Robot Learning From a Human Expert Using Inverse Reinforcement Learning

A Deep Reinforcement Learning Approach for Industrial Applications

Rasmus Eckholdt Andersen, Emil Blixt Hansen, Steffen Madsen

Manufacturing Technology

Master's Thesis

Department of Materials and Production
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Robot Learning From a Human Expert Using Inverse Reinforcement Learning

Theme:
Master's Thesis

Project Period:
Spring Semester 2019

Project Group:
vt4groupc-f19

Participants:
Rasmus Eckholdt Andersen
Emil Blixt Hansen
Steffen Madsen

Supervisor:
Simon Bøgh

Page Numbers: 97

Date of Completion:
June 3, 2019

Abstract:

The need for adaptable models, e.g. reinforcement learning (RL), have in recent years been more present within the industry. However, the number of commercial solutions using RL is limited, one reason being the complexity related to the design of RL. Therefore, a method to identify complexities of RL for industrial applications is presented in this thesis. It was used on 15 applications inspired from four industrial companies. Complexity was especially identified in relation to the reward functions. Thus two Linear Inverse RL (IRL) algorithms in which the reward function is represented as a linear combination of features, was tested using expert data. Some of the tests indicated a visual better result than tests carried out using RL. The process of designing features shared similarities with the process of designing a reward function. The added complexity of implementing Linear IRL and constructing expert data is thus not always a simpler approach. The IRL method GAIL, which requires no feature construction, was furthermore tested showing potential.

Contents

Resumé

Indenfor industriområdet er der begyndt at være et fokus på en samling af teknologier der kan forstærke produktion, både indenfor omkostninger, kvalitet og tilpasning af produkterne. En af de disse teknologier er autonome robotter der bruger modeller der kan tilpasse sig selv til omgivelserne, f.eks. reinforcement learning. Dette speciale undersøger hvordan moderne reinforcement learning metoder kan bruges i industrielle use cases inspireret fra danske virksomheder.

Dette speciale undersøger først Markov Desicion Process (MDP), som er den fundamentale baggrund for reinforcment learning metoder. Dernæst er de første reinforcement learning metoder (såsom Monte Carlo, Q-learning og Deep Q-Networks) undersøgt som kan løse simple diskrete problemer. Ydermere er en række moderne metoder også undersøgt som bruger neurale netværks. Det er inklusiv metoderne Deep Deterministic Policy Gradient, Trust Region Policy Optimisation, Soft Actor-Critic og Guided Cost Learning. Alle disse reinforcment learning metoder er afhængige af at designeren kan lave en reward function der afspejler opgaven. Hvis dette ikke er gjort, kan metoderne ikke løse det givne problem. Derfor er metoder indenfor inverse reinforcment learning undersøgt, da disse bruger data fra en ekspert til at lære den omtalte reward function.

Da reinforcement learning ikke er brugt meget i industrien, er en ny metode kaldet Reinforcement Learning Complexity (RLC) introduceret. Denne bruges til at vurdere de komplekse områder af en MDP, samt fungere som et fundament for en diskussion om et reinforcement learning projekt. RLC-metoden er testet på de inspireret use case fra de industrielle partnere.

På baggrund af analysen er en inspireret use case fra DMRI brugt til at teste problemformuleringen. Problemformuleringen består af tre forskningsspørgsmål, hvor det første omhandler hvordan ekspertdataene kan blive samlet og brugt for DMRI use casen. Det andet spørgsmål omhandler hvordan en software arkitektur og simuleringsmiljø skal strukturers. Det sidste spørgsmål omhandler hvordan inverse reinforcement learning kan bruges, og hvad dens performance er i forhold til traditionel reinforcement learning.

Ekspertdataene er opsamlet med brugen af et HTC VIVE VR-system, hvor en hand-eye kalibrering er lavet for at relatere koordinatsystemet fra VR og robotten. Derudover, er det analyseret at der findes forskellige metoder til at opsamle det såkaldte ekspertdata. De forskellige metoder har hver deres fordele og ulemper. Derfor er konceptet om Direct Task Space Learning introduceret som prøver at få robottens og ekspertens arbejdsrum til at være det samme som opgaven. Derved er det nemt at flytte ekspertdataene til robot uden at gå på kompromis med helheden af dataene.

Softwaren er bygget op omkring ROS og skrevet i Python 2.7. Strukturen er bygget op omkring det standardiseret miljø fra OpenAI Gym. Pakken Keras-rl blev brugt til at standardisere de implementerede inverse- og reinforcement learning metoder. Simuleringsmiljøet Gazebo er brugt og det blev bemærket, at der er nogle stabilitetsproblemer. Derfor er et alternativt miljø ved navn TCP Simulation introduceret. TCP Simulation miljøet bliver brugt til at træne de kinetiske bevægelserne af robot end-effectoren før dynamikken bliver trænet i Gazebo. Hvorefter den trænede policy kan blive brugt på robotten. Dette gør, at den samlet simuleringstid bliver reduceret samt, at de fysiske aspekter stadig bliver trænet.

Forskellige reinforcement learning algoritmer blev først testet på standarde OpenAI Gym miljøer såsom CliffWalking, MountainCar og Pendulum. Ydermere, blev disse miljøer også testet med lineær inverse reinforcement learning metoder ved brug af de trænede reinforcement learning policies til at generere ekspertdata. Dette er gjort for at sammenligne de to metoder og validere om inverse reinforcement learning kan bruges. Derefter blev de optagede ekspertdata brugt til at træne en inverse reinforcement learning agent. Metoden der er brugt, er viapunkter lagt ind i mellem start og målet. Derudover blev der også testet en kvadratisk programmerings metode. Resultaterne viste sig ikke at afspejle eksperten som kan betyde, at metoden med viapunkter ikke virker optimalt.

Preface

This master's thesis is based on the work of the 4^{th} and final semester of the MSc. in Engineering in Manufacturing Technology, during the spring semester of 2019. The project was a part of the Danish research and knowledge sharing robotic community RoboCluster.

The authors would to give a special thanks to Simon Bøgh, the project supervisor, whom throughout both this and previous projects, encouraged the authors to do their best and to write this thesis. Moreover, thanks will go out to members of Simon Bøgh's research group, Nestor Arana Arexolaleiba and Nerea Urrestilla Anguiozar, for always Helpful discussions and knowledge sharing for this thesis.

Reader's Guide

To get the best understanding of this thesis, it is recommended to follow the guide below.

Content

- It is recommended to read the full thesis, following the order kept by the authors.
- This thesis is a research thesis and therefore, much emphasis is spent on the analysis, and thus, a thorough analysis is expected.
- Words or abbreviations written in italic, e.g. *ML*, are keywords of a certain section.

Figures and Tables

- All figures and tables have captions below.
- Figures and tables not made by the authors contains a source in the caption.

Bibliography

- The bibliography is placed after the final chapter of the report before the appendix.
- Entries in the bibliography are ordered alphabetically.
- Each entry contains the following information: authors, year, and title.

- Amount of information depends on type of entry and availability of information.
- Entries in bibliography are referenced using author last name and year.
- Entries directly referenced to in text is without parentheses.
- If the reference is placed before the dot, it is referring only to the specific sentence.
- If the reference is placed after the dot, it is referring to the whole prior paragraph.

Appendix

- Appendices are placed after the bibliography in the end of the report.
- Appendices have assigned capital letters starting from A.
- Appendices are referred in text using the assigned letters.

All the source code, extra material and report can be found in the project repository by scanning the QR-code in Figure 1 or by link: `http://bit.ly/irlVt4Repository`

Figure 1: Project repository: `http://bit.ly/irlVt4Repository`.

Glossary

ACN Actor-Critic Networks

AI Artificial Intelligence

ANN Artificial Neural Network

DDPG Deep Deterministic Policy Gradient

DQN Deep Q-Network

GAIL Generative Adversarial Imitation Learning

GUI Graphical User Interface

IRL Inverse Reinforcement Learning

MDP Markov Decision Process

ML Machine Learning

NN Neural Network

NPC Non-Player-Character

PPO Proximal Policy Optimisation

RL Reinforcement Learning

SAC Soft Actor Critic

SARSA State-Action-Reward-State-Action

SVM Support Vector Machine

TCP Tool Center Point

TD Temporal Difference

TPMT Technology-Push Manufacturing Technology

TRL Technology Readiness Level

TRPO Trust Region Policy Optimisation

Aalborg University, June 3, 2019

Rasmus Eckholdt Andersen
<rean14@student.aau.dk>

Emil Blixt Hansen
<ebha14@student.aau.dk>

Steffen Madsen
<smad14@student.aau.dk>

Chapter 1

Introduction

This master's thesis is created as a part of the Danish robotic network *RoboCluster*, whose primary goal is to share robotic and manufacturing knowledge between company members and educational institutions (RoboCluster Webpage 2019). One of the focuses in RoboCluster is to introduce learning in robotics using adaptable models such as neural networks. This thesis does especially investigate how to transfer human expert knowledge to a robot using Inverse Reinforcement Learning. The motivation for this is to have robots learn from humans to investigate automation possibilities of complex tasks where traditional manufacturing technologies are not sufficient. The following sections introduce concepts within the future of manufacturing technologies, and a brief literature study of existing applications of robot learning.

1.1 Manufacturing Automation

In the era of the 4^{th} industrial revaluation, new demands are, according to Madsen et al. 2014, given to the manufacturing industry which among other is caused by the following factors: globalisation, product regulations, low product life cycles, rapid technological development and customisation. The globalisation is increasing the number of potential competitors within different manufacturing fields. Therefore, manufacturing companies should have increasingly rapid product development and explore new innovative manufacturing technologies to stay competitive. Additionally, the globalisation brings new markets where product regulations are varying among countries. Product life cycles are shortened due to customer demands and rapid technological product development, and customers are at the same time increasingly demanding customised products. The rapid development has moved the limits of the production, which has brought opportunities for new innovative products, services, manufacturing processes and automation technologies. All the above-mentioned factors create a need to continuously develop, test, and implement new products, services, or processes. This is contributing to a dynamic and unpredictable production environment. Thus modern production systems often need to be flexible, reconfigurable, adaptable and at the same time efficient. (Rüßmann et al. 2015)

The potentials in using automation equipment are recognised throughout many different manufacturing fields. Some industries contain processes which are too complicated to be automated with existing automation equipment. Many such

processes are found in the meat industry, since the structure of meat vary significantly and thereby causing a high product variation. The often repetitive and physically demanding processes are therefore easier solved by employing manual labour, than using existing manufacturing equipment. Humans have sophisticated sensory, reasoning, adaptability, and manipulation abilities. Whereas traditional automation equipment has a limited ability to adapt, interpret, and manipulate variations, or changes in a production environment. These challenges are not limited to the meat industry but are also present in other industries such as industrial laundry and robot painting. (Purnell 2013)

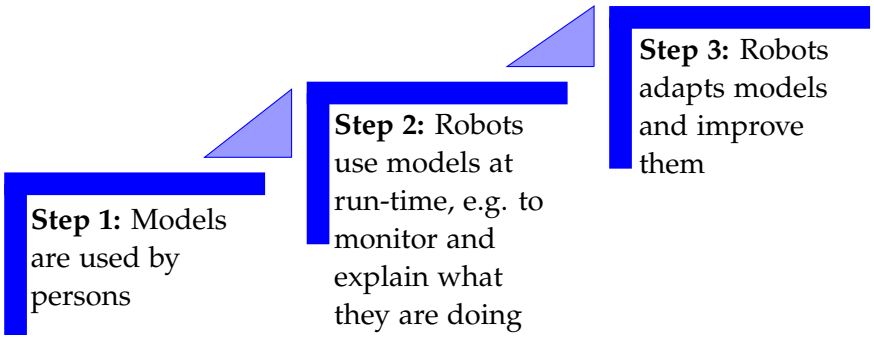


Figure 1.1: An abstract model showing the changes in robotic software development. (SPARC 2016)

The European robotic partnership SPARC has a multi-annual roadmap (SPARC 2016) for robotics in different industries. In this roadmap SPARC specifies different relevant robotic abilities in a manufacturing context which should be a target for research. Three of these abilities have direct relevance to this thesis; *Adaptability*, *Decisional Autonomy*, and *Cognitive Abilities*. *Adaptability* refers to the ability to adapt to a new environment. *Decisional Autonomy* is the ability to act autonomously in a complex and potential unknown environment. *Cognitive Abilities* is when the system can interpret different environments and tasks such that it can execute accordingly. These abilities are used to discoverer the necessary technologies in order to reach the performance needed for a specific robot solution. Figure 1.1 illustrates the three levels of robot development, going from models used manually to adaptable models used by robots. The following section present the concept of adaptable models.

1.2 Adaptable Models & Machine Learning

Throughout the development of new technologies, both in the form of manufacturing, information and communication, the need for adaptable models has become more present to automate, increase revenue, and customer's demands (Geniar 2016) (Yip 2018). An example of an early adaptable model was presented by Åström 1980, where an adaptable PID-controller was used to control ship-tankers through wind and waves. Another example on adaptive models are *Potential Fields*, which are commonly used for motion planning for mobile robots (Choset et al. 2005). Potential Fields has, e.g. been used to enable a mobile robot equipped

with SONAR sensors to navigate an unknown environment successfully (Cosío and Castañeda 2004).

In recent time, where computation power and the availability of data have increased exponentially, *Artificial Neural Networks* (ANN or NN) have become a popular *Machine Learning* (ML) techniques. Companies like OpenAI and DeepMind has shown significant progress in the field of *Artificial Intelligence* (AI) over the last decade, with new methods and publications emerging with a high frequency. DeepMind has demonstrated how *Deep Q-Networks* (DQN) can achieve human level performance in Atari games with just pixels and score as input (Mnih et al. 2015). Silver et al. 2016 from the DeepMind team beat the world champion in the Chinese board-game GO. A year later Silver et al. 2017 presented a new model that learned by playing against itself and successfully beat the model from 2016. Furthermore, the real-time strategy game StarCraft II has a similar story of an adaptable model beating the best players (Vinyals et al. 2019). OpenAI has developed an adaptable *Natural Language* model (named GPT-2) which can write multiline samples of any topic the user gives as input (Radford et al. 2019). The fully trained model of GPT-2 was not released to the public out of fear for malicious use, and consequentially Irving and Askell 2019 described the need for social scientists in AI development.

The different progresses within AI often uses a combination of different ML techniques: *Supervised*, *Unsupervised* and *Reinforcement Learning*. In *Supervised Learning*, a model is trained with context-specific training data, and each element has a label corresponding to a class. Supervised learning algorithms thus adapt its variables to the given training data according to the labels given. Supervised learning is, therefore, an adaptable model; However, it is only adaptable at compile time. *Unsupervised Learning* can be used when there are no distinct labels available for the data. Clustering is a method in unsupervised learning which clusters the data and potentially discover hidden patterns. The last technique of ML is *Reinforcement Learning* (RL), where an *agent* traverses an environment guided by a reward function as illustrated in Figure 1.2. In RL, the agent is not given examples of optimal actions but instead must discover them through trial and error. The agent/environment model is built up of the *Markov Decision Processes* which is described in Section 2.1. (Bishop 2006)

The reward function in RL is generally engineered to a specific task. For many problems, the reward function is not simple to engineer, and thus the general RL problems are hard to solve. In such situations *Inverse Reinforcement Learning* (IRL) can be used. IRL flips the problem by trying to find a reward function that a given policy is trying to optimise instead of the policy optimising the reward function. In many cases, the optimal policy could be an expert doing the task (Russell 1998). An example of this is *Apprenticeship Learning* presented by Abbeel and Ng 2004 where they used it to drive a car simulation by recording expert data from a human driver.

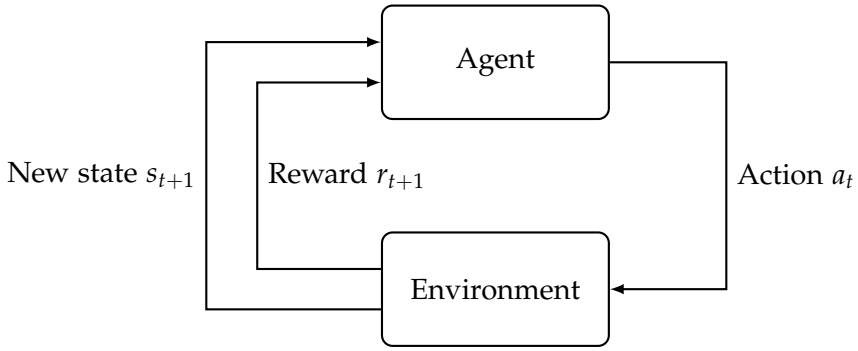


Figure 1.2: The agent/environment model of RL. Here the agent selects an action a_t which interacts with the environment. The environment then outputs a state s_{t+1} and a reward r_{t+1}. Each state can consist of multiple observations, e.g. robot joint values. The reward is a value corresponding to how good the state is, and is often engineered to every RL use case. (Sutton and Barto 2018)

1.3 Existing Applications

In research of RL techniques, games (including video-, board- and card-games) are the go-to platform to test and develop algorithms. Due to the nature of games having a well-defined set of rules and scoring system and thereby making it possible to compare results directly between algorithms. As mentioned, examples of implementation of ML in games are Atari (Mnih et al. 2015), Go (Silver et al. 2017), and StarCraft II (Vinyals et al. 2019). The last two games indicate the most complicated board-game and video-game (regarding strategy) and are a good indication of how far RL has come. Nonetheless, available commercial games where an RL algorithm (controlling, e.g. a Non-Player-Character (NPC)) is lacking. An example of a commercial available video-game that implemented RL is *Creatures* by Grand et al. 1997, seen in Figure 1.3.

