Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Sindre Benjamin Remman

Robotic manipulation using Deep Reinforcement Learning

Master's thesis in Cybernetics and Robotics

Supervisor: Anastasios Lekkas

June 2020

NTNU
Norwegian University of Science and Technology

Sindre Benjamin Remman

Robotic manipulation using Deep Reinforcement Learning

Master's thesis in Cybernetics and Robotics
Supervisor: Anastasios Lekkas
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Preface

This thesis serves as the final work on my master's degree in Cybernetics and Robotics at the Norwegian University of Science and Technology (NTNU). It was created under the supervision of Anastasios Lekkas, during the spring of 2020.

This thesis has three main goals. The first goal is to examine how methods from Deep Reinforcement Learning can be used to control a robotic manipulator to perform various tasks. The second goal is to examine which properties are needed from a simulation framework if a Deep Reinforcement Learning agent is trained in a simulator before it is transferred to the real world. The final goal is to examine how methods from Explainable Artificial Intelligence can be used to interpret the decisions of a Deep Reinforcement Learning agent. It is assumed that the reader has a background in machine learning and mathematics. However, the theory behind this work is also presented.

The robotic manipulator named OpenMANIPULATOR-X, created by ROBOTIS, was provided by NTNU for the work on this thesis. A lever that is used in some of the experiments in the thesis was created in the fall of 2019, together with Karl Ylvisaker and the workshop of the Department of Engineering Cybernetic. The lever was created during the work on the specialization project that preceded this thesis. The Robotic Operating System (ROS) was used to control the manipulator using packages provided by ROBOTIS [1]. The Deep Deterministic Policy Gradient and Hindsight Experience Replay implementation that was used in this thesis was adapted from an implementation by Alishba Imran [2] during the work on the specialization project. This implementation is based on the machine learning library PyTorch. The two simulators Gazebo and PyBullet was used, and the code was written in Python 2.7. My personal computer was used for all experiments. The figures in the thesis have been created by me using draw.io unless otherwise stated. Some plots were create using MATLAB, some using the Python package shap, and the remaining was created using matplotlib.

In the middle of the semester, the COVID-19 pandemic began in Norway, and the country was under lock-down for the rest of the semester. This resulted in the semester being even more demanding than initially envisioned.

A huge thank you goes to Anastasios Lekkas for his encouraging support and supervision throughout the semester. Lastly, a special thank you goes to my friends and family who had to put up with me in a period where most of what was on my mind were Deep Reinforcement Learning, robots, and this thesis.

Abstract

The task of improving robots' autonomy has been a goal for researchers for many decades. Various tools have been used to do this, and today's robots are more autonomous than ever. However, there is still something missing. For robots to be truly autonomous, they have to have the ability to learn from their experience and improve the performance based on what they have encountered. Based on the advancements in Artificial Intelligence in the last decade, more research is now being done into Reinforcement Learning. Reinforcement Learning is a tool that can enable robots to improve their performance gradually and might be part of what will take future robots to a new level of autonomy.

Inspired by the arrival of several new Reinforcement Learning algorithms and methods in the last few years, this thesis aims to examine the combination of Reinforcement Learning and robotics. Specifically, the version of Reinforcement Learning that is used in this thesis is named Deep Reinforcement Learning. This variant combines Reinforcement Learning with Artificial Neural Networks and has already had great success.

A simulated and a real version of the OpenMANIPULATOR-X by ROBOTIS was used in this thesis's work. This robotic manipulator was provided by the Norwegian University of Science and Technology (NTNU). The robotic manipulator was used to manipulate a lever created during the work on the specialization project that preceded this thesis. The lever was made in collaboration with fellow master student Karl Ylvisaker and the workshop of the Department of Engineering Cybernetics. To create an intelligent agent that managed to manipulate this lever, the agent was first trained in the simulators PyBullet and Gazebo, before being transferred to the real-world environment. It managed to complete the task in the real environment; however, the performance was lower than it was in the simulated environments. A video demonstration was made of this real-world experiment, and the video is delivered with this thesis.

Several other tasks were done in the simulated environments, such as some tasks where the base of the manipulator is oscillating along the world z-axis, in an attempt to emulate underwater currents. The plan had initially been to also transfer these oscillating tasks to the real world. However, because of the COVID-19 pandemic, a device that could cause such oscillations in the real world was never made, and the results from these tasks are solely from the simulated environments. Another video, which shows the agent's performance in three simulated environments with oscillations, is also delivered with this thesis. It can be seen in this video that the agent manages to complete its goal sufficiently well in all three tasks, even though some behavior that would likely not transfer well to the real world can be seen.

One of the problems with Deep Reinforcement Learning methods is that it is challenging to determine how the agents arrive at their decisions. Therefore, a method from Explainable Artificial Intelligence named SHapley Additive exPlanations (SHAP) was used in an

attempt to interpret an agent's decision making. Even though this agent performs very well on its task, the results from using SHAP shows that the agent can not be fully trusted.

Sammendrag

Det å forbedre roboters autonomi har lenge vært et mål for forskere. Ulike verktøy har blitt brukt for å gjøre dette, og dagens roboter er mer autonome enn noen gang før. Det er imidlertid fortsatt noe som mangler. For at roboter skal være virkelig autonome, må de ha evnen til å lære av sin erfaring og forbedre seg basert på hva de har opplevd. Basert på fremskritt innen kunstig intelligens det siste tiåret, forskes det nå mer på forsterkende læring (eng. Reinforcement Learning). Forsterkende læring er et verktøy som kan gjøre det mulig for roboter å forbedre prestasjonene sine gradvis og kan være en del av det som vil ta fremtidige roboter til et nytt nivå av autonomi.

Inspirert på stadig flere nye algoritmer og metoder innen forsterkende læring de siste årene, tar denne oppgaven for seg å videre undersøke kombinasjonen av forsterkende læring og robotikk. Spesifikt heter den versjonen av forsterkende læring som brukes i denne oppgaven for dyp forsterkende læring (eng. Deep Reinforcement Learning). Denne varianten kombinerer forsterkende læring med kunstige nevrale nettverk og har allerede hatt stor suksess.

En simulert og en ekte versjon av robotmanipulatoren OpenMANIPULATOR-X av ROBOTIS ble brukt i arbeidet på denne oppgaven. Denne robotmanipulatoren ble supplert av Norges teknisk-naturvitenskaplige universitet (NTNU). Robotmanipulatoren ble brukt til å manipulere en spak som ble lagd under arbeidet med fordypningsprosjektet som ble gjort i forkant av denne masteroppgaven. Spaken ble laget i samarbeid med Karl Ylvisaker, som også var masterstudent, og det tekniske verkstedet ved Instituttet for teknisk kybernetikk. For å lage en intelligent agent som klarer å manipulere denne spaken, ble agenten først trent i simulatorene PyBullet og Gazebo, før den ble overført til det virkelige miljøet. Den klarte å fullføre oppgaven i det virkelige miljøet, men prestasjonen var imidlertid lavere enn i de simulerte omgivelsene. Det ble lagd en videodemonstrasjon av dette eksperimentet, og videoen blir levert med denne oppgaven.

Flere andre oppgaver ble utført i de simulerte miljøene, for eksempel noen oppgaver der basen til manipulatoren oscillerer langs z-aksen, i et forsøk på å emulere undervannsstrømmer. Planen var i utgangspunktet å også overføre disse oscillerende oppgavene til den virkelige verden. På grunn av COVID-19-pandemien ble det imidlertid aldri laget et redskap som kan forårsake slike oscilleringer i den virkelige verden, og resultatene fra disse oppgavene er utelukkende fra de simulerte miljøene. En annen video, som viser hvordan agentene oppfører seg i tre simulerte miljøer med oscilleringer, er også levert med denne oppgaven. Det kan sees i denne videoen at agenten klarer å fullføre målet sitt tilstrekkelig godt i alle tre oppgavene, selv om en viss oppførsel som sannsynligvis ikke vil overføres godt til den virkelige verden kan sees.

Et av problemene med metoder innen dyp forsterkende læring er at det er utfordrende å tolke hvordan agentene kommer til sine beslutninger. Derfor ble en metode fra forklarbar

kunstig intelligens (eng. Explainable Artificial Intelligence) kalt SHapley Additive exPlanations (SHAP) brukt i et forsøk på å tolke en agents beslutningstaking. Selv om denne agenten har god prestasjon på sin respektive oppgave, viser resultatene fra å bruke SHAP at agenten ikke kan stoles på.

Contents

List of Tables

List of Figures

Acronyms

AI Artificial Intelligence. 2, 33

ANN Artificial Neural Network. 1, 2, 4, 6–8, 12, 13, 24, 30, 55, 59

CNN Convolutional Neural Network. 79

DDPG Deep Deterministic Policy Gradient. 4, 25, 26, 32, 45, 46, 55, 67–70

DL Deep Learning. 2, 32, 33

DOF Degrees of Freedom. 36, 40, 77

DP Dynamic Programming. 18

DQN Deep Q-Network. 24–26

DRL Deep Reinforcement Learning. 2–4, 6, 15, 24, 26, 28, 30, 32, 39, 48–51, 63, 68, 70, 77–79

GPU Graphics Processing Unit. 38

HER Hindsight Experience Replay. 28, 29, 45, 52, 68, 77, 79

MDP Markov Decision Process. 18, 21, 25, 26

MSE Mean Squared Error. 13, 16

MuJoCo Multi-Joint dynamics with Contact. 39

OM-X OpenMANIPULATOR-X. 3, 40

OPM-RL-ENVS open_manipulator_rl_environments. 3, 48, 73

ReLU Rectified Linear Unit. 10, 11, 45, 46, 68

RL Reinforcement Learning. xi, 1, 2, 4, 6, 15–24, 26, 28, 38, 48

ROS Robot Operating System. 3, 37–40, 43, 48, 64

SAC Soft Actor-Critic. 77, 79

SGD Stochastic Gradient Descent. 15

SHAP SHapley Additive exPlanations. iii, iv, vi, 4, 33, 34, 73, 79

Tanh Hyperbolic Tangent. 9, 45

URDF Unified Robot Description Format. 40, 51, 52

UUV Unmanned Underwater Vehicle. 50

XAI Explainable Artificial Intelligence. 2–5, 32, 78, 79

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* [3].

Robotic systems have contributed to the advancement of human society for decades. From industrial robots that can perform tasks that are tiresome, repetitive, and possibly dangerous for humans to medical robots that can help save human lives, it is clear that robots have come to play an essential role in today's world. The potential for the deployment of robots in still more fields is vast, and researchers are still working to develop new and innovative ways that can improve the capability of robots. One possible way to improve robots' capabilities is to improve their autonomy. Control theory and cybernetics have done a great deal to improve the autonomy of robots, but arguably, this may not be enough. When using such tools, the robot is in the end, acting and reacting to situations that the engineers can anticipate beforehand. Pre-programming the behavior of robots has its limitation; if a situation happens that can not be predicted during the development of the robot, it is anyone's guess what will happen. For robots to improve their autonomy further, it has to be possible for robots to improve their behavior based on what they experience. This is where Reinforcement Learning (RL) appears as a tool to enable robots to improve their autonomy further. In contrast to traditional methods for robot control, RL can enable a robot to learn new strategies from its experiences. RL and control engineering share some of the same concepts, albeit with different terminology, as can be seen in Table 1.1.

During this decade, RL has achieved success in various fields. Previously, a problem with RL was what is called the curse of dimensionality [7]. This curse refers to the problem that RL methods have when it comes to high-dimensional inputs and outputs. In 2013, DeepMind found a way to alleviate this curse by successfully combining RL with Arti-

Control Engineering	Reinforcement Learning
Controller	Agent (policy)
Controlled system	Environment
Control signal	Action

Table 1.1: The difference in terminology between control engineering and reinforcement learning [3, lecture 1].

ficial Neural Networks (ANNs), which are powerful modeling tools [8]. The result from combining RL and ANNs is called Deep Reinforcement Learning (DRL). By using DRL, DeepMind managed to create an agent that learned to play Atari 2600 by solely using the raw pixels of the screen as input. DeepMind has also created AlphaZero, a system that uses DRL to play chess, shogi, and Go. AlphaZero managed to beat world-champion computer programs on all these games. Recently, research has also been done into applying DRL to robotic systems [9, 10, 11, 12, 13].

A problem with robotic DRL is that current methods require a significant amount of experience to learn how to solve a task. If a real robot gathers this experience, this can, for instance, lead to degradation of the robot due to overuse. For the work on this thesis, this problem was solved by first learning from experiences gathered in a simulated environment similar to the real one, before transferring the learned behavior to the real robot.

When it comes to working with ANNs in general, a challenge is that it is hard to know how the ANN arrives at its decision. This is a challenge that becomes unfeasible for a human to solve when the network is sufficiently complex. It is not easy to trust a decision-making agent whose actions can not be explained, even though it may perform well in general. It is too easy to use an ANN just because it has achieved good performance. A more reasonable interpretation of the ANN's good results is that it has achieved good results within the tested data. However, what if the network encounters something that it has not been tested on? Then there are no guarantees that the network will make a correct decision.
Furthermore, what if someone tries to exploit the network's weaknesses to make it arrive at a wrong decision? One pixel attacks have become famous for fooling ANNs to arrive at the wrong decision when it comes to image recognition [14]. It is conceivable that similar attacks can be made for DRL, which can lead to especially dangerous situations when it comes to robotic DRL. These are some of the reasons why Explainable Artificial Intelligence (XAI) has become an increasingly popular research topic in the last years [5]. XAI methods can be applied to Artificial Intelligence (AI) solutions so that humans can understand their decisions. XAI for Deep Learning (DL), and especially for robotic DRL, is a relatively new research topic, so more research into this is required.

1.2 Objectives and research questions

The main goal of this thesis is to answer the following research questions:

Research questions:

- Which advantages and disadvantages does Deep Reinforcement Learning (DRL) have when it comes to controlling a robotic manipulator under various conditions?
- Which properties should a simulation framework hold to make the transfer to the real-world efficient for robotic deep reinforcement learning, and what can be done to improve the efficiency of this transfer?
- How can methods from Explainable Artificial Intelligence (XAI) be used to interpret the results from this thesis?

To answer these research questions, a series of objectives has been made. These objectives track the progress during the work on this thesis.

Objectives:

1. Research simulation framework alternatives to the simulator Gazebo for robotic DRL.
2. Create a DRL framework that can transfer between simulators and the real-world manipulator on the tasks done in this thesis.
3. Create various robotic DRL tasks that use the OpenMANIPULATOR-X (OM-X) in the DRL framework just mentioned.
4. Design a strategy for transferring from the simulated environment to the real environment.
5. Transfer the DRL agent trained in the simulator(s) to the real world.
6. Find a method from XAI that can help interpret the results from this thesis, and use it to interpret some of the results.

1.3 Contributions

A package for Robot Operating System (ROS) called open_manipulator_rl_environments (OPM-RL-ENVS) was created during the work on this thesis. This package serves as a framework that can be used to create DRL tasks for the OM-X. The framework makes it possible to transfer the tasks between the two simulators PyBullet and Gazebo, and the real-world manipulator. The interface of the OpenAI Gym is used, which means that all existing DRL algorithms that use this interface can be applied to the tasks created by using this framework. The package is written in the programming language Python 2.7. Multiple tasks were created using this framework, with the main tasks being a lever manipulation task and three tasks where the base of the manipulator is exposed to an oscillating disturbance that is meant to emulate underwater currents.

By using the computer graphics software Blender, a simulation model of a lever was created during the work on this thesis. This model serves as a representation of the lever created during the author's specialization project [15]. The model can be used in both of the simulators mentioned above. The model's dimensions and appearance are accurate

compared to the real lever. However, if the model is going to be used in further work, the model's dynamics should be tuned to make it behave like the real lever.

This thesis also shows that DRL can be used to control a robotic manipulator to manipulate a lever to a specified goal angle very successfully in simulators, and to a lesser extent, but still successfully in the real world. Furthermore, by showing that this task can also be performed while the manipulator is exposed to an oscillating disturbance, which is supposed to emulate underwater currents, this strengthens the possibility of applying DRL to subsea tasks. However, the lever manipulation with oscillations was only done in simulations due to real-world constraints.

Finally, this thesis shows how SHapley Additive exPlanations (SHAP) values, a technique from XAI, can be used to interpret the decisions of a DRL agent. The results for this gave reasons for not trusting an agent trained on a simple reaching task.

1.4 Outline of the report

This thesis consists of six main chapters, including this introductory chapter. Following is an overview of the next five chapters:

- Chapter 2: Theory
  - This chapter introduces terminology and theory that is important for the rest of the thesis. It starts with an overview of machine learning and ANNs. After this, an introduction to RL is given before transitioning to DRL, which is the main subject of this thesis. An overview of the Deep Deterministic Policy Gradient (DDPG) algorithm, which is used in the experiments later and the algorithms that lead up to DDPG, is given throughout the RL and DRL sections. The machine learning section finishes with a discussion of reward shaping, sim-to-real transfer of DRL agents, and XAI. The final section of the theory chapter involves robotic manipulators and the theory and terminology of these, which is used throughout the thesis.
- Chapter 3: Equipment and setup
  - An overview of the software and the hardware used in this thesis is given.
- Chapter 4: System design
  - The design of the system used to solve the tasks in this thesis is explained. This explanation includes the DRL algorithm implementation and the DRL framework that has been created for the thesis.
- Chapter 5: Results and discussion
  - The main results of this thesis are shown and discussed.
- Chapter 6: Conclusion
  - The research questions are answered, and an overview of the further work that could be done on this topic is given.

Chapter 2

Theory

As this thesis is a continuation of the specialization project done in the fall of 2019, the required theory is similar. This means that the theory chapter is mostly an extended and updated version of the author's earlier work [15], except for Section 2.1.6, which deals with Explainable Artificial Intelligence (XAI).

2.1 Machine learning

Machine learning algorithms are algorithms that can improve their performance based on the data that they process. This means that the algorithm can be seen as learning more about how to solve a particular problem as it processes more data related to the problem. The motivation for using machine learning is to solve problems that are difficult to engineer a solution to by hand. Some examples of such problems are computer vision tasks, medical imaging, speech- and text recognition. There are three primary variants of machine learning:

- Supervised learning
  - It concerns learning from a set of labeled examples [16, p.2]. Often used in, for instance, image recognition.
- Unsupervised learning
  - It concerns finding structures in collections of unlabeled data [16, p.2].
- Reinforcement learning
  - It is more similar to how humans learn compared to other variants. Mainly concerns how a decision-making agent can obtain information about its environment by exploring, and then exploiting this information to maximize a feedback metric [16, pp. 1-3].

This chapter and thesis mainly consider Reinforcement Learning (RL), although some examples from supervised learning are used when discussing Artificial Neural Networks (ANNs). Specifically, the variant of RL that is used in the experiments of this thesis 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 that are more specific for robotic RL are also explored, such as transferring from simulator to real-world (sim-to-real).

2.1.1 Artificial Neural Networks

This section on ANNs takes inspiration from Chapters 1, and 2 in the book *Neural Networks and Deep Learning* [17], but is written in the author's own words.

ANNs are 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 2.1 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 was the first type of artificial neuron. Frank F. Rosenblatt conceived the perceptron in 1958. He created the perceptron "to illustrate some of the fundamental properties of intelligent systems in general" [18]. For the perceptron in Figure 2.1 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 more suitable for artificial neurons. Certain properties are needed from the activation functions of artificial neurons to be able to use the learning algorithms described

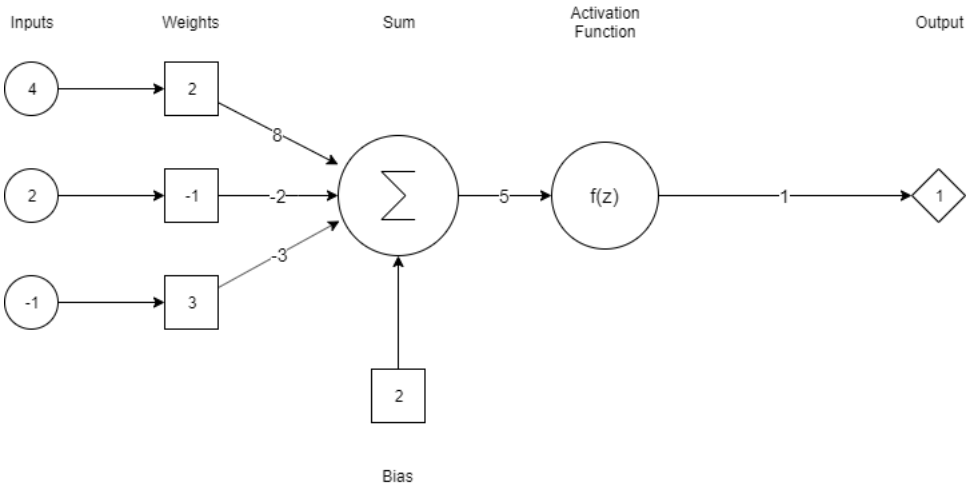


Figure 2.1: Example of a perceptron

below. One of these properties is that a small change in the neuron's biases and weights will result in a small change in the neuron's output. 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 (or the reverse), which is a radically different change in output. Another of the properties is that the gradient of the activation function has to be defined, and ideally have some areas where it is not equal to zero. For the perceptron, the gradient is not defined for $w^T x + b = 0$, and for $w^T x + b \neq 0$, the gradient is zero. This means that the learning algorithms commonly used for ANNs would work poorly for perceptrons. 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 [17, ch. 1].

