2.2 Inverse kinematics

Inverse kinematics concerns deriving the values of the joint variables that will give a specified position and orientation of the end-effector. This means that the inverse kinematics describes how to transfer coordinates given in the task space to coordinates in the joint space. This problem is, as given by the name, the opposite of the forward kinematics problem. This is more difficult than the forward kinematics problem. There are possibly many joint configurations that correspond to a single end-effector position and orientation, and there are also singularities that may arise.

Chapter 3

Equipment and setup

3.1 Software

3.1.1 Robot Operating System

Robot Operating System (ROS) is an open-source, meta-operating system that aims to make robot development more robust, easier and more flexible. It is a collection of tools, libraries, and conventions that simplifies the development of robots across various robotic platforms. ROS can be used with both C++ and Python. ROS during operation is essentially a collection of nodes; a node is a process that can communicate with other nodes, sensors, and actuators. The nodes communicate using services and topics. A topic is a transport system over which nodes can send and receive messages. To send messages on a topic, a node has to publish to the topic; to receive messages it subscribes to a topic. Several different types of messages can be sent and received from a topic, and these are language independent. This means that one node that runs on C++ can send messages on a topic to a node that runs on Python.
Topics provide a way for nodes to communicate, but this is a many-to-many communication. This means that all nodes can send messages to every topic, and all nodes can listen to every topic. The communication is also just one way since there is no way for a node to respond to the message it receives by using just a single topic. To provide the request-reply interaction that is often needed in distributed systems, nodes can also use services. A service is defined by a pair of message structures, one for the request and one for the reply. A providing node can offer a service, and a client node can use this service by sending a request message and then wait for the reply [22, 23].

3.1.2 Gazebo

Gazebo is a 3D dynamic robotics simulator which can be controlled by ROS using a collection of packages called gazebo_ros_pkgs. Gazebo provides multiple physics engines, a big library of robot models and environments and many different sensors [24, 25].

3.1.3 PyTorch

PyTorch is a Python library for deep learning. This library is used for the implementation of the DRL used in this thesis. It provides features such as automatic backpropagation, enabling the GPU in deep learning and simple setup of all the most common types of neural network layers. PyTorch makes it very simple to build a neural network, both convolutional and feedforward [26].

3.1.4 OpenAI Gym

OpenAI has developed a toolkit called Gym, which provides several environments in which a user can develop and compare different RL algorithms. These environments cover among others, tasks such as robotic manipulators, classic control problems and video games. What is especially useful in OpenAI Gym is that all these environments share a common interface, which means that the same learning algorithm can be used in all the environments. The main functions used in the Gym environments are [27]:

- make(environment_name)
  - Sets up a new instance of the environment given by the argument.
  - Returns an object of the environment class
- step(action)
  - Applies a step to the environment based on the action used as the argument.
  - Returns the observation, the reward for the transition, whether the next state is a terminal state and a dictionary of diagnostic information that is specific to the environment.
- reset()
  - No arguments, but it resets the environment.
  - Returns the first observation of the reset environment.

3.1.5 openai_ros package

As mentioned in the previous subsection, the OpenAI Gym provides several useful tools for implementing RL algorithms. Unfortunately, it does not provide environments for training ROS based robots using Gazebo simulations. This means that if OpenAI Gym is going to be used with Gazebo, the environment has to be created by the user. To create the OpenAI Gym environment in Gazebo, an ROS package called openai_ros was used. This package provides a useful structure for setting up the OpenAI Gym environment. The structure is illustrated in Figure 3.1, and is as follows [3]:

- GazeboEnvironment class
  - This is the class that connects to the Gazebo simulator. This is the only class that does not need to be changed to suit the task at hand.

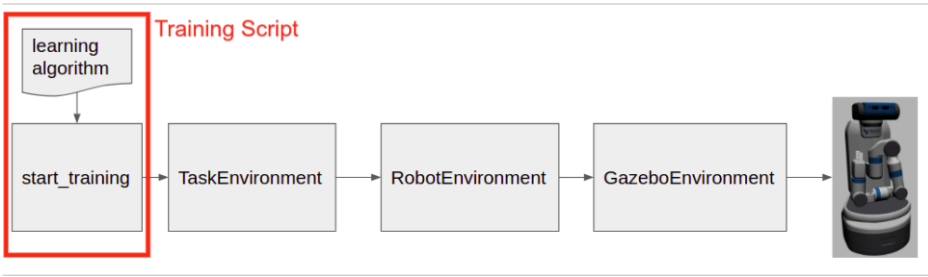


Figure 3.1: The structure used in openai_ros, taken from [3]

- RobotEnvironment class
  - This class specifies the robot that is going to be used in the OpenAI Gym environment. How to get the state of the manipulator, and how to operate it needs to be included in this class. This class had to be rewritten for the manipulator used in this thesis. Inspiration from a similar robot environment provided by the creators of openai_ros was used, although many changes had to be made. The robot environment that was used as inspiration was the fetch_env_v2 robot found in [4].
- TaskEnvironment class
  - In this class, the actual task that is going to be performed needs to be defined. All communication between the learning algorithm and the robot needs to be done here. Elements such as the reward function, how to get observations, whether the episode is finished and how to apply the selected action to the robot have to be included in this class. This class also had to be rewritten for the tasks in this thesis. Inspiration from a similar task environment provided by the creators of openai_ros was used, although many changes had to be made. The task environment that was used as inspiration was the fetch_reach task found in [4].