Figure 1.3: In game footage of Creatures. (Julia 2013)

Since Creatures, significant advancement has happened to the RL field. Despite this, no real commercial game using RL has been released since then. The reason for this can be many, but most likely, it is the cost of developing RL for games. Game producers might not see the benefits of creating an RL NPC where the game

is not directly focused around that subject, as the case was with Creatures. In the field of industrial applications RL has shown potential within e.g. **maintenance** (Xanthopoulos et al. 2018) (Compare et al. 2018), **motion planning** (Chen et al. 2017) (Peng et al. 2017), **routing** (Khodayari and Yazdanpanah 2005) (Lin et al. 2016), **scheduling** (Gabel and Riedmiller 2007) (Kim et al. 2016), and **technical processes control** (Hafner and Riedmiller 2011). Since this thesis is focused on RL from a robotics point-of-view, a small sub-area of industrial applications with existing RL publications are shown in the following list:

Path Planning
Examples of implementing RL in path planning are presented by Park et al. 2007 and Meyes et al. 2017. RL has proven to be successful in solving path planning in both 2D and 3D environments.

Welding
Robot manipulators are used to a high extent in automated welding processes with examples such as shown by Casler Jr 1986, Lipnevicius 2005, and Lee et al. 2011. Because of the widespread use of robotic welding, this area has also been explored with RL techniques as presented by Takadama et al. 1998, Günther et al. 2016, and Jin et al. 2019.

Pick-and-Place
Pick and place is a fairly used application for robot manipulators and has also received attention from RL approaches with examples presented by Gu et al. 2017, Andrychowicz et al. 2017, and Nair et al. 2017.

Rehabilitation
There exist a considerable amount of research in the field of rehabilitation, with examples presented by Pehlivan et al. 2015 and Vallés et al. 2017. Furthermore, companies such as Life Science Robotics with their product ROBERT has used a collaborative robot manipulator to aid rehabilitation (Life Science Robotics 2019). Examples of applications utilising RL are (Huang et al. 2015) and (Hu and Si 2018).

As of this thesis, there exist no known commercially available solutions that include RL in any of the above-mentioned areas, despite the scientific interest.

1.4 Initial Project Hypothesis

This thesis investigates the usage of RL in a robotic context where expert data is used. The expert data is captured from a specific task performed by an expert and is then used to solve an RL problem. Moreover, this thesis is part of the Danish robotic network *RoboCluster*, which is a collection of companies and research institutions with the primary goal of sharing robotics knowledge and aid the research within the field. Use cases inspired from different RoboCluster companies are analysed, and one is selected as the use case for this thesis. As presented in Section 1.2 and Section 1.3, the number of commercially available solutions incorporating RL is limited, e.g. due to the complexity of designing reward functions.

Some methods have addressed this challenge by using expert data from humans, as was shown with Apprenticeship Learning. From this, the initial hypothesis is formulated as:
The complexity of Reinforcement Learning in use cases from RoboCluster companies can be aided by the use of a human expert.

Chapter 2

Background

This chapter introduces the fundamental theory behind RL problems, i.e. *Markov Decision Process* (MDP). Additionally, tabular methods for solving RL problems are introduced followed by an introduction to value-based, policy gradient and actor-critic RL methods in Chapter 3. The content of this chapter is based on Sutton and Barto 2018.

2.1 Markov Decision Process

The following section gives an introduction to the mentioned *Markov Decision Processes* (MDP), which is the backbone of the RL problem. In general RL, an agent traverses an environment by taking actions, and observing states and numerical rewards. Thus MDPs are a formalisation of the traditional sequential decision making where actions influence the observed states as shown in Figure 2.1. Depending on the environment, there may be a probability that the action execution fails - in Figure 2.1 this is shown as returning to the initial state, however, it could be an entirely new state. A traditional MDP contains the following tuple:

- S: a set of observable states
- A: a set of actions
- $P_{sa}(\cdot)$: the transition probability when taking action a in state s (i.e. a model of the dynamics in the environment - this is only required for model based reinforcement learning)
- $R : (S, A) \rightarrow \mathbb{R}$: a map from states and actions to a single numerical value

The sequential aspect comes when observing an episode of state, action, and rewards: $\{(s_0, a_0, r_0), \ldots, (s_T, a_T, r_T)\}$. The problem RL is trying to solve is to maximise the cumulative reward G, of which the simplest case is shown in Equation 2.2.

$$G = R_0 + \cdots + R_t + \cdots + R_{T-1} \tag{2.1}$$

$$= \sum_{t=0}^{T} R_t \tag{2.2}$$

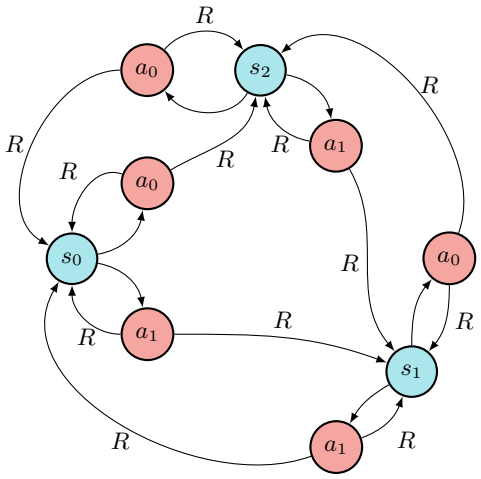


Figure 2.1: The relation between states, actions, and rewards in a Markov Decision Process.

If the MDP is finite, the sets of states, actions and rewards contain only a finite number of T elements. Such a finite MDP is called an episode or trajectory. If there is no guarantee the episode will ever end, or the episode may be significantly large, it can be infeasible to give rewards far into the future the same weight to the current situation as the more immediate rewards. Therefore a discount factor can be added to G to discount rewards which may not affect the immediate steps.

$$G = \gamma^0 R_0 + \cdots + \gamma^t R_t + \ldots \tag{2.3}$$

$$= \sum_{t=0}^{\infty} \gamma^t R_t \tag{2.4}$$

$$= R_0 + \sum_{t=1}^{\infty} \gamma^t R_t \tag{2.5}$$

Where $\gamma \in [0; 1]$ is the discount factor. Thereby G becomes bounded as long as $R \in [R_{min}, R_{max}]$.

Assuming the agent traversing the MDP is optimal (i.e. it will always take the action that maximises Equation 2.5) Equation 2.5 can be used as a value of how good the current state is to be in. Due to the repeating aspect of Equation 2.5, the value of following a policy π can be expressed as shown in Equation 2.6.

$$V^{\pi}(s_t) = R_t + \gamma V^{\pi}(s_{t+1}) \tag{2.6}$$

Here a policy refers to the selection of an action. This selection could, for instance, be based on some probability ϵ of selecting a random action. Similarly, a value for taking an action in state s and thereafter following policy π can be defined as Equation 2.7.

$$Q^{\pi}(s_t, a_t) = R_t + \gamma Q^{\pi}(s_{t+1}, a_{t+1}) \tag{2.7}$$

For such an action value functions, a policy could, for instance, be $\epsilon = 0.1$, i.e. there is a 10% probability of selecting a random action. This means there is 90% probability of selecting the action with the highest action value. In RL, Equation 2.6 and 2.7 are typically estimated from experience, e.g. by keeping an average of the achieved reward successive to a given state. This way of estimating the value-functions is called Monte Carlo methods and is just one of the multiple possible approaches of estimating the value-functions from experience as described in the following section.

2.2 Tabular Methods

As mentioned in Section 2.1, there are multiple approaches to solving a standard MDP. This section describes some of the popular approaches to estimating the value of an action in a given state. Since these methods only estimate a value of an action, and not the action itself, they are referred to as value-based methods in a tabular representation. For these methods to be applicable, the MDP can only have a finite number of selectable actions, such as in video-games (move up, down, left, or right) or the colour of traffic lights in an intersection. Similarly, the states can only contain discrete observations when using tabular methods; however, a value-based method for using continuous observation is presented in Chapter 3. The section starts with the most straightforward method known as Monte Carlo and advances into modern approaches that allow for temporal-difference and off-policy learning.

2.2.1 Monte Carlo Method

The simplest form of estimating the value of an action is to average the reward received when being in a state and selecting an action. After each episode, the estimate can be updated in order to converge to the optimal values. Algorithm 1 shows an implementation of a Monte Carlo method where it can be seen how the average of the reward is calculated. G_t can be calculated using Equation 2.2.

This type of Monte Carlo is called *first visit Monte Carlo prediction*. It is also possible to count every time a state is encountered by omitting **line 6** in Algorithm 1 (thereby becoming *every visit Monte Carlo*). Every visit Monte Carlo extends itself more naturally to general function approximation. As it can be seen in Algorithm 1, the Monte Carlo method requires the possibility to count the number of times a state and action has been encountered. This means this method is only applicable when the environment can be discretised to a grid-space.

Algorithm 1 First-Visit MC Prediction (*for action values*).

Input: Policy π, positive integer *num_episodes*
Output: Value function Q ($\approx q_\pi$ if *num_episodes* is large enough)

Initialise $N(s,a) = 0$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Initialise $return\_sum(s,a) = 0$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
for $i \leftarrow 1$ **to** *num_episodes* **do**
    Generate an episode $S_0, A_0, R_1, \ldots, S_T$ using π
    for $t \leftarrow 0$ **to** $T-1$ **do**
        if (S_t, A_t) *is a first visit* (*with return* G_t) **then**
            $N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$
            $return\_sum(S_t, A_t) \leftarrow return\_sum(S_t, A_t) + G_t$
        end if
    end for
end for
$Q(s,a) \leftarrow return\_sum(s,a)/N(s,a)$ for all $s \in \mathcal{S}, a \in \mathcal{A}$
return Q

2.2.2 Sarsa Method

One of the limitations of Monte Carlo methods is the fact that the agent can only learn after each episode. This can be circumvented by applying temporal-difference (TD) learning as done in Sarsa methods. Thus Sarsa and Monte Carlo methods are similar but with the difference that Sarsa will update the action value estimation after each timestep instead of after each episode.

Temporal-difference is a way of optimising the value estimation by minimising the temporal-difference error. This error comes from the recursive function in Equation 2.7 where the action value of the current state s_t should be equal to the next reward plus the discounted action value estimated by the same function. If this is not the case, the action value can be updated by minimising this error as shown in Equation 2.8.

$$\delta = R_t + \gamma Q^\pi(s_{t+1}, a_{t+1}) - Q^\pi(s_t, a_t) \tag{2.8}$$

The current estimate $Q(s_t, a_t)$ can then be updated by adding this error as in Equation 2.9.

$$Q^\pi(s_t, a_t) = Q^\pi(s_t, a_t) + \alpha \overbrace{\left(R_t + \gamma Q^\pi(s_{t+1}, a_{t+1}) - Q^\pi(s_t, a_t)\right)}^{\text{TD error } \delta \text{ from Equation 2.8}} \tag{2.9}$$

Where α is a scaling factor in order to avoid oscillation. Much like with Monte Carlo methods, the Sarsa methods requires a finite number of actions to choose from since it will estimate a value for each action. The policy π will then select which action to choose based on the estimated action value.

2.2.3 Q-learning

Q-learning is similar to Sarsa in the sense that it also uses temporal-difference learning, but instead of using the value determined by the policy π when performing the temporal update in Equation 2.8 it uses the action that maximises the accumulated reward, i.e. it follows a greedy policy for $Q^{\pi}(s_{t+1}, a_{t+1})$ as shown in Equation 2.11.

$$\delta = R_t + \gamma \max_{a_{t+1}} Q^{\pi}(s_{t+1}, a_{t+1}) - Q^{\pi}(s_t, a_t) \tag{2.10}$$

$$Q^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) + \alpha \overbrace{\left(R_t + \gamma \max_{a_{t+1}} Q^{\pi}(s_{t+1}, a_{t+1}) - Q^{\pi}(s_t, a_t) \right)}^{\text{TD error } \delta \text{ from Equation 2.10}} \tag{2.11}$$

Thus Sarsa and Q-learning are very similar with only a small difference in their update procedure. They also share the same limitations of requiring a finite number of actions and a finite number of states.

Chapter 3

State of the Art

In this chapter, the variety of presented methods are extended to include those which can solve problems in domains with arbitrary large state or action spaces, unlike the tabular methods described throughout Section 2.2. Instead of discretising a continuous RL problem, an alternative solution is to obtain a continuous function approximation of the value function. In this thesis, this process will be referred to as function approximation. In an RL problem with an infinite state-space, it is necessary to generalise states, such that decisions can be made based on previously visited states which are similar to the current. This is necessary while most possible states in such problem are never encountered more than once if at all. The aim is to generalise based on examples obtained from previously visited states to estimate e.g. a value function. The problem of generalising and estimating functions is one of the critical problems, related to arbitrary large state and action spaces. This chapter describes methods which use general function approximators to solve an MDP. (Sutton and Barto 2018)

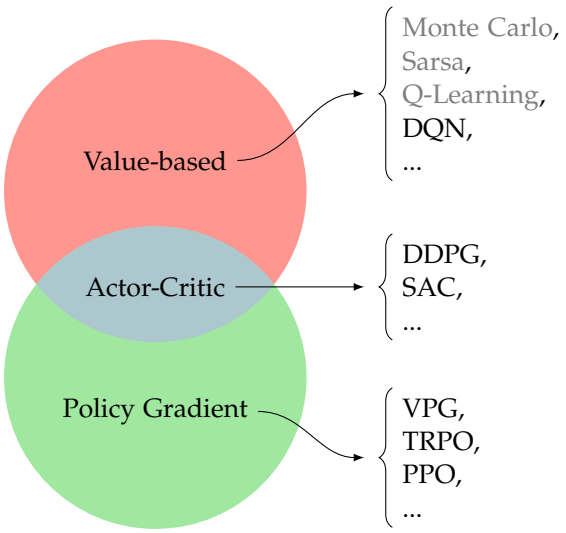


Figure 3.1: The three main categories of reinforcement learning. Note Monte Carlo, Sarsa, and Q-Learning are described in Section 2.2, but are included here for completeness.

Figure 3.1 gives an overview of the three main categories of RL described in this chapter. The first category is the already mentioned value-based method from Chapter 2. The following sections present additional value-based methods that overcome some of the limitations mentioned in Section 2.2. The next category is policy gradient methods, which operates by learning probability distributions of the optimal actions. These methods do, therefore, not necessarily rely directly on

a value function. The last category, actor-critic methods, are a combination of both value-based methods and policy gradient methods where a value function is used to update a policy. One of the advantages of these methods is that they allow for estimating continuous actions directly.

Additionally, Section 3.4 introduces Inverse Reinforcement Learning (IRL), and finally, a summary of the presented methods can be found in Section 3.5.

3.1 Value-Based Methods

As mentioned, the disadvantage with the tabular methods from Section 2.2 is the sensitivity to the number of dimensions in the state-space and action-space. This section introduces a generalisation of the Q-learning algorithm using a general function approximator.

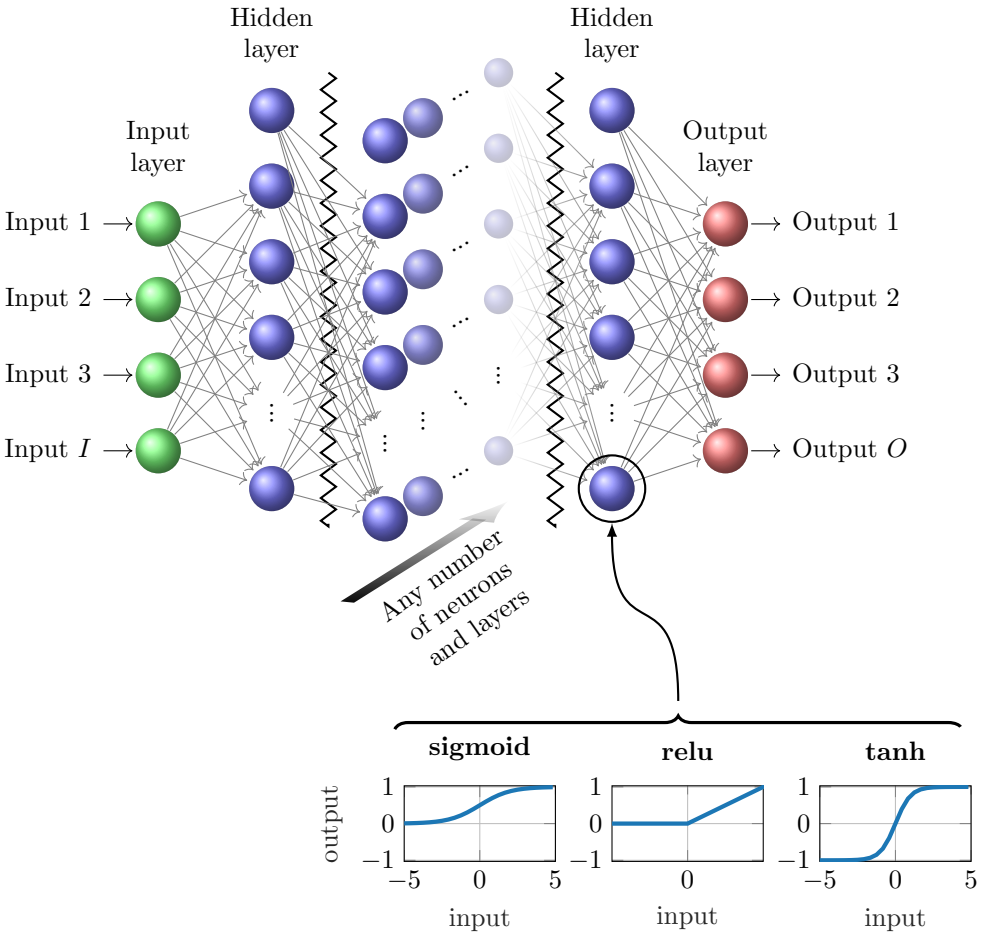


Figure 3.2: A general setup of a neural network. The function variables are represented by arrows to each node. The nodes can optionally contain an activation function that introduces non-linearity. The number of hidden layers and nodes in each hidden layer is a design choice.

Deep Q-Learning

The only value-based method using a general function approximator considered in this thesis is an extension of Q-learning called Deep Q-learning or Deep Q-network (DQN). Here a generic function approximator in the form of an NN is used to estimate the action-value. A general NN can be seen in Figure 3.2, where the inputs are observations from the current state and the outputs are the action-value of each possible action.