For artificial neurons to be able to approximate complex functions, they must be put together into a network, an ANN. An ANN is structured into layers, which each consist of a set of neurons. 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 2.2. In this example, there are three neurons in the input layer, four neurons in the hidden layer, and two neurons in the output layer. This type of network is called a feedforward network, which means that the 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[1]. The number of neurons and the number of layers in a neural network can be used to describe the depth and width of the network. The depth of a neural network is defined as the number of layers (including the output layer, but excluding the input layer). The width of a neural network is defined as the maximum

[1] 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.

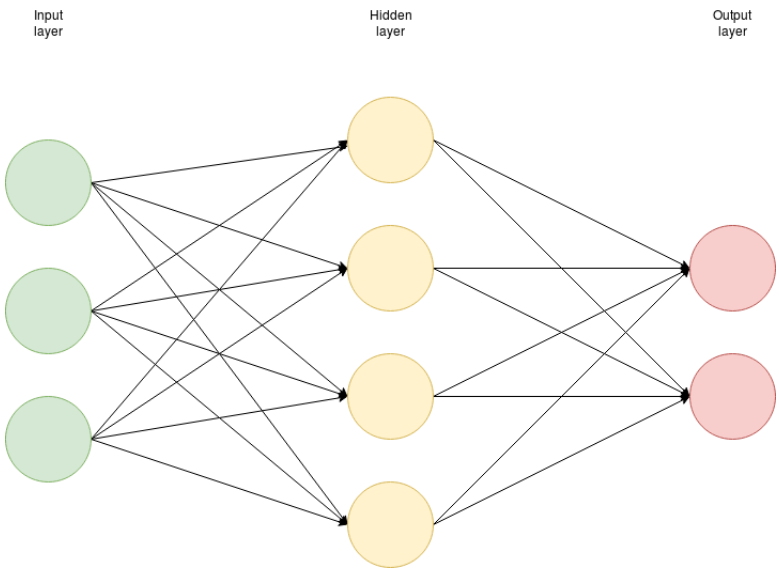


Figure 2.2: Example of a neural network with a single hidden layer

number of neurons in a layer [19].

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 work very well for data-driven approaches [20]. The function that an ANN approximates is often quite complex and is often nearly 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 are needed. These are the learning algorithms. Before examining the learning algorithms, an overview of some common activation functions is given.

Activation functions

It is challenging to find an excellent way to evaluate activation functions. The most common way to compare activation functions is to test them on various tasks, and then compare the results empirically. This is not an ideal way to this because the results depend on the tasks that are used for testing. It is reasonable to assume that an activation function can perform better on some tasks and worse on others, and therefore comparing activation functions on a set of tasks may prove unfair for some activation functions. Nevertheless, this is probably the best way to compare them today, and this thesis' choice of activation functions is primarily based on empirical evidence from the results in Section 5.1.

For simplicity, in all the following activation functions $z = w \cdot x^T + b$.

Perceptron As mentioned above, the perceptron's activation function is defined by

$$f(z) = \begin{cases} 1, & \text{for } z \geq 0 \\ 0, & \text{for } z < 0 \end{cases},$$

and the derivative of the perceptron's activation function is given by

$$f'(z) = \begin{cases} 0, & \text{for } z \neq 0 \\ \infty, & \text{for } z = 0 \end{cases}.$$

Plots of the perceptron and its activation function can be seen in Figure 2.3, where the cross at $[x, y] = [0, 0]$ represents where the derivative is not defined.

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

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

and the derivative of the sigmoid function is given by

$$f'(z) = \sigma'(z) = \frac{e^{-z}}{(1 + e^{-z})^2}.$$

As previously stated, an intrinsic 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 2.3 shows that this is true for the sigmoid neuron, and not true for the perceptron.

Hyperbolic Tangent The Hyperbolic Tangent (Tanh) activation function is given by

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}},$$

which means that its derivative is given by

$$\begin{aligned} f'(z) = \frac{d}{dz} \tanh z &= \frac{\frac{d}{dz}(e^z - e^{-z})(e^z + e^{-z}) - \frac{d}{dz}(e^z + e^{-z})(e^z - e^{-z})}{(e^z + e^{-z})^2} \\ &= \frac{(e^z + e^{-z})^2 - (e^z - e^{-z})^2}{(e^z + e^{-z})^2} \\ &= 1 - \tanh^2(z) \end{aligned}$$

A problem with the Tanh activation function and the Sigmoid activation function described above is called the *vanishing gradient* problem. The reason for this problem can be illustrated by the plots of Tanh and Sigmoid's gradients. The derivative will be close to zero if the input to these activation functions is large (either positive or negative). This makes it very difficult to use the learning algorithms to learn, and this problem becomes more severe the deeper a neural network is [21].

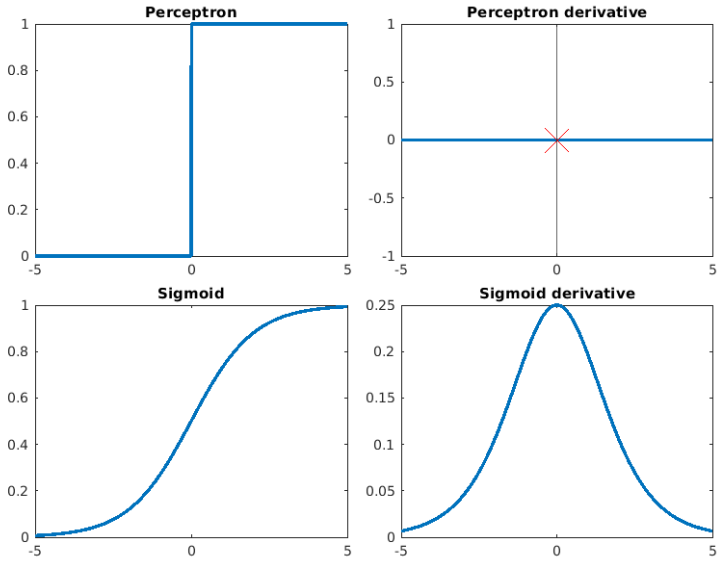


Figure 2.3: Plots of Sigmoid and Perceptron activation functions and their derivatives

Rectified Linear Unit The Rectified Linear Unit (ReLU) activation function has been the most commonly used activation function for deep learning in the last years [22, 23].

ReLU is the same as the input for all positive inputs, and zero for all negative inputs, that means that it is defined as

$$f(z) = \begin{cases} z, & \text{for } z \geq 0 \\ 0, & \text{for } z < 0 \end{cases},$$

or simply

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

This means that the derivative of ReLU is given by

$$f'(z) = \begin{cases} 1, & \text{for } z > 0 \\ \nexists, & \text{for } z = 0 \\ 0, & \text{for } z < 0 \end{cases}.$$

Even though the derivative of ReLU does not exist at $z = 0$ it is common to define either $f'(0) = 1$ or $f'(0) = 0$. It will happen extremely rarely that $z = 0$, which means that doing this is not a problem in practice.

Swish Motivated by finding a better alternative to the ReLU activation function, Ramachandran et al. used a combination of exhaustive and reinforcement learning-based search to find new novel activation functions [23]. The best performing activation function that they discovered, which they named Swish, is given by

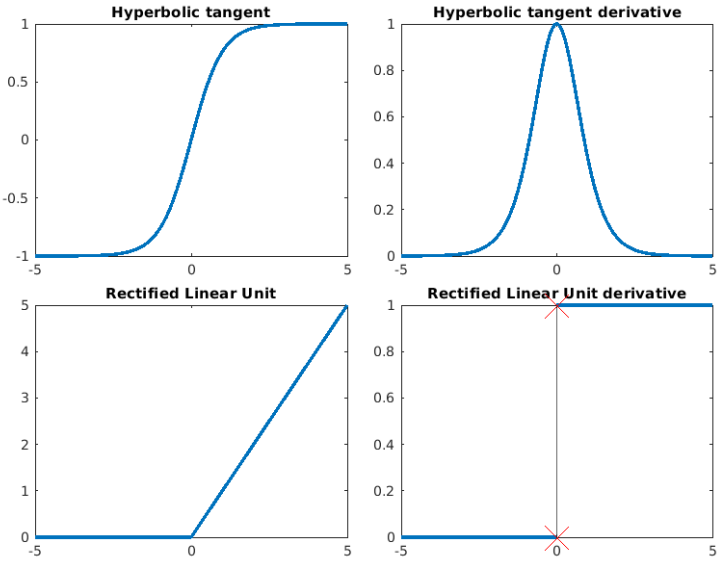


Figure 2.4: Plot of Tanh and ReLU activation functions and their derivatives

$$f(z) = z \cdot \sigma(z),$$

where $\sigma(z)$ is the Sigmoid activation function, which means that

$$f(z) = z \cdot \frac{1}{1 + e^{-z}},$$

and its derivative is given by

$$\begin{aligned} f'(z) &= \sigma(z) + z\sigma'(z) \\ &= \sigma(z) + z\frac{e^{-z}}{(1 + e^{-z})^2} \\ &= \sigma(z) + \sigma(z)f(z)(e^{-z} + 1) - \sigma(z)f(z) \\ &= f(z) + \sigma(z)(1 - f(z)) \end{aligned}$$

Swish, in contrast to ReLU, is smooth and non-monotonic[2]. However, similar to ReLU, Swish is bounded below, and unbounded above [24]. From Ramachandran et al.'s experiments, Swish has better performance than ReLU, and because of Swish's similarity to ReLU, it is possible to directly change the activation function from ReLU to Swish in existing applications.

[2]For a function to be non-monotonic, it has to be increasing on some interval and decreasing on another interval

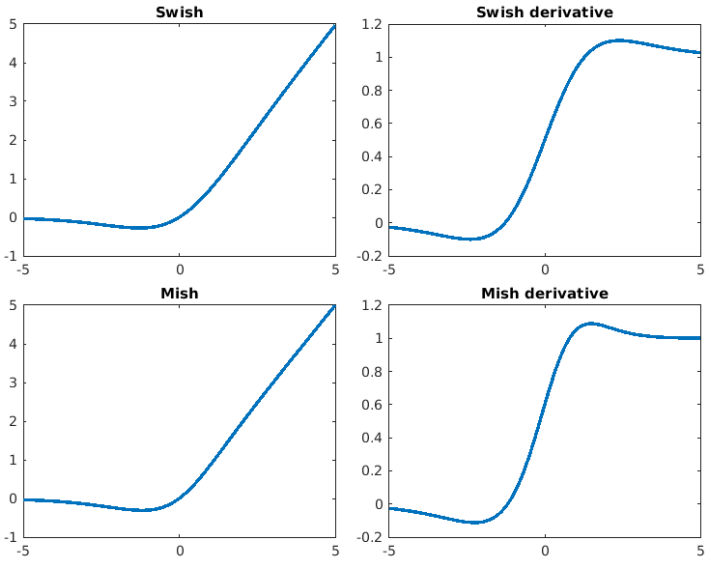


Figure 2.5: Plot of Mish and Swish activation functions and their derivatives

Mish Mish is an activation function that is very similar to Swish. It has many of the same properties as Swish: it is smooth, non-monotonic, bounded below, and unbounded above. D. Misra found that this activation function has better performance than both ReLU and Swish for several tasks [24]. The activation function is given by

$$f(z) = z \cdot \tanh\left(\zeta(z)\right),$$

where $\zeta(z)$ is the softplus activation function,

$$\zeta(z) = \ln\left(1 + e^z\right).$$

The derivative of the Mish function is

$$f'(x) = \frac{e^x(4(x+1) + 4e^{2x} + e^{3x} + e^x(4x+6))}{(2e^x + e^{2x} + 2)^2}.$$

As can be seen in Figure 2.5, both Swish and Mish looks very similar.

Learning algorithms

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

$$C(w,b) = \frac{1}{2n}\sum_x ||y(x) - f(w^T x + b)||^2. \tag{2.1}$$

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 (2.1) 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. However, 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

$$\nabla_{z^L} C = \delta^L = \nabla_{a^L} C \odot f'(z^L), \tag{2.2}$$

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

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot f'(z^l) \tag{2.3}$$

The motivation for finding the errors is to make it easier to change the network's biases and weights 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. By using the error in each layer, these partial derivatives are short and concise:

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l \tag{2.4}$$

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \tag{2.5}$$

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 (2.2), Equation (2.3), Equation (2.4), and Equation (2.5) that was just derived [17, 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 $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ and $\frac{\partial C}{\partial b^l_j} = \delta^l_j$

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 in supervised learning is 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 of the examples are in the training set and the test set. After all the epochs are done, the best practice is 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. In RL, an epoch is done when a pre-specified number of training episodes has been completed. It is common, especially in DRL, to do some test episodes at the end of each epoch to track the training progress.

Regularization

Overfitting is a problem with neural networks that often occurs if steps are not taken to avoid it. Overfitting means that the neural network no longer generalizes to unseen data. This problem can come from training too much on the available training data. When a neural network becomes overfitted, it gets better at recognizing the training data, but worse at recognizing data not seen during training. This is a very undesirable situation. For example, consider a neural network that is going to be used for image recognition. The point of this is not to be able to recognize images that have already been labeled but to recognize new images that have not yet been categorized. A way to reduce overfitting is to increase the available training data. This can make it more difficult for the neural network to memorize the features of the specific images, and instead make it learn features that are general for all images of the categories that should be classified. A second way to avoid overfitting is to reduce the complexity of the neural network (for instance, depth and width). A smaller neural network is less likely to overfit to training data because it will not have the capability to memorize each image. A third 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 [17, ch. 3].

2.1.2 Reinforcement learning

Reinforcement Learning (RL) is the variant of machine learning that is arguably the most similar to how humans learn. Consider a child that learns not to hot things because the child gets negative feedback in the form of pain for doing this action. Similarly, RL employs positive and negative feedback in the form of a scalar reward signal to learn how to perform various tasks. This form of feedback is often called *reinforcement* or *reward*. In RL, a multitude of algorithms and methods can be used to try to maximize the reward. Before going into the specifics of how some of these algorithms and methods work, an overview of the terms used in this section is given. This overview is inspired by [25, pp.10-13].

- Environment
  - In the context of RL, the environment defines the task that is going to be solved.

- Agent
  - An agent is an entity that can observe the environment (using sensors in the real world) and act on the environment (using actuators in the real world). In RL, the objective is to create an *intelligent agent* that can discover a way to maximize the reward it receives from its environment.
- State
  - A state, denoted by s, is a "unique characterization of all that is important in a state of the problem that is being modelled" [25, p.10]. The set of all possible states is called the state space, $\mathcal{S}$. The state space is also some times called the observation space.
- Action
  - Actions, denoted by a, are how the agent can influence the state of the environment. The set of all actions is called the action space, and is denoted by $\mathcal{A}$. The set of actions that can be performed in a given state $s \in \mathcal{S}$ is denoted by $A(s) \subseteq \mathcal{A}$. $A(s)$ is a proper subset of $\mathcal{A}$, because for some environments, there may be actions that cannot be performed in some states.
- Transition model
  - A transition model, $T : S \times A \times S \rightarrow [0, 1]$, is a probability distribution over the possible next states, given the previous states of the environment and the action that is done in the current state. A Markovian transition model assumes that the next state only depends on the current state and the action done in the current state, that is $T(s, a, s') = P(s'|s, a)$. A transition model is a proper probability distribution over the possible next states. This means that

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

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

- Reward function
  - A reward function is a function that determines how rewards should be given. A reward is a scalar number that ideally gives a performance measurement of how well the agent is performing. The reward function can be dependent on state $R(s_t) = r_t$, the state and action $R(s_t, a_t) = r_t$ or the transition, $R(s_{t+1}, s_t, a_t) = r_t$.
- Policy
  - The policy of an agent is what determines what action the agent should take based on the state of the environment. A policy is denoted by π, and can be either deterministic or stochastic. In the case of a deterministic policy, the policy is a direct mapping from state to action $\pi(s_t) = a_t$; in the case of a stochastic policy, the policy is a probability distribution over all actions available in a given state, $\pi(s_t, a_t) = P(a_t|s_t)$

- Off-policy and on-policy algorithms
  - In RL, the term off-policy is used to describe an algorithm that uses a different policy to act in the environment (and generate data) than the one that it tries to improve. An on-policy algorithm uses the same policy for both of these [16, p.82].
- The RL process
  - 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 2.6.

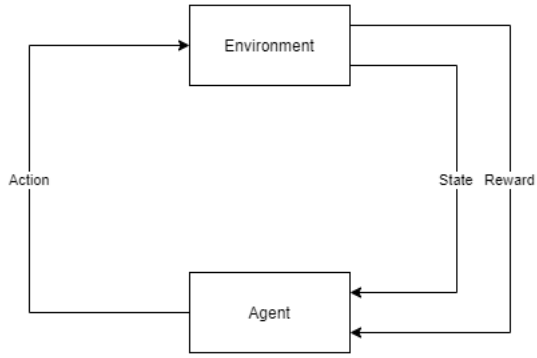


Figure 2.6: Illustration of the RL process

Markov Decision Process

This section and all the sections up until, but not including, Policy-based, value-based, and Actor-Critic algorithms, take inspiration from the book *Reinforcement Learning: An Introduction* [16].

RL concerns solving sequential decision processes in a way that maximizes the reward. To create a framework for doing this, the problem is generally modeled as a Markov Decision Process (MDP). An MDP is defined as a tuple $< \mathcal{S}, \mathcal{A}, T, R >$. The entries of this tuple are: a state space, an action space, a Markovian transition model, and a reward function (each of these is described above) [25, p.12]. The transition model and the reward function are together, usually called the *model* of the MDP. In RL, the model is often not known. This leads to some significant distinctions in the different frameworks used to solve MDPs. When the MDP's model is known, the problem can be solved using methods from Dynamic Programming (DP), which is beyond the scope of this thesis. When the MDP's model is not known, the problem can be solved using RL. From this point, there are two ways to solve the problem using RL. One way is to make an algorithm that can try to learn the model of the MDP, and then learn a policy using this learned model. This is called *model-based* or *indirect* RL and is also beyond the scope of this thesis. The other

way to solve a problem where the model is unknown using RL is by using methods from *model-free* RL, which is what is done in this thesis.

To maximize the reward, the goal is for the agent to find the optimal policy π^*, which is the policy that maximizes the expected reward. There are several ways to define what it means to maximize the expected reward; the following are three ways to define this [25, pp.13-15].

- The finite horizon model
  - $E[\sum_{t=0}^{h} r_t]$, where the notation $E[\dots]$ 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.

Value functions and Bellman Equations

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:

$$V^\pi(s) = E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t = s\right\}, \tag{2.6}$$

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 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 (2.6).

$$\begin{aligned}
V^{\pi}(s) &= E_{\pi}\left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t = s \right\} \\
&= E_{\pi}\left\{ r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{\infty} r_{t+\infty} | s_t = s \right\} \\
&= E_{\pi}\left\{ r_t + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s \right\} \\
&= E_{\pi}\left\{ r_t + \gamma V^{\pi}(s_{t+1}) | s_t = s \right\} \qquad (2.7) \\
&= \sum_{s'} T(s, \pi(s), s')(R(s, \pi(s), s') + \gamma V^{\pi}(s')), \text{ for } s = s_t,\ s' = s_{t+1} \qquad (2.8)
\end{aligned}$$

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:

$$V^{\pi *}(s) \geq V^{\pi}(s), \text{ for all } s \in \mathcal{S} \text{ and all policies } \pi. \qquad (2.9)$$

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 approach is used, Q-functions are often used instead of value functions. It frequently happens that the transition model is not known in RL, which means that Q-functions are useful and necessary in many situations. The relationship between the optimal value function and the optimal Q-function is as follows

$$Q^*(s, a) = \sum_{s' \in S} T(s, a, s')\Big(R(s, a, s') + \gamma V^*(s') \Big) \qquad (2.10)$$

$$V^*(s) = \max_a Q^*(s, a) \qquad (2.11)$$

Exploration and Exploitation

The probably most important element of RL is the agent's ability to discover new strategies. To do this, the agent cannot only try to *exploit* the task by performing the actions that are currently considered to be best; to discover new strategies, the agent has to *explore* the environment by trying new strategies. The exploration-exploitation trade-off is about balance. If the agent explores too much, the knowledge obtained about the environment will not be sufficiently used. If the agent explores too little, it will take a long time to find new strategies.