3.2 Hardware

3.2.1 OpenMANIPULATOR-X

The robotic manipulator that is used for the real-world testing in this thesis is the OpenMANIPULATOR-X by Robotis. This is a manipulator that has four revolute joints that correspond to four DOF. This manipulator lacks a spherical wrist, which means that it has two less DOF than what is typical in an industrial robotic manipulator [21]. This means that there are some configurations that it cannot reach. More specifically, these configurations correspond to rotations on the Z- and X-axis of the end-effector. The manipulator is mounted on a wooden board. The OpenMANIPULATOR-X is based on ROS and OpenSource [1].

Figure 3.2: The old valve with the manipulator

3.2.2 Lever

The valve that was previously used for similar projects to this is not suitable for the OpenMANIPULATOR-X. This valve is shown in Figure 3.2. As described in Section 3.2.1 this manipulator does not have a rotating wrist, so a valve is an unsuitable objective for this manipulator. A better task for this manipulator is to operate a lever. Based on this, we had a conversation with the mechanical workshop of the Department of Engineering Cybernetics and provided specifications for a lever that we wanted for the project. This lever has an SM-S4303R Continuous Rotation Servo attached to it so that it can automatically reset itself during the training phase of the DRL. This can make training in the real-world faster and more precise. This new lever is shown in Figure 3.3.

Figure 3.3: The new lever with the manipulator

Chapter 4

Implementation and results

4.1 FetchReach-v1 task in OpenAI Gym

To test the DDPG+HER implementation, an environment was selected from the OpenAI Gym which is called FetchReach-v1. The goal of this environment is to position the end-effector of a robotic manipulator within the vicinity of a goal point. A random goal point is selected at the start of each episode. This goal point is uniformly selected to be inside a cube with 30 cm sides, where the origin of the cube is at the initial position of the end-effector. This environment does not run in Gazebo, it instead runs in the simulator MuJoCo. The DDPG+HER implementation used in this thesis was created for OpenAI Gym, so the setup, other than the installation of MuJoCo, was minimal. To install and use MuJoCo a student license had to be obtained from [12]. For this task, the agent was trained for 30 epochs, with 30 episodes in each epoch. For the evaluation between epochs, 20 evaluation episodes were done. The results from the evaluation episodes are shown in Figure 4.2. The hyperparameters used for this task are shown in Table 4.1. How this task looks in MuJoCo can be seen in Figure 4.1.

4.2 Fetch-reach task in Gazebo

An OpenAI Gym environment was created for Gazebo using openai_ros as a framework. To test this environment, a task was created. This is a simple reacher task, which is inspired by the FetchReach-v1 task. In this new task, a random point in the work-space of the manipulator is selected. Rewards are only given when the origin of the end-effector is in the close vicinity of this point (inside a sphere with a 4 cm radius which has its origin at the point), and the objective of this task is then to place the end-effector at this point. An episode is labeled as a success if the end-effector of the manipulator is near the point when the episode is finished. This task essentially requires the agent to learn the inverse kinematics of the robot. To illustrate in the simulator where the goal point is, a green ball is spawned with its origin at the goal point. How the task setup looks in Gazebo is shown