Though this method uses a general function approximator instead of a tabular, the methods are equivalent except for the update step. Using a general function instead of a tabular method means that the TD update from Equation 2.9 and 2.11 is not possible. Instead, the TD update becomes a minimisation problem to minimise the TD error in Equation 2.8 as shown in Equation 3.1 with respect to the network parameters θ.

$$\underset{\theta}{\text{minimise}} \quad R_t + \gamma \max_{a_{t+1}} Q^\pi(s_{t+1}, a_{t+1}|\theta) - Q^\pi(s_t, a_t|\theta) \tag{3.1}$$

The advantage of using a function approximator is that it is easily scalable with the size of the state, i.e. the inputs can both be continuous and discrete and more inputs can be added without changes to the underlying optimisation.

Even though NN has the potential to approximate complex underlying functions, making them converge to optimal solutions can be difficult. Therefore several improvements have been developed to stabilise the convergence. The following is a short list of some of the popular improvements.

Target Networks (TN): The optimisation step in Equation 3.1 may make the neural network unstable as the weights are changed due to the recursive property of the action-value function in Equation 2.7. To limit this, a separate target network with parameters θ' is used to compute the action-value for state s_{t+1} as shown in Equation 3.2.

$$Q^\pi(s_t, a_t|\theta) = R_t + \gamma \max_{a_{t+1}} Q^\pi(s_{t+1}, a_{t+1}|\theta') \tag{3.2}$$

The parameters of the target network can then be updated by setting $\theta' \leftarrow \theta$ after a given number of timesteps, or it can be set gradually after every timestep as $\theta' \leftarrow \alpha\theta' + (1-\alpha)\theta$. (Mnih et al. 2015)

Dueling Networks: Instead of estimating the action-value function, the neural network consists of two streams - one for estimating the state-value and one for estimating an action advantage. A visual representation can be seen in Figure 3.3. (Wang et al. 2015)

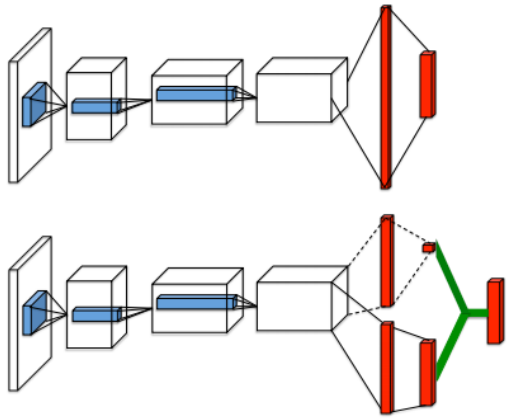


Figure 3.3: (top) A traditional Q-network. **(bottom)** A dueling network with the advantage and state-value estimation. (Wang et al. 2015)

Double Networks: Q-networks are known to overestimate the value function as shown by Hasselt et al. 2015. In the same paper, they propose to use a separate network to compute the TD target shown in Equation 3.2. They do this by first rewriting it as shown in Equation 3.3 and then selecting the action with respect to the online network instead of the target network, as shown in Equation 3.4.

$$\text{(DQN + TN) } Q^{\pi}(s_t, a_t|\theta) = R_t + \gamma Q^{\pi}\left(s_{t+1}, \arg\max_{a_{t+1}} Q^{\pi}\left(s_{t+1}, a_{t+1}|\theta'\right) \middle| \theta'\right) \quad (3.3)$$

$$\downarrow$$

$$\text{(Double DQN) } Q^{\pi}(s_t, a_t|\theta) = R_t + \gamma Q^{\pi}\left(s_{t+1}, \arg\max_{a_{t+1}} Q^{\pi}\left(s_{t+1}, a_{t+1}|\theta\right) \middle| \theta'\right) \quad (3.4)$$

This means that the action is selected with respect to the online network, but the value of that action is computed using the target network. (Hasselt et al. 2015)

Multi-Step Reinforcement Learning: So far all the shown algorithms have assumed a single TD update. Asis et al. 2017 proposes an algorithm to compute the TD error by looking n-steps ahead. This gives a TD error based on multiple steps instead of a single step. In practice, Equation 3.1 is updated to Equation 3.5.

$$\underset{\theta}{\text{minimise}} \quad \sum_{k=0}^{n-1}\left(\gamma^k R_{t+k}\right) + \gamma^n \max_{a_{t+n}} Q^{\pi}(s_{t+n}, a_{t+n}|\theta) - Q^{\pi}(s_t, a_t|\theta) \quad (3.5)$$

This allows the agent to use its experience data more efficiently.

Prioritized Experience Replay: Updating a Q-network can be a slow process that requires multiple passes with the same data before any significant improvements. Additionally, the ability to replay state transitions that generated a high accumulated reward can increase performance, as shown by Mnih et al. 2015. Initially, the experience is sampled uniformly from the replay buffer, whereas newer methods prioritise the experience to replay according to the TD error. Since the TD error indicates the current accuracy of the value estimation, there is more to learn from transitions with high TD errors. By replaying transitions with a high TD error more frequently, the accuracy of the value estimation is improved. (Schaul et al. 2016)

Noisy Nets: Having a constant epsilon-greedy exploration introduces a non-learnable parameter, which means that the agent's sentiment to explore is uniform for the entire state-space. To avoid this, Fortunato et al. 2017 introduces noise directly on the NN layers, which is updated with gradient descent. This allows the agent to learn how to ignore the noisy stream. Since the noisy parameters are updated using gradient descent over time, the noise decays at different rates in the state-space while allowing for a form of annealing exploration. The noisy net exploration is added to the traditional parameters in a layer of a network, as shown in Equation 3.6.

$$y = \underbrace{b + Wx}_{\text{Traditional layer}} + \underbrace{b_{noisy} \odot \epsilon^b + \left(W_{noisy} \odot \epsilon^W\right) x}_{\text{Noisy Net}} \tag{3.6}$$

Where x is the input to the layer, W and b are the weights and bias of the layer, W_{noisy} and b_{noisy} are learnable parameters for the noise (i.e. a scaling of the noise ϵ^W and ϵ^b). Note that $\odot$ denotes element-wise multiplication. Lastly, y is the preactivation values of the given layer.

Each of these improvements has been combined in a single Deep Q Network agent set to learn Atari2600 games. The algorithm has the name Rainbow and currently outperforms all existing implementations of DQNs on the Atari2600 game benchmark. (Hessel et al. 2017)

3.2 Policy Gradient Methods

Instead of estimating the actions based on a value, a probability distribution can be used to determine which action to select in a given state. This approach is referred to as policy gradient methods and works for both discrete and continuous action spaces. Policy gradient methods aim to learn some policy parameters θ by using the gradient of a performance measure $J(\theta)$ concerning the policy parameters. From these policy parameters, a stochastic policy π_θ is generated with a probability distribution over action probabilities given observations. Following a policy is thus equal to sampling an action from this probability distribution. This stochastic

policy results in smooth changes in the probability distribution when the policy is updated, unlike value-based methods where small changes can result in a dramatic change. (Sutton and Barto 2018)

The goal in policy gradient methods is to find a policy which maximises the expected sum of rewards over all state-action sequences, i.e. $J(\theta)$ in Equation 3.8.

$$J(\theta) = \mathbb{E}\left[\sum_{\tau} R_\tau \middle| \pi_\theta\right] \tag{3.7}$$

$$= \sum_{\tau} P(\tau|\theta) R_\tau \tag{3.8}$$

Where τ denotes a state-action sequence $\{(s_0, a_0), ..., (s_T, a_T)\}$ as defined in Section 2.1, R_τ is equal to the sum of rewards in a state-action sequence (in this case $\gamma = 1$, $R_\tau = \sum_{t=0}^{T} R_t$), π_θ is the current policy and $P(\tau|\theta)$ is the probability of τ when having the policy parameter θ.

The gradient of the performance measure $J(\theta)$ is equal to the expected value of the log probability of the state-action sequences τ multiplied by R_τ as seen in Equation 3.10. This gradient is called the likelihood ratio policy gradient.

$$\nabla_\theta J(\theta) = \sum_{\tau} P(\tau|\theta) \nabla_\theta \log P(\tau|\theta) R_\tau \tag{3.9}$$

$$= \mathbb{E}[\log P(\tau|\theta) R_\tau] \tag{3.10}$$

Here $\nabla_\theta \log P(\tau|\theta)$ is equal to $\nabla_\theta \log \sum_{t=0}^{T} \pi_\theta(s_t, a_t)$, i.e. the transition probabilities do not depend on θ and thus disappears in the gradient. While, the gradient of $J(\theta)$ can be expressed as an expected value it can be estimated using sample paths under a specific policy as seen in Equation 3.11.

$$\nabla_\theta J(\theta) \approx \hat{g} = \frac{1}{m} \sum_{i=1}^{m} \nabla \log P(\tau|\theta) R(\tau^{(i)}) \tag{3.11}$$

In that way, a gradient is estimated, as a function of θ which points in the direction which increases $P(\tau|\theta)$ for state-action sequences with a positive reward and decreases sequences with a negative reward. This estimate does, however, in general produces a significant variance and thereby a low convergence time. The following sections contain examples on algorithms which strives to improve the above policy gradient principles by, e.g. lowering the produced variance in the gradient.

3.2.1 Reduction of Gradient Variance & Vanilla Policy Gradient

Baselines and temporal structures are two methods which can be applied for lowering the variance and noise in the likelihood ratio policy gradient estimate, presented in Equation 3.11.

When estimating the gradient, a baseline value can be subtracted from $R(\tau^{(i)})$, as seen in Equation 3.12.

$$\nabla_\theta J(\theta) \approx \hat{g} = \frac{1}{m} \sum_{i=1}^{m} \nabla \log P(\tau|\theta)(R(\tau^{(i)}) - b) \tag{3.12}$$

Where b is the baseline value e.g. the average of the sum of rewards, $\frac{1}{m} \sum_{i=1}^{m} R(\tau^{(i)})$. The baseline is used to centre the estimate around a baseline, such that the probability of a state-action sequence is increased if the representative reward is above the baseline and decreased if it is below the baseline. The baseline can e.g. be a constant value, time-dependent or dependent on the current state. The aim of a state-dependent baseline is to estimate the expected return of being in a state when following a policy, i.e. a state-dependent baseline can be seen as a state value function, $V^\pi(s_t)$ which is updated using sample trajectories.

The variance in the gradient estimate can be further lowered by looking at the temporal structure of the estimate. By decomposing Equation 3.12 into Equation 3.13 it can be seen that the sum of rewards over a whole trajectory is multiplied with $\nabla \log \pi_\theta(s_t, a_t)$ at each timestep t i.e. the sum of rewards thus contributes to the estimation at each timestep. Removing the terms, in Equation 3.13, which is independent from the current action a_t the variance in the gradient estimate can be lowered.

$$\hat{g} = \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{T} \nabla \log \pi_\theta(s_t, a_t) \left(\left(\sum_{k=0}^{t-1} R(s_k, a_k) \right) + \left(\sum_{k=t}^{T} R(s_k, a_k) \right) - b \right) \tag{3.13}$$

While, the probability of taking actions at state T is independent from the reward obtained at e.g. t_2 the term $\sum_{k=0}^{t-1} R(s_k, a_k)$ can be removed from Equation 3.13, and Equation 3.14 is thus obtained.

$$\hat{g} = \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{T} \nabla \log \pi_\theta(s_t, a_t) \left(\sum_{k=t}^{T} R(s_k, a_k) - b \right) \tag{3.14}$$

The two above principles are both incorporated in the policy gradient method called *Vanilla Policy Gradient* (VPG). Which can be found in Algorithm 2. VPG is simple but has limitations when it comes to data efficiency, robustness and finding a gradient stepsize α. (Schulman et al. 2017)

Algorithm 2 Vanilla Policy Gradient. (Sutton and Barto 2018) (Schulman et al. 2017)

1: **for** $k = 0, 1, 2, ...$ **do**

2: Collect m state action-sequences using the current policy π_θ.

3: Compute the discounted reward at each timestep and for each sequence:

$$R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$$

4: Compute the advantage estimates $\hat{A}$ at each timestep for each sequence:

$$\hat{A} = R_t - b_\phi(s_t)$$

5: Estimate the policy gradient:

$$\hat{g}_k = \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{T} \nabla \log \pi_\theta(s_t^{(i)}, a_t^{(i)}) \hat{A}_t{}^{(i)}$$

6: Update the policy parameters using the gradient e.g. by gradient ascent:

$$\theta_{k+1} = \theta_k + \alpha \hat{g}_k$$

7: Update the baseline value, $b_\phi(s_t)$, e.g. using mean square error.

$$\phi_{k+1} = \arg\ \min\ \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{T} \left(b_\phi(s_t^{(i)}) - \left(\sum_{k=t}^{T} R(s_k^{(i)}, a_k^{(i)}) \right) \right)^2$$

8: **end for**

3.2.2 Trust Region Policy Optimisation

This section presents a policy gradient method presented by Schulman et al. 2015 called *Trust Region Policy Optimisation* (TRPO). TRPO uses the principles of trust region optimisation to control the stepsize used when updating a policy using a gradient. The presented method was demonstrated to have a robust performance in a wide range of tasks, e.g. simulated robotic swimming and hopping.

The gradient used in the vanilla algorithm presented in the previous section is updated by taking a step with a gradient which can be written according to Equation 3.15.

$$\hat{g} = \mathbb{E}\left[\nabla \log \pi_\theta(s_t, a_t) \hat{A} \right] \tag{3.15}$$

Where $\hat{A}$ is an advantage estimate, e.g. $\hat{A} = R_t - b_\phi(s_t)$ as in Algorithm 2. This gradient is in TRPO formulated as a cost function as seen in Equation 3.16.

$$L^{PG} = \mathbb{E}\left[\log \pi_\theta(s_t, a_t)\hat{A}\right] \tag{3.16}$$

Optimising a policy using this objective function, e.g. as in the vanilla, do however according to Schulman et al. 2017 often results in large and destructively policy updates, and it is therefore not well justified to do.

The presented method does, therefore, use another formulation of the objective function. The function uses the theory of importance sampling, and a KL diverges constraint on the size of a given policy update, as see in Equation 3.18.

$$\underset{\theta}{\text{maximise}} \quad \mathbb{E}\left[\frac{\pi_\theta(s_t, a_t)}{\pi_{\theta_{old}}(s_t, a_t)}\hat{A}\right] \tag{3.17}$$

$$\text{s.t.} \quad \mathbb{E}\left[KL[\pi_{\theta_{old}}(s_t, a_t), \pi_\theta(s_t, a_t)]\right] \leq \delta \tag{3.18}$$

Where $\pi_{\theta_{old}}$ is the old policy, π_θ is the current, and KL is a measure of difference between the two policies. In this way, the optimisation can only happen in a region where the constraint is fulfilled, i.e. large and destructive policy updates are restricted.

3.2.3 Proximal Policy Optimization

One of the difficulties with TRPO is the KL diverge constraint which adds a large overhead on the optimisation. In order to avoid this computationally heavy constraint (Schulman et al. 2017) presented a method called *Proximal Policy Optimization* (PPO) that modifies the objective function such that the KL-divergence constraint is no longer necessary, as seen in Equation 3.19.

$$L^{PG} = \mathbb{E}\left[min\left(\underbrace{\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{Old}}(a_t|s_t)}\hat{A}}_{\text{Undo bad gradients}}, \underbrace{clip\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{Old}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon\right)\hat{A}}_{\text{Clip gradient step size}}\right)\right] \tag{3.19}$$

This new objective function is similar to the objective function of TRPO in Equation 3.17, but consists of three parts; namely the *min* operator, the same objective function as in Equation 3.17, and a clipping of the objective function. The clipping function ensures not to take too large gradient steps, i.e. it is not possible to improve more than a proportion of ϵ over the old policy. The *min* operator selects the term that has the lowest value between the gradient and the clipped gradient. In practice, this happens when the current policy has a higher probability of an action than the old policy, while at the same time producing a negative advantage value $(\hat{A})$. This means that the policy now wants to select actions that produce a smaller accumulated reward which is not desired. Therefore the *min* operator

ensures that the policy does not take too large gradient steps. While at the same time allows the policy to recover from policies that increase the probability of an action that produces a negative advantage. Since the advantage will be negative and thus move the gradient in the opposite direction (i.e. undoing the original failed gradient update).

3.3 Actor-Critic Network

This section covers two versions of an *actor-critic network* (ACN), which like the policy gradient methods overcomes the limitation of value-based algorithms by operating in a continuous action-space. An ACN consists of two networks, an actor which learns the policies and a critic which evaluates the actions chosen by the actor. The critic is learning the action-value function for the actor's current policy, through a TD algorithm, this allows the critic to criticise the actor. The basic principles of an ACN can be seen in Figure 3.4. (Sutton and Barto 2018)

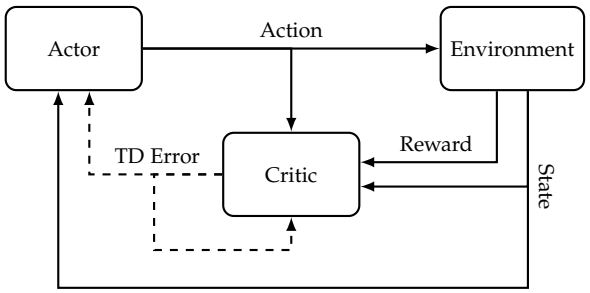


Figure 3.4: Basic principles of an actor-critic network setup.

One of the significant advantages of an actor-critic setup over a strictly value-based method or a strictly policy gradient method is the ability to estimate continuous actions while not suffering from the instability of the simple policy methods.

There do exist numerous actor-critic setups, such as Policy Gradient with Function Approximations (Sutton et al. 1999), Sample Efficient Actor-Critic with Experience Replay (Wang et al. 2016), and Asynchronous Actor-Critic (Mnih et al. 2016). The two actor-critics investigated in this section are a *Deep Deterministic Policy Gradient* method and a *Soft Actor-Critic* method. Though they do share many similarities, some key differences make their performance vary.