The exploration strategies used in this thesis are:

- ϵ-greedy exploration
  - For this strategy, the agent does the greedy action (exploiting by doing what it considers to be the best action) with probability ϵ, and it does a random action with probability $1 - \epsilon$. Here $\epsilon \in [0, 1]$ is a hyperparameter (which can change over time if the user designs it so). By setting $\epsilon = 1$, the agent will always select the greedy action; by setting $\epsilon = 0$, the agent will always select random actions. This is a simple way to explore, and it often yields good results. When evaluating the agent, the agent will select the greedy action.
- Exploration noise
  - By adding some form of noise to the greedy action selected by the agent, the agent can discover new strategies while also using the knowledge that it has learned about the environment. If a is the greedy action selected by the agent, and $\mathcal{N}$ is the noise process, then $\tilde{a} = a + \mathcal{N}$ is the action that is performed in the environment for exploration purposes. This type of exploration is only suitable when the action space is continuous, which it generally is in robotics problems.

Q-learning algorithm

For model-free RL approaches, such as those used in this thesis, the model (transition model and reward function) of the MDP cannot appear in the algorithms, since this is unknown. One of the most fundamental traditional RL algorithms is the Q-learning algorithm. This is a tabular method, which means that the Q-values learned by the algorithm are inserted into a table of size $|\mathcal{S}| \times |\mathcal{A}|$, where $|\mathcal{S}|$ and $|\mathcal{A}|$ is the size of the state and action space, respectively. Q-learning is also a *temporal difference* algorithm, which means that it learns estimates of values (for instance, value functions or Q-functions) based on other estimates. Temporal difference methods are often used in RL since they do not require the model of the MDP. The Q-learning algorithm can be seen in Algorithm 1, where the update rule is

$$Q(s, a) := Q(s, a) + \alpha\left(r + \gamma \max_{a' \in A(s')} Q(s', a') - Q(s, a)\right).$$

Deriving this update rule entirely is beyond the scope of this thesis, but a partial derivation follows. For temporal methods, the estimates are usually made in an online fashion.

This means that the estimates are calculated incrementally during the RL operation. Consider Equation (2.7). From this equation, it can be seen that an estimate for $V(s_t)$ is $r_t + \gamma V(s_{t+1})$, given that r_t is the reward received by going from s_t to s_{t+1} by doing action a. The goal is to calculate the optimal value function V^*, which means that the error between the optimal policy and the current policy at a given state is $V^*(s) - V(s)$. If the optimal policy was known, it is possible to change the current policy to the optimal policy incrementally using gradient ascent (gradient ascent since the optimal value function maximizes the value for all states according to Equation (2.9))

$$V(s) := V(s) + \alpha(V^*(s) - V(s)).$$

However, since the optimal policy is not known, this is not possible. Instead, an estimate of the optimal policy is used. An estimate of the optimal policy that is available is

$$V^*(s_t) \sim r_t + \gamma V^*(s_{t+1}) \sim r_t + \gamma V(s_{t+1}).$$

This is a poor estimate since there is no guarantee that $V(s_{t+1})$ is a good approximation of $V^*(s_{t+1})$, but it can be shown that as the update rule is applied repeatedly to the value function, the optimal value function can be reached. This means that the update rule is now

$$V(s) := V(s) + \alpha(r_t + V(s_{t+1}) - V(s)), \tag{2.12}$$

which is the update rule for another temporal difference algorithm called TD(0). A similar derivation can now be done, where instead of using value functions, Q-functions are used. Applying gradient ascent as if the optimal Q-function was known, gives the update rule

$$Q(s,a) := Q(s,a) + \alpha(Q^*(s,a) - Q(s,a)).$$

Equation (2.10) and Equation (2.11) can be combined to give

$$\begin{aligned} Q^*(s,a) &= \sum_{s' \in S} T(s,a,s') \Big(R(s,a,s') + \gamma \max_{a' \in A(s')} Q^*(s',a') \Big) \\ &= E\Big\{ r_t + \gamma \max_{a' \in A(s')} Q^*(s',a') \Big\} \end{aligned}$$

Which means that

$$Q^*(s,a) \sim r_t + \gamma \max_{a' \in A(s')} Q^*(s',a') \sim r_t + \gamma \max_{a' \in A(s')} Q(s',a').$$

With $r_t + \gamma \max_{a' \in A(s')} Q(s',a')$ used as an estimate for $Q^*(s,a)$, inserted into the update rule above gives the update rule for the Q-learning algorithm

$$Q(s,a) := Q(s,a) + \alpha\Big(r + \gamma \max_{a' \in A(s')} Q(s',a') - Q(s,a) \Big). \tag{2.13}$$

It can be shown that with a learning rate $\alpha \in (0,1)$ the update rules Equation (2.12) and Equation (2.13) will converge at the optimal value function and Q-function [26].

The Q-learning algorithm follows the general RL operation structure, as illustrated in Figure 2.6. The agent receives a state, chooses an action based on this state, and performs it. After that, it observes a new state and reward, which it then uses to update the Q-value associated with the old state and the action it took. After that, the new state is used as the current state, and the process repeats.

Algorithm 1 Q-learning, taken from [25, p.31]

Require: discount factor γ, learning rate parameter α
  initialize Q arbitrarily (e.g. $Q(s,a) = 0, \forall s \in S, \forall a \in A$
  for all episode **do**
    s is initialized as the *starting* state
    repeat
      choose an action $a \in A(s)$ based on Q and an exploration strategy
      perform action a
      observe the new state s' and received reward r
      $Q(s,a) := Q(s,a) + \alpha\left(r + \gamma \max_{a' \in A(s')} Q(s',a') - Q(s,a)\right)$
      $s := s'$
    until s' is a goal state
  end for

Policy-based, value-based, and Actor-Critic algorithms

For simplicity, the term "value function" refers to both Q-functions or standard value functions in this section. This is because an actor-critic method or critic method can both estimate the Q-function and the value function.

The Q-learning algorithm just described is what is called a value-based, or critic-only, algorithm. This means that it tries to calculate a value either for each state (a value function) or for each state-action pair (a Q-function). The policy of a value-based algorithm is then generated based on maximizing the expected value. On the other side of the spectrum are policy-based, or actor-only, algorithms. These algorithms do not calculate a value function at all. Instead, they directly estimate the policy that the algorithm is going to follow. These two ways to do RL have different advantages and disadvantages:

- Value-based algorithms
  - These algorithms can be seen as indirect since they do not optimize directly in the policy space. They are often more stable during learning than policy-based algorithms. Value-based algorithms may approximate the value function well, but they lack any specific guarantees that the policy will be close to optimal [27].
- Policy-based algorithms
  - These algorithms estimate the gradient of the performance directly in the policy space. One of the disadvantages of these methods is that the gradient estimators may have significant variance [27].

In an attempt to combine the best of both worlds, there also exists algorithms that calculate a value function and/or a Q-function, in addition to also estimating the policy that the algorithm is going to follow. These algorithms are called actor-critic algorithms. The actor part of the algorithm is the policy, and the critic part is the value function. The value function estimated by the critic is used to update the parameters of the actor. Actor-critic methods can have better convergence properties compared to critic methods, and they can also have faster convergence compared to actor methods, because of lower variance [27].

2.1.3 Deep reinforcement learning

The curse of dimensionality posed by Bellman [7] has been a long-standing and unsolved challenge in RL. As the number of states increases, the amount of computations needed to explore the state-space exponentially increases. For example, consider the state space in some of the experiments done in Chapter 5. There are four states for the joint positions, four states for the joint velocities, and three states for the goal. This means that there is an 11-dimensional state space. If this state space is then discretized into ten levels (which probably will also result in a too coarse discretization), this means that there are 10^{11}, or in other words, one hundred billion different states in the problem. This amount of states would take an enormous amount of computations, memory, and time to explore.
In 2013, the first RL method to successfully learn control policies directly from a high-dimensional sensory input was created by DeepMind [8]. This method is called Deep Q-Network (DQN) and is based on the Q-learning algorithm shown in Algorithm 1. The DQN algorithm can be seen in Algorithm 2. In this method, RL is combined with ANNs, to create Deep Reinforcement Learning (DRL). The fundamental idea behind DQN is that a nonlinear function approximator can be used to map state and action to a value. As discussed in Section 2.1.1, neural networks can be used for this. In addition to using neural networks, two other elements enabled the success of DQN. These two elements are the *replay buffer* and a separate *target network* to stabilize training.

Replay buffer/Experience replay One of the problems that DRL faces is that most optimization algorithms assume that all samples are independent of each other. In this case, a sample is a transition. 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 why the replay buffer is used. 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. It is worth noting that replay buffers can only be used by off-policy algorithms [8].

Target network The reason for using a target network is to lessen the correlation with the target [4]. A target that changes with every timestep makes training difficult because this becomes similar to trying to catch up to a moving target [8]. The next Q-value in the

Algorithm 2 deep Q-learning with experience replay, taken from [4]

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocesssed sequence $\phi = \phi(s_1)$
  for t=1, T **do**
    With probability ϵ select random action a_t
    otherwise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta))$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a', \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  end for
end for

Q-learning algorithm, which can be seen in Algorithm 1 is calculated by

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

DQN uses a similar cost function

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

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. When initializing, the target network has the same parameters as the Q-network. During operation, the target networks copy the parameters of the Q-network every C timestep.

The replay buffer and the target network's value can be seen in the paper *Human-level control through deep reinforcement learning*, where a comparison of DQN with and without replay buffer and target network is shown [4, p.12]. This table is copied and shown in Table 2.1.

Deep Deterministic Policy Gradient

The inspiration to the Deep Deterministic Policy Gradient (DDPG) algorithm comes from the success of the DQN algorithm. Where DQN can only be used in MDPs with discrete

Game	With replay, with target Q	With replay, without target Q	Without replay, with target Q	Without replay, without target Q
Breakout	316.8	240.7	10.2	3.2
Enduro	1006.3	831.4	141.9	29.1
Rivere Raid	7446.6	4102.8	2867.7	1453.0
Seaquest	2894.4	822.6	1003.0	275.8
Space Invaders	1088.9	826.3	373.2	302.0

Table 2.1: The effects of replay and separating the target Q-network (higher numbers are better in this case) [4]

action spaces, DDPG can be used in MDPs with continuous action spaces. DQN does, besides, work best with low-dimensional action spaces, whereas DDPG can be used in higher-dimensional action spaces. Most real-world applications will have a continuous, possibly high-dimensional, action space, so this provides a significant incentive for using DDPG [9].
DDPG is a model-free off-policy actor-critic algorithm. Since DDPG is an actor-critic algorithm, it learns a policy in addition to a Q-function. The policy is the actor part of the algorithm, and the Q-function is the critic part. The main elements from DQN that are included in DDPG are the replay buffer and separate target networks. DDPG uses two target networks, a Q-value target network, and a policy target network. These target networks slowly track the critic and actor networks by

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

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

where $\tau \in \mathbb{R} : \tau \in (0,1), \tau \ll 1$.

Because DDPG uses a deterministic policy, a method has to be employed to make the agent explore. To do this, DDPG uses exploration noise, as described in the section on exploration and exploitation above.

The DDPG algorithm is the algorithm that is used in all the DRL experiments of this thesis.

2.1.4 Reward shaping

One of the most challenging parts of doing RL is the task of goal specification. This mainly comes down to specifying the reward function. As previously stated, an RL agent must receive feedback on its performance to learn how to do a task. However, in designing a reward function, one has to walk a thin line: give rewards too frequent, and one is essentially pre-defining the agent's behavior (which is precisely what aims to be avoided by using RL); give rewards too infrequently, and the agent may never get a reward and may never learn (or at least learn very slowly). The problem of designing an excellent way to give rewards such that the RL agent can accomplish its goal is called the curse of goal

Algorithm 3 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

specification [28]. A proper reward function can make a huge difference in whether an agent can solve an RL task or not. It also can significantly impact the time it takes to solve a task.

Hindsight Experience Replay

Traditionally, designing a useful reward function requires a great deal of domain knowledge. In some cases, it may not be known what kind of behavior is necessary to accomplish the goal, making the design of reward functions very difficult. In addition to this, reward functions can often be quite complex. This means that the reward function becomes another element that needs to be tuned, making the development of RL applications even more difficult. The simplest way to design a reward function would be to use a binary reward, which states whether the agent has achieved the goal. However, as stated above, sparse rewards may lead to an agent learning exceptionally slowly or not at all. A binary reward that is only given when the agent has accomplished its goal is likely the most sparse reward possible.

Recently, a technique has been discovered which enables DRL agents to learn from sparse rewards, even rewards as sparse as the binary rewards just described. This technique is called Hindsight Experience Replay (HER) and was created by Andrychowicz et al. [11]. The principle behind HER is that even if the desired goal was not reached, the state that instead was achieved can also, in a way, be considered as a goal. This is called a substituted/achieved goal. For example, assume that an agent has done an episode and achieved the state sequence $s_1, \ldots, s_T$, but the goal state $g = s_g$ was not in this state sequence. This means that, with sparse rewards, the agent has achieved the same reward for the entire episode. Even though the agent did not discover how to achieve the state s_g, it has discovered how to reach every other state in the state sequence. Using HER, the states in the state sequence can then be used as substituted goals, and the agent can learn from these [11]. When storing transitions in the HER buffer, the transitions are stored with the original goal, but also a subset of substituted goals. How to choose the substituted goals is then a decision that has to be made when using HER. Andrychowicz et al. describe four different strategies for selecting substituted goals:

- final
  - Replay with k substituted goals, which come from the same episode as the transition 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 drawn from all states encountered during the whole training procedure.

This means that for HER, this *replay strategy* is a new hyperparameter that has to be decided on before the training starts. The number of substituted goals, k, is another new hyperparameter that also has to be defined when using HER. The HER algorithm can be seen in Algorithm 4.

By using HER, the process of goal specification is hugely simplified, and all results from this thesis are achieved by only using sparse rewards. The reward function for all experiments done is

$$r = \begin{cases} 0, & \text{if goal is not reached} \\ -1, & \text{if goal is reached (within a small error tolerance)} \end{cases}.$$

Algorithm 4 HER algorithm, taken from [11]

Require: an off-policy RL algorithm $\mathbb{A}$ ▷ e.g. DQN, DDPG, NAF, SDQN
Require: a strategy $\mathbb{S}$ for sampling goals for replay ▷ $\mathbb{S}(s_0, \ldots, s_T) = m(s_T)$
Require: a reward function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{G} \rightarrow \mathbb{R}$ ▷ e.g. $r(s, a, g) = -[f_g(s) = 0]$
  Initialize $\mathbb{A}$ ▷ e.g. initialize neural networks
  Initialize replay buffer R
  for episode = 1, M **do**
    Sample a goal g and an initial state s_0
    for t=0, T-1 **do**
      Sample an action a_t using the behavioral policy from $\mathbb{A}$:
        $a_t \leftarrow \pi_b(s_t||g)$ ▷ $||$ denotes concatenation
      Execute the action a_t and observe a new state s_{t+1}
    end for
    for $t = 0, T-1$ **do**
      $r_t := r(s_t, a_t, g')$
      Store the transition $(s_t||g', a_t, r_t, s_{t+1}||g')$ in R ▷ standard experience replay
      Sample a set of additional goals for replay $G := \mathbb{S}(\textbf{current episode})$
      for $g' \in G$ **do**
        $r' := r(s_t, a_t, g')$
        Store the transition $(s_t||g', a_t, r', s_{t+1}||g')$
      end for
    end for
    for $t = 1, N$ **do**
      Sample a minibatch B from the replay buffer R
      Perform one step of optimization using $\mathbb{A}$ and minibatch B
    end for
  end for

2.1.5 Sim-to-real transfer

The choice of which domain to train agents in is an essential decision within robotic reinforcement learning. Currently, there does not exist an ideal option for this, as all alternatives have advantages and disadvantages. One approach that can be chosen is to train the robot in the real world, which is the domain where it is inevitably going to end up. A significant advantage of training the agent on the real robot is that nothing needs to be modeled. This means that there are no modeling errors, and the performance achieved during training will most likely perfectly transfer to its operation later. There are also many problems with training on the real robot because this approach suffers from the curse of real-world samples [28]. Some of the main problems of training on the real robot are [28, pp. 1250-1251]:

- Degradation of the robot due to overuse can be very costly and may lead to needing to replace the robot altogether.
- At the beginning of the training, when the ANN is completely randomized, there is no way to know what kind of motions the robot may perform. The robot may damage people, expensive equipment in the area around it or itself.
- Training in the real world cannot be sped up, so if one trains the agent in the working environment, the algorithm used needs to be 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 robot itself or its environment. 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. Nevertheless, there are many difficulties in training the agent in a simulator. This is mainly because of the *reality gap* between the simulator and the real-world. A 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.

Transfer learning

When two different tasks have similarities, it may be that behavior learned in one of the tasks can be useful for the other task. This is where transfer learning comes in. If there is a deep learning agent trained on one task, it is possible to transfer the resulting neural network to the other task, and then continue training. When transfer learning, it is common not to update the parameters of the earliest layers in the neural network, and instead only update the layers that are closer to the neural network output. The motivation behind this is that the earlier layers can act as a form of feature extractor, while the latter can put these features together into an output. This also has the advantage of faster training time. This is because the gradient for the earlier layers will not need to be calculated, which can save a significant amount of time. For an illustration of this, see Figure 2.7, where the parameters of all layers except the output layer are frozen during the transfer learning phase.

Another way to do transfer learning is by using a differential learning rate. When doing this, the first layers are not frozen, but instead, use an increasingly lower learning rate when going from the output layer to the input layer. For an illustration of this, see Figure 2.8 where ($a > b > c > d > e$).

One final way to do transfer learning is simply to update the parameters of all the layers as it is done during regular training, with the same learning rate for all layers. Note that in this case, it could be beneficial to use a lower learning rate than during regular training. This is because the model has already been trained on a task, and the assumption is that only small adjustments are needed to make it have success on a new task.

One way to improve the success of sim-to-real transfer is to use transfer learning. By, for example, first training the agent in a simulator, and then transfer learning in the real world. This can lessen some of the disadvantages by training on a real robot that are described above. Since the agent has already learned about the task in the simulator, the agent will most likely learn faster, which will lead to less degradation on the robot. It is also possible to have a better intuition of how the agent will behave when transfer learning from simulator to the real world, which can lead to less chance of the robot damaging itself and its environment.
In some cases, training at all in the real world may be unwanted. Transfer learning can still help in this case. It is possible first to train the agent in a simulator that is fast but may be inaccurate compared to the real world robot and environment. After that, transfer learning can be done in a simulator that is slower, but more similar to the real world. This can give satisfactory results, and this is what is done in this thesis in later chapters.