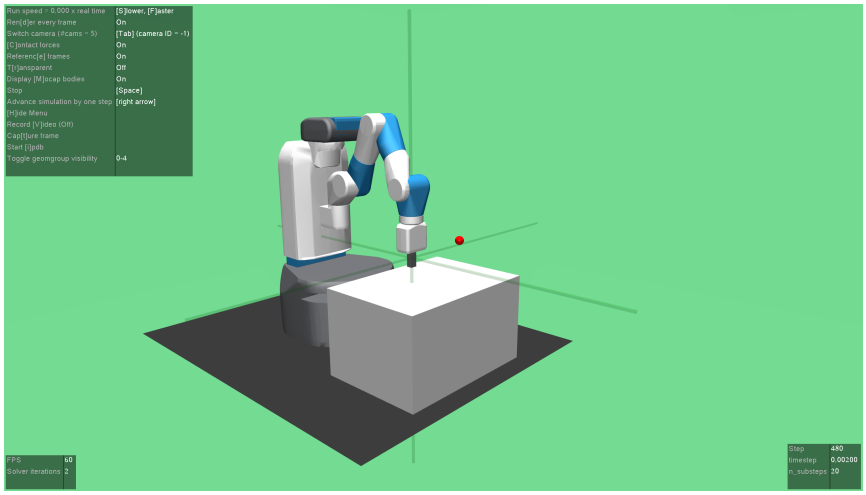


Figure 4.1: How the FetchReach-v1 task looks like in MuJoCo

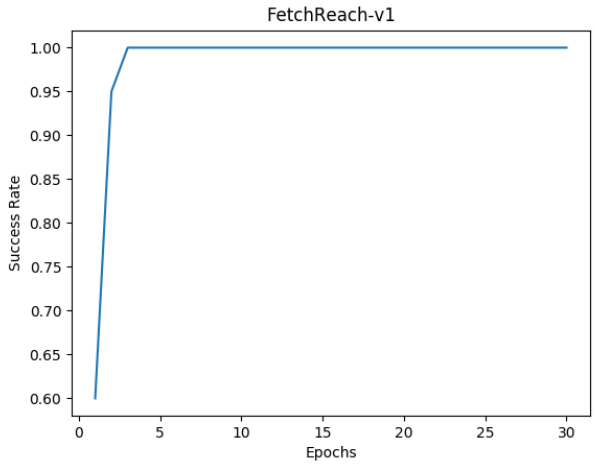


Figure 4.2: The success rate for the FetchReach-v1 task

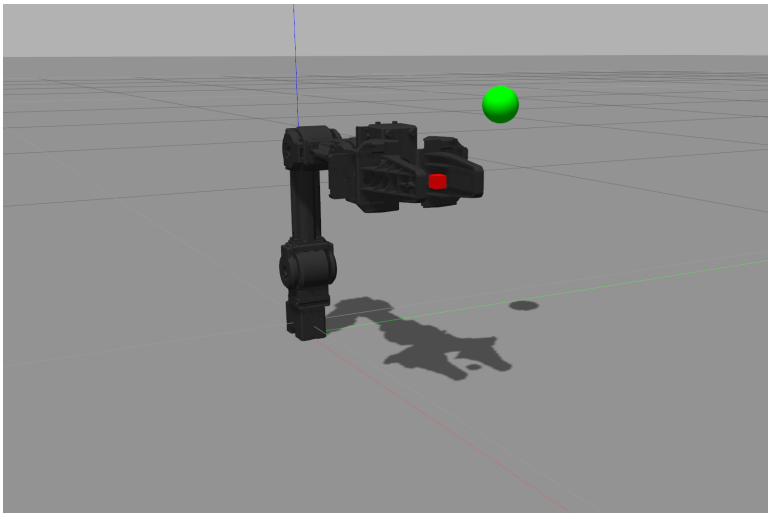


Figure 4.3: Gazebo task setup

in Figure 4.3.

The observation/states that the agent receives from the environment are the position and velocity of the joints, in addition to the goal point given in Cartesian coordinates. This means that the state vector is a 11×1 vector, as there are four joints and three entries in the Cartesian coordinates (X-Y-Z). The input to the environment, in other words, the action, is a 4×1 vector where each entry corresponds to how much the joints 1-4 should change. The action is then scaled by a constant and sent to a ROS service called "/open_manipulator/goal_joint_space_path_from_present". This service creates and executes a trajectory from the present joint angles, and how much the joint angles should change is what is sent to the service. Both the state space and the action space are continuous in this case. For robotic RL tasks, this is the most suitable approach, since this gives the agent more precise control compared to a discretization of the spaces. Moving all the joints at the same time also makes the task more difficult for the agent to learn, but also gives the potential for more effective trajectories. This task would have been more or less impossible without HER since the chance that the end-effector would have been in the vicinity of the point during the episode is low when doing random exploring. To solve this without HER, either a more complex reward function would have to be created, or the distance from the goal point to the end-effector would have had to been given as a state. To get the substituted goals used for HER, the forward kinematics of the manipulator is used. See Section 2.2.1 for a description of the forward kinematics.

First, a simpler version of this task was created. In this version, the goal point is in the same fixed location every episode. The result of this can be seen in Figure 4.6a and Figure 4.6b. The success rate goes to 1.0 for all epochs after epoch 10, even though this may not be clear from the plot. Note that the average reward plotted here is not the reward that the agent uses to optimize its parameters. The agent uses binary rewards which say whether or not it was in the vicinity of the point at the end of the episode, as described

in Section 2.1.4. The average reward that is plotted in Figure 4.6a has the purpose of illustrating how fast the end-effector is positioned near the point; -1 for each timestep the end-effector is not in the vicinity of the point, and 0 when it is in the vicinity of the point. The hyperparameters used during the training for this task can be seen in Table 4.1. The training procedure was as follows: Train the agent for 30 epochs, where each epoch consists of 30 episodes. After each epoch, 4 evaluation episodes are done, and the results from the evaluation episodes are what is plotted in Figure 4.6a and Figure 4.6b. During the training period, Gaussian noise is added to the action selected by the actor-network. This noise is multiplied by a hyperparameter ϵ_n before it is added to the action. The noise is added because the agent should explore when training, as explained in Section 2.1.3. For exploration, in addition to the exploration noise, a random action is taken with probability defined by the hyperparameter $\epsilon_r \in [0, 1]$. When evaluating the performance between epochs, no exploration noise is added and no random actions are taken. The reason for the low amount of 4 evaluation episodes during this task is that the goal is in the same position for all evaluation episodes. Since no exploration and no updates to the neural networks are done when evaluating, the agent will choose the same series of actions for each evaluation episode. The model architecture of the actor and critic neural networks used in this task is illustrated in Figure 4.4 and Figure 4.5, and is as follows:

- Actor-network
  - The input to this network is the state vector, which consists of the four joint positions, the four joint velocities and the three Cartesian coordinates of the goal point. The input is passed through a ReLU activation function into the hidden layers.
  - After that, there are three fully-connected hidden layers with 256 neurons in each. The activation function between these hidden layers is ReLU.
  - After the third fully-connected layer is the output layer which has four neurons. The activation function between the third fully-connected layer and the output layer is Tanh. The neurons in the output layer, labeled 1-4 correspond to how much the joints 1-4 should change. The Tanh activation function is used instead of ReLU because the required change in the joints could be either positive or negative, and should also be limited in both directions.
- Critic-network
  - In addition to the state vector, the action that is evaluated is used as input to the critic network. The input is passed through a ReLU activation function into the hidden layers.
  - As in the actor-network, there are three fully-connected hidden layers each consisting of 256 neurons after the input, with the ReLU activation function between each of these layers.
  - After the third fully-connected layer comes the output layer, which has a single neuron, which corresponds to the Q-value of the state-action pair used as input. There is no activation function between the third fully-connected layer and the output layer.