3.3.1 Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient (DDPG) is an algorithm, presented by Lillicrap et al. 2015 which combines policy gradient methods with the recent success of the DQN. Specifically, the critic is a standard DQN with the only difference being that instead of estimating the action value for each action based on observations. It estimates an action value for an action given as input - i.e. the output is a single

value. The action inputs to the DQN are policy values. A small example of a DDPG network is shown in Figure 3.5.

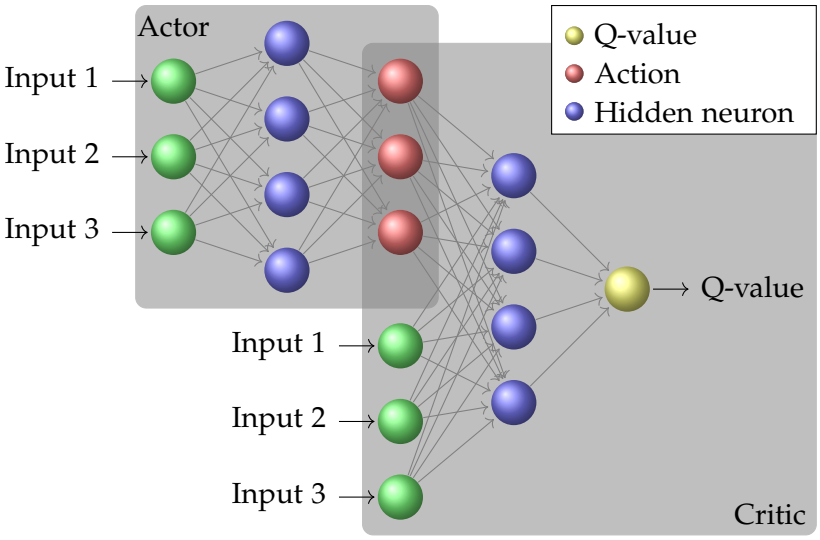


Figure 3.5: An ACN showing the relationship between the actor and the critic.

From Figure 3.5 it can be seen how the critic is a function of the actor. Using this property, Lillicrap et al. 2015 proved that the gradient of the network with respect to the parameters of the actor can be approximated using the chain-rule and a batch of N states as shown in Equation 3.20.

$$\nabla J \approx \frac{1}{N} \sum_{i=1}^{N} \nabla Q(s_i, \mu(s_i)|\theta^Q) \nabla \mu(s_i|\theta^\mu) \tag{3.20}$$

In other words; this gradient will move the actor in the direction that maximises the critic, which in turn is guided by the minimisation of the TD error as shown in Equation 2.8 and 2.9.

The complete algorithm for implementing a DDPG is shown in Algorithm 3. The advantage of DDPG is that the policy can predict action values directly, thus they can be continuous. This is useful for robotic tasks that often are difficult to discretise without loss of information.

3.3.2 Soft Actor-Critic

A more recent algorithm for continuous action control is the *Soft Actor-Critic* (SAC) algorithm (Haarnoja et al. 2018). Instead of only maximising the accumulated reward like with traditional DQNs or even DDPG, SAC tries to additionally maximise the entropy of the policy producing the actions. In short, entropy is a measure of how predictable a random variable is. An example of the entropy of a constant variable is zero since it is easy to predict what value the constant is going to take, whereas a random variable from an interval has a higher entropy depending on

Algorithm 3 DDPG Algorithm with single timesteps. (Lillicrap et al. 2015)

Initialise critic $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ
Initialise target critic Q' and target actor μ' with weights $\theta^{Q'} \leftarrow \theta^Q$ and $\theta^{\mu'} \leftarrow \theta^\mu$
for $e = 1$, M **do**
  for $t = 1$, T **do**
    Get action $a_t = \mu(s|\theta^Q) + \mathcal{N}$ ▷ $\mathcal{N}$ is some noise on the action
    Execute action a_t in environment, observe r_t and s_{t+1}
    Store transition (s_e, a_e, r_e, s_{t+1}) in R
    Sample N random 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 the critic by minimising the loss:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}\Big(y_i - Q(s_i, a_i|\theta^Q)\Big)^2$$

    Update the actor policy using the sampled policy gradient:

$$\nabla J \approx \frac{1}{N}\sum_{i=1}^{N}\nabla Q(s_i, \mu(s_i)|\theta^Q)\nabla\mu(s_i|\theta^\mu)$$

    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
return Q

the distribution its sampled from. By maximising the entropy, SAC ensures that the policy will always explore actions that have similar Q-values and thus the risk of ending in a local minimum is decreased.

The objective function for SAC is thus modified from the standard sum of expectation of the rewards as in Equation 2.5 to be a more general objective function containing an entropy term $\mathcal{H}$ as shown in Equation 3.21.

$$J(\pi) = \sum_{t=0}^{T}\mathbb{E}[R_t + \alpha\mathcal{H}(\pi(\cdot|s_t))] \tag{3.21}$$

As already mentioned, SAC uses both a critic and a policy network like DDPG, and this is in principle all that is needed, but according to (Haarnoja et al. 2018), keeping a separate network for the value function (see Equation 2.6) helps with convergence. Thus the objective functions J for the value function, critic, and actor are as shown in Equation 3.22-3.24.

$$J_V = V(s_t) - Q(s_t, a_t) - log(\pi(a_t|s_t)) \quad \text{State value update} \quad (3.22)$$

$$J_Q = Q(s_t, a_t) - R_t + V(s_{t+1}) \quad \text{Action value update} \quad (3.23)$$

$$J_\pi = KL\left(\pi(\cdot|s_t), \frac{\exp\left(Q(s_t, \cdot)\right)}{Z(s_t)}\right) \quad \text{Policy update} \quad (3.24)$$

The update for the value function in Equation 3.22 is similar to the equivalent update of Equation 2.6, but has the added factor that the entropy of the current policy is added to the objective function. The update for the action value in Equation 3.23 is the same as for DDPG, which is to minimise an error. In the policy update in Equation 3.24, KL denotes the KL-divergence which, in short, is a way of measuring the similarity of two probability distributions - in this case, the policies. $Z(s_t)$ is a normalisation factor that normalises the distribution, but since it does not affect the gradient with respect to the new policy it can be ignored. (Haarnoja et al. 2018)

3.4 Expert Learning & Inverse Reinforcement Learning

One of the main challenges with defining a problem as an optimisation of an MDP is the designing of a reward function that reliably describes what is to be optimised without imposing any bias to the solver. Engineering a reward function often requires comprehensive domain knowledge and trial-and-error testing in order to achieve a suitable reward (Sutton and Barto 2018). To quote Ng and Russell 2000, *"the entire field of reinforcement learning is founded on the presupposition that the reward function, rather than the policy, is the most succinct, robust, and transferable definition of the task"*. Thus a reward function is an import part of a traditional MDP, and the designing of this has a significant impact on the performance of the overall system. The process of inferring a reward function based on some expert data is called *Inverse Reinforcement Learning*. Section 3.4.1 introduces different approaches to constructing this expert data. Section 3.4.2 presents two methods for using expert data to model a reward function as a linear combination of features based on observations of the MDP. A second approach to IRL described in Section 3.4.3 is to represent the reward function using a general function approximator in the form of an NN.

3.4.1 Constructing Expert Data

The expert data related to an IRL problem should capture an ideal policy π^*. One of the main issues when obtaining expert data is the correspondence between the expert *teacher*, and the *learner*, e.g. the teacher might not direct state changes in the same way as the learner. According to Argall et al. 2009 expert data collection can be divided into two categories: demonstrations and imitations.

When constructing expert data via demonstrations, the teacher demonstrates some behaviour to the learner, which only collect observations through its own sensors

i.e. the true expert data from the teacher is not directly recorded. Demonstration expert data can according to Argall et al. 2009 furthermore be divided into two subcategories: teleoperations and shadowing. In teleoperations the learner is operated by the teacher demonstrating some behaviour. This means that there often is a direct map between the constructed expert data and the true expert data related to the demonstrated behaviour. Since the only observed data is from the learner and not the expert directly, it can be difficult to capture exact low-level behaviour of the teacher. Shadowing is where a learner is collecting observations through its own sensors when trying to mimic the behaviour of an observed teacher. This means there is no direct map between the teacher space and the learner space and thus an additional algorithm is often needed for the learner to be able to observe the teacher.

The category of using demonstration data for robot learning has been the focus of several studies. Teleoperation in the form of kinesthetic robot teaching was used by Finn et al. 2016 for solving dish placement and pouring task using IRL. In another study, a block stacking problem was solved by a robot using teleoperations in an HTC VIVE virtual reality systems (VR) (Nair et al. 2017) (HTC VIVE 2019). Ghalamzan and Ragaglia 2017 used kinesthetic robot teaching to solve robot tasks in environments with moving obstacles. Abbeel and Ng 2004 presented an IRL algorithm which was demonstrated for control of a car driving simulation. The demonstration data was recorded using teleoperations. Demonstration data has furthermore been applied to a range of other applications, such as toy helicopter control (Ng et al. 2006), object grasping (Sweeney and Grupen 2007), navigation tasks and maze solving (Argall et al. 2009).

When constructing expert data via imitations, the data collection is carried out on a platform which is not the learner itself. A mapping between the learner and the teacher is therefore often needed to ensure a correct correspondence. Imitation data can be obtained by, e.g. attaching sensors on the teacher during execution to obtain the expert data or via external sensors. This expert data can provide a precise and low-level description of the optimal policy but does often require special build sensors and thus brings overhead and complexity compared to demonstration data. One of the main difficulties in obtaining imitations is to ensure that the mapping between the learner and the teacher is intact. Muelling et al. 2014 presented a table tennis robot using IRL and imitation expert data obtained by sensors external to the human players. Kim and Pineau 2016 collected expert data for socially adaptive path planning via an RGB-D camera external to walking pedestrians. Calinon and Billard 2007 used motions collected via motion sensors on arms and head of a teacher and projected the motions to a humanoid. The same approach was used in Koskinopoulou et al. 2016, where expert data for human-robot collaboration tasks were collected via internal motion trackers attached to wrist, shoulder and elbow of a human expert.

As a summary, there are several approaches to set up a teacher-learner scenario. Demonstrations and imitations both have advantages and disadvantages. Imitation expert data are often complicated to collect due to the overhead related to sensor equipment. It can furthermore be complicated to create a correspondence

between the teacher and the learner. Imitation expert data can, however, capture a more exact behaviour of the teacher. Demonstration data can be simpler to collect, but it can be more difficult to capture the exact low-level behaviour of the teacher. The correspondence between the learner and teacher is often well defined in demonstrations, and the expert data is obtained directly through the sensors of the learner.

3.4.2 Linear Inverse Reinforcement Learning

In Linear Inverse Reinforcement Learning (IRL), a reward function is assumed to be representable by some linear combination of features and weights. The first approach to Linear IRL is by finding a reward function for which the expected value of some observed trajectories generated by an unknown optimal policy π^* is higher than the expected value of some observed trajectories following a policy π as shown in Equation 3.25. (Ng and Russell 2000)

$$\mathbb{E}[V^{\pi^*}(s_0)] \geq \mathbb{E}[V^{\pi}(s_0)] \tag{3.25}$$

Where s_0 is a fixed starting state. The value $V(s_0)$ is calculated as a linear combination of some static basis feature functions $\phi_i(s)$ chosen at design time. When the reward function is defined as $R = \phi_i$, then the value of a basis function is computed as shown in equation Equation 3.26.

$$V_i^{\pi}(s_0) = \sum_{t=0}^{T} \gamma^t \phi_i(s_t) \tag{3.26}$$

The value for a state is then a weighted sum of all the basic feature functions as shown in Equation 3.27.

$$V^{\pi}(s_0) = \sum_{i=0}^{k} w_i V_i^{\pi}(s_0) \tag{3.27}$$

Where the weights w_i are the parameters to fit such that Equation 3.25 is true. This gives the linear programming problem posed in Equation 3.28.

$$\begin{aligned} \text{maximise} \quad & \sum_{i=1}^{k} \left(V^{\pi^*}(s_0) - V^{\pi}(s_0) \right) \\ \text{s.t.} \quad & |w_i| \leq 1, \quad i = \{1, \ldots, k\} \end{aligned} \tag{3.28}$$

The algorithm for approximating a linear reward function can be seen in Algorithm 4.

Algorithm 4 Linear IRL using value estimation. (Ng and Russell 2000)

Input: Trajectories S^E generated from an expert policy
Input: k basis feature functions ϕ
Output: Reward function R
Output: Optimal policy π for R
1: Initialise w_i from a uniform distribution $[-1, 1]$ for $i, \ldots, k$
2: Compute the value of the expert data using Equation 3.27
3: **for** $j \leftarrow 1$ to J **do**
4: Let π be the policy that solves the MDP with reward function $R = \sum_{i=0}^{k} w_i \phi_i(s)$
5: Sample a set of trajectories S from π
6: Compute the value of S and S^E as shown in Equation 3.27
7: Update w_i by solving the linear programming problem in Equation 3.28
8: **end for**
9: **return** R, π

The second approach to Linear IRL, called apprenticeship learning, comes from Abbeel and Ng 2004. The approach is overall similar to the method presented by Ng and Russell 2000, as it also set up a linear combination of feature functions that are weighted. While the first algorithm tries to match some value of a trajectory as shown in Equation 3.27, the algorithm presented by Abbeel and Ng 2004 tries to match feature expectation *vectors* estimated as shown in Equation 3.29 given m trajectories.

$$\mu = \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0}^{\infty} \gamma^t \phi(s_t^{(i)}) \tag{3.29}$$

Equation 3.29 is then used the calculate the feature expectation for both the expert μ_E and a policy μ_π. The idea is that by matching these feature expectations, the policy π will produce trajectories that perform as good as the expert. Another difference is that Abbeel and Ng 2004 puts a $\| w \| \leq 1$ constraint on the weights meaning the problem is a quadratic programming problem rather than a simpler linear one as shown in Equation 3.30. The complete algorithm for apprenticeship learning can be found in Algorithm 5.

$$\begin{aligned} \text{maximise} \quad & w^T (\mu_E - \mu_\pi) \\ \text{s.t.} \quad & ||w||_2 \leq 1 \end{aligned} \tag{3.30}$$

This approach is similar to how *Support Vector Machines* (SVM) optimise, by adding a label of 1 to the expert feature expectations data and a label of -1 to all other samples, i.e. the goal is to find a reward function where the expert does better by a margin than the learned policy. Thus the output of the algorithm is at least one policy that performs as good as the expert minus some margin on the learned reward.

Algorithm 5 Apprenticeship Learning. (Abbeel and Ng 2004)

Input: Trajectories S^E generated from an expert policy
Input: k basis feature functions ϕ
Output: Reward function R
Output: optimal policy π for R

Compute the expert feature expectation μ_E using Equation 3.29
Randomly pick some policy π and compute feature expectation μ_π
for $j \leftarrow 1$ to J **do**
    Update w by solving Equation 3.30
    Solve the MDP using the reward $R = w^T \phi(s)$
    Let π be the policy that solves the MDP with reward function:

$$R = w^T \phi(s_t)$$

    Sample a set of trajectories S from π
    Compute the feature expectation vector μ_π of S as shown in Equation 3.29
end for
return R, π

One of the significant challenges with the linear reward representations mentioned in this section is that there exist multiple reward functions for which Equation 3.25 is true (e.g. when $R = 0$ for all actions). Another challenge with these approaches is that not only is it necessary to design the features; it is a requirement that the optimal reward function can be represented as a linear combination of expected feature functions. Additionally as it was shown in both Algorithm 4 and 5 it is necessary to solve the MDP inside the loop, meaning that every time a new set of weights is generated, a complete MDP problem has to be solved which can be computationally heavy or even infeasible to do multiple times. The following section is an approach to tackle the latter by representing the reward function as an NN.

3.4.3 Deep Inverse Reinforcement Learning

As mentioned in Section 3.4.2, one of the biggest challenges with a linear representation of the reward function is the need for feature functions that often requires domain knowledge to design. Furthermore, they lack the ability to express complex non-linear reward functions often required in high-dimensional spaces such as robotic tasks. (Doerr et al. 2015)

Some of the newer approaches that address the challenges with linear feature combination methods use general non-linear function approximators in the form of NN's. One of these methods is introduced by Ziebart et al. 2008 where they model sub-optimal behaviour as noise and tries to maximise the entropy in order to get a descriptive reward function. The original paper used a discrete observation and action-space, which heavily limits the usability in real-world robotic applications. A deep version of the same approach has also been developed by Wulfmeier et al.

2015. With the nature of an NN, it is capable of using continuous observations but is still limited to discrete actions. *Guided Cost Learning* (GCL) by Finn et al. 2016 is based on the same principles as Ziebart et al. 2008, but extends it to continuous observations and actions. Furthermore, by updating the reward function simultaneously with solving the MDP, the computationally heavy need for solving multiple MDPs is negated. They applied the method to stack dishes in a rag, and pour into cups, both of which they successfully solved with a robotic manipulator.

Another IRL method *Generative Adversarial Imitation Learning* (GAIL) by Ho and Ermon 2016 poses the IRL problem as an adversarial problem which contains two networks as shown in Figure 3.6; a discriminator network that associates a label with the with the expert data and tries to predict the probability that a given input trajectory belongs to the set of expert trajectories. In other words, the discriminator tries to separate expert data from 'fake' expert data. The second generative network is the policy that generates these 'fake' trajectories. The goal of this policy is to confuse the discriminator to classify the generated trajectories as expert data. Thus the problem is no longer to match linear feature expectations, but a classification problem that can be trained simultaneously with updating the policy. This method has a big potential within multiple fields such as image upscaling (Ledig et al. 2016), music generation (Yang et al. 2017) (Mogren 2016), speech enhancement (Pascual et al. 2017) (Kaneko et al. 2017), and text to image (Zhang et al. 2016).