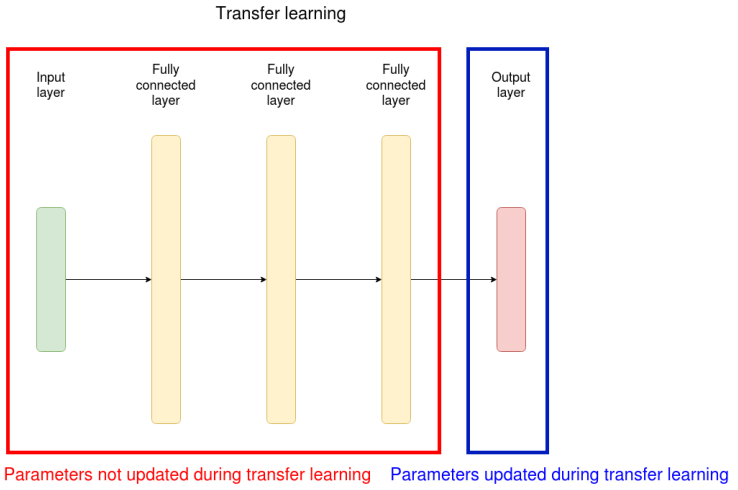


Figure 2.7: Example of transfer learning where all layers except the last layer are frozen during training

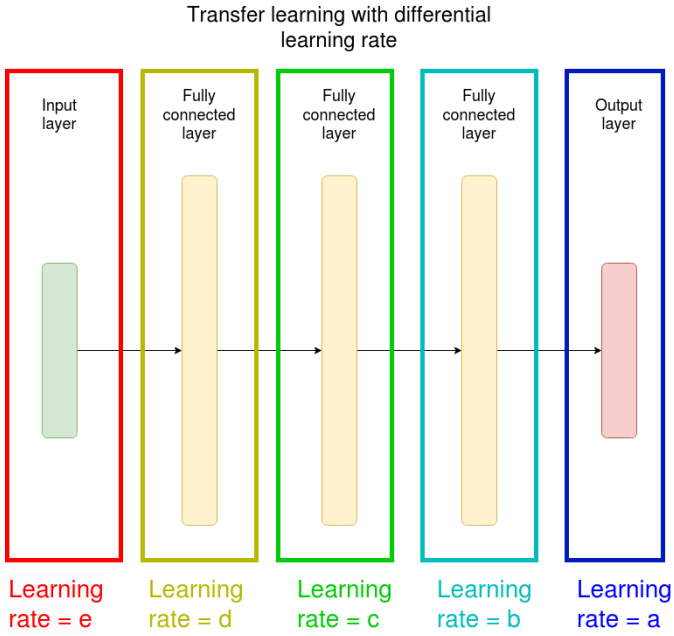


Figure 2.8: Example of transfer with differential learning rate

2.1.6 Explainable Artificial Intelligence

One of the significant disadvantages of state-of-the-art Deep Learning (DL) and DRL methods is the lack of interpretability of the results, for instance, which input features or combinations of input features are important for determining the decision of the neural network, and which features are not important. This is an essential question, but it is challenging to answer. A neural network may have an enormous amount of neurons and connections (vertices) between neurons, which can make it unfeasible for a human to determine how different input features are combined into the output. For an example of the enormous amount of connections in a neural network, consider one of the neural networks used in this thesis. This neural network is the DDPG actor-network in Figure 4.1, which has 11 input neurons, three hidden layers with 256 neurons in each, and four output neurons. This means, for instance, that the input neurons have $11 \times 256 = 2816$ connections to the first hidden layer. The total number of connections is then

$$11 \times 256 + 256 \times 256 + 256 \times 256 + 256 \times 4 = 134912.$$

This means that even a relatively shallow and narrow neural network such as this has over a hundred thousand connections. This number of connections is unfeasible for a human to consider.

XAI has become an increasingly popular research topic in just the last few years, since

the success of DL. For a picture of this, look at Figure 2.9, which was taken from [5]. Without the ability to explain in some way the decisions that an Artificial Intelligence (AI) agent makes, the application of AI in many fields, especially safety-critical ones, can be considered reckless.

Shapley Additive Explanations

This section is inspired by the paper *A Unified Approach to Interpreting Model Predictions* [29].

SHapley Additive exPlanations (SHAP) is a framework that can help identify how the features that are used as input to a neural network contribute to the neural network output. As stated above, a neural network is too complex to be understood by humans. This leads to having to use a model that is easier to understand, which serves as an approximation of the neural network. One such class of models is called the *additive feature attribution methods*. A definition that holds for these methods is first given.

The original prediction model is denoted by f (in the case of this thesis, this will be a neural network), and the explanation model is denoted by g. Local methods are designed to explain a prediction $f(x)$ from a single input x, and the focus is on local methods. It is common for explanation models to use simplified inputs, denoted by x'. These simplified inputs map to the original inputs using a mapping function $x = h_x(x')$. Local methods try to make $g(z') \approx f(h_x(z'))$ given that $z' \approx x'$. Additive feature attribution methods use an explanation model that is a linear combination of binary variables, and following is the definition of additive feature attribution methods:

$$g(z') = \phi_0 + \sum_{i=1}^{M} \phi_i z'_i, \tag{2.14}$$

where z' is a vector of binary variables with size M, M is the number of simplified input features and $\phi_i \in \mathbb{R}$ is an effect that is assigned to each feature by the method.

A single unique solution for the class of additive feature attribution models has three desirable properties:

Feature 1, Local accuracy:

$$f(x) = g(x') = \phi_0 + \sum_{i+1}^{M} \phi_i x'_i \tag{2.15}$$

Feature 2, Missingness:

$$x'_i = 0 \rightarrow \phi_i = 0 \tag{2.16}$$

Feature 3, Consistency:

$$\begin{aligned} &\text{if } f'(h_x(z')) - f'(h_x(z'_i = 0)) \geq f(h_x(z')) - f(h_x(z'_i = 0)), \forall z' \in \{0,1\}^M \\ &\text{then } \phi_i(f', x) \geq \phi_i(f, x) \end{aligned} \tag{2.17}$$

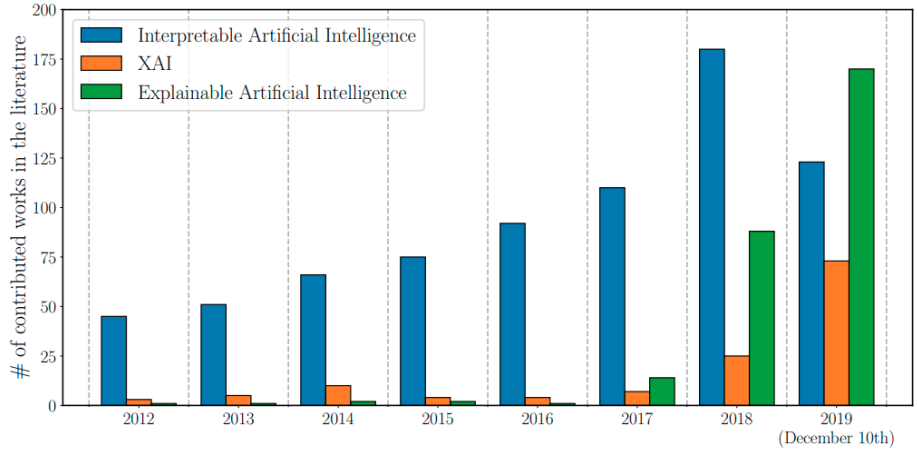


Figure 2.9: Evolution of the number of total publications whose title, abstract and/or keywords refer to the field of XAI during the last years. Data retrieved from ScopusR (December 10th, 2019) by using the search terms indicated in the legend when querying this database. It is interesting to note the latent need for interpretable AI models over time (which conforms to intuition, as interpretability is a requirement in many scenarios), yet it has not been until 2017 when the interest in techniques to explain AI models has permeated throughout the research community. This figure is taken from [5].

For any two models f and f'.

Only one explanation model g can follow Equation (2.14) and satisfy the properties stated in Equation (2.15), Equation (2.16) and Equation (2.17) is given by:

$$\phi_i(f,x) = \sum_{z' \subseteq x'} \frac{|z'|!(M-|z'|-1)!}{M!}[f(h_x(z')) - f(h_x(z'_i = 0))], \tag{2.18}$$

where "$|z'|$ is the number of non-zero entries in z' and $z' \subseteq x'$ represents all z' vectors where the non-zero entries are a subset of the non-zero entries in x'" [29]. The SHAP values are then the solution to Equation (2.18). Each SHAP value indicates how much a given state contributes to the magnitude of a given output according to the explanation model. For example, if there are 20 inputs and ten outputs, the number of SHAP values will be $20 \times 10 = 200$.

2.2 Robotic manipulators

This section is inspired by the book *Robot modeling and control* [30].
Robotic reinforcement learning treats the robotic manipulator as a black box. This means that the only relevant information is how the input (actions) affect the output (the next state). Therefore, the information that traditionally is used to create a controller for a manipulator, the dynamics of the manipulator, for instance, is not relevant. Nevertheless,

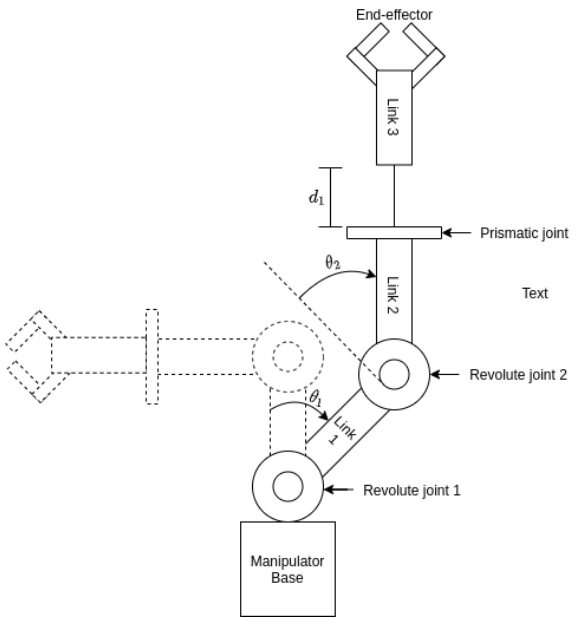


Figure 2.10: Illustration of a Revolute-Revolute-Prismatic (RRP) manipulator

some terminology on robotic manipulators is useful for describing the objectives of different tasks and describing how the agent behaves when controlling the manipulator.

There are two main categories of robotic manipulators, open-chain manipulators, and closed-chain manipulators. Within these two categories, there also exist sub-categories. The manipulator used in this thesis is an open-chain manipulator, and therefore only these are discussed in this section. An open-chain manipulator has one fixed end and one moving end. Between these two ends, there are links and joints. The links are rigid, and the joints are movable. Usually, the links and joints are alternating, as can be seen in Figure 2.10. Following is an overview of some useful concepts that concern robotic manipulators:

- The end-effector of a manipulator
  - The term "end-effector" refers to the tool at the end of the manipulator's arm. The end-effector can, for instance, be a gripper, a suction cup, or a welding tool.
- Joints
  - There are mainly two types of joints that exist for robotic manipulators, revolute joints and prismatic joints. A revolute joint function acts as a hinge and allows relative rotation between two links. A prismatic joint allows relative linear motion between two links [30]. The current value of a specific joint is called a *joint variable*. The notation of a joint variable is often given by q_i for an arbitrary joint and θ_i or d_i if the joint is specified as revolute or prismatic. In

this notation, i is the number of the joint; for example, q_1 will refer to the first joint, and θ_3 will refer to the third revolute joint. There also exist other types of joints, such as spherical joints, but these other types can be described by using multiple revolute and/or prismatic joints (for instance, a spherical joint can be modeled by using three revolute joints). An open-chain manipulator is often described by the first three joints. The number of joints determines the Degrees of Freedom (DOF) of the manipulator.

- Joint/configuration space
  - The joint space is the set of all values the joints of a manipulator can have.
- 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 Cartesian 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 joint variables' values. 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 most straightforward 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 describes 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 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 a meta-operating system that dramatically simplifies the usage of robots in various applications. It is a collection of tools, libraries, and conventions that simplifies the development of robots across various robotic platforms [31]. There exist several packages that can be used with ROS. These packages cover everything from sensor interface to computer vision to robot control. A ROS node is an executable that can use ROS to communicate with other nodes [32]. The code for ROS can be written in both C++ and Python. ROS uses three types of communication between nodes: topics, services, and actions. In this thesis, only topics and services are used, so actions will not be discussed. The following information is inspired by the ROS concept site [32].

Topics ROS topics use asynchronous message passing. Nodes can either publish or subscribe to a topic. When a node publishes to a topic, it sends messages to the topic, without waiting for any reply. When a node subscribes to a topic, it reads the messages that have been published to the topic. The nodes that subscribe or publish to a topic do not know about each other, and the only thing they consider is either sending or reading the messages.
An example of this is a node that has some sensor and another node that needs the readings from this sensor. The first node can then publish a message to a topic with information about the state of the sensor whenever this information is ready, or at a pre-specified time interval. When the other node sees that a new message has arrived at the topic, it can use this information in its operation. Because any node can send messages to any topic, and any node can read messages from any topic, topics can be thought of as a many to many type of communication [31].

Services Where topics use asynchronous message passing, ROS services use synchronous message passing. Each service is defined by a pair of messages. These messages are called request and reply messages, and their structure is pre-defined. Services work in a client-server fashion. A server can initiate a service at the start of the operation. A client can then send a request message to the server, and after it has sent the request message, it waits for a reply message from the server.
In contrast to publishing and subscribing to topics, nodes that use services have to know about each other and have to have a connection made between each other. For example, consider one node that has direct control over a robot manipulator (for instance, that it can change the set point of the controller of the joints), and another node that has some sequence of joint states that together form a path which it wants the manipulator to follow. The first node can initiate a service at the start of its operation, then wait for messages telling how to change the joints' set points. The client can then send a request message to the service for the joint position that it wants. When the server receives this message, it can change the set point of the controller, and then send a reply messages back to the client which either says that the set point has been successfully changed, or that it failed to make the request. Because there needs to be a connection between two nodes, a server and a client, to use services, services can be thought of as a one to one type of communication. It is also possible to create a persistent connection to a service. This can make the message passing faster since the client does not need to connect to the server each time it needs to send a request message. Persistent connections are although less robust (since there is no built-in detection of whether a server has stopped working), and a mechanism that checks whether the server is still up, needs to be implemented to increase the robustness.

3.1.2 PyTorch

PyTorch is a Python library for deep learning. It supports features such as automatic back-propagation, Graphics Processing Unit (GPU) for parallel computing, and uncomplicated setup of most common types of neural networks. PyTorch's ability to support dynamic computation graphs makes it convenient to use compared to other deep learning frameworks such as Tensorflow, which uses static computation graphs. This can make debugging much easier, and also enables techniques such as dynamic neural networks [33].

3.1.3 OpenAI Gym

The OpenAI Gym is a framework that can simplify the creation of RL environments and the testing of RL algorithms [34]. It has several environments that can be used to develop and compare different RL algorithms. It also provides an interface that is common for all gym environments, which means that, in theory, an RL algorithm that can run on one environment can run on all environments without adaption. This greatly simplifies the testing of RL algorithms, because it makes it as easy as changing one line of code to test on different environments. The main functions that need to be defined for Gym environments are [34]:

- 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 an observation of the environment, the reward for the transition, whether the next state is a terminal state, and a dictionary of diagnostic information specific to the environment.
- reset()
  - No arguments, but it resets the environment.
  - Returns the first observation of the reset environment.

3.1.4 Simulators

Choosing which simulator to use for robotic DRL is not a trivial task. There exist many different simulations that are more or less suitable for DRL. Two of these simulators are used in this thesis, Gazebo, and PyBullet.

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 sensors [35, 36].
Gazebo was used during the author's previous work on this project. As stated in [15], this simulator has many shortcomings when it comes to DRL, to name a few: problems with resetting the environment, and low real-time factor. Gazebo also has some positive sides: Gazebo can be controlled with the robot meta-operating system ROS and can use the same ROS services as the physical manipulator, which makes the sim-to-real process easier.

PyBullet

After the author's work on the pre-project, the plan was initially to try using the simulator Multi-Joint dynamics with Contact (MuJoCo) for this master thesis. By doing some more research on MuJoCo, it was discovered that since this is proprietary software, it might not be possible to use the student license provided for a thesis such as this. Also, since it is proprietary, software developed using MuJoCo cannot be used by anyone without a license, so the idea was that this would make the work done on this thesis less available.
When researching open-source alternatives to MuJoCo, the simulator PyBullet was discovered. On PyBullet's website, they specifically list robotics and reinforcement learning as some of the areas in which PyBullet is intended to be used [37]. Based on this, PyBullet was deemed an excellent simulator to use for this thesis.
PyBullet has several features that make it well suited for a project such as this:

- PyBullet has a built-in step function, which moves one timestep forward in the simulation when called.

  - This functionality is well suited for reinforcement learning since the assumption is that the environment does not change without the agent performing an action.

- PyBullet can use the URDF models that are used in Gazebo without any "adaption". This means that the Unified Robot Description Format (URDF) model of the manipulator already used in Gazebo could also be used in PyBullet.
- PyBullet runs much faster than, for instance, Gazebo.

3.2 Hardware

3.2.1 OpenMANIPULATOR-X

The robotic manipulator used for the real-world testing in this thesis is the OpenMANIPULATOR-X (OM-X) by Robotis. This is a manipulator that has four revolute joints that correspond to four DOF. The numbering of these joints is illustrated in Figure 3.2. It also has an end-effector, which is a gripper, which means that the total number of DOF is five. This manipulator lacks a spherical wrist, which means that it has two less DOF than what is typical in an industrial robotic manipulator (not counting the end-effector) [30]. 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 OM-X is based on ROS and OpenSource [1]. The manipulator, together with the lever described below, can be seen in Figure 3.1.

Figure 3.1: Manipulator and lever in the real world setup

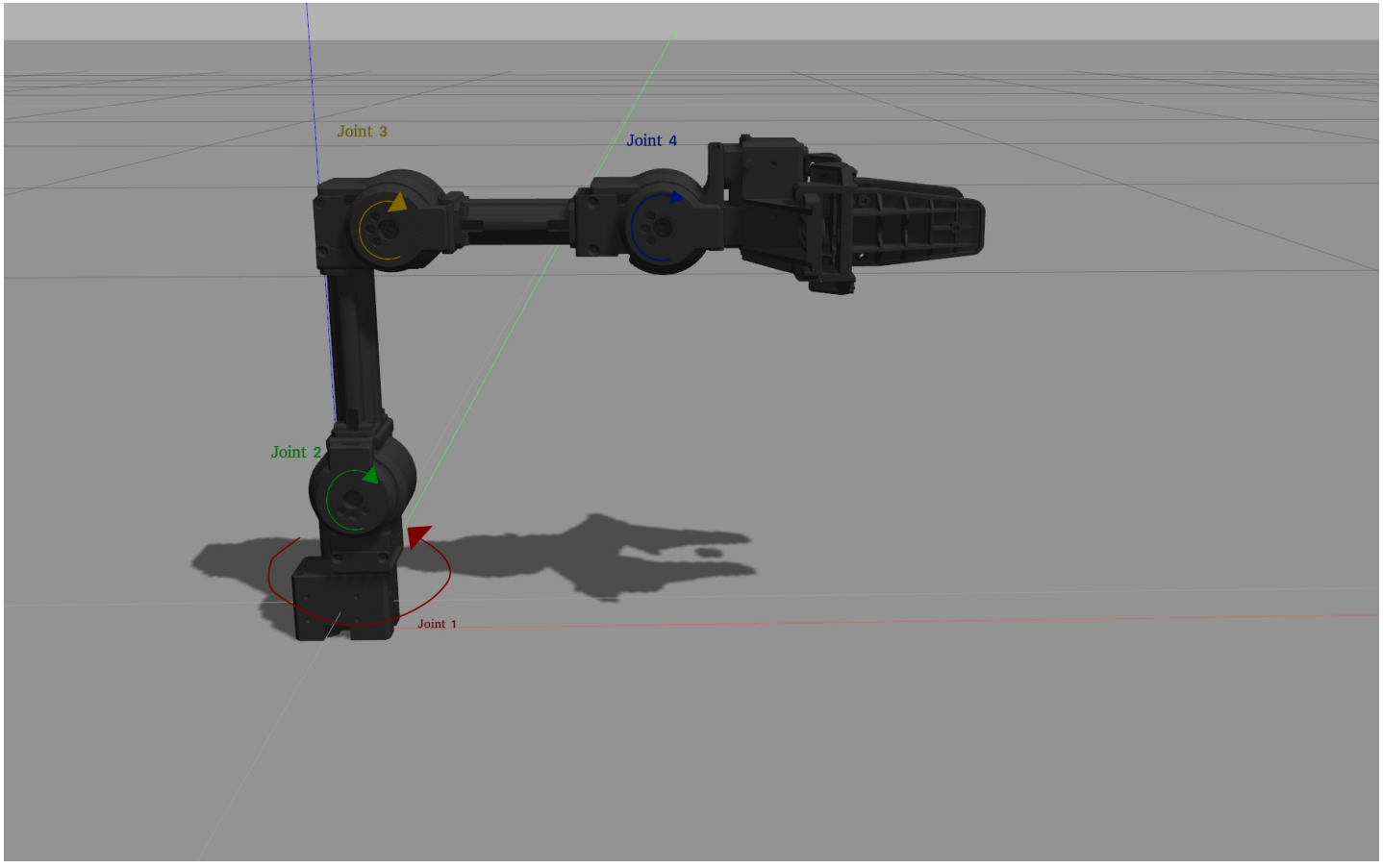


Figure 3.2: Illustration of the joint numbering of the manipulator (screenshot taken in Gazebo)

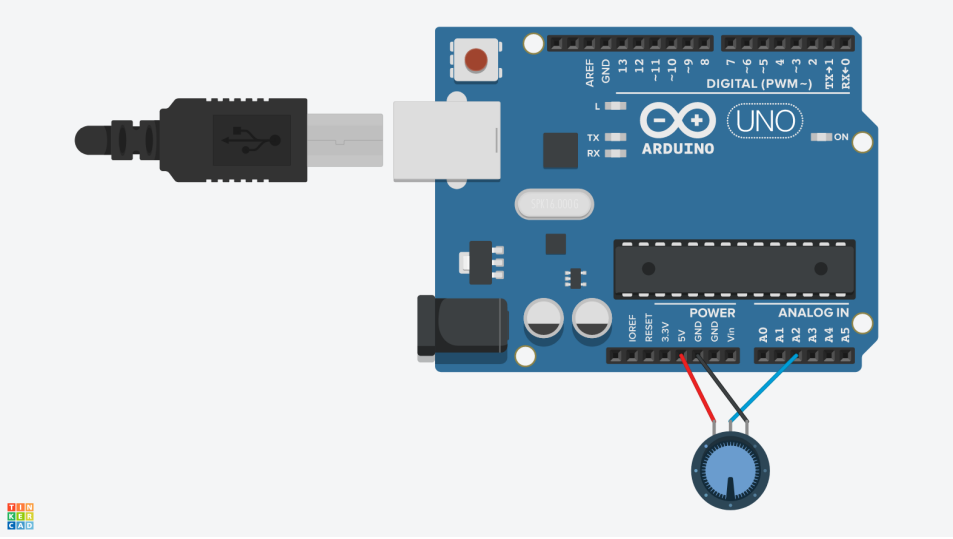


Figure 3.3: Potentiometer setup with Arduino Uno. Image is made using Tinkercad [6].