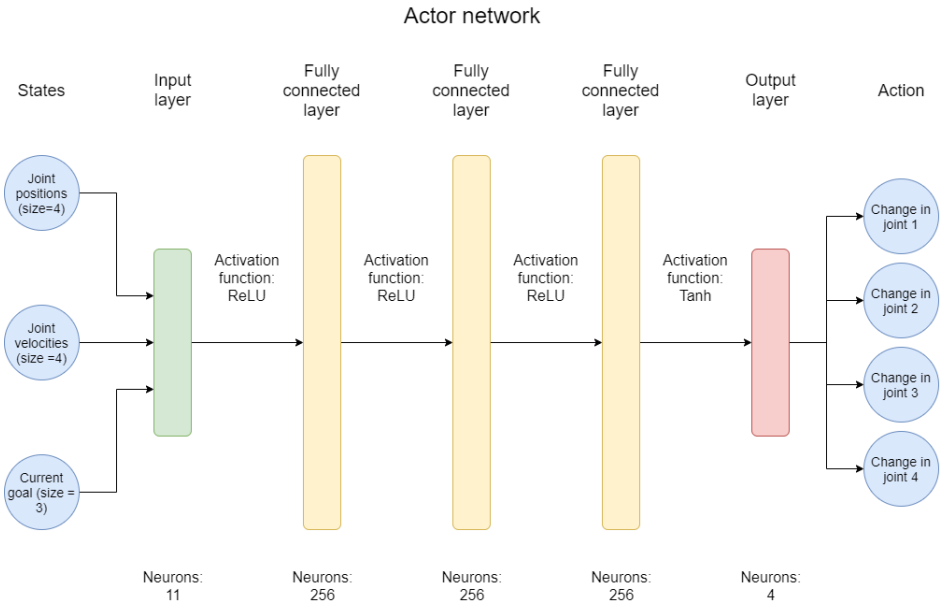


Figure 4.4: Actor-network architecture

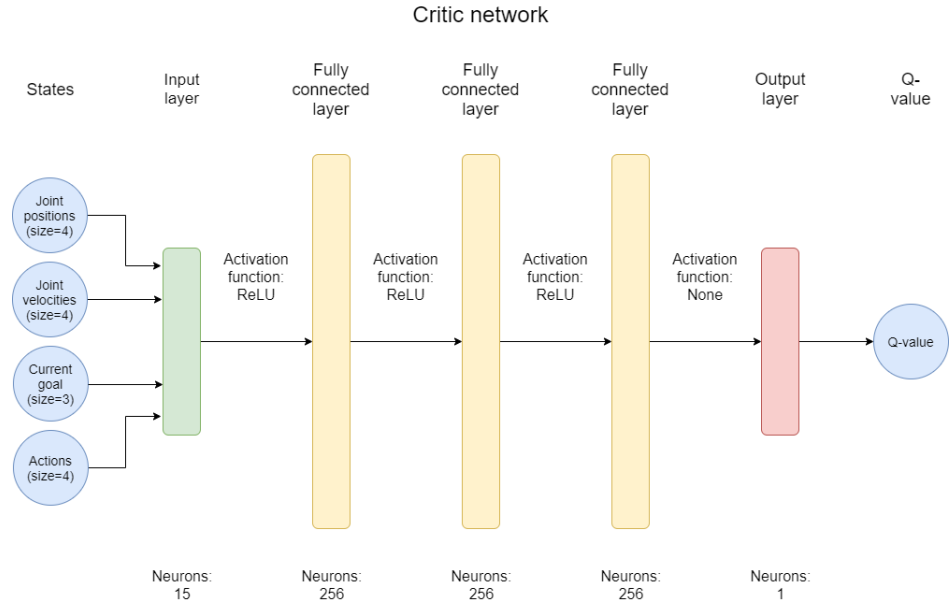
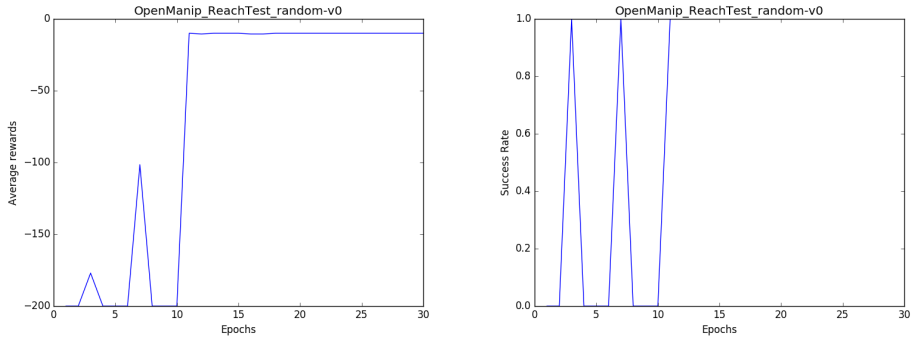


Figure 4.5: Critic-network architecture

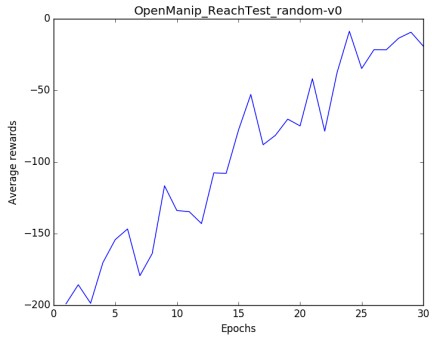


(a) Average reward **(b)** Success rate

Figure 4.6: Success rate and average reward when the goal is at a fixed point in front of the manipulator.