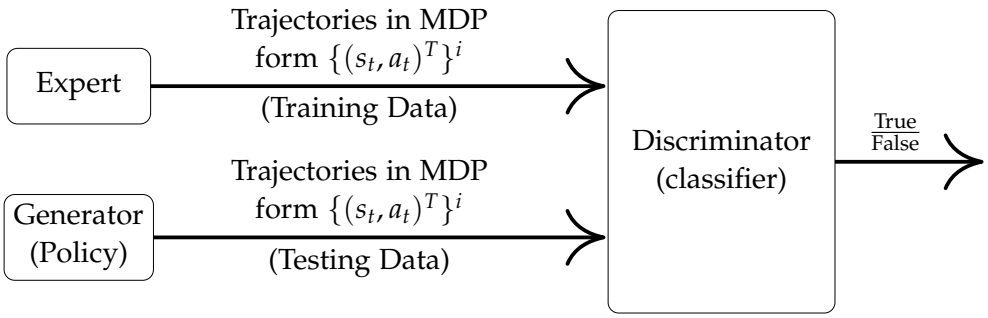


Figure 3.6: General setup of Generative Adversarial Imitation Learning.

One of the disadvantages with GAIL is that it does not construct a traditional reward function, but rather a classifier that classifies state transitions similar to the expert data. Thus generalising this method to dynamic environments, such as driving a car, may prove difficult.

3.5 Summary

The following section is a summary of the RL algorithms presented in this chapter. Some of the key aspects and differences are listed along with a short description of what that means in practice. Afterwards, the same is done for the IRL methods.

The RL algorithms presented in this chapter is a small sample of the many algorithms that get developed every year. However, they do present some of the core methodologies on which most RL algorithms are based. They each address challenges such as the size of the action and observation space and how they handle episodes and trajectories. Table 3.1 lists some of the key properties mentioned in this chapter.

From Table 3.1, it is apparent that when selecting an RL algorithm, it is important to consider what type of observations and actions the problem contains. If the tasks allow observing simple features such as binary signals and the actions likewise are discrete signals, there exist some fundamental approaches to RL algorithms, e.g. Monte Carlo and Q-Learning. Additionally, depending on how much knowledge already known about the environment, some of the algorithms require, e.g. transition probabilities in order to produce policies that converge to a useful result. However, none of the algorithms described in this chapter requires this.

Likewise, with regular RL, there are some differences in how the IRL problem gets solved. Some of the aspects are dependent on the same limitations as with regular RL, such as requiring a model of the environment in order to represent a reward function properly. Other limitations are assumptions such as that the reward function can be represented as a linear combination of engineered features. Engineering these features is also a challenging aspect to overcome complex robotic tasks. Table 3.2 summarises some of the aspects considered in this chapter.

Along with the assumption of the properties of the function approximator (i.e. linear or a NN), the most important aspect to consider when selecting an IRL algorithm is whether the MDP is solved in-loop. Solving the MDP in-loop requires efficient simulation tools since, depending on the task, the MDP has to be solved multiple times, which can be slow and inefficient.

	Action size	Observation size	Approximation function	TD method	Model free	Reference
Monte Carlo	D	D	Table	No	Yes	(Sutton and Barto 2018)
Sarsa	D	D	Table	Yes	Yes	(Sutton and Barto 2018)
Q-learning	D	D	Table	Yes	Yes	(Sutton and Barto 2018)
Deep Q-learning	D	C	NN	Yes	Yes	(Mnih et al. 2015)
DDPG	C	C	NN	Yes	Yes	(Lillicrap et al. 2015)
SAC	C	C	NN	Yes	Yes	(Haarnoja et al. 2018)
TRPO	D & C	C	NN	No	Yes	(Schulman et al. 2015)
PPO	D & C	C	NN	No	Yes	(Schulman et al. 2017)

Table 3.1: A summary of selected properties of the RL methods presented in Section 3.1-3.3. D=discrete, C=continuous, NN=neural network.

	Action size	Observation size	Approximation function	MDP in-loop	Reference
Linear IRL	D & C	C	Linear	Yes	(Ng and Russell 2000)
Apprenticeship Learning	D & C	C	Linear	Yes	(Abbeel and Ng 2004)
Maximum Entropy	D	D	Linear	Yes	(Ziebart et al. 2008)
Deep Maximum Entropy	D	C	NN	Yes	(Wulfmeier et al. 2015)
Guided Cost Learning	C	C	NN	No	(Finn et al. 2016)
Generative Adversarial Imitation Learning	C	C	NN	No	(Ho and Ermon 2016)

Table 3.2: A summary of selected properties of the IRL methods presented in Section 3.4. D=discrete, C=continuous, NN=neural network.

Chapter 4

Reinforcement Learning Complexity

Different methods and basic background for RL was described in Chapter 2 and 3. To determine the industrial applicability of these methods, different use cases inspired from four RoboCluster companies are evaluated using an approach called *Reinforcement Learning Complexity* (RLC). The approach serves as a pilot study to identify the complex aspects in RL for industrial applications. RLC is inspired by an existing method called Technology-Push Manufacturing Technology (TPMT), which is used to validate if a new manufacturing technology can be pushed to the market. TPMT will be introduced in Section 4.1, followed by Section 4.2 which introduces RLC.

4.1 Technology-Push Manufacturing Technology Definition

Different methods have been proposed to quantify how market ready a certain technology is. One of these is the *Technology Readiness Levels* (TRL) described in the white-paper (Mankins 1995). TRL is defined in nine stages of how ready to market the specific technology is. TRL was developed by NASA and was used for their missions; however, the model can be used on any arbitrary technology. The EU's research and innovation program *H2020* has, for example, adapted the model by adding a stage 0 and using it to validate new projects within the program (EU 2018). A problem with TLR is that an expert sets the score with no specific guidelines.

A method called *Technology-Push Manufacturing Technology* (TPMT) by Bøgh et al. 2012 can be used in order to compensate for low dimensional validation of TRL. TPMT works by multiplying a weight (e.g. α) with a general variable (e.g. X_1) describing a specific aspect related to a given technology. These weight/variable pairs are normalised with the sum of all weights, as shown in Equation 4.1.

$$\text{Suitability Score} = \frac{\alpha X_1 + \beta X_2 + \sigma Y_1 + \mu Y_2}{\alpha + \beta + \sigma + \mu} \tag{4.1}$$

The weights are used to empathise the importance of the variables for a given manufacturing technology. Moreover, the variables are scored on a scale from 1-5 going from suitable to not suitable:

Score 1: Very suitable

Score 2-3: Intermediate suitable

Score 4-5: Not suitable

The suitability score in Equation 4.1 is then an indication of the overall implementation horizon related to a technology. The formula is not dependent on the number of variables and therefore, in the thesis, the formula is simplified with weight vector $\boldsymbol{w}$ and variable vector $\boldsymbol{x}$, shown in Equation 4.2.

$$S = \frac{\boldsymbol{w} \cdot \boldsymbol{x}}{\sum_{i=1} w_i} \tag{4.2}$$

In the TPMT paper, the four following areas were the general variables: input/output, environment, technology, and process. The complete explanation of the variables is shown in Table 4.1.

Variable	Definition	Factors
x_1 Input/Output	All of the input aspects which interact with the system e.g. the part/product	The different factors related to the inputs, such as physical properties of the parts, input variance
x_2 Environment	The environment surrounding the system such as the layout	Factors of the environment are areas as safety and accessibility
x_3 Technology	This includes if there are existing technologies which can solve the task	Factors are technologies such as plug-and-play solution and other robotic equipment that can solve it
x_4 Process	The process to be performed is validated on the complexity of the task	The factors of the process are aspects such as positioning, tolerances, processing capabilities etc.

Table 4.1: The four general variables along with a description and which factors plays into the score. Note that x_3 and x_4 corresponds to Y_1 and Y_2 in Equation 4.1.

4.2 Reinforcement Learning Complexity Method Definition

As it was shown in Section 1.3, there is a lack of commercially available solutions that utilises RL. One possible reason for this is the complexity of traditional RL. Due to this, it was deemed necessary to identify the complex aspects of RL itself in industrial contexts. Thus for this thesis, a new method for analysing general RL applications is presented called RLC (*Reinforcement Learning Complexity*). It is built up the same way as TPMT, and it evaluates the complexity of an MDP (described in Section 2.1) of a task along with the complexity of simulation. An RLC score is defined by Equation 4.2 with a scoring from 1-5 and a corresponding colour scheme: green, yellow, orange, red, and blue. The scoring for both the RLC score and the general variables are as follows:

Score 1: Simple

Score 2: Simple with some complexity

Score 3: Complex but achievable

Score 4: Very complex

Score 5: Not achievable

General variables related to the scoring are described in Table 4.2. The RLC score is then used to rank the complexity of creating and solving an MDP in the different use cases. Any use case containing one or multiple general variables of 5 are left out of this ranking.

Variable	Definition	Factors
x_1 Observations	All of the observations available and needed for the MDP state-space	For the observations factors are joint values, TCP, trajectory and other relevant observations (e.g. part observations)
x_2 Actions	The actions which can be performed by the robot and RL agent (MDPs action-space)	Factors of actions are torques, joint values, external activation's and robot configurations
x_3 Reward	The reward giving back to the RL agent after an action has been performed	Factors are constraints of the system, required quality and user feedback
x_4 Simulation	The training environment for the RL agent to train the solution	The factors of the simulation is software available, online training

Table 4.2: The four general variables of RLC along with a description and what factors plays into the score.

15 use cases have been inspired by four different RoboCluster industrial members. The use cases cover the subjects of robot painting, rehabilitation, and pick-and-place processes found in the meat industry. Note that the uses cases are not directly taken from the industry, but only inspired by the companies. Each of these use cases from Section 4.3 to 4.6 are analysed, scored with RLC and is compared in Section 4.7. The scoring is done with an immediate assessment made by an expert panel of six people, with knowledge within *manufacturing production*, *robotics and automation*, *computer vision*, *machine learning*, and *reinforcement learning*. This scoring is done as a pilot study, and thus no prior thorough research or experiments is done for each of the use cases.

4.3 Inropa Use Cases

The field of robotic spray-painting has been gaining traction throughout the years. As stated by Chen et al. 2008, the process is complicated, and the success of the automated procedures depends on the part structure. Therefore, companies specialised in robotic painting has emerged where Inropa is an example. Inropa uses their own software for detecting parts (from a point cloud to a segmentation), paint planning, scheduling, and compute the needed robot configurations, which is illustrated in Figure 4.1. Inropa has stated that the last three processes are where they have the most difficulties since there are many parameters to tune, which is

currently solved with brute-force. For this pilot study, the processes *Paint Planner* and *Motion Planner* have been chosen as two processes, where the parts to be painted is a beam, a window frame, and a random part. These three are illustrated in Figure 4.2. (Inropa 2016)

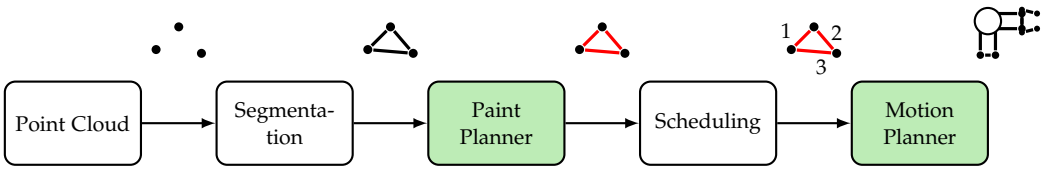
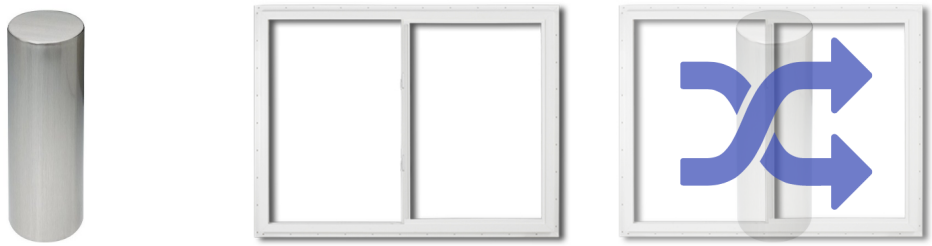


Figure 4.1: The five processes every component goes through at Inropa's solution. The two processes marked with green has been chosen.



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

Figure 4.2: The three different parts used for the Inropa use cases. **(a)** First part - a metal beam. **(b)** Second part - a window frame (Indiamart Webpage 2019). **(c)** Third part - random part.

RLC for Paint Planner

The paint planning process is executed on every new part entering the painting cell. The input is a segmentation of a part, and here different parameters such as paint stroke size, stroke lengths, stroke angles, stroke speed, etc. need to be computed for each segment of the part. Hereafter this info is sent to the *scheduling* process which computes the order of the execution. Three uses cases are defined and scored below, with a complete overview of the concrete scoring in Table 4.3:

Use case 1:

A known (CAD files are also available) metal beam in a fixed, known position, with a segmented polyhedron available. The beam should be completely covered in paint, but paint overlap, thickness and smoothness are not important. From the scoring in Table 4.3, it can be seen that the actions are scored to be the simplest to implement. Contrary, designing the reward function was scored to be difficult by the expert panel. The RLC score is 2.8.

Use case 2:

The window frame is non-convex part compared to use case 1. Otherwise, all the other mentioned aspect of use case 1 also apply for use case 2 (known, fixed, paint requirements, etc.). The RLC score is 2.9, where only a weight in

the simulation is increased due to the increased importance of a representative simulation when the parts are non-convex.

Use case 3:

The only information available for the third use case is a segmented polyhedron and the position of an otherwise unknown part. The paint requirements are the same as use case 1 and 2. The RLC score is 3.8 due to the increased risk of only being able to partially observe the part.

The expert panel discussed an MDP where the actions could consist of stroke width and size, speed of stroke, angle of the stroke. Likewise, the states discussed were stroke width and size, material properties, paint thickness, and paint vendor. As shown in Table 4.3, the reward is deemed difficult to model but, according to the expert panel, could consist of uniform painting and paint consumption.

Inropa - Paint Planner

		RLC			
		Observations	*Actions*	*Reward*	*Simulation*
1st use case	**General variable**	3	1	3	3
	Weight	0.8	0.2	0.8	0.8
	RLC score	2.846			
2nd use case	**General variable**	3	1	3	3
	Weight	0.8	0.2	0.8	0.9
	RLC score	2.852			
3rd use case	**General variable**	4	3	4	4
	Weight	0.8	0.8	0.8	1.0
	RLC score	3.765			

Table 4.3: The scoring of Inropa's use cases with paint planner. The RLC score's colour is rounded to the nearest integer.

RLC for Motion Planner

The second process is the motion planner, which outputs robot configurations for the segments to paint. There is no memory of prior planned configurations. Thus the motion planner needs to plan from scratch for every part, which is a time consuming task. Initial robot configurations are essential since it is not allowed to break a paint stroke. The three use cases presented in the previous section, are for the motion planner defined and scored below, with a complete overview in Table 4.4:

Use case 1:

The input to the motion planner is a sequence of trajectories related to the metal beam, which is to be executed. Since all the trajectory planning and scheduling are done at prior, the expert panel rated the actions as simple. The observations were rated a grade higher since the environment may only be a partially observable (e.g. cables and wires might limit robot movement).

Simulation has the highest weight due to the ability to model some of the hidden observations. The RLC score is 1.61.

Use case 2:
For use case 2, trajectories related to a window frame in a fixed position is the input. The scoring for the window is close to identical to the prior use case, with the exception of the simulation weight. This weight was scored higher since the input requires more trajectories and is non-convex. The RLC score is 1.63.

Use case 3:
The input for use case 3 is a sequence of trajectories. It is not scored much higher than use case 1 and 2. The expert panel evaluated that there might be problems related to observations not being fully represented if a segmentation failed. However, the object is still not a complex task and thus has an RLC score of 1.64.

The expert panel discussed an MDP where the actions could consist of joint values, TCP positions, the position of wires on the robot. Likewise, the states discussed were joint values, part dimension, paint planning schedule. The reward could, according to the expert panel, consist of execution speed, number of joint rotations.

Inropa - Motion Planner

		RLC			
		Observations	*Actions*	*Reward*	*Simulation*
1st use case	**General variable**	2	1	1	2
	Weight	0.5	0.3	0.6	0.9
	RLC score	1.609			
2nd use case	**General variable**	2	1	1	2
	Weight	0.5	0.3	0.6	1.0
	RLC score	1.625			
3rd use case	**General variable**	2	1	1	2
	Weight	0.6	0.3	0.6	1.0
	RLC score	1.640			

Table 4.4: The scoring of Inropa's use cases with motion planner. The RLC score's colour is rounded to the nearest integer.

4.4 Life Science Robotics Use Cases

Rehabilitation and mobilisation of bedridden patients are hard on the physiotherapist due to the heavy lifting the job involves. The product ROBERT, by Life Science Robotics, aims to alleviate some of the heavy liftings in the rehabilitation of legs by using a KUKA manipulator. (Life Science Robotics 2019)

To investigate the use of RL in devices as ROBERT, different use cases are set up. All of them concerns the movement of a patients leg, as illustrated in Fig-

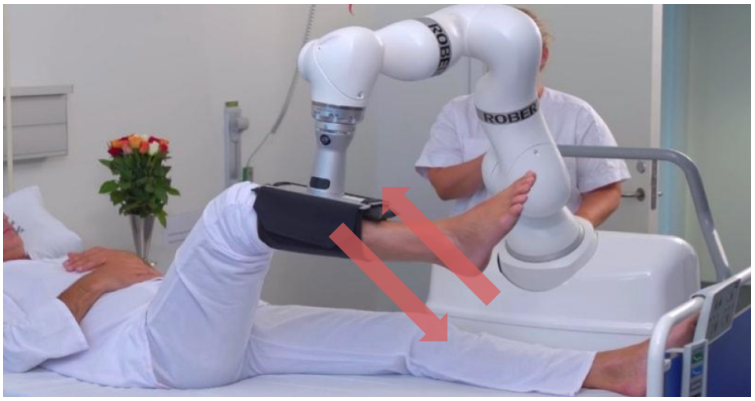


Figure 4.3: The path in which ROBERT will move the leg. (Life Science Robotics 2019)

ure 4.3.