3.2.2 Lever

The lever that was created during the pre-project [15] was deemed unnecessarily complicated for the work on this thesis. It also needed an external power source in addition to a microcontroller to work. Therefore some changes to the lever were made. The servo motor that was initially installed was taken out and replace with a potentiometer. This potentiometer was wired to an Arduino Uno. The Arduino Uno can use ROS to send messages, using the rosserial package[1]. The code was written in C++ to periodically send messages with the joint angle of the lever to a ROS topic. The wiring of the potentiometer and Arduino can be seen in Figure 3.3 and the real-world setup can be seen in Figure 3.4.

[1] http://wiki.ros.org/rosserial

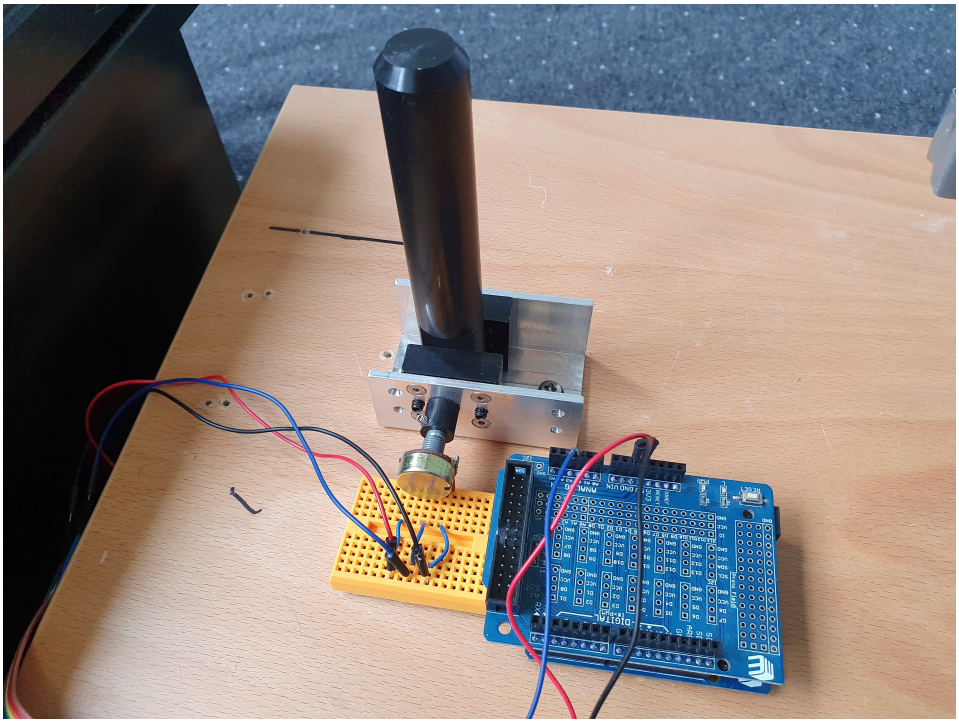


Figure 3.4: The lever, arduino and potentiometer setup

Chapter 4

System design

4.1 DDPG and HER

The DDPG and HER implementation used in this thesis is the one used in the author's earlier work [15]. This was adapted from [2] during the author's work on the pre-project. The implementation is written in the programming language Python 2.7 and uses the deep learning framework PyTorch. As stated in Section 2.1.3, DDPG is an actor-critic algorithm, the model architecture of the actor and critic network is shown in Figure 4.2 and Figure 4.1, and is as follows[1]:

- 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.
    - For some of the experiments done in Chapter 5, the state vector may be changed. The reasoning behind this and how it is changed is explained in the respective experiments' sections in Chapter 5.
  - 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 bounded in both directions.

[1] The network description here is specifically for the reaching task described in the next section.

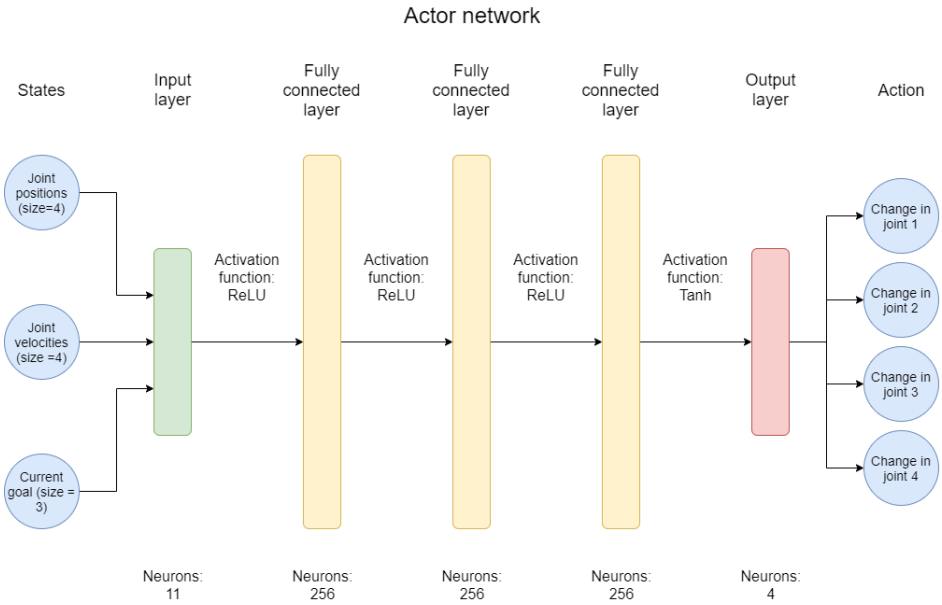


Figure 4.1: Actor-network architecture

- 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 described in the actor-network above, this state vector is also extended in some of the experiments in Chapter 5
  - 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 used in the output layer.

The hyperparameters used in the DDPG implementation are shown in Table 4.1. In this table of hyperparameters, ϵ_n corresponds to the standard deviation of the gaussian noise that is added to the actions, and ϵ_r is the chance of the agent taking a random action instead. This means that this implementation combines both of the exploration techniques described in the theory chapter.

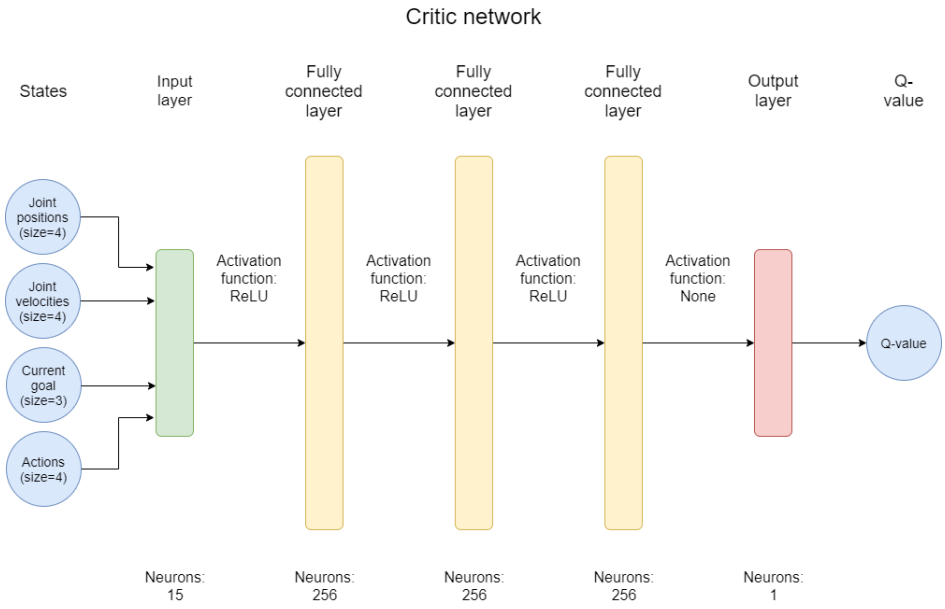


Figure 4.2: Critic-network architecture

Hyperparameter	Value
Learning rate actor, α_a	0.001
Learning rate critic, α_c	0.001
Discount factor, γ	0.98
l2 regression, λ	1
noise_eps, ϵ_n	0.2
random_eps, ϵ_r	0.18
HER ratio to be replaced, k	4
HER replay strategy	future
Mini-batch size	256

Table 4.1: DDPG hyperparameters

4.2 open_manipulator_rl_environments

open_manipulator_rl_environments (OPM-RL-ENVS) is a package for ROS created by the author during the work on this thesis. It uses the interface of the OpenAI Gym described in Section 3.1.3, so in theory, any RL algorithm that can use continuous state and action spaces can be used with this package. The end-goal of this package is to make it simpler to design new reinforcement learning tasks for the OpenMANIPULATOR-X and to simplify the transfer between PyBullet, Gazebo, and the real-world manipulator on these tasks. The inspiration for this package comes from the author's work on the pre-project preceding this thesis [15], and the openai_ros package [38], which was used during the work on the pre-project. This package was written in Python 2.7.

4.2.1 Robot environments

The robot environments handle the communication with either the simulators or the real manipulator, hereafter called the domains. The robot environments function as a link between the domains and the DRL agent. An effort was made to make the interface to the robot environments as similar as possible. This was done because to make it so a change to a task will transfer to any of the domains without needing to adapt the robot environments.

OpenManipPyBulletRobot

This robot environment uses a direct connection to PyBullet. As opposed to the other two robot environments, no process-to-process communication is done. This increases the simulation speed even further compared to Gazebo.

OpenManipGazeboRobot

This robot environment uses ROS services and topics to communicate with Gazebo. It uses the ROS topic open_manipulator/joint_states to subscribe to the joint positions and velocities, and it uses the topic open_manipulator/gripper/kinematics_pose to subscribe to the task space position of the end-effector. To change the joint angles of the manipulator, it uses the ROS service /open_manipulator/goal_joint_space_path_from_present. To open and close the gripper, it uses the /open_manipulator/goal_tool_control service. It has a persistent connection to both of these services to improve the response time.

OpenManipPhysicalRobot

This is the robot environment that is used to control the real manipulator. An advantage with the Gazebo simulated manipulator is that it publishes the same topics and services as the real manipulator. This means that this robot environment uses the same topics and services as the OpenManipGazeboRobot.

4.2.2 Task environments

The following are the environments that define the tasks done in this thesis. Since all of these environments use sparse rewards, an episode is classified as a success if the last step of the episode gives a reward of 0. For all of these environments, the actions correspond to how much the joint angles should change relative to the current joint angle (except for the gripper joint, which is described soon). The action is scaled by a constant $a_{scaling} \in (0, 1)$. This is done so the agent does not move too far in a single action. After experimenting with different values for $a_{scaling}$, a good value was found in $a_{scaling} = 0.2$. This means that if for instance the action $a = [1.0, -1.0, 0.6, 0.0]$ is chosen in the reach environment described below, joint 1 will move $0.2\,\mathrm{rad}$, joint 2 will move $-0.2\,\mathrm{rad}$, joint 3 will move $0.12\,\mathrm{rad}$, and joint 4 will have the same angle. For the tasks that use the gripper, the first joint is not used. This is to make the task a little bit easier to solve. Instead, the 4th entry in the action array (a_4) correspond to whether the gripper should be open or closed:

$$\begin{cases} a_4 \geq 0, & \rightarrow \text{Gripper should be open} \\ a_4 < 0, & \rightarrow \text{Gripper should be closed} \end{cases}.$$

This means that if the action $a = [0.6, -1.0, 0.4, -0.1]$ is done in the tasks that use the gripper (the two lever environments), joint 2 will move $0.12\,\mathrm{rad}$, joint 3 will move $-0.2\,\mathrm{rad}$, joint 4 will move $0.08\,\mathrm{rad}$, and the gripper will be closed. Moreover, the state space will be augmented by four states for the environments that use the gripper. Two states for the position of the gripper's two parts, and two states for the velocity of these two parts.
When an action is done in these environments, the environment will wait until either the action is complete or a certain amount of time has passed (if, for instance, the action can not be done, because the action would lead to, for example, going through the floor). Waiting until the action is complete is done for two reasons. The first reason is to make it easier to transfer between domains, because of differences in the controllers. The second reason is that the agent cannot take actions too frequently. "In continuous control domains, the performance goes to zero as the frequency of taking actions goes to infinity" [39]. The DRL agent controls of how the joint angles should change, and not what the torque of the servo motors should be. This means that the agent is not controlling the robot on the lowest level, and another controller handles how to change the torque of the servo motors so that the desired joint angles are achieved. It is feasible to let an agent control the torques directly, however, to train such an agent in a simulator before transferring to the real world requires very accurate information about the dynamics of the manipulator in the simulator, and this was therefore not done. Letting an agent directly control the torques of the servo motors is done, for instance, in the paper *End-to-End Training of Deep Visuomotor Policies* [12].

OpenManipulator_reach_task-v0

This is a recreation of the primary task done in the pre-project [15], using the open_manipulator_rl_environments package. This is a task where the goal is to place the end-effector of the manipulator in the vicinity of a randomly selected point in the

manipulator's workspace. If the end-effector is within a sphere with a radius of 2.0cm and origin in the randomly selected point, it is classified as being in the vicinity of the point. The points are randomly selected between episodes using randomized spherical coordinates. These spherical coordinates are uniformly selected between the ranges

$$r, \theta, \phi \in \mathbb{R} : r \in [0.18, 0.30], \theta \in [0, \frac{\pi}{3}], \phi \in [-0.89\pi, 0.89\pi]. \tag{4.1}$$

In addition to the randomly selected point the manipulator should reach, the manipulator's starting position is also partly randomized. The starting position of the first joint of the manipulator (see Section 3.2.1 for the numbering of the joints) is randomized using a uniform distribution in the range $\theta_1 \in [-\frac{\pi}{2}, \frac{\pi}{2}]$. The other joints of the manipulator start each episode with

$$\theta_2 = \theta_3 = \theta_4 = 0.$$

The gripper is not used in this task, and is therefore always in the starting position

$$d_1 = d_2 = 0.$$

This task essentially requires the DRL agent to discover the position inverse kinematics of the manipulator. The term inverse kinematics is described in Section 2.2.2.

OpenManipulator_reach_task_oscillating-v0

This task was inspired by the idea of using DRL to control a manipulator that is attached to an Unmanned Underwater Vehicle (UUV). This task is the same as the regular reach task described above, but the manipulator is also oscillating along the world z-axis. This is done to emulate the disturbance caused by underwater flows. In reality, the flows are much more complex, and the disturbance is not only acting along the z-axis.
The oscillations are created using

$$o_t = A - A\cos\left(2\pi(t + \phi)/P\right),$$

where A is the amplitude of the oscillations given in meters, t is the current timestep in the episode, ϕ is the phase shift with respect to the timestep, P is the period of the oscillations with respect to the timestep, and o_t is the z position of the base in the current timestep. The amplitude, the phase shift and the period is randomized uniformly each episode between the bounds

$$A \in \mathbb{R}, \phi, P \in \mathbb{Z} : A \in [0.02, 0.07], \phi \in [0, 120], P \in [240, 720]$$

To make it feasible for the DRL agent to solve this task, the state vector has to be extended. Two new states are added to the state vector, the position of the manipulator's base given in world z-coordinates, and the velocity of the manipulator's base in world z-coordinates. It is questionable whether the information in these two new states (especially the position) can be given with reasonable accuracy in an underwater setting, but for this task, it is assumed that these can be given to perfect accuracy.

OpenManipulator_line_follow_oscillating-v0

To further examine the feasibility of using DRL to control the robotic manipulator while it is under an oscillating disturbance, an additional task was created using the open_manipulator_rl_environments framework. This task is the same as the reaching task with oscillations described above, except that the goal is not to reach a single point. Here, the goal is to make the end-effector of the manipulator follow a straight-line path. This straight-line path is randomized between episodes by first randomly selecting a start point for the line, then randomly selecting an end point for the line, and finally making a line between the two points. The start point and the end point is selected uniformly according to

$$\begin{aligned} x_{start}, x_{end} \in \mathbb{R} : x_{start} = x_{end} &\in [0.1\text{m}, 0.23\text{m}] \\ y_{start} \in \mathbb{R} : y_{start} &\in [0.15\text{m}, 0.25\text{m}] \\ y_{end} \in \mathbb{R} : y_{end} &\in [-0.25\text{m}, -0.15\text{m}] \\ z_{start} \in \mathbb{R} : z_{start} &\in [0.1\text{m}, 0.3\text{m}] \\ z_{end} \in \mathbb{R} : z_{end} &\in [0.1\text{m}, 0.3\text{m}] \end{aligned}$$

To give information to the DRL agent about the line, the line is discretized into points. Experiments with different numbers of discretization points in the line were done, and 20 discretization points proved a good number. Since the maximum length of the line is

$$\begin{aligned} \sqrt{(y_{start\_max} - y_{end\_min})^2 + (z_{start\_max} - z_{end\_min})^2} &= \sqrt{(0.5\text{m})^2 + (0.2\text{m})^2} \\ &\approx 0.5385\text{m} \end{aligned}$$

(the x-position is randomized, but the same for both the start point and the end point) this means that the maximum length between each discretization point is approximately $\frac{0.5385\text{m}}{20} = 0.026925\text{m}$. The minimum length between each discretization point is

$$\frac{y_{start\_min} - y_{end\_max}}{20} = \frac{0.3\text{m}}{20} = 0.015\text{m},$$

because, for the minimum length of the line, the z-coordinates are the same.

OpenManipulator_lever_manipulate_task-v0

One of the main goals of this thesis is to create a DRL agent that can use a real-world manipulator to manipulate a lever. To do this, a task environment has to be created that has a lever. The real-world lever described in Section 3.2.2 was measured, and a mesh that is similar to the real lever was created using the software Blender [40]. To use the lever model in the simulators, a URDF model was created, which uses the mesh made in Blender for visual and collision. In addition to this, an URDF also needs to specify the properties of the lever. The properties that needed to be defined were inertia, mass, friction, and damping. Trial and error were used to make the properties of the simulated lever similar to the real lever. Gazebo and PyBullet can, fortunately, both use URDF models, but the two simulators use different parameters when specifying the properties. This lead to requiring

one URDF for Gazebo, and one URDF for PyBullet. The lever model in the simulator Gazebo can be seen in Figure 4.3. During training, the lever's position, the start angle of the lever joint, and the goal angle of the lever joint are randomized between episodes. The position of the real lever relative to the real manipulator was measured to be about $0.256\mathrm{m}$ along the x-axis. Based on this, the position of the lever during training in the simulator was randomized uniformly according to

$$x_{lever} \in \mathbb{R} : x_{lever} \in [0.23\mathrm{m}, 0.27\mathrm{m}].$$

This was done to make the agent experience different lever positions to account for measuring inaccuracies in the real lever's position. The y- and z- coordinates of the lever are zero for all episodes. The start angle and goal angle of the lever are randomized uniformly according to

$$\theta_{start}, \theta_{goal} \in \mathbb{R} : \theta_{start} \in [-1.0 \text{ rad}, 1.0 \text{ rad}], \theta_{goal} \in [-1.0 \text{ rad}, 1.0 \text{ rad}],$$

with the additional constraint that $|\theta_{start} - \theta_{goal}| > 0.4$ rad.

Rewards in this environment are sparse because HER should be used. The reward is given according to

$$r = \begin{cases} -1, & \text{if } |\theta_{lever} - \theta_{goal}| \geq 0.025 \text{ rad} \\ 0, & \text{if } |\theta_{lever} - \theta_{goal}| < 0.025 \text{ rad} \end{cases}.$$

Inspired by the paper where HER was introduced [11], the manipulator starts with a grasp on the lever for 50% of the episodes during the training. This is done to encourage exploration and make training faster.