Hyperparameter	Value
Learning rate actor, α_a	0.0009
Learning rate critic, α_c	0.0009
Discount factor, γ	0.98
l2 regression, λ	1
noise_eps, ϵ_n	0.2
random_eps, ϵ_r	0.2
HER ratio to be replaced, k	4
HER replay strategy	future
Mini-batch size	256

Table 4.1: Hyperparameters for FetchReach-v1 and the tasks created for Gazebo



(a) Average reward

(b) Success rate

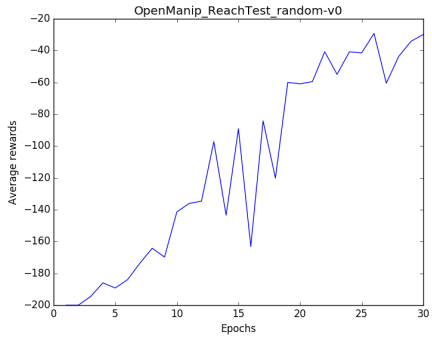
Figure 4.7: Success rate and average reward when the goal is at a random point 180 degrees in front of the manipulator

One problem that arose during training for this task was the simulation speed in Gazebo. During the whole training for this version of the task, the simulation speed was unable to go higher than real-time speed. This means that the whole training from start to finish took just over seven hours for just 1000 episodes of training and 200 episodes for evaluation. Because of this, an attempt was made to contact Anders Haver Vagle, who also used the same simulation on his master's thesis in the spring of 2019, and who managed to simulate at almost four times real-time speed.

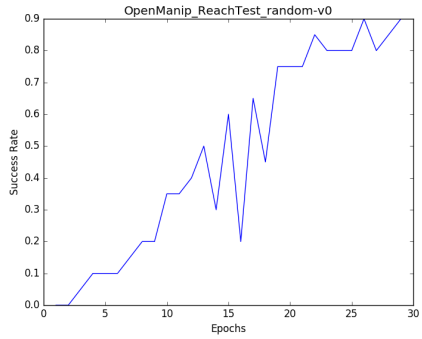
After a brief email correspondence with Anders Haver Vagle, the problem with the simulation speed was discovered. In the Gazebo environment of the openai_ros package, the max_update_rate parameter was set to 1000.0 in the initialization. This means that together with a timestep of 0.001, the real-time factor becomes 1.0. After this was discovered, the max_update_rate parameter was changed to 4000.0. This means that the simulation speed runs at around 4 times real-world speed. After this change, the simulation is still not running as fast as it ideally should, but the training time is significantly shortened.

After the fixed-target version of the task, a new task was created where a random point in front of the manipulator is selected. This random point changes between every episode. The results of this can be seen in Figure 4.7a and Figure 4.7b. In this task, the training procedure was almost identical to the previous task. The only difference is that instead of using 4 evaluation episodes, 20 evaluation episodes were used. This is because the goal is not in the same location each episode, which means that the agent must choose different series of actions in each episode. The same hyperparameters and model architecture was used in this task as in the fixed-target task.

A final expansion on this reach task was created, where the point that the manipulator needs to reach is not only in front of the manipulator. The point is now a uniformly selected



(a) Average reward



(b) Success rate

Figure 4.8: Success rate and average reward when the goal is at a random point in the work-space of the manipulator (360 degrees around it).

random point given in spherical coordinates by

$$r, \theta, \phi \in \mathbb{R} : r \in [0.18, 0.30], \theta \in [-\frac{\pi}{3}, \frac{\pi}{3}], \phi \in [0, 2\pi] \tag{4.1}$$

The location of the point also changes between every episode. The results of this final task can be seen in Figure 4.8a and Figure 4.8b. The same hyperparameters and model architecture was used in this task as in the two previous tasks. The training procedure was similar to the task when the target was in a random position in front of the manipulator. The only difference was that each epoch had double the amount of episodes (from 30 to 60, for a total of 2400 episodes including evaluation) to account for the more difficult task.

4.3 Transferring the learned policy to the real-world manipulator

To see how well the policy learned on the last task transfers to the real world manipulator, a comparison between the actual inverse kinematics and the learned inverse kinematics was done. This comparison was done by selecting three random points from the area given by Equation (4.1). Then the learned kinematics was used to try to make the manipulator reach these points. Afterward, the actual inverse kinematics of the manipulator was also used to try to reach the points for comparison. The results of this experiment are shown in Table 4.2. The scale in this table is in meters.

The learned inverse kinematics was also used to demonstrate how the manipulator can reach a grasping position on the lever described in Section 3.2.2.