RLC for ROBERT

For ROBERT, three use cases have been defined. They are described and scored below, with a complete overview in Table 4.5:

Use case 1:
The patient is unconscious and unable to move the leg. ROBERT will only follow already predefined trajectories and not create new ones by itself. The movement is done to prevent bedsore and activate the leg. There will be a physiotherapist in the room at all time. Due to the individuality of human legs, the simulation got a score of 3 even though this use case only concerns predefined movements. The RLC score is 1.8.

Use case 2:
The patient is now conscious, and ROBERT is replaying trajectories demonstrated by the physiotherapist. The patient still has a paralysed leg. The expert panel evaluated the observations and actions to be simple, but the reward and simulation have risen in complexity and thus, the RLC score of 2.5.

Use case 3:
The third use case is active training, where a conscious patient tries to move a leg together with ROBERT. The trajectories are not strictly predefined since the patient should be able to move the leg. Both the simulation and reward was scored 5 from the expert panel. This was due to the difficulty of defining a reward function that distinguishes robot movements from patient movements. A simulation would have to simulate a conscious patient interacting with ROBORT, which itself is a difficult task. This use case got an RLC score of 3.8, and contains two general variables with a score of 5, i.e. the task is deemed not achievable.

The expert panel discussed an MDP where the actions could consist of joint torques, TCP forces. Likewise, the states discussed were joint torques, joint values, TCP

forces, patient feedback. As shown in Table 4.5, the reward is deemed difficult to model but, according to the expert panel, could consist of forces, patient feedback.

Life Science Robotics

		RLC			
		Observations	*Actions*	*Reward*	*Simulation*
1st use case	**General variable**	1	1	2	3
	Weight	0.9	0.9	0.9	0.9
	RLC score	1.750			
2nd use case	**General variable**	1	1	4	4
	Weight	0.9	0.9	0.9	0.9
	RLC score	2.500			
3rd use case	**General variable**	4	1	5	5
	Weight	1.0	0.9	1.0	1.0
	RLC score	3.821			

Table 4.5: The scoring of Life Science Robotics' use cases. The RLC score's colour is rounded to the nearest integer.

4.5 RobNor Use Cases

RobNor, like Inropa, is a company specialised in robot painting. They offer a wide variety of product/services for their customers concerning the topic of automated painting (RobNor 2019a). The use cases from RobNor has been selected to be *powder painting*. RobNor specifies that they solve powder painting applications with varying complexity depending on the client's requirements. The part to be painted in these use cases is the window frame in Figure 4.2b. Note that these use cases are end-to-end, meaning the input is a window frame and the output is a painted window frame with everything in between being a black box of RL. (RobNor 2019b)

RLC for Powder Painting

Three use cases have been defined and scored below, with a complete overview in Table 4.6:

Use case 1:
This use case is a complete robotic painting system, which means it both control the robot and the painting. The window frame is in a known position, and the painting smoothness is not considered. The expert panel evaluated the reward and simulation as the hardest aspects to solve in the case. The RLC score is 3.53.

Use case 2:
The same window frame from use case 1 is used, along with the requirements of the painting. Here linear spray units are also painting the part and the

robot should thus function together with those. The expert panel evaluated that the linear units resulted in increased complexity in both the design of a reward function and the observation space. The RLC score is 3.78.

Use case 3:

The third use case is identical to the first use case with an added camera for vision feedback. The simulation weight has increased since it is hard to simulate vision feedback, which is needed to train the RL agent. The RLC score is 3.79.

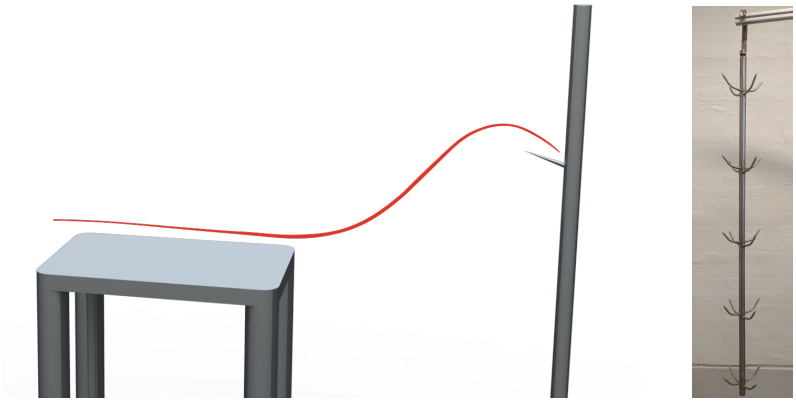
The expert panel discussed an MDP where the actions could consist of coating amount, speed of stroke, TCP position. Likewise, the states discussed were stroke width and size, material properties, vision feedback, TCP position. As shown in Table 4.6, the reward is deemed difficult to model but, according to the expert panel, could consist of powder waste, process time, equally distributed powder.

RobNor					
		RLC			
		Observations	*Actions*	*Reward*	*Simulation*
1st use case	**General variable**	3	3	4	4
	Weight	0.8	0.8	0.9	0.9
	RLC score	3.529			
2nd use case	**General variable**	4	3	4	4
	Weight	1.0	0.8	1.0	0.9
	RLC score	3.784			
3rd use case	**General variable**	4	3	4	4
	Weight	1.0	0.8	1.0	1.0
	RLC score	3.789			

Table 4.6: The scoring of RobNor's use cases. The RLC score's colour is rounded to the nearest integer.

4.6 Danish Meat Research Institute Use Cases

The Danish Technological Institute has a subdivision called *Danish Meat Research Institute* (DMRI) who specialises in production equipment for the meat industry. The use cases from DMRI are picking and placing large pieces of meat from pigs. The meat is picked up from a conveyor and is hanged on a hook (referred to as the *Christmas Tree*). This task is currently done manually, but since the meat is heavy, the operators are not allowed to do this for extended periods of time. In Figure 4.4, a trajectory related to hooking meat on the Christmas tree is shown. (DMRI 2019)



(a) (b)

Figure 4.4: (a) The setup of the DMRI use cases. **(b)** The *Christmas tree* where the middle pieces of meat is hanged.

RLC for Pick-and-Place of Meat

The three use cases are defined and scored below, with a complete overview in Table 4.7:

Use case 1:
In the first use case, the robot picks up a piece of meat from a known position and places it on a static hook. The agent's only task is to learn a trajectory and place the meat. The expert panel evaluated the simulation high since it was deemed complex to simulate the properties of meat. The RLC score is 2.6.

Use case 2:
Instead of one specific hook, multiple hooks are available. The initial position of the meat is the same as in use case 1, and the positions of the multiple hooks are static and known. The agent's task is to learn a trajectory for hooking the meat as well as which hook to hang it on. The expert panel evaluated the reward as being difficult to model due to the increasing amount of parameters to consider. The RLC score is 2.7.

Use case 3:
A single Christmas tree hangs from the ceiling and is not fixed. The meat is placed on a table but is at a random initial position. The observation has been increased since information concerning the Christmas tree and the position of the meat are deemed relevant. The RLC score is 3.1.

The expert panel discussed an MDP where the actions could consist of TCP position, joint values, torque control. Likewise, the states discussed were TCP position,

joint values, torque control. As shown in Table 4.7, the reward is deemed difficult to model but, according to the expert panel, could consist of distance to hook, TCP position, joint forces.

DMRI		RLC			
		Observations	*Actions*	*Reward*	*Simulation*
1st use case	**General variable**	1	1	3	4
	Weight	0.5	0.5	0.8	1.0
	RLC score	2.642			
2nd use case	**General variable**	1	1	4	4
	Weight	0.6	0.7	0.8	1.0
	RLC score	2.742			
3rd use case	**General variable**	3	1	4	4
	Weight	0.8	0.7	0.9	1.0
	RLC score	3.147			

Table 4.7: The scoring of DMRI's use cases. The RLC score's colour is rounded to the nearest integer.

4.7 Summary of RLC Use Cases

RLC aims to identify the complexity of RL for industrial use cases. The method evaluates each of the aspects of an MDP and tries to indicate which aspects may prove to complicate an RL project. It is recommended that the method is applied in the initial phases of an RL project as it also functions as a discussion of RL in the context of the task.

General Variable Distribution										
	Observation					**Action**				
Score	**1**	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**
Amount	4	3	4	4	0	11	0	4	0	0
	Reward					**Simulation**				
Score	**1**	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**
Amount	3	1	3	7	1	0	3	3	8	1

Table 4.8: The distribution of scores for each of the general variables. The amount refers to the number of times the given score was given.

Table 4.8 shows the overall distribution of scores given in this chapter. The *amount* refers to the number of times the specific score was given, e.g. an amount of 4 at score 1 means a score of 1 was given four times in that category. A comparison of the scoring of *observations*, *actions*, *reward*, and *simulation* shows that the lowest scoring general variable is *actions*. The low score indicates it is not a complex aspect of an MDP for these industrial application. This could be due to most of the presented use cases already being automatised today, meaning the actions for solving the tasks already exists. The *observations* variable is almost uniformly distributed between score 1 and 4, both inclusive. This gives an image of it being use

case dependent. The two highest scoring general variable are *reward* and *simulation*. Simulation is scored high, which indicates that it can be difficult to obtain efficient and representative simulations of industrial tasks for RL. The reward is also generally scored high, which indicates that designing a reward function is a complex aspect of an industrial RL solution. One of the potential difficulties is quantifying the progress and quality of the process while maintaining a simple reward structure, e.g. a convex function with a single maximum.

Using the RLC score obtained in this chapter, the use cases can be ranked on their complexity, as shown in Table 4.9. The use cases with the lowest RLC score are the tasks which have the least complexity and thereby the best chance at incorporating RL, according to the expert panel.

The three lowest scored industrial applications are all motion planning tasks. The reason for this can be found in the problem lending itself naturally to an MDP setup. For instance, each timestep in a terministic episode for these tasks would only consist of a sparse negative reward, encouraging the agent to remember process parameters and robot configurations that require as few replans as possible. It should be noted that RLC is just an attempt of identifying the complexity of RL, and thus there can exist legal or ethical reasons that prohibit any use of RL within specific industrial fields. One example being healthcare where strict legislation dictate when and how machinery can interact with patients.

RLC Score Index

Index	Use case	RLC score	Index	Use case	RLC score
1	I-MP 1	1.609	**9**	I-PP 2	2.852
2	I-MP 2	1.625	**10**	DMRI 3	3.147
3	I-MP 3	1.640	**11**	RN 1	3.529
4	LSR 1	1.750	**12**	I-PP 3	3.765
5	LSR 2	2.500	**13**	RN 2	3.784
6	DMRI 1	2.642	**14**	RN 3	3.789
7	DMRI 2	2.742	**15**	LSR 3	3.821
8	I-PP 1	2.846			

Table 4.9: The RLC score index for all 15 use cases in ascending order. The abbreviations are: **I-PP** = Inropa - Paint Planner, **I-MP** = Inropa - Motion Planner, **LSR** = Life Science Robotics, **RN** = RobNor, **DMRI** = Danish Meat Research Institution. The number following the abbreviation referrers to the use case from that company.

Chapter 5

Problem Formulation

As it was introduced in Chapter 1, there is an increasing demand for flexibility and adaptability in modern production. This is due to increasing customer demands, customisation, global market competition, etc. These demands increase the need for adaptable models in production that can quickly and efficiently adapt to customers and market fluctuations. One area of adaptable models is machine learning, which has received great attention from the research community in the last few decades.

Chapter 2 and 3 described the general background and terminology of MDP and RL and what a state transition consists of. Different RL methods were presented to solve MDPs in order to give an overview of the different methods available. The RL methods were separated into three classes; *value-based*, *policy gradient*, and *actor-critic*. As part of the value-based methods, tabular methods are a powerful tool for small and discrete state-spaces. However, when the state-space is increased, and the actions are not discrete, the tabular methods become infeasible. To this end, a deep function approximator alleviates some of these limitations. Another approach is the policy gradient methods, which uses a probabilistic approach to select actions given the current state. The last class is actor-critic, which combines both policy gradient and value-based methods by using a value function to guide a policy. Moreover, as described in Chapter 3, one of the fundamental parts of RL is the reward function, which is often specifically engineering in a trial and error process. Therefore the topic of *Inverse Reinforcement Learning* was presented in Section 3.4. IRL aims to solve the problem of designing a reward function by using expert data to model a reward function. Finally, Section 3.5 discusses some of the pros and cons of different RL methods. This includes the size of the state and action-space along with properties such as whether they are a TD method or are model free. Similarly, some properties of different IRL methods were presented, including if they solve the MDP in-loop, which sets higher demand for efficient RL algorithms and simulations.

In Chapter 4 a method named *Reinforcement Learning Complexity* (RLC) was used to validate the RL complexity of 15 different use cases, inspired from four RoboCluster companies. The method investigated the complexity of *observations*, *actions*, *reward*, and *simulation*. It was concluded that the most complex aspects of RL in the use cases were the reward and simulation.

It is not possible to confirm the initial hypothesis which stated that *the complexity of reinforcement learning in RoboCluster use cases can be aided by the use of a human expert*. However, there does exist a comprehensive amount of literature on the topic of

RL. One of the main complexities of RL is the design of a reward function which IRL tries to alleviate. The first use case from DMRI has been chosen for a further investigation of IRL for industrial applications. Thus the initial project hypothesis can be refined to the following problem formulation:

How can an IRL algorithm with a robot manipulator be set up and used to solve a pick-and-place task at DMRI with the use of recorded expert data?

This problem formulation is made up of three research questions:

1. How can expert data be collected and utilised for the task at DMRI?
2. How should a training and simulation environment be structured for the task at DMRI?
3. How can an IRL algorithm be used for a pick-and-place task as the use case and does the performance differ from an RL approach?

5.1 Research Work Plan

Based on the problem formulation and the research questions, a research work plan has been created with five entries. These entries serve as guidelines to accomplish the problem formulation and answer the research questions.

1. Develop a simulation environment to accelerate training
Since simulation plays an intrinsic role in RL, a virtual environment has to be created for the DMRI use case. To secure generality and transferability, the developed simulations should all comply with the unofficial standard for environment creation defined by OpenAI (Brockman et al. 2016). This also allows for faster development since existing implementations of RL algorithms support this standard such as (Plappert 2016), (Hill et al. 2018), and (Zhang and Tai 2017).

2. Implement an RL method to use in a simulation environment
As it was presented in Section 3.4.2, some of the IRL methods requires solving of the MDP multiple times. It is, therefore, important that at least one RL method is implemented and ready to use. This includes RL methods that support continuous states or actions. Since the simulation environment should follow the OpenAI standard, there exist multiple implementations of different RL algorithms ranging from value-based methods such as DQNs to policy gradient methods such as TRPO and PPO to actor-critic such as DDPG and SAC.

3. Make an experimental setup to capture expert data
As shown in Section 3.4.2, the expert data has to be captured before any IRL can take place. This requires a setup that can facilitate the recording of relevant observations and actions. Since the observations are dependent on the method of IRL used and the available equipment, there is no requirement to the form of the recording (i.e. it can be both virtual or physical).

4. Implement IRL using expert data
The fourth step of the work plan is to implement one or multiple versions of the IRL methods described in Section 3.4. Since the goal of this thesis is to model a reward signal based on expert data, this step is an intrinsic part of this thesis.

5. Transfer the learning from a simulation to a real-world experimental setup
Finally, the last step is to transfer a learned policy from any simulation environment to a real robot in order to execute a trajectory. The generated trajectory should solve the same task as the task from which the expert data was recorded. In this thesis, this amounts to picking and placing a piece of meat from a table to a hook.

5.2 Delimitation

The selected use case from DMRI includes simulation of a piece of meat. It was shown that the simulation of meat is a complex task, and since the simulation is not the main objective of the thesis. the task has been simplified by discarding simulation of meat. Therefore the picking and placing of meat has also been omitted and reserved for future work.

The development part of the thesis is split into three chapters. Each of these chapters will focus on the three research questions and the work plan:

Chapter 6: The Experimental Setup & Human Expert Data
The first research question is investigated, where the chapter also gives a complete overview of the experimental test setup. In this chapter the third work plan is carried out.

Chapter 7: Software & Simulation Environment
The software architecture is presented in this chapter, which also investigates research question number 2. Furthermore, the simulation environment is presented and described, along with tests regarding its stability. This includes work plan steps 1 and 5.

Chapter 8: Trajectory Learning
The final development chapter presents the problem of IRL and how it was solved in this thesis, and thus investigates research question number 3. This includes both OpenAI environments and the use case from DMRI, whereas the chapter includes work plan 2 and 4.

The following three chapters contains relevant tests and results within the chapters themselves.

Chapter 6

The Experimental Setup & Expert Data

This chapter contains a description of the experimental setup and how human expert data are acquired according to research question 1, from Chapter 5.

As presented in Section 3.4.1, there are two approaches for constructing expert data. The first approach is using teleoperations, i.e. constructing expert data using the learner directly. The second approach is using imitation data, where sensors are attached to the teacher or placed externally. Imitation data have the advantage of accurately representing the teacher, while teleoperations have a simpler map from teacher to learner as shown in Figure 6.1. In the context of robots learning from humans, the teacher is a human and the learner is a robot. The concept of *Direct Task Space Learning*, presented in the following section, aims at finding a subspace of a Task space which both the human expert and the robot can observe.