In this task, the state vector is also different than what was described in Section 4.1. The state still contains information about the position and velocity of the joints. The Cartesian coordinates of the base of the lever with respect to the manipulator base, the Cartesian distance from the end-effector to the base of the lever and the current angle of the lever is also given. Also, the goal in the form of a scalar that represents the desired angle of the lever in radians is given.

OpenManipulator_lever_manipulate_oscillating-v0

This is the final task that was created for this package. It adds oscillations to the lever manipulation task. It randomizes the parameters of the oscillations between episodes as in the previous oscillation tasks, and it also randomizes the position and angle of the lever as in the lever manipulation task above.

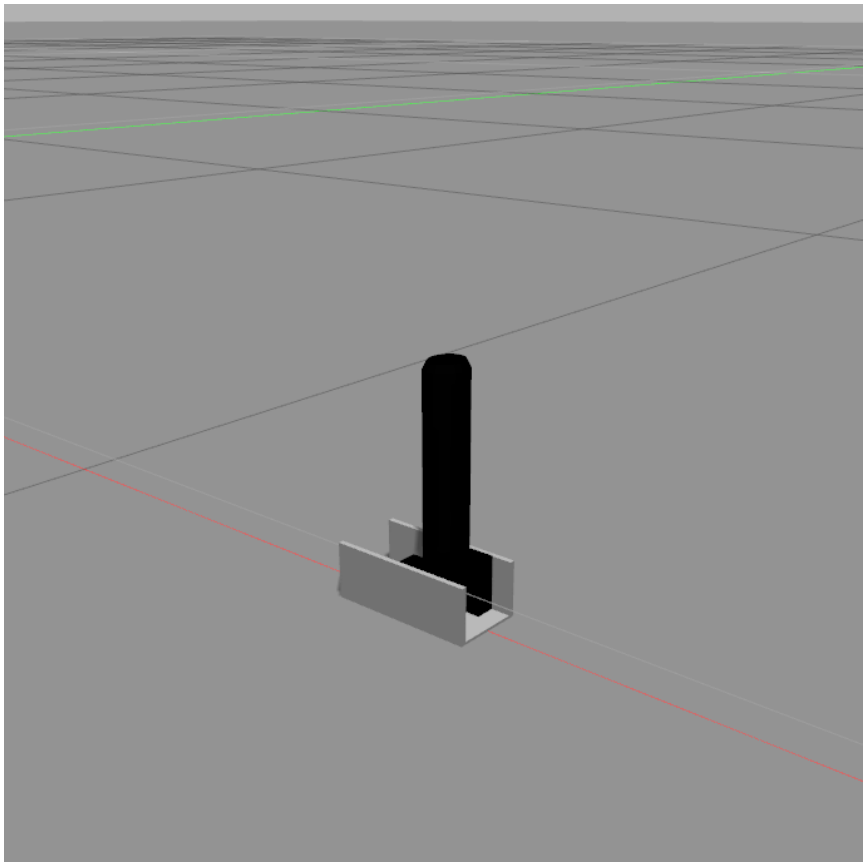


Figure 4.3: Lever model in Gazebo

Chapter 5

Results and discussion

5.1 Activation function comparison

5.1.1 Results

To provide some motivation for the choice of activation function for the ANNs used in this thesis, a comparison between the activation functions in question is made here.
To create this comparison, the OpenManipulator_reach_task-v0 environment from the open_manipulator_rl_environment described in Section 4.2 is used. This is the most straightforward environment in the package, which means that it uses the shortest amount of time to train. Nevertheless, reaching a point is common for all tasks that can be performed by a robotic manipulator, so this environment is considered a fair way to test the activation functions. To test the activation functions, the training procedure is as follows:

- Five agents were trained concurrently on the reaching task using DDPG with the same hyperparameters for each agent. The hyperparameters are shown in Table 4.1. This was done for each activation function, so this means that 25 different agents were trained in total for this comparison.
- Each agent was trained for 50 epochs with 15 training episodes in each epoch.
- After each epoch, 30 test episodes were done.
- Each episode consists of 300 timesteps.
  - In the test episodes, the agents choose the greedy actions, which means that they follow a strictly deterministic policy.
- The results from the test episodes are plotted in Figure 5.1 and Figure 5.2. In the plot, the mean of the results from the five agents is plotted, and the similarly colored area around the lines corresponds to half a standard deviation.

– Quadratic interpolation was used when creating the plots, to make them easier to read.

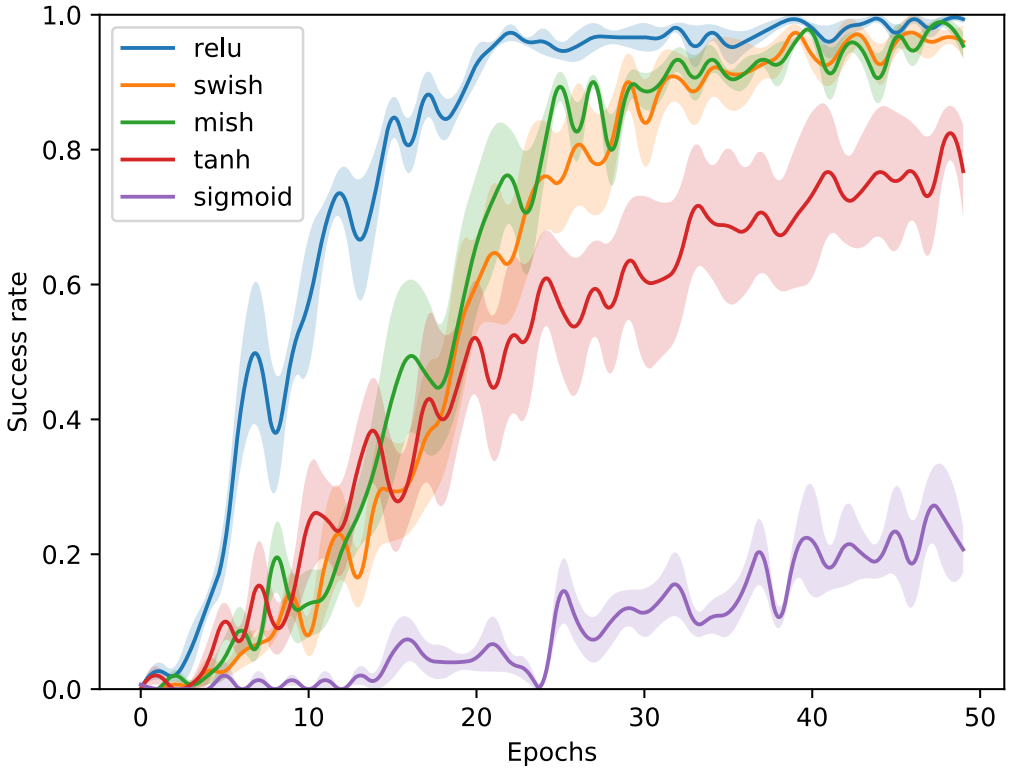


Figure 5.1: Success rates from activation function test

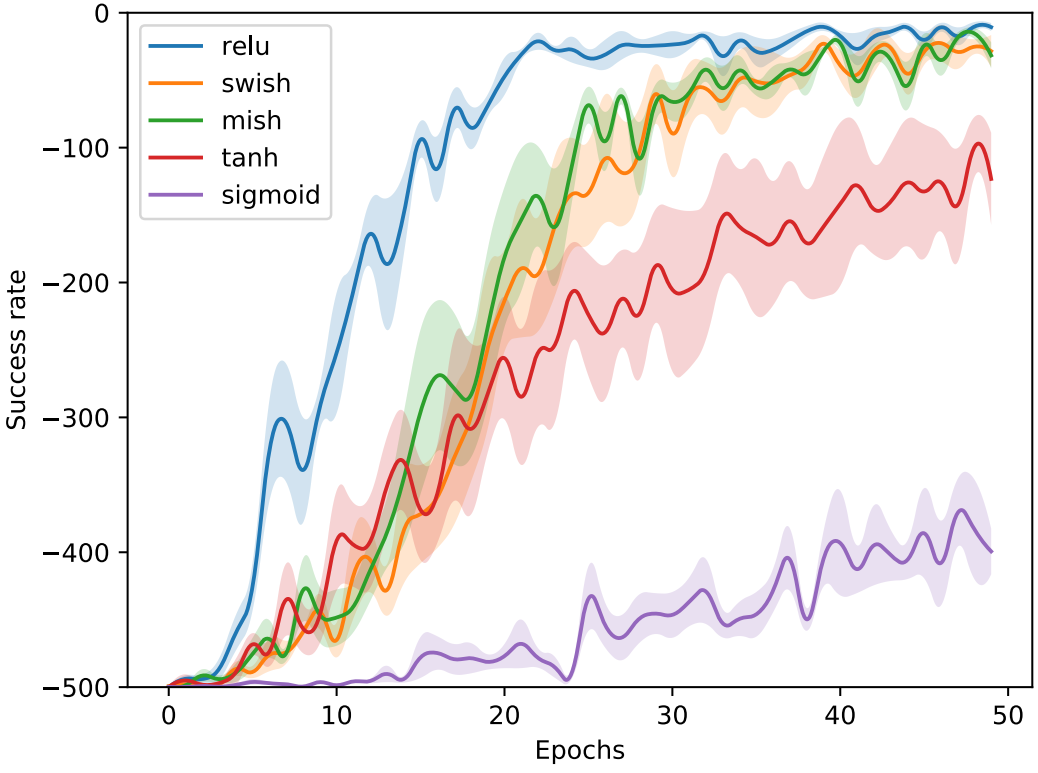


Figure 5.2: Average rewards from activation function test (the label on the y-axis is wrong)

5.1.2 Discussion

From Figure 5.1 and Figure 5.2 the activation functions can be arranged according to their performance on this task:

1. ReLU
2. Swish and Mish
3. Tanh
4. Sigmoid

The results were more or less as expected. ReLU has, by far, the best performance on this task. It stabilizes at a success rate of ≈ 1.0 faster than any other activation function, and it has the lowest variance on the task (measured in the standard deviation over the five agents trained). The expectation before doing the experiment was that Mish or Swish would have the best performance (or at least similar performance to ReLU), but this was not the result. It may be that Mish's and Swish's performance boost only comes into play when the ANN is deeper than it was in this experiment. For a shallow ANN such as this, ReLU's simplicity may be some of what makes its performance better.
Tanh and sigmoid never reach a success rate of ≈ 1.0. Tanh has, as expected, better performance than sigmoid with success rate ≈ 0.8, and sigmoid being the worst performing activation function on this comparison, with success rate ≈ 0.2 at the end of the training. Tanh and sigmoid might have reached the same success rate as the other three activation functions if given more training episodes, but for this thesis, this experiment is enough to exclude them from further usage as activation functions for hidden layers. It should again be mentioned that all the agents use tanh as the activation function for the output layer of the actor ANN.

5.2 Transfer learning from PyBullet to Gazebo

5.2.1 Results and discussion

When an agent trained in PyBullet on the OpenManipulator_reach_task-v0 was transferred to Gazebo on the same task, the agent did not perform as well as it did in PyBullet. The agent reaches the vicinity of the point, but it does not stop moving as it does in PyBullet, and instead moves around the vicinity of the point for the entirety of the rest of the episode. Comparing Figure 5.3, which is the error in PyBullet, with Figure 5.4, which is the error in Gazebo, this is apparent. Note that the agent/neural network that gave these results is the same in both of these figures, only the simulator is different. Because of these unsatisfactory results, an attempt was made to employ transfer learning to improve the performance in Gazebo.

The idea is to pre-train the agent in PyBullet, and after that, continue to train the agent in Gazebo on the same task. Gazebo has very similar performance to the real-world manipulator, so it is assumed that good performance in Gazebo equals good performance on the real-world manipulator.

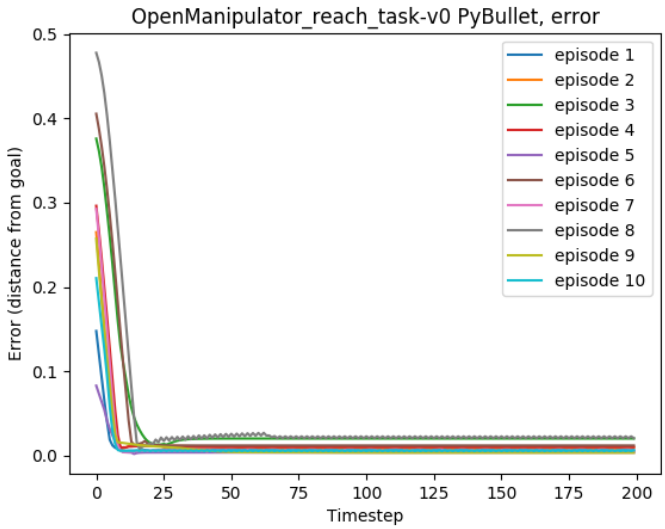


Figure 5.3: The error in PyBullet

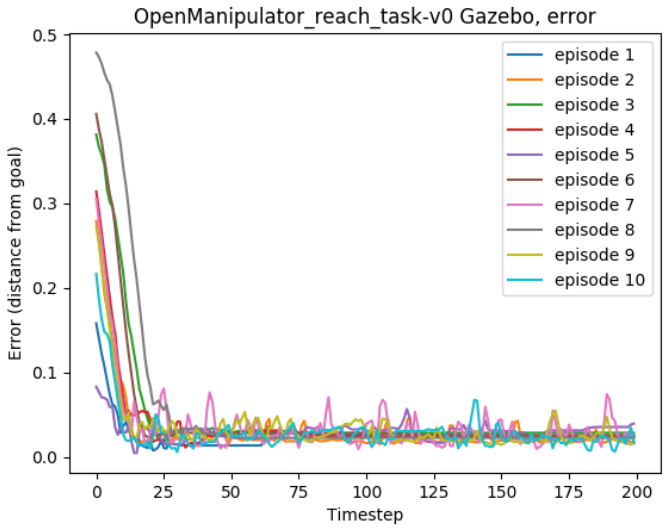


Figure 5.4: The error in Gazebo before transfer learning (same neural network as in Figure 5.3)

Training in PyBullet is much faster than training in Gazebo. Ten training episodes with 300 timesteps in each episode takes about 206 seconds in Gazebo; the same amount of training episodes only takes about 15 seconds in PyBullet. Therefore, the prediction before attempting to do this transfer learning is that the total amount of time it will take to train will be much shorter than solely training in Gazebo.

To better illustrate how the manipulator moves its end-effector towards the goal, the distance between the end-effector and the goal point was plotted over ten test episodes. These ten test episodes use the same ten goals for all the comparisons. The euclidian distance from the end-effector to the goal was plotted over the entire episodes using

$$\sqrt{(x_{ee} - x_g)^2 + (y_{ee} - y_g)^2 + (z_{ee} - z_g)^2}$$

to calculate the euclidian distance. Here x_g, y_g, z_g are the euclidian coordinates of the goal, and x_{ee}, y_{ee}, z_{ee} are the euclidian/task-space coordinates of the end-effector. This euclidian distance can be interpreted as the error over the episode since the objective of the agent is to place the end-effector in the goal point. In other words, the objective is to minimize the distance from the end-effector to the goal.

The results from testing the agent in PyBullet after training are excellent, see Figure 5.3. If these results could transfer to Gazebo, that would have been ideal. However, based on Figure 5.4, this is not the case. As explained above, the manipulator oscillates around the goal point when transferring to Gazebo. Similar behavior would most likely occur if the agent was transferred to the real-world manipulator, so because of the risk of damaging the real manipulator from this, it was not attempted. Based on these results, a transfer learning plan was created. This plan is as follows:

- Two hundred transfer learning episodes was done, using the actor, critic, target actor, and target critic networks from the results in Figure 5.3 and Figure 5.4 as the initial neural networks.
- Ordinarily, in transfer learning, the first layers of the neural networks are frozen, and only the last layers are updated during transfer learning. However, for this, all of the layers are updated during transfer learning. Because the robots are identical, and only the platform is different, this was theorized to be the best way to do it.
- The hyperparameters for this transfer learning scheme are the same as in Table 4.1, except the learning rate. The learning rate that was used was somewhat lower, 0.0008 for both the actor and the critic.

The results from Figure 5.3, Figure 5.4 and Figure 5.5 are all done with the same 10 goals for a true comparison. These ten goals were randomly selected beforehand using the spherical coordinates described in Equation (4.1). The initial position of the manipulator is not randomized between episodes; the initial joint angles are $\theta_1 = \theta_2 = \theta_3 = \theta_4 = 0$ for all ten episodes.

The results after transfer learning for 200 episodes are shown in Figure 5.5. Although it can be seen that the agent moves slower to the goal than in Figure 5.3 and Figure 5.4, the

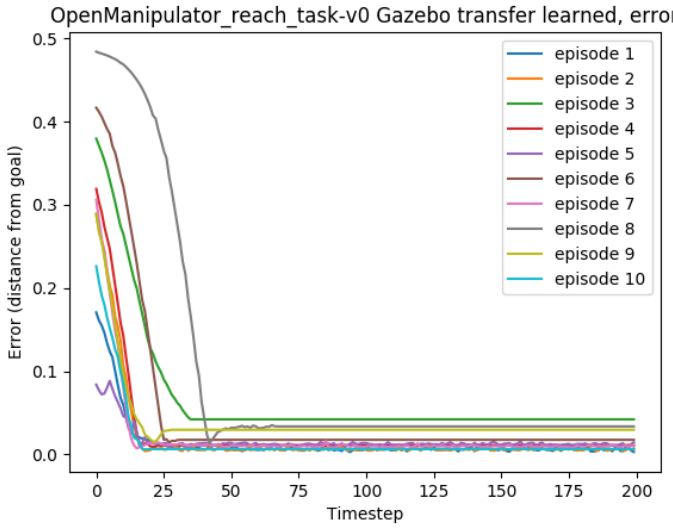
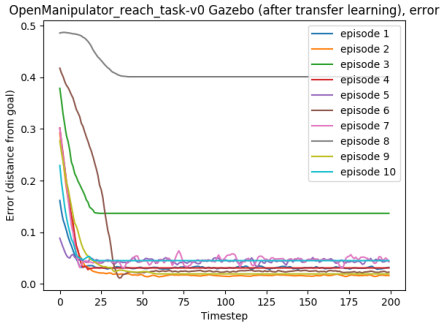


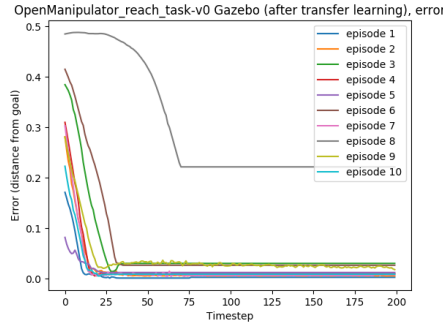
Figure 5.5: The error in Gazebo after transfer learning

manipulator is much more stable than in Figure 5.4. It is also apparent that for some goals, the transfer learned agent is less accurate than before transfer learning (consider, for instance, the green line that corresponds to episode 3). However, the goal of this experiment was to see whether the agent could learn to keep the manipulator more steady after reaching the goal in Gazebo after transfer learning. This was, without a doubt, accomplished. By transfer learning for more than 200 episodes, the agent will most likely also be faster and more accurate.

Transfer learning was also tried using differential learning rate and frozen parameters in all but the last layer, as described in Section 2.1.5. This gave much worse results than updating all layers, as can be seen in Figure 5.6a and Figure 5.6b. The reasons for these unfortunate results are challenging to understand, because of the complexity of neural networks.



(a) Transfer learning with frozen weights in all layers except the last layer.



(b) Transfer learning with differential learning rate.

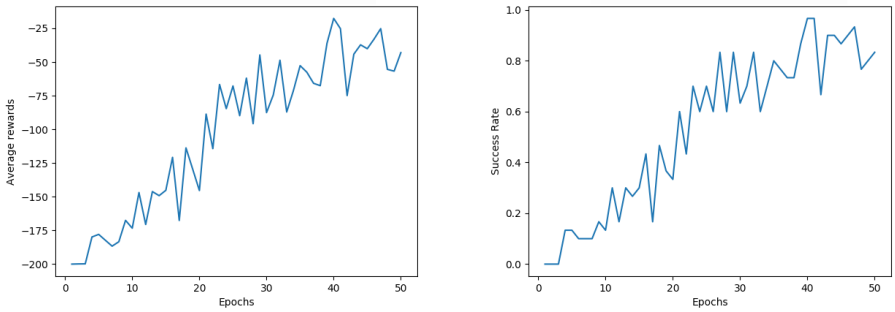
Figure 5.6: Transfer learning with frozen weights and differential learning rate

5.3 Lever manipulation task

5.3.1 Results

As stated earlier, one of the main goals of this thesis is to train a DRL agent so that it can learn to pull a lever. This is what is done in this section. In addition, the agent trained in the simulators is also transferred to the real manipulator.