Both the experiment that compares the learned inverse kinematics with the actual inverse kinematics and the experiment where the inverse kinematics is used to reach a grasp-

Goal point	Learned policy achieved goal	Learned policy goal error	Inverse kinematics achieved goal	Inverse kinematics goal error
$\begin{bmatrix} -0.096 \\ -0.0275 \\ 0.253 \end{bmatrix}$	$\begin{bmatrix} -0.1016 \\ -0.0324 \\ 0.2102 \end{bmatrix}$	$\begin{bmatrix} 0.0056 \\ 0.0049 \\ 0.0428 \end{bmatrix}$	$\begin{bmatrix} -0.0959 \\ -0.0276 \\ 0.2513 \end{bmatrix}$	$\begin{bmatrix} -0.0001 \\ 0.0001 \\ 0.0017 \end{bmatrix}$
$\begin{bmatrix} -0.006 \\ 0.14 \\ 0.195 \end{bmatrix}$	$\begin{bmatrix} 0.0035 \\ 0.1422 \\ 0.1916 \end{bmatrix}$	$\begin{bmatrix} -0.0095 \\ -0.0022 \\ 0.0034 \end{bmatrix}$	$\begin{bmatrix} -0.0053 \\ 0.1403 \\ 0.1925 \end{bmatrix}$	$\begin{bmatrix} -0.0007 \\ -0.0003 \\ 0.0025 \end{bmatrix}$
$\begin{bmatrix} 0.2085 \\ -0.0038 \\ 0.149 \end{bmatrix}$	$\begin{bmatrix} 0.1959 \\ 0.0147 \\ 0.1590 \end{bmatrix}$	$\begin{bmatrix} 0.0126 \\ -0.0185 \\ -0.0100 \end{bmatrix}$	$\begin{bmatrix} 0.2091 \\ -0.0036 \\ 0.1435 \end{bmatrix}$	$\begin{bmatrix} -0.0006 \\ -0.0002 \\ 0.0055 \end{bmatrix}$

Table 4.2: The results from transferring the learned policy to the real-world manipulator and comparing with the actual inverse kinematics

ing position on the lever was made a video demonstration of. This video demonstration is enclosed with this report.

Chapter 5

Discussion and analysis

5.1 MuJoCo versus Gazebo

As the DDPG+HER implementation used in this thesis was designed to work for the fetch tasks in OpenAI Gym, the excellent results shown in Figure 4.2 are not that surprising [2]. The application of this task is more useful as a comparison between the simulators MuJoCo and Gazebo when used for RL. When training on the FetchReach-v1 task in MuJoCo, the simulation speed easily went over 256 times the real-world speed. For comparison, Gazebo struggled to maintain a simulation speed of 4 times real-world speed. The manipulator used in FetchReach-v1 also has 3 more joints than the OpenMANIPULATOR-X, which should make it more expensive to simulate, so this comparison seems very much in MuJoCo's favor.

Another problem that Gazebo has in the domain of RL is resetting between episodes. In MuJoCo this is done very quickly, as the episode is just instantly restarted. In Gazebo, the manipulator has to be moved back to the starting position. This takes a lot of unnecessary time, especially considering that Gazebo is already simulating far slower than MuJoCo. This problem with resetting has also been encountered by previous master students Anders Haver Vagle and Arild Vermedal [28, 29] who also solved the resetting problem by making the manipulator move back to its starting position on its own.

Most of the research papers on robotic RL found during the work on this thesis use MuJoCo as the simulator. This suggests that in the robotic RL community, MuJoCo is often the preferred simulator [9, 20, 30, 31]. It would be interesting to see how the tasks that were simulated in Gazebo in this thesis would be simulated in MuJoCo for a completely fair comparison. Unfortunately, as far as to the author's knowledge, a model of the OpenMANIPULATOR-X does not yet exist in MuJoCo. More on this is discussed under further work in Section 6.2.

5.2 Performance of the tasks

When comparing the results from the Gazebo tasks and the results from FetchReach-v1, it seems as if the Gazebo tasks are significantly under-performing. The two first Gazebo tasks and the FetchReach-v1 task were trained for the same amount of epochs, with the same amount of episodes in each epoch. Comparing plots Figure 4.2 and Figure 4.7b which both use a randomly selected point each episode as the goal point, the FetchReach-v1 task has a success rate of 1 after just a couple of epochs, and the Gazebo task requires all 30 epochs to get close to similar results. The reason for this difference may simply be that the hyperparameters are better tuned for the FetchReach-v1 task, but the Gazebo tasks can also be argued that are more difficult.

In the FetchReach-v1 task, the input to the manipulator is done in the task space, which has 3 dimensions (X-Y-Z); in the Gazebo tasks, the input to the manipulator is done in joint space, which has 4 dimensions (joint 1-4). This gives the Gazebo tasks an action space that has one more dimension and thus makes the action space more time-consuming to explore.

In both of these tasks, the goal position is given in Cartesian coordinates, which correspond to the task space. It may be that it is easier for an agent to learn how to go to a point that is given in coordinates that correspond to the coordinates of its action space. This may also be one of the reasons why the FetchReach-v1 task had a significantly better performance than the tasks created for Gazebo.

The difference in how the environments are set up may also be a reason for the different performances. The Gazebo environment was created by the author of this thesis, an individual student; the MuJoCo environment was created by OpenAI, a relatively big corporation in the RL community. The difference in the quality of the environments with regards to RL is therefore probably significant.

5.3 Transferring the policy learned in the simulator to the real world manipulator

The results from Section 4.3 are obtained by solely training the agent in the simulator. Considering that the agent was never trained on the real manipulator, the results are sufficiently good. The largest error was when the manipulator was going to reach the first point in Table 4.2. The distance from the goal point to the end-effector was in this case

$$\sqrt{(0.0056)^2 + (0.0049)^2 + (0.0428)^2} = 0.043442,$$

which is just over 4.3 cm. This point also was the one that corresponded to the most deviating behavior in the learned inverse kinematics compared to the actual kinematics. For points that are approximately right behind the manipulator, the agent learned a behavior where it bends over backward to reach the point. Since it is not specified in the reward function what the orientation of the end-effector should be, only the position, the agent's objective was to reach the goal regardless of the orientation. To be able to also specify the orientation of the end-effector, this could have also been included in the goal, and the reward could have been given based on whether the end-effector was in both the correct

orientation and position.