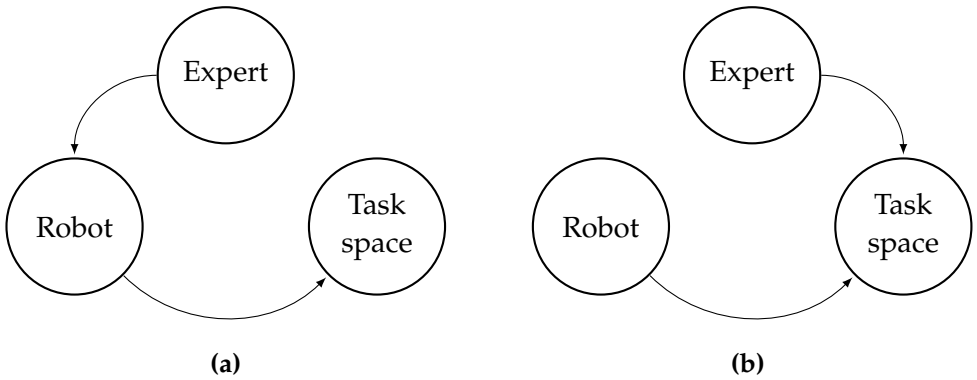


Figure 6.1: (a) The relation between the robot, a human expert and the task space for expert data via teleoperations. **(b)** The relation between the robot and the tasks, and a human expert and the task space for expert data via imitations.

6.1 Direct Task Space Learning

A Task space represents the physical space in which a task is located. It thus includes the observations required to solve a given task. The task specific knowledge of the human expert is based on interactions and experience with this space. In direct task space learning, the human and the robot will share a common observation space. Thus there is no need for the human to teleoperate the robot, since they already share a subspace of the Task space. Furthermore, the overhead of mapping the expert data often present in imitation data collection is omitted. A way

to achieve the shared subspace, could be to use sensors that are directly relateable by both the robot and human expert. One of the potential shortcomings of using a subspace of the full Tasks space, is that this subspace may simply not contain enough task specific information to solve the task. A practical example of this is a pilot operating a plane by observing both altitude and air plane velocity. If the pilot is teaching a student how to fly, but only showing how to read the current velocity, the student will never be able to fly.

The aim of this thesis is to find a policy for a robot agent, which uses a set of expert trajectories to solve the selected use case in an experimental setup. These expert trajectories will be represented as a 3D point cloud of positions directly recorded from the Task space, as seen in Figure 6.2. These points representing the trajectories will function as a common subspace between the human expert and the robot, as they are observable for both with little to no additional mapping.

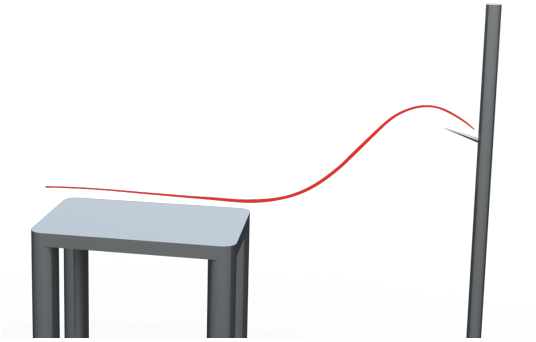


Figure 6.2: An example of a simulated expert trajectory for the DMRI use case.

In Section 3.4.1 it was found that some studies captured expert data directly from a human expert using attached motion sensors (Koskinopoulou et al. 2016), (Calinon and Billard 2007). With inspiration from these studies, a similar approach will be used in this thesis. A sensor will be attached on the object which the human expert is operating, e.g. the meat, from where the expert trajectories are captured. Since the expert trajectories are represented as a 3D point cloud, it is sufficient that a used sensor technology captures the points which the human expert enters. The HTC VIVE virtual reality system is a simple and off the shelf technology, which can provide this information, see Figure 6.3. A VR-tracker is thus deemed as a sufficient sensor technology for the collection of expert trajectories, see Figure 6.3b.

(a) **(b)**

Figure 6.3: (a) A HTC VIVE virtual reality system. **(b)** A VR-tracker. (HTC VIVE 2019)

The robot agent used for the experimental setup is a UR5 robot manipulator. This agent, together with a Christmas tree, the HTC VIVE system and an aluminium

table constitute the experimental setup, from where the human expert trajectories are collected, see Figure 6.4. The raw output from the VR-tracker is its position relative to a world frame within the HTC VIVE system, this output is thus not related to the robot agent. To create this relation, the expert trajectories should be collected in relation to the robot base to secure the correspondence between the human expert and the robot. The following section consists of a description of how this correspondence is secured.

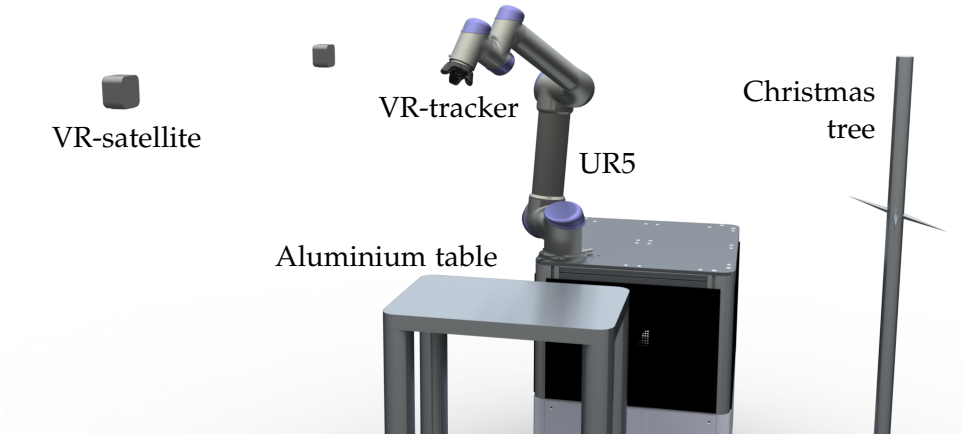


Figure 6.4: The experimental setup containing the VR-satellites, VR-tracker, UR5, aluminium table, and the Christmas tree where the meat is hanged.

6.2 Human Expert & Robot Correspondence

The initial outputs from the VR-tracker are sets of m points constituting a trajectory relative to a world frame within the VR-system ${}^{world}_{point_k}T$, $k = \{1, \ldots, m\}$. To secure a correspondence between the robot agent and the human expert, this initial output is transformed such that the m points in the expert trajectories can be captured relative to the base of the robot agent ${}^{base}_{point_k}T$. If the static transformation from the VR world frame to the robot base frame ${}^{world}_{base}T$ is known expert trajectories can be obtained relative to the robot agent by standard matrix multiplication as seen in Equation 6.1.

$$ {}^{base}_{point_k}T = \left({}^{world}_{base}T\right)^{-1} {}^{world}_{point_k}T, \quad k = \{1, \ldots, m\} \tag{6.1} $$

${}^{world}_{base}T$ is however unknown and needed in order to relate the robot base to the points in the expert trajectories.

Hand-Eye Calibration

A hand-eye calibration aims to obtain extrinsic camera parameters, i.e. the static transformation from a robot tool to a camera placed on the tool ${}^{tool}_{cam}T$. The same approach as in a hand-eye calibration will in this section be used to find ${}^{world}_{base}T$ and thereby ${}^{base}_{point_k}T$, $k = \{1, \ldots, m\}$. When doing the hand-eye calibration, a VR-tracker will be attached to the tool of an UR5 as seen in Figure 6.5 and the transformation from the robot tool to the tracker is obtained according to Figure 6.6. The robot will then be moved to n different TCP positions. For these positions, transformations from the robot base to the robot tool ${}^{base}_{tool_j}T$ and transformations from the world to the VR-tracker ${}^{world}_{tracker_j}T$ are obtained for $j = \{1, \ldots, n\}$.

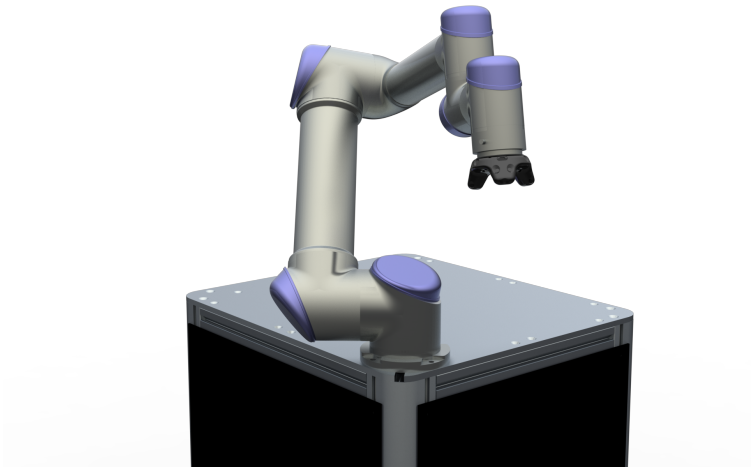


Figure 6.5: The VR tracker attached to the UR5.

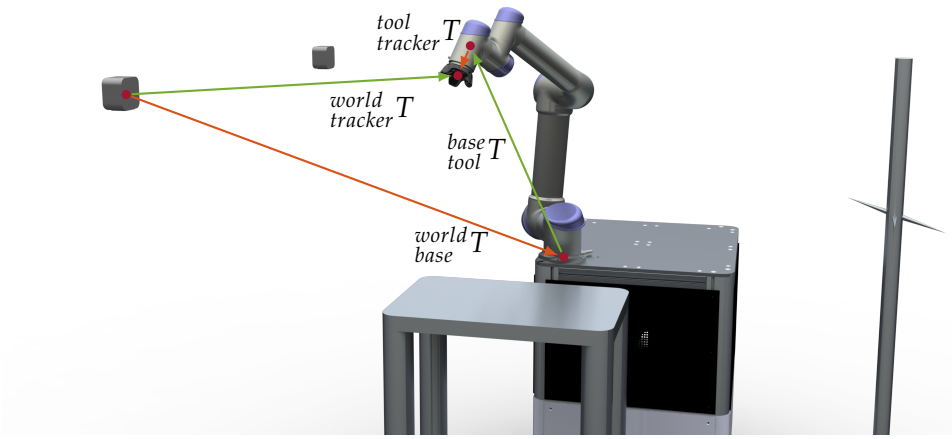


Figure 6.6: The hand eye calibration of the robot manipulator and the VR-system. The green lines indicates the known transformations and the red the unknown.

From the positions the transformations ${}^{tracker_i}_{tracker_j}T$ and ${}^{tool_i}_{tool_j}T$ are found for all possible combinations, as seen in Equation 6.2 and 6.3.

$$A_{i,j} = {}^{tool_i}_{base}T \, {}^{base}_{tool_j}T \qquad \forall \text{ possible combinations of } i,j = \{1,\dots,n\} \wedge i \neq j \quad (6.2)$$

$$B_{i,j} = {}^{tracker_i}_{base}T \, {}^{base}_{tracker_j}T \qquad \forall \text{ possible combinations of } i,j = \{1,\dots,n\} \wedge i \neq j \quad (6.3)$$

Using Equation 6.2 and 6.3, ${}^{tool}_{tracker}T$ can then be obtained by solving the equality seen in Equation 6.4.

$$A_{i,j}X = XB_{i,j} \quad (6.4)$$

Where X is equal to the static transformation ${}^{tool}_{tracker}T$. When ${}^{tool}_{tracker}T$ is known, ${}^{world}_{base}T$ can be estimated using the n calibration points, as seen in Equation 6.5.

$${}^{world}_{base}T = \frac{1}{n}\sum_{i=1}^{n} {}^{world}_{tracker_i}T \left({}^{tool}_{tracker}T\right)^{-1} {}^{tool_i}_{base}T \quad (6.5)$$

An approach for doing a hand-eye calibrations was presented by Tsai and Lenz 1988. This approach aimed at being simple, efficient, and accurate. Due to the simplicity in the implementation, it was deemed suitable for collecting the expert trajectories for this thesis.

The approach presented by Tsai and Lenz 1988 finds the transformation, ${}^{tool}_{tracker}T$ by first obtaining the rotation ${}^{tool}_{tracker}R$ and then the related translation. Using all the calibration points, an over-determined system of linear equations can be created using the last three coefficients from the quaternions of the following rotation matrices: ${}^{tool_i}_{tool_j}R$ and ${}^{tracker_i}_{tracker_j}R$, as seen in Equation 6.6.

$$\left({}^{tool_i}_{tool_j}P + {}^{tracker_i}_{tracker_j}P\right) \times {}^{tool}_{tracker}\hat{P} = {}^{tracker_i}_{tracker_j}P - {}^{tool_i}_{tool_j}P \quad (6.6)$$

Where P is found by multiplying the quaternions of the related rotation matrix by 2. This system of linear equations can be solved using least square, where a unique solution for ${}^{tool}_{tracker}\hat{P}$ is found. After finding ${}^{tool}_{tracker}\hat{P}$ it can be used to find ${}^{tool}_{tracker}P$ according to Equation 6.7, as proven by Tsai and Lenz 1988.

$${}^{tool}_{tracker}P = \frac{2 \; {}^{tool}_{tracker}\hat{P}}{\sqrt{1 + |{}^{tool}_{tracker}\hat{P}|^2}} \quad (6.7)$$

When ${}^{tool}_{tracker}P$ is found from Equation 6.7, the rotation matrix ${}^{tool}_{tracker}R$ can be found analytically. Having ${}^{tool}_{tracker}R$ the translation ${}^{tool}_{tracker}t$ can be found using Equation 6.8, where a unique solution can be found using least square optimisation.

$$\left({}^{tool_i}_{tool_j}R - I\right) {}^{tool}_{tracker}t = {}^{tool}_{tracker}R \; {}^{tracker_i}_{tracker_j}t - {}^{tool_i}_{tool_j}t \tag{6.8}$$

Thus following the method presented in this section, it is possible to relate any given point in the VR-system to the base of the robot agent.

6.3 Human Expert Data Collection

The VR-system can be interfaced with ROS using an existing ROS-driver called *vive_ros*, which, e.g. can return the position of a VR-tracker relative to a VR world frame (RoboSavvy 2019). Using the calibration theory presented in the previous section, it is possible to obtain expert trajectories from the VR-tracker relative to the base of the robot agent. The expert trajectories which will be used throughout this thesis can be seen in Figure 6.7. These trajectories are generated by one of the authors by moving the VR-tracker according to Figure 6.2.

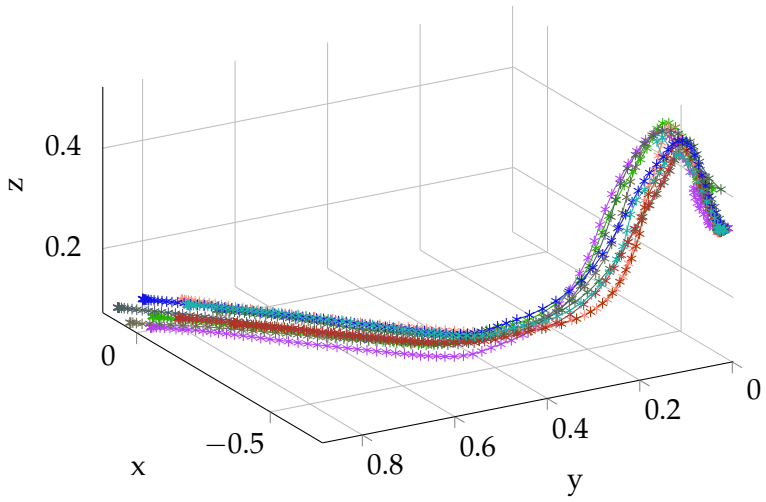


Figure 6.7: 10 expert trajectories from the experimental setup.

The precision of the hand-eye calibration returned by the implementation was evaluated through a test. In this test the precision of two calibrations each containing 4, 6, 8, 10 and 12 calibration points, were evaluated, i.e. ten calibrations. Two transformations, ${}^{world}_{base}T$ and ${}^{tool}_{tracker}T$, were found using the implemented hand-eye calibration, for all ten sets of calibrations. Using these transformations, translational and rotational errors were obtained using Equation 6.9 and the ${}^{world}_{tracker}T$ returned by the ROS-driver. This was done with 50 reference points and the result were averaged over the five different calibration groups.

$${}^{world}_{tracker}T = {}^{world}_{base}T \; {}^{base}_{tool}T \; {}^{tool}_{tracker}T \tag{6.9}$$

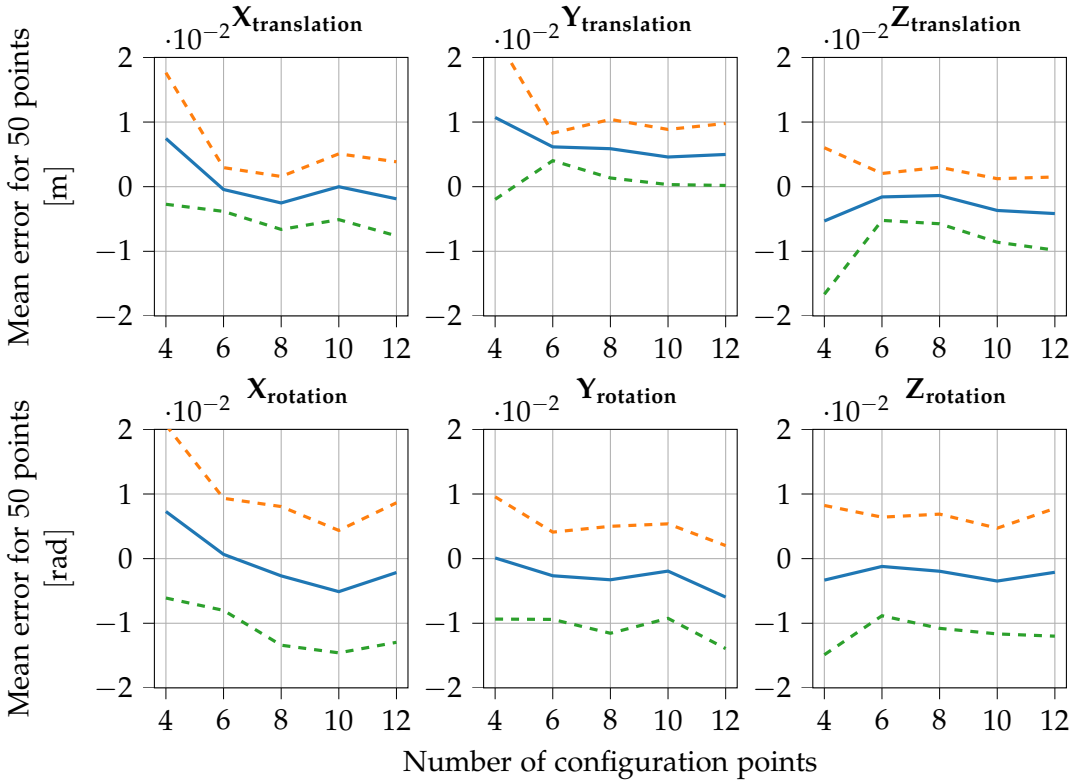


Figure 6.8: The mean errors for in translation and rotation over 50 test points and two different calibrations for 4, 6, 8, 10, and 12 points. The dashed lines indicates the standard deviation.