- First, an agent was trained in PyBullet, using the hyperparameters in Table 4.1.
  - The agent was trained for 50 epochs, with 30 episodes in each epoch. After each epoch, 30 test episodes were done.
  - The results from this training can be seen in Figure 5.7.
- After the agent was finished training, five episodes were done with five preselected start and goal positions for the lever. These preselected positions were randomly selected according to the approach described in the OpenManipulator_lever_manipulate_task-v0 part of Section 4.2.2.
- Then the same five test episodes with the preselected start and goal positions of the lever were done in Gazebo, and the results can be seen in Figure 5.9.
- The same transfer learning plan as was described in Section 5.2 was used on this task, with the only difference being that the transfer learning was done for 300 episodes instead of 200 episodes. This was motivated by the results in Section 5.2 where the agent was somewhat slow and inaccurate after only 200 episodes of transfer learning. The results after transfer learning in Gazebo on the five test episodes can be seen in Figure 5.10.
- Finally, the task was transferred to the real world. The results from this can be seen in Figure 5.11. This is done using the same five start and goal positions as in the others. A video was made of the performance of the agent during this, and this video



(a) Average rewards

(b) Success rates

Figure 5.7: Average rewards and success rates from training on the lever manipulation task in PyBullet

is delivered with this thesis. It should be noted that sometimes during the operation on the real manipulator, the ROS service that is used to change the joint angles takes a long time (about five seconds) to respond to a command. During these times, the video is cut, to keep the video shorter and more enjoyable to watch. Note that the results in Figure 5.11 are done with only 70 steps in each episode in contrast to the other figures in this section, which has 100 steps in each episode. This was done to keep the video shorter. This video's filename is lever_manipulation.mp4.

5.3.2 Discussion

As was expected, this task was more complicated for the agent to solve than the reaching task. It required twice the number of episodes to reach close to a 100% success rate. As can be seen in Figure 5.7, the performance has not stabilized at 100% success rate at the end of the training. This means that it could most likely have benefited from training for more episodes. Figure 5.8 shows the results from doing the 5 test episodes in PyBullet after finishing the training. The performance is accurate and fast, even though the agent first moves the lever in the wrong direction for all episodes except episode 2. When the agent trained in PyBullet is transferred to Gazebo in Figure 5.9, it still maintains this behavior where it moves the lever in the wrong direction for all episodes but episode 2. However, this behavior is amplified in Gazebo. It also uses more steps to reach its goal, and when it reaches the goal, it oscillates somewhat around the goal, especially in episode 2 and 5. It is clear that this could benefit from transfer learning in Gazebo. Figure 5.10 shows the performance after transfer learning has been done, and the performance has become much better. It can, in a way, be argued that the performance in Gazebo now is better than the performance in PyBullet, since the manipulator does not first move the lever in the wrong direction. Then, the agent that was transfer learned in Gazebo is transferred to the real manipulator. When this was done, the $a_{scaling}$ parameter was changed from 0.2 to 0.1. This was done to make the manipulator move less in each step so that if the agent chooses to something to the manipulator that would damage it, it would be easier to

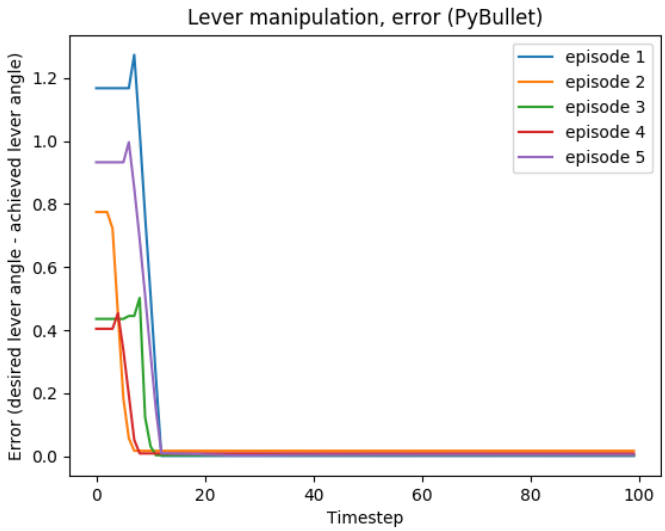


Figure 5.8: The error in PyBullet on lever manipulating task

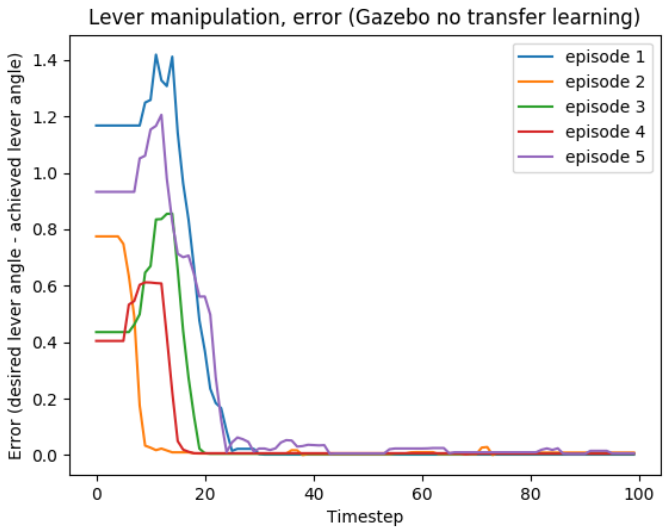


Figure 5.9: The error in Gazebo before transfer learning on lever manipulating task

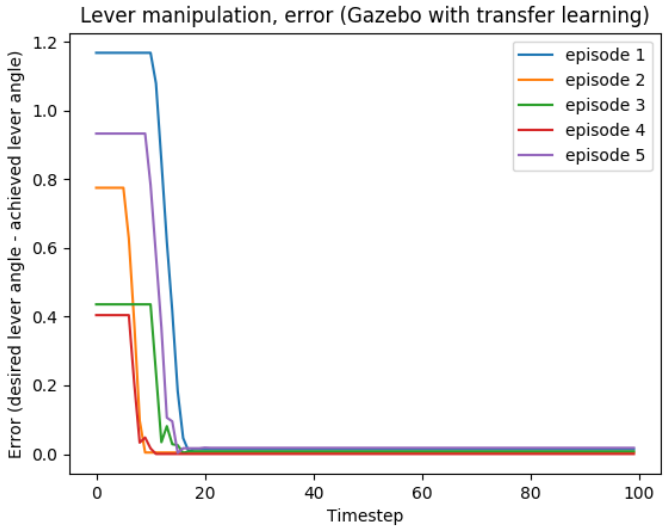


Figure 5.10: The error in Gazebo after transfer learning on lever manipulating task

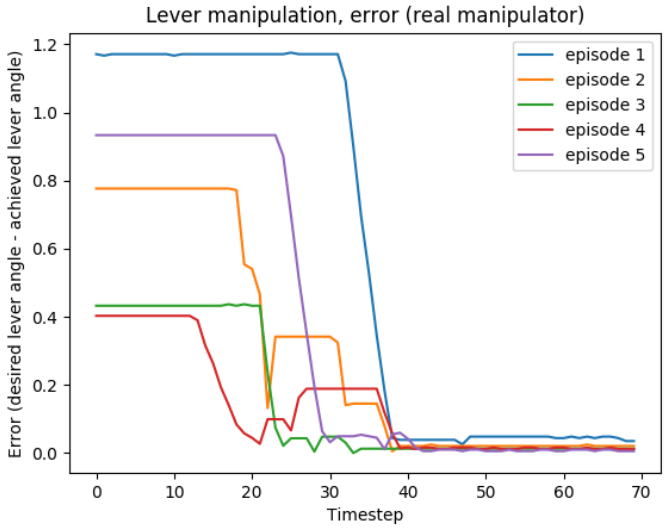


Figure 5.11: The error on the real world manipulator on lever manipulation task

anticipate and turn off the operation preemptively. This change in the $a_{scaling}$ parameter explains at least partly why the lever is moved slower to the goal in Figure 5.11 than in the others. The lever is also moved back and forth for some episodes (episodes 2 and 4 specifically), and the lever is not moved as close to the goal angle as it was when doing the same task in the simulator. This is more than likely because of the simulator to the real-world gap, described in Section 2.1.5. It is assumed that the dynamics and behavior of the real manipulator are similar to the simulated manipulator. However, the real lever seems to have very different properties when compared to the simulated lever. As the properties of the simulated lever were estimated by using trial-and-error, and without any measuring of, for instance, the friction, this is quite likely. Even though the performance on the real manipulator is much worse than in the simulations, the agent manages to reach the goal of moving the lever within $0.025\,\mathrm{rad}$ of the goal angle for all episodes except episode 1 (in episode 1 it is about $0.035\,\mathrm{rad}$ away from the goal angle). This transfer from the simulator to the real world could likely have benefited from techniques such as dynamics randomization [13], especially randomization of the dynamics of the lever. Unfortunately, this was never implemented during the work on this thesis.

5.4 Oscillating tasks

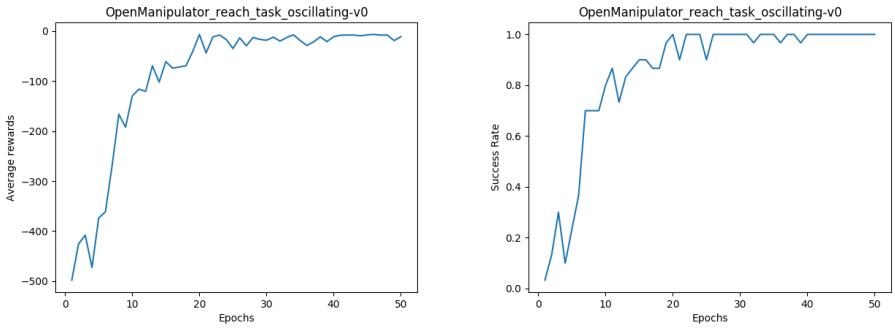
As described in Section 4.2.2, the motivation behind these oscillating task environments is to emulate underwater currents. Results from the tasks described in Section 4.2.2 are shown and discussed below, and these results are solely from the simulator PyBullet. The idea was originally to create a device that could cause similar oscillations in the real-world setting, which the real-world manipulator could be attached to. This was so that the tasks could have been transferred to the real world. However, because of the Covid-19 pandemic, creating this real-world oscillating device proved unfeasible for this thesis. A video that shows the performance of these three oscillating tasks in the simulator is also delivered with this thesis. The filename of this video is oscillating_tasks.mp4.

5.4.1 Oscillating reach task

Results

As previously mentioned, the motivation behind this task is to see whether it is feasible to control the manipulator while under an oscillating disturbance along the z-axis. The following training procedure was used to achieve these results:

- A single agent was trained using DDPG with the hyperparameters in Table 4.1.
- The agent was trained over 50 epochs, with 15 training episodes in each epoch
- After each epoch, 30 test episodes were done, and as in Section 5.1, the agent chose only greedy actions during the test episodes.
- The results from training this task in PyBullet is shown in Figure 5.12a and Figure 5.12b.



(a) Average reward

(b) Success rate

Figure 5.12: Success rate and average reward from training one agent on the reaching task with oscillations in PyBullet

After training, five test episodes were done, where the distance from the end-effector to the goal (the error) is plotted. This error was calculated the same as in Section 5.2. Results from this are shown in Figure 5.13, and this is the error from the episodes shown in the video.

Discussion

This experiment indicates that it is feasible for a DRL agent to control a robotic manipulator that is under an oscillating disturbance, given that the state gives information about the oscillations with respect to the position and velocity. When comparing the results on the reach-task in Section 5.1 (specifically the results with ReLU) with Figure 5.12b and Figure 5.12a, the agent, surprisingly, learns at a similar rate for both the non-oscillating and oscillating task. A reaching task is still a quite simple task, so even if oscillations and two new states are added, it seems that this still is a simple task for a DRL agent using DDPG and HER to solve.
From Figure 5.13, it is also apparent that the fully-trained agent manages to minimize the distance from the end-effector to the goal fast, and this distance stays low for the rest of the episode. It should again be mentioned that the goal and the start configuration of the manipulator are different for each episode, because of the randomization described in Section 4.2.2.

5.4.2 Oscillating line follow task

This task was done to see whether the manipulator can not only go to a specific point and hold the end-effector in this point while under oscillations, but also see whether the manipulator can move the end-effector in a straight path during oscillations.

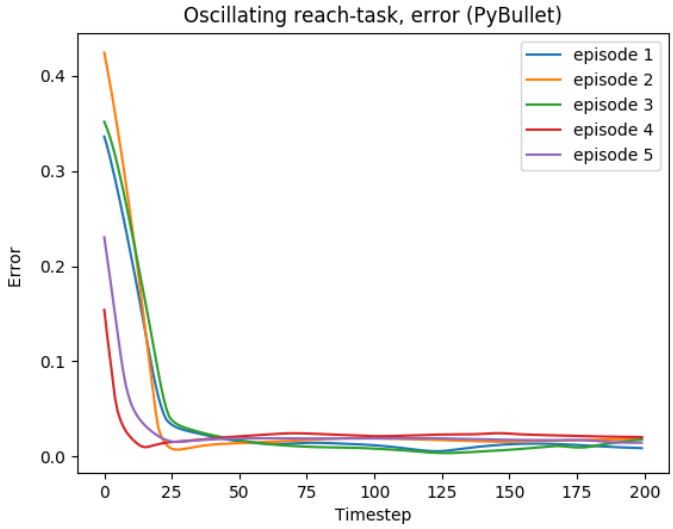


Figure 5.13: A plot of the euclidian distance from the end-effector to the goal for five episodes for an agent that has finished its training on the reach-task with oscillations in PyBullet.

Results

- A single agent was trained using DDPG with the hyperparameters in Table 4.1.
- The agent was trained over 50 epochs, with 20 training episodes in each epoch.
- After each epoch, 30 test episodes were done, where greedy actions were chosen.
- The results from training this task in PyBullet is shown in Figure 5.14a and Figure 5.14b.

In addition to this, five episodes were done where the distance from the end-effector to the line was plotted over the episodes. This distance was calculated according to:

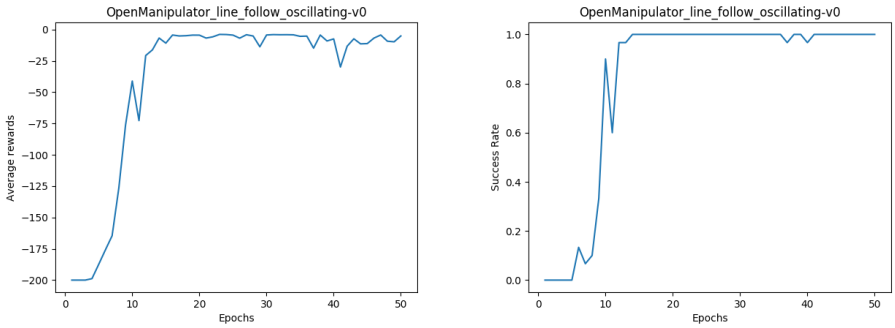
$$d = \begin{cases} \frac{|(\boldsymbol{x}_0 - \boldsymbol{x}_1) \times (\boldsymbol{x}_0 - \boldsymbol{x}_2)|}{|\boldsymbol{x}_2 - \boldsymbol{x}_1|}, & \text{if } -1 < t < 0 \\ \min(|\boldsymbol{x}_0 - \boldsymbol{x}_1|, |\boldsymbol{x}_0 - \boldsymbol{x}_2|), & \text{otherwise} \end{cases}$$

for

$$t = -\frac{(\boldsymbol{x}_1 - \boldsymbol{x}_0)\dot{(}\boldsymbol{x}_2 - \boldsymbol{x}_1)}{|\boldsymbol{x}_2 - \boldsymbol{x}_1|^2}$$

where $\boldsymbol{x}_0$ is the current task-space position of the end-effector, $\boldsymbol{x}_1$ is the start point of the line, and $\boldsymbol{x}_2$ is the end point of the line[1]. These points are given in Cartesian coordinates. The first case where $-1 < t < 0$, gives the distance when the closest point from the end-effector to the line is somewhere between the start and end point of the line. The

[1] This formula was taken from https://mathworld.wolfram.com/Point-LineDistance3-Dimensional.html



(a) Average reward

(b) Success rate

Figure 5.14: Success rate and average reward from training one agent on the line follow task with oscillations in PyBullet.

second case gives the distance when the closest point from the end-effector to the line is either the start point or the end point. This distance can also be considered the error over the episodes. The error is shown in Figure 5.17, and is the error from the video that is delivered with this thesis.

Discussion

It can be seen from the plots in Figure 5.14 that the training stabilizes around 100% success rate fast, after about 15 epochs. It seems like this also is not a very difficult task for the agent to solve. When looking at the error in Figure 5.15, it also seems like the agent manages to hold the end-effector close to the line for the entirety of the episodes although some erratic behavior can be seen for episode 4 around steps 25 and 50 as well as at the end of the episode. When comparing with Figure 5.13, it seems like the oscillations are more apparent in the line following task than the reaching task. However, it should be noted that the scaling on the y-axis is different in both of these plots. These results indicate that a DRL agent controlling a robotic manipulator can control the manipulator to make it follow a straight line, even under oscillations, even though its performance is partially decreased by the oscillations.

5.4.3 Oscillating lever manipulation task

Results

- A single agent was trained using DDPG with the hyperparameters in Table 4.1.
- The agent was trained over 50 epochs, with 100 training episodes in each epoch.
- After each epoch, 30 test episodes were done, where greedy actions were chosen.
- The results from training this task in PyBullet is shown in Figure 5.16a and Figure 5.16b.

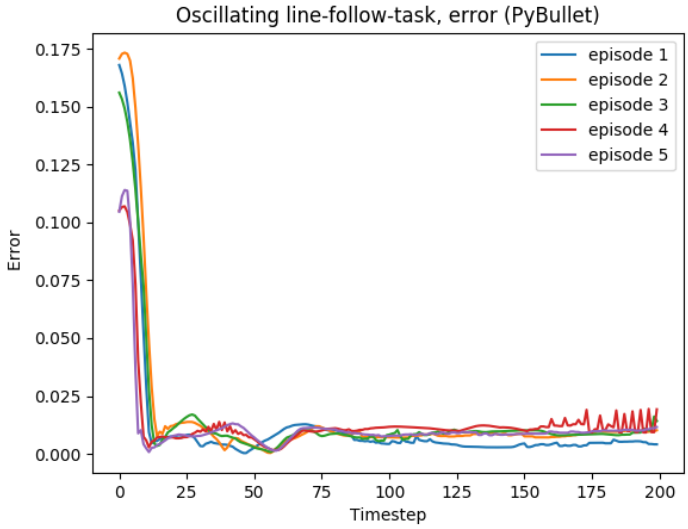


Figure 5.15: A plot of the euclidian distance from the end-effector to the line for five episodes for an agent that has finished its training on the line following-task with oscillations in PyBullet

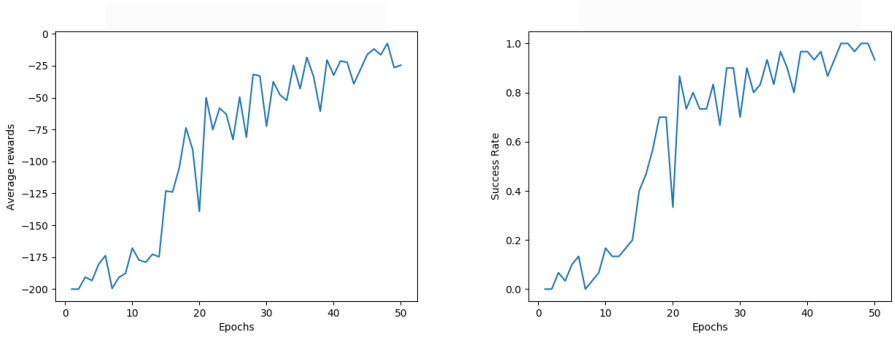
After the training was finished, five episodes were done with random start and goal positions of the lever angle. The results from this can be seen in Figure 5.17 where the error is plotted, and in the video delivered with this thesis, where the episodes that gave the results in Figure 5.17 are shown. In the video, a green sphere spawns each episode, indicating where the goal position of the lever is.

Discussion

This is clearly the most challenging task for the agent to learn that was done in this thesis. The agent requires more training episodes than any of the other environments to achieve success, as can be seen in Figure 5.16 where each epoch contains 100 training episodes. For all the five test episodes after the agent was fully trained, it manages to move the lever very close to the goal angle. However, in episode 3, the agent is struggling to move the lever to the goal position, as can be seen in both Figure 5.17 and the video delivered with this thesis.