Following is some of what could have been done to possibly improve the performance on the real world manipulator:

- The agent could have been trained longer in the simulator
- The area around the goal point where the agent receives the reward was maybe too big for these tasks, and could probably have been made smaller for more accurate learned inverse kinematics
- The agent could have been trained on the real world manipulator in addition to the simulator

Reaching the lever using the learned inverse kinematics

As shown in the last part of the video demonstration, the agent can make the manipulator reach the lever in a grasping position. What remains here is for the agent to learn how to grasp, and then how to pull or push the lever. The location of the lever was in this case found by measuring with a ruler where the lever was in comparison to the manipulator. To get the location during a more realistic operation, a camera could be used. Then one could use for instance supervised learning to make a computer program that can find the location of the lever relative to the manipulator based on an image. A depth camera could also be used to more accurately determine the location of the lever relative the manipulator.

Chapter 6

Conclusion

6.1 Answering the research questions

In this section, the research questions presented in Section 1.2 will be answered.

Is it feasible to let reinforcement learning handle (at least some parts) of the operation of a robotic manipulator in safety-critical applications?
One of the main difficulties in doing safe RL is how to handle exploration. In this thesis, the exploration was done by adding noise to the action selected by the neural network, and also by having a low chance of taking a random action. For a safety-critical operation, this is most likely not sufficiently safe. Taking random actions is not compatible with safe operation. For RL to be used online in a safety-critical application, the exploration has to be done in a better and more predictable way than what was done in this thesis.
There are situations where RL can be useful even in safety critical-applications though. This is when RL is used to solve a task where the solution is difficult to obtain. This can, for instance, be the case when a robot has very many DOF. Then the RL agent can be trained in a simulator, and the pure policy can be transferred to the real world and can be applied to the safety-critical operation. In this thesis, when the agent had been sufficiently trained on the tasks, there was almost no irregular behavior when the pure policy was applied during the evaluation episodes. When the policy was transferred over to the real world manipulator, the policy also performed well and quite predictable.

How practical is it to use the simulator Gazebo and the framework ROS for reinforcement Learning?
Using Gazebo and ROS for RL can be an arduous task. Setting up an environment for RL in Gazebo and ROS is not easy, but the package openai_ros makes this process more manageable. The biggest disadvantage of this arrangement is the simulation speed in Gazebo. For relatively simple tasks such as those in this thesis, the simulation speed is adequate, as the number of required episodes is low. For more complex tasks, which require a large number of episodes for training, the simulation speed will be a major deterrent. If one is

to do robotic RL, other simulators such as MuJoCo will probably be preferred. This is under the condition that designing an RL environment for these other simulators is not too difficult.
Another disadvantage with Gazebo, when used for RL, is the trouble with resetting the environment after an episode has completed. In this thesis, the manipulator had to move back to the initial position on its own, and this can lead to troublesome situations if the manipulator is stuck. A better way to reset the environment can likely be found in Gazebo, but a good solution was not found during the work on this thesis.

6.2 Further work

This work will be continued in the spring of 2020, during the work on the author's master's thesis. The work done on this pre-project thesis has laid the groundwork for the master's thesis both when it comes to the theory used in the thesis, but also when it comes to familiarity with the equipment and task. The loop will be closed by adding a camera that can detect the location of for instance a lever, and the task will be transferred to the real-world manipulator. The camera used for this will be the Raspberry Pi camera which will be mounted on the wrist of the manipulator. Additionally, the Intel Realsense D435 depth camera may be placed in a position where it has a more general overview of the environment. The idea is that when combining the view from both of these cameras, a more complete understanding of the task and environment may be obtained by the agent.

In the master's thesis work, other DRL algorithms than DDPG will be tried out. A promising algorithm that first will be investigated is the Soft Actor-Critic (SAC) algorithm. This is a state-of-the-art reinforcement learning algorithm. SAC is off-policy and can, fortunately, be combined with HER. In the research paper on HER by Andrychowicz et al. [13], it is stated that HER works very well with multi-goal tasks. The task on the master's thesis will most likely be to create an agent that can learn how to pull and push the lever described in Section 3.2.2. This is a task that easily can be structured into multiple goals. The first goal can be to move the end-effector into a grasping position on the lever. Then the second goal can be to grasp the lever and then pull or push it. A third goal can then, for instance, be to move the manipulator back into the starting position. This first goal is similar to the tasks already done in Gazebo in this thesis. The main difference between the tasks done in this thesis and the first goal described here is that the orientation of the end-effector may have to be specified in addition to the position, for easier grasping during the second goal. The location of the lever relative to the manipulator also has to be detected by a camera, instead of specified by the user.

Because of the difficulties discovered in this thesis in using Gazebo for RL, another approach will be tested. Initially, during the master period, an attempt will be made to create a model of the OpenMANIPULATOR-X and the experimental setup for MuJoCo. As discussed in Section 5.1, it seems like MuJoCo is much more suited for RL than Gazebo. The author does not yet know how difficult it is to create an RL environment for MuJoCo, or if it even is possible, but this is something that will be discovered early on in the work on the master's thesis.