This task was originally going to be the main result of this thesis when it was transferred to the real manipulator. However, no device that could create oscillations in the real world was made, and the results are only from PyBullet. The results indicate that it could be possible to do it in a real-world setting, but the transfer from the simulator to the real world would likely be challenging. When viewing the video from this task in the simulations, it seems like the agent learns a behavior that would be very specific to the simulator.

If this task was transferred to the real world, the risk of damaging the manipulator would



(a) Average rewards **(b)** Success rates

Figure 5.16: Success rates and average rewards from training on the lever manipulation task with oscillations in PyBullet

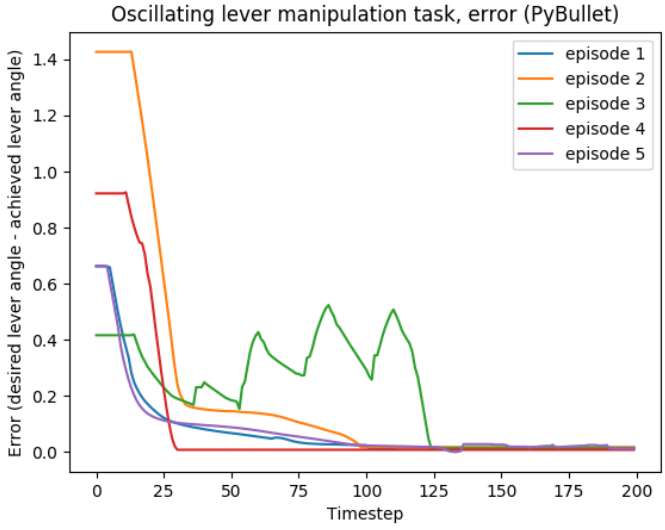


Figure 5.17: A plot of the distance between the achieved lever angle and the desired lever angle over the test episodes

be high. If, for instance, the oscillating device moves the robot's base towards the ground while the end-effector is already close to the ground, this could likely damage the servo motors. To transfer this agent successfully to the real world with less risk of damaging the real manipulator, the reward function could, for instance, give more reward if the manipulator is moved away from the lever when the lever has been moved sufficiently close to the goal. The force and torque that is applied to the manipulator could also be part of the reward function, and the agent could try to minimize this while also achieving the goal.

5.5 Explainable Artificial Intelligence

5.5.1 Results

SHAP values were used to interpret some of the results of this thesis. An agent that was previously trained on the reaching task in the OPM-RL-ENVS was used for this. Two situations were used, and in each of these situations, the agent is just a single step away from the goal. The situations are shown in Figure 5.18a and Figure 5.18b, hereafter called situation one and two, respectively. For the situation in Figure 5.18a, the agent only needs to choose the action [1.0, 0.0, 0.0, 0.0] to reach the goal, which corresponds to moving the first joint $0.2\,\mathrm{rad}$. For the situation in Figure 5.18b, the agent needs to choose the action [0.0, 0.0, 0.0, 1.0], which corresponds to moving the fourth joint $0.2\,\mathrm{rad}$. The SHAP explainer was trained on ten thousand randomly selected poses and goals, which are in the vicinity of these two situations, and then used to evaluate how the different states are used to create the output of the neural network. The local result is shown for situation one is shown in Figure 5.20 and Figure 5.21 show the same for situation two.

More global results are also calculated, by training the SHAP explainer on 10 thousand randomly selected poses and goals that could arise during the agent's operation and showing a summary of how much the different states contribute to the magnitude of the different actions. These results are shown in Figure 5.19.

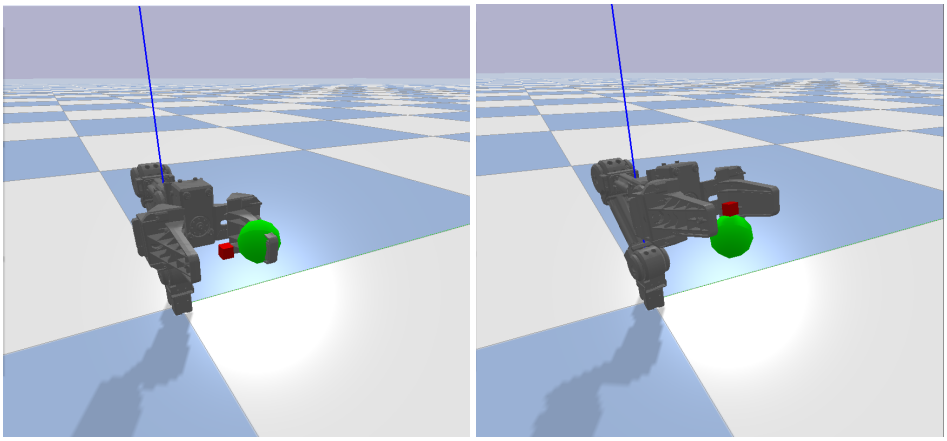
The python library shap[2] was used to calculate the shap values and to plot the graphs [29].

5.5.2 Discussion

The prediction before calculating the SHAP values is that for situation 1, joint 1 will be the essential factor of the agent's decision since this is the only joint that needs to be moved to reach the goal. The same is true for situation 2, except that joint 4 needs to be moved. For the global results, intuition says that all three coordinates of the goal will be principal. It is also predicted that the position of the joints themself will be necessary for how the joint positions should be changed. For example, the position of joint 1 will be relevant for how joint 1 should be moved, and so on. In addition to this since joints 2-4 move the end-effector in the same plane[3], the position of these three joints should be important for

[2] https://github.com/slundberg/shap

[3] for instance if the y-coordinate is zero, changing joints 2-4 will only result in the x and z-position of the end-effector changing.



(a) Situation 1 **(b)** Situation 2

Figure 5.18: The two situations used for XAI, shown in PyBullet

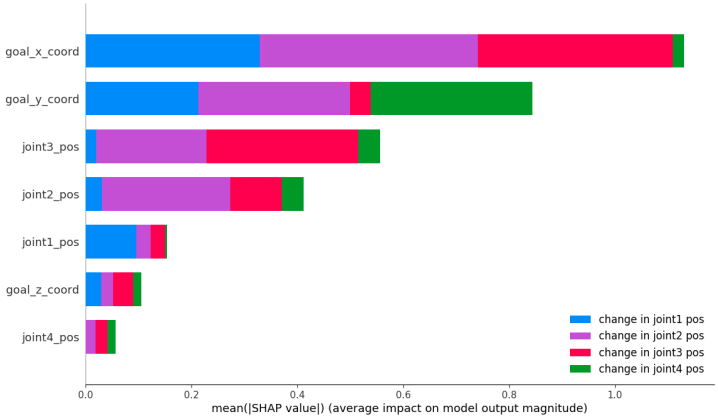


Figure 5.19: The global result from SHAP values

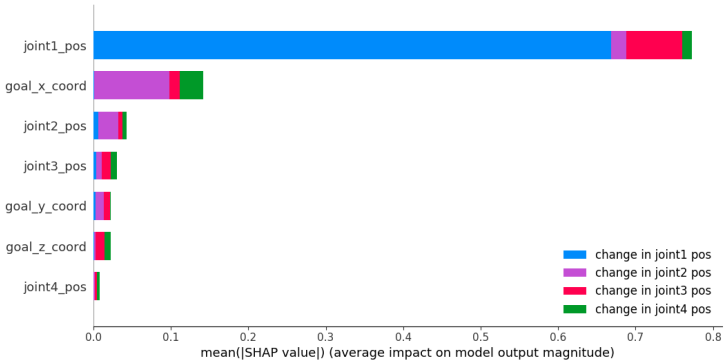


Figure 5.20: Summary of how the states influence the actions for situation 1

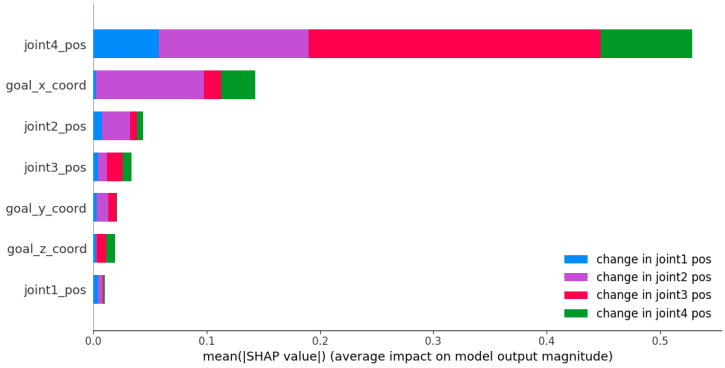


Figure 5.21: Summary of how the states influence the actions for situation 2

how they all should change. Another prediction is that the z-position of the goal will not influence how joint 1 should move because these are not connected in any way since joint 1 is a rotation around the base, which will not influence the z-coordinate of the end-effector.

Figure 5.20 shows that the prediction was accurate. The position of joint 1 is the most important, and it is by far, most contributing to how joint 1 should be changed. What is peculiar is that the position of joint 1 also is important for how joint 3 should change.

Figure 5.21 shows that for situation two, the prediction was also correct. The position of joint 4 is the most important for deciding how the different joints should change. It is also apparent that the agent does not choose the path to the goal that would be most logical (changing only joint 4). However, joint 2 and joint 3 could also be changed to reach the goal in a single step, so the results are not unreasonable. It is nevertheless not good that the position of joint 4 influences how joint 1 should change. Joint 1 is in the perfect position for reaching the goal, regardless of how the other joints are moved to reach the goal, so it should not have been moved at all.

The global result from Figure 5.19 shows something reasonably disconcerting. The z-position of the goal is relatively important for how joint 1 should be changed. As explained above, it is not logical for this to happen, which means that this decreases the trust in the agent's behavior. It is also peculiar that the x-coordinate is vital for how joint 3 should change but not that important for how joint 4 should change; the opposite is true for the y-coordinate, which is vital for how joint 4 should change, but not very important for how joint 3 should change. This is not that disconcerting, because as already mentioned, these two joints change the position of the end-effector in the same plane. It is also peculiar that the z-coordinate influences the actions to a meager extent. It would be logical if how joints 2-4 should be changed was very influenced by the z-coordinate of the goal since these joints influence the z-coordinate of the end-effector regardless of the position of joint 1, but this was not the case.

The trustworthiness of the agent after this experiment has decreased because of this experiment. The prediction that the positions of joints 1 and 4 are crucial for, respectively, situations one and two was correct, which increases the trust. However, the other results discussed here decrease the trust far more. This means that even if the agent has excellent performance on its task, there are reasons not to trust it.

Chapter 6

Conclusion

6.1 Answering the research questions

Which advantages and disadvantages can Deep Reinforcement Learning (DRL) provide when it comes to controlling a robotic manipulator under various conditions?

DRL can be used to solve a wide variety of different problems for robotics, given that the reward function is appropriately made, that the states contain appropriate information and that the actions give the DRL agent sufficient control over the robot. One of the techniques used in this thesis, called Hindsight Experience Replay (HER), dramatically simplifies the reward function's creation. This all together gives DRL the ability to solve problems for robotics, where it is not easy to find a solution using traditional methods—for example, grasping tasks or controlling a robot with a significant number of Degrees of Freedom (DOF).

Another advantage of DRL is that the agent has the potential to improve its performance during operation. This is a significant advantage over traditional methods. However, it would not be recommended to let robots controlled by the DRL algorithms used in this thesis explore during operation. The exploration mechanisms used during this thesis were ϵ-greedy exploration and exploration noise. These mechanisms are unsuitable for a safety-critical operation, especially ϵ-greedy exploration, where the agent has a certain probability of taking a random action. Other DRL algorithms deal with exploration in a more suitable way for online learning during operation—for instance, the Soft Actor-Critic (SAC) algorithm, which tries to maximize entropy while also maximizing the reward [10]. In the future, more DRL algorithms that can explore safely during operation are needed for DRL to help improve robots' autonomy.

One problem with DRL and especially HER is that it is difficult to predict what the agent will do after it has completed its goal. A solution to this could be to make the reward function give more reward if the robot is moved to a safe position after completing its

goal. Another solution can be to use traditional robot control methods to automatically move the robot to this safe position after the goal is done.

Another problem with robotic DRL is related to the gap between the simulator and the real world. If the agent is trained in a simulator, then transferred to the real world, the agent may learn strategies specific to the simulated environment that do not transfer well to the real environment. Solutions to this can be, for instance, to train using the real robot as was done in the paper *The Ingredients of Real World Robotic Reinforcement Learning* [41], or to use techniques such as dynamics randomization [13], which can let an agent estimate the fundamental properties of the real environment.

The most significant problem when it comes to robotic DRL is that agents may not be trustworthy. From the Explainable Artificial Intelligence (XAI) results in this thesis, it is apparent that even if an agent manages to complete its goal approximately every time, this is not enough to trust it. As long as the basis of the agent's decision is different from what human intuition says, it will be challenging to trust the agent. This can lead to a significant roadblock for robotic DRL's widespread acceptance. For more complex tasks than this, it could even be challenging to determine what human intuition would be. These are some of the reasons why more research into XAI for robotic DRL is needed.

Which properties should a simulation framework hold to make the transfer to the real-world efficient for robotic deep reinforcement learning, and what can be done to improve the efficiency of this transfer?

There are two features that simulation frameworks can have that are arguably more important than any else: the simulation speed, and the simulation accuracy. By simulation speed, it is here meant how much faster than real-world speed the simulation can run, and by simulation accuracy, it is meant how similar the simulated environment is to the real-world environment. As mentioned throughout this thesis, the simulators PyBullet and Gazebo has been used in this work. PyBullet showed a high simulation speed but was lacking when it came to simulation accuracy. Gazebo, on the other hand, was accurate compared to the real-world environment, but the simulation speed was very slow.

It may be that these two features are simply at odds with each other; for a simulator to provide more accurate simulation, more calculations have to be made, which takes more time. To do robotic DRL successfully by first training the agent in a simulator and then transfer to the real world, both of these two features should be present. Unfortunately, neither of the simulators used in this thesis had both of these features. In this thesis, this was solved by first training with PyBullet, the faster simulator, and then transfer learning in Gazebo, the more accurate simulator. This is a solution that can prove feasible but requires some extra work compared to just training in a single simulator.

As mentioned during the answer to the last research question, dynamics randomization could likely have been used to make the transfer from the simulator to the real world more efficient. If this had been implemented, it is possible that the agent would only need to train in PyBullet, and then be transferred directly to the real world.

How can methods from XAI be used to interpret the results from this thesis?

For this thesis, the method from XAI that was used was SHapley Additive exPlanations (SHAP). This method can explain how a DRL agent arrives at its decisions by showing how the agent uses different states to decide what action to take. By using domain knowledge, a human can see if the agent makes decisions based on reasonable states. In this thesis, for instance, it was discovered that an agent trained on a reaching task uses the z-coordinate of the goal to decide how to move a joint that does not influence the z-coordinate of the end-effector at all. This, and other results, gave reasons for suspecting that the agent did not decide on the actions based on reasons which follow human intuition. After doing this experiment, the trust in the agent was decreased.
To use SHAP on a DRL problem, the user has to have a good deal of domain knowledge. It may be unfeasible to understand how the agent ideally should arrive at its decisions for tasks that are more involved than this reaching task. This leads to the conclusion that SHAP is useful for DRL tasks where a human can understand how the problem should be solved, and not that useful for more complex tasks. Nevertheless, even for more complex tasks, it should be tried to use SHAP as a tool for a better understanding of how the task should be solved.

6.2 Further work

There is much work that could still be done on this topic. It would be interesting to see how the combination of SAC and HER would perform on the tasks in this thesis. Also, it would have been interesting to see whether the method from *The Ingredients of Real World Robotic Reinforcement Learning* [41] could be applied to an environment such as this, which is somewhat more complicated than the one used in that paper. As already mentioned in this chapter, it would also be interesting to see how dynamics randomization could improve the results of transferring to the real world.

It can also be attempted to transfer the oscillating tasks created for this thesis to the real world. This will likely be a challenging task because of the simulator to the real world gap. If this is going to be done, a device that creates oscillations on the real-world manipulator has to be made.

Computer vision can be used in this topic to detect the position of the lever. End-to-end DRL could also be tried, where the images generated by a camera are directly mapped to actions that the manipulator should perform. This can, for instance, be done by first using Convolutional Neural Networks (CNNs) for feature extraction, and fully connected layers afterwards.

Lastly, more methods from XAI could be used on this topic. For instance, the Linear Model U-trees that are shown in the paper *Toward Interpretable Deep Reinforcement Learning with Linear Model U-Trees* [42]. It was hard to find papers where XAI was used for robotic DRL during the work on this thesis, so this is a subject that needs more research.

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

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. Bellemare, A. Graves, M. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–33, 02 2015.

[5] A. Barredo Arrieta, N. Díaz-Rodríguez, J. Del Ser, A. Bennetot, S. Tabik, A. Barbado, S. Garcia, S. Gil-Lopez, D. Molina, R. Benjamins, and et al., "Explainable artificial intelligence (xai): Concepts, taxonomies, opportunities and challenges toward responsible ai," *Information Fusion*, vol. 58, p. 82–115, Jun 2020. [Online]. Available: http://dx.doi.org/10.1016/j.inffus.2019.12.012

[6] I. Autodesk, *Tinkercad*, 2020 (accessed 27- May- 2020). [Online]. Available: https://www.tinkercad.com/

[7] R. E. Bellman, *Dynamic Programming*. Princeton University Press, 1957.

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

[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] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," 2018.

[11] 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.

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

[13] X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel, "Sim-to-real transfer of robotic control with dynamics randomization," 2018.

[14] J. Su, D. V. Vargas, and K. Sakurai, "One pixel attack for fooling deep neural networks," *IEEE Transactions on Evolutionary Computation*, vol. 23, no. 5, p. 828–841, Oct 2019. [Online]. Available: http://dx.doi.org/10.1109/TEVC.2019.2890858

[15] S. B. Remman, "Deep reinforcement learning for robotic manipulation," 2019, unpublished.

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

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

[18] F. F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological review*, vol. 65 6, pp. 386–408, 1958.

[19] Z. Lu, H. Pu, F. Wang, Z. Hu, and L. Wang, "The expressive power of neural networks: A view from the width," 2017.

[20] F. J. Montáns, F. Chinesta, R. Gómez-Bombarelli, and J. N. Kutz, "Data-driven modeling and learning in science and engineering," *Comptes Rendus Mécanique*, vol. 347, no. 11, pp. 845 – 855, 2019, data-Based Engineering Science and Technology. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1631072119301809

[21] 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

[22] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, "Activation functions: Comparison of trends in practice and research for deep learning," 2018.

[23] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," 2017.

[24] D. Misra, "Mish: A self regularized non-monotonic neural activation function," 2019.

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

[26] H. van Hasselt and M. A. Wiering, "Convergence of model-based temporal difference learning for control," in *2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*, 2007, pp. 60–67.

[27] V. R. Konda and J. N. Tsitsiklis, "Actor-critic algorithms," in *Advances in Neural Information Processing Systems 12*, S. A. Solla, T. K. Leen, and K. Müller, Eds. MIT Press, 2000, pp. 1008–1014. [Online]. Available: http://papers.nips.cc/paper/1786-actor-critic-algorithms.pdf

[28] 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.

[29] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 4765–4774. [Online]. Available: http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf

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

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

[32] *ROS concepts*, (accessed 13- May- 2020). [Online]. Available: http://wiki.ros.org/ROS/Concepts

[33] B. Lorica, *Why AI and machine learning researchers are beginning to embrace PyTorch*, 2017 (accessed 08- June- 2020). [Online]. Available: https://www.oreilly.com/radar/podcast/why-ai-and-machine-learning-researchers-are-beginning-to-embrace-pytorch/

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

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

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

[37] E. Coumans and Y. Bai, "Pybullet, a python module for physics simulation for games, robotics and machine learning," http://pybullet.org, 2016–2020.

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

[39] 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/

[40] T. B. Foundation, *Blender*, 2020 (accessed 27- May- 2020). [Online]. Available: https://www.blender.org/

[41] H. Zhu, J. Yu, A. Gupta, D. Shah, K. Hartikainen, A. Singh, V. Kumar, and S. Levine, "The ingredients of real world robotic reinforcement learning," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=rJe2syrtvS

[42] G. Liu, O. Schulte, W. Zhu, and Q. Li, "Toward interpretable deep reinforcement learning with linear model u-trees," 07 2018.

Sindre Benjamin Remman

Robotic manipulation using Deep Reinforcement Learning

NTNU
Norwegian University of
Science and Technology