A major part of the master's thesis will concern transferring the learning agent successfully from the simulator to the real world. This is often a big problem when it comes

to robotic RL, as discussed in Section 2.1.5. Two techniques that can make the transition from the simulator to the real-world better are *dynamics randomization* and *domain randomization* [20, 30]. Dynamics randomization was already discussed in this thesis in Section 2.1.5. Domain randomization is somewhat similar to dynamics randomization, except that it concerns randomizing the environment/domain in specific ways such that the most important visual information for the solving of the task is constant between episodes. This way the learning agent used for visual recognition of the environment can learn what visual information is most important to solve the task. These two techniques will be investigated during the master project, and possibly be included as a part of the final result. As a side note, both of these techniques are done in MuJoCo in the papers where they are presented, so it is assumed that it is not overly complicated to implement the techniques in this simulator if a model of the OpenMANIPULATOR-X can be made for MuJoCo.

Bibliography

[1] emanual.robotis.com, *OpenMANIPULATOR*, 2019 (accessed 25- Sept- 2019). [Online]. Available: http://emanual.robotis.com/docs/en/platform/openmanipulator_x/overview/

[2] A. Imran, *Robotics-DDPG-HER*, 2019 (accessed 06- Dec- 2019). [Online]. Available: https://github.com/alishbaimran/Robotics-DDPG-HER

[3] A. Ezquerro, M. A. Rodriguez, and R. Tellez, *openai_ros package*, 2019 (accessed 05- Dec- 2019). [Online]. Available: http://wiki.ros.org/openai_ros

[4] TheConstructCore, *openai_ros source*, 2019 (accessed 10- Dec- 2019). [Online]. Available: https://bitbucket.org/theconstructcore/openai_ros/src/kinetic-devel/

[5] S. Russel and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Prentice Hall, 2010.

[6] A. Lekkas, "Lecture notes in TTK23 Introduction to Autonomous Robotics Systems for Industry 4.0," October-November 2019.

[7] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013.

[8] D. Silver, T. Hubert, J. Schrittwieser, and D. Hassabis, *AlphaZero Shedding new light on chess, shogi, and Go*, 2018 (accessed 15- Nov- 2019). [Online]. Available: https://deepmind.com/blog/article/alphazero-shedding-new-light-grand-games-chess-shogi-and-go

[9] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," 2015.

[10] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, pp. 1238–1274, 2013.

[11] Open Source Robotics Foundation, *Gazebo - website*, 2014 (accessed 17- Dec- 2019). [Online]. Available: http://gazebosim.org/

[12] Roboti LLC, *MuJoCo - advanced physics simulation*, 2018 (accessed 09- Dec- 2019). [Online]. Available: http://www.mujoco.org/

[13] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba, "Hindsight experience replay," 2017.

[14] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. The MIT Press, 2017.

[15] M. Wiering and M. van Otterlo, Eds., *Reinforcement Learning: State-of-the-Art*. Springer, 2012.

[16] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015. [Online]. Available: http://neuralnetworksanddeeplearning.com/

[17] S. Avinash, *Understanding Activation Functions in Neural Networks*, 2017 (accessed 15- Nov- 2019). [Online]. Available: https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0

[18] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," *31st International Conference on Machine Learning, ICML 2014*, vol. 1, 06 2014.

[19] M. Plappert, M. Andrychowicz, A. Ray, B. McGrew, B. Baker, G. Powell, J. Schneider, J. Tobin, M. Chociej, P. Welinder, V. Kumar, and W. Zaremba, *Ingredients for Robotics Research*, 2018 (accessed 05- Dec- 2019). [Online]. Available: https://openai.com/blog/ingredients-for-robotics-research/

[20] X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel, "Sim-to-real transfer of robotic control with dynamics randomization," 2017.

[21] M. W. Spong, S. Hutchinson, and M. Vidyasagar, *Robot Modeling and Control*. John Wiley and Sons, Inc, 2006.

[22] *About ROS*, (accessed 06- Dec- 2019). [Online]. Available: https://www.ros.org/about-ros/

[23] *ROS - Getting started tutorial*, (accessed 06- Dec- 2019). [Online]. Available: http://wiki.ros.org/ROS/Introduction

[24] Open Source Robotics Foundation, *Gazebo - beginner tutorial*, 2014 (accessed 16- Dec- 2019). [Online]. Available: http://gazebosim.org/tutorials?cat=guided_b&tut=guided_b1

[25] J. Hsu, N. Koenig, and D. Coleman, *gazebo_ros_pkgs*, 2018 (accessed 16- Dec- 2019). [Online]. Available: http://wiki.ros.org/gazebo_ros_pkgs

[26] Torch Contributors, *PyTorch - Documentation*, 2019 (accessed 16- Dec- 2019). [Online]. Available: https://pytorch.org/docs/stable/index.html

[27] OpenAI, *OpenAI Gym - Documentation*, (accessed 10- Dec- 2019). [Online]. Available: http://gym.openai.com/docs/

[28] A. H. Vagle, "Reinforcement learning for robotic manipulation," Master's Thesis, NTNU - Norwegian University of Science and Technology, 2019.

[29] A. Vermedal, "Deep reinforcement learning for valve manipulation," Master's Thesis, NTNU - Norwegian University of Science and Technology, 2019.

[30] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Sep 2017. [Online]. Available: http://dx.doi.org/10.1109/IROS.2017.8202133

[31] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," 2015.