Figure 6.8 displays the error between the calculated ${}^{world}_{tracker}T$ and the one returned by the *ros_vive* driver. From the test, it can be seen that 4 calibration points produce a larger error than any of the other calibrations, which is also clearly shown in Figure 6.9. The precision of a calibration will, according to Tsai and Lenz 1988 improve when the amount of calibration points increases. A reason for this not being the case in this test could be that no common strategy was used when obtaining the calibration points. Some of the calibrations do, therefore capture all six degrees of freedom better than others.

There was a tendency that the VR-system returned noisy measurements when the tracker was not visible or only partly visible, which could influence the calibration result. This made it difficult to calibrate the whole work-space of the robot since some positions created noise and could, therefore not be visited during the calibration. As it can be seen in the norm error for the translation, see Figure 6.9, there is no significant change in the performance for 6, 8, 10 and 12 calibration points. Having 6 calibration points produces a better result when looking at the norm rotation error. This difference is, however, not significant, and no clear conclusion about an optimal amount of calibration points can be made. Despite inaccuracies in the hand-eye calibration, the implementation was still deemed sufficient for the expert data collection for this thesis.

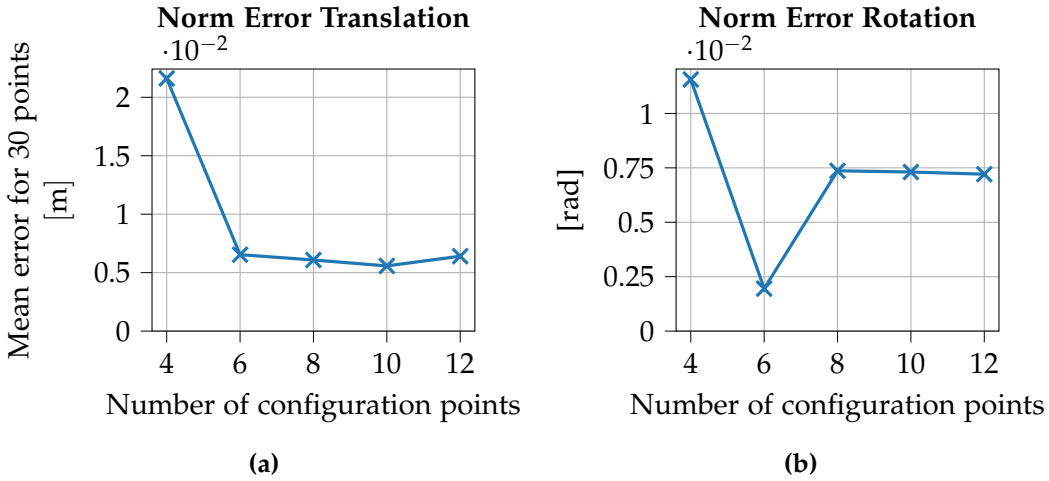


Figure 6.9: The norm error related to the rotation and the translation obtained through the experiment.

Chapter 7

Software & Simulation Environment

As it was shown in Chapter 4, an essential aspect of RL is the accessibility of an efficient simulation environment. This chapter presents the simulation environments developed and used for the DMRI use case and will thereby answer the second research question from Chapter 5. The chapter furthermore includes the different ML frameworks, along with stability tests of the simulation and a newly proposed TCP Simulation environment.

7.1 Software Architecture

In order to achieve the research objectives, a structured software architecture was deemed necessary. Moreover, an infrastructure with RL in mind is a requirement. The RL environment called Gym by OpenAI is a standardised platform with different built-in environments to test and train RL algorithms on. Gym is written in Python and is open source, and therefore, users can implement their own environments (Brockman et al. 2016). Due to the standardisation OpenAI Gym provides, it was chosen as the base of all simulation frameworks related to this thesis.

Besides the standardised environment, an additional package called *Keras-rl* by Plappert 2016 is used. Some of the state of the art RL algorithms are implemented in the Keras-rl framework, which uses the deep ML framework Keras. Keras-rl supports the Gym standard out of the box and also supports adding custom algorithms not already implemented. With the frameworks for developing and testing RL algorithms in place, a framework for controlling the robot agent is required. Here the open source middleware ROS is used to control the robot and works as the underlying foundations for the whole software architecture. ROS limits the software in this thesis to Python 2.7. In Figure 7.1, an overview of how the software architecture is set up. Moreover, in Appendix A, an UML diagram of the created UR environment can be found. The created environment is further explained in Section 7.1.2.

For any ML method, training is essential for a successful result. When training with real hardware such as a robot manipulator, both safety and time is a concern. Safety is a concern since an RL agent learns by exploring the environment, which is hazardous for itself and its surroundings. The second concern of time is relevant since general training an RL agent requires a significant amount of timesteps and thus it would be infeasible to do on a real robot. Therefore, a physics simulation was

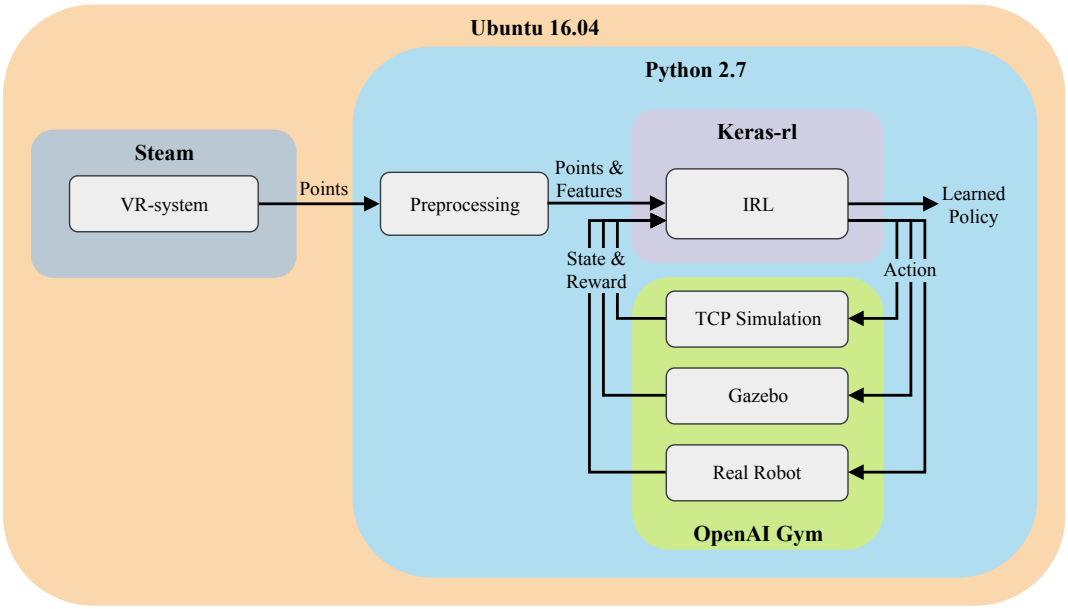


Figure 7.1: An overview of the software architecture. The *VR-system* and *preprocessing* are where the acquisition of data is done. Both *Keras-rl* and *OpenAI Gym* implements IRL algorithms and environments built by the authors. The three environments are not executed at the same time but can be executed in an arbitrary order. The output is a *learned policy* which can be executed on a desired environment or on a robot manipulator.

required to pre-train the robot in. This is further examined in Section 7.2. The following section introduces the interface standard defined by OpenAI's Gym.

7.1.1 OpenAI Gym Interface

The mentioned environment called Gym has some standardised function calls and a standard way to describe different part of the program. The standard function calls are shown in Table 7.1.

Gym Function Calls	
make(id)	Initialises the environment specified by the ID. It is the first function to call and is only called once.
step(action)	The step function makes the agent take the action specified in the call.
render(void)	To get visual feedback, the render function should be called. This requires no additional parameters.
reset(void)	When an episode is terminated (either the goal is met, or the maximum numbers of timesteps are reached) the reset function is called. This resets the environment to some initial state. Resetting requires no additional parameters from the agent.
close(void)	The close function is called to tell Gym that the program is finished with the environment. Thus this function is not used by the RL agent directly.

Table 7.1: The fundamental function calls for Gym interaction.

The functions *step* and *reset* returns different information regarding the environment and performance, which can be seen in Table 7.2.

Gym Return Parameters	
Next state	A set of observations relevant to the environment after an action has been taken.
Reward	The reward received after applying an action.
Done	If this is returned as true, the action either completed the environment or the maximum allowed timesteps has been reached.
Info	Additional info regarding the environment could be used for debugging.

Table 7.2: The returned parameters from the functions step and reset.

Every environment also contains *spaces*, which specifies the different actions and observations available which is explained in Table 7.3.

Gym Spaces	
Action space	Specifies the type and size of actions the agent can take in the environment, e.g. continuous or discrete actions.
Observation space:	Specifies the type and size of the observation space the agent can observe.

Table 7.3: The two spaces type of Gym.

All the mentioned functions and variables need to be implemented in any new custom environment for it to function with the other software aspects such as Keras-rl.

7.1.2 UR Environment

For this thesis, a custom simulation environment was created of a UR5 robot manipulator available in the research facilities at Aalborg University. Thus, the *UREnv* environment was created. Appendix A contains a full UML diagram of the UREnv along with its child environments. A simplified overview of the environments can be seen in Figure 7.2. The UREnv is not meant to be used directly but serves as an abstract class which all future UR environments inherit from. Therefore, the UREnv contains all the required functionalities, e.g. how to handle a reset call.

A child class of UREnv was created called URReach. URReach overwrites some of the functions from UREnv and specifies them to the reach objective. The URReach environment defaults to discrete actions; therefore, a child class for the URReach class has been made, called *ContinuousURReach*, which only overwrite the action function.

7.2 Physics Simulation

There exist different physic simulations where many can integrate a robot manipulator. Examples of such are ARGoS (Pinciroli et al. 2012), V-Rep (E. Rohmer 2013), and Gazebo (Koenig and Howard 2004) which are all free to use. Moreover, paid alternatives exist with RoboDK (RoboDK 2019) and Mujoco (Todorov et al. 2012)

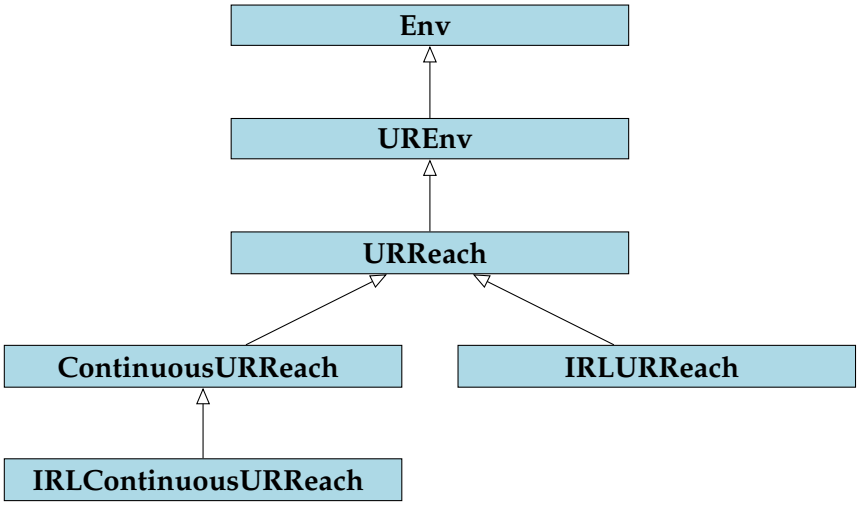


Figure 7.2: An overview of the structure of the different environment classes. The arrow symbolises class inheritance. A full UML diagram can be found in Appendix A

as examples. Some of the paid versions include the correct controllers for the different robot manufactures, whereas the free alternative relies on other solutions to plan, e.g. TCP movements. A survey by Ivaldi et al. 2014 on different robotic simulators showed Gazebo was one of the most known and used tools. Likewise, Gazebo was chosen for this thesis.

7.2.1 Gazebo

To simulate a robot in Gazebo, a description file of the robot is needed. This description file, called *urdf*, describes the joints and links along with the physicals properties for computing the dynamics of the robot. For this thesis, a UR5 urdf file was already provided. The Gazebo environment with the UR5 can be seen in Figure 7.3.

An additional aspect of Gazebo is the *sim time* which specifies how much time has passed in the simulation. Meaning with a *real time factor* of 1.0 the sim time and real time is the same. Since this thesis focuses on RL and a reason for using simulation is to speed up the training time, the real time factor is desirably set as high as possible. Different factors are at play to increase the real time factor. These are computer hardware and the complexity of the environment. Thus the following measures were done to increase the real time factor:

Delete sunlight
: Since the goal is to learn an RL agent, light has no other purpose than aesthetics in this use case, and thus deleting it removes the need to compute, e.g. shadows.

Delete ground plane
: When an RL agent explores its environment, it takes random actions which potentially collides with the ground floor. Even when the robot is not colliding with the ground floor, the physics simulations still has to check for

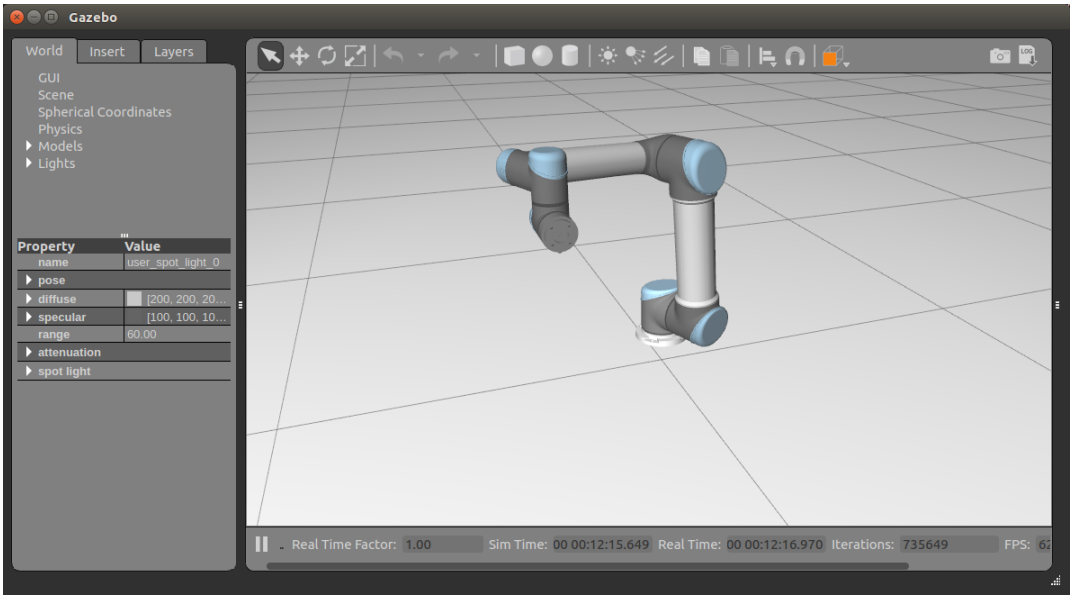


Figure 7.3: The UR5 in the Gazebo simulation.

collision. Since a ground floor can be added virtually in the UR environment, the need for a ground floor in Gazebo is redundant.

Headless mode

The Gazebo environment has a GUI, which is used for debugging and visual feedback. But when training an RL agent, the GUI is not needed for extended training periods, and since it requires computation power to show a GUI, a headless mode is preferred.

Reset robot

When resetting the environment, the robot should be reset to a specific initial configuration. However, in Gazebo, when specifying this initial position, the simulation still needs to plan a path for the robot to reach this position. This includes computing physics and waiting for it to reach the initial pose. When resetting the environment, the physics are unimportant since the only goal is to reach the initial position as fast as possible. Some of the steps to increase the resetting speed were disabling gravity and specifying that the number of timesteps to reach the initial pose must be one, hence it jumps to the position.

With these alterations done, the execution speed is at its maximum allowed by the hardware in use.

Besides the simulation, a path planning tool for the robot is required. The optimal solution would be to use the planner built into the UR5. However, this capability is not available in Gazebo, and therefore, the general purpose planner called *MoveIt* is used as a substitute. Since MoveIt is a general purpose planner, its planning capabilities are not optimised for robot manipulators which caused some issues. Moreover, MoveIt has problems with the maximum rotation of UR5's joints. Therefore a joint limited UR5 is used, which has joint movements restricted to $\pm\pi$ rad. (Moveit Webpage 2019)

When training the UR5 in the simulation, it was discovered that either MoveIt or Gazebo does not handle collision correctly. When the robot collides with itself, the model breaks and the UR5 becomes uncontrollable. To recover from this, the only known solution is to delete the model in Gazebo and re-spawn. This takes seconds to do and is therefore not desirable. Additionally, it was discovered that this caused instability in execution time, which is investigated further in the following section.

7.2.2 Stability & Performance

In this section, the stability and performance of the Gazebo environment are tested. The performance measured is the average time per timestep per episode. For this test, the UR5 had a start and goal position as (also illustrated in Figure 7.4):

Start: $[0.03134514, 0.78650704, 0.08298199]$

Goal: $[-0.6829526, 0.01094951, 0.33524344]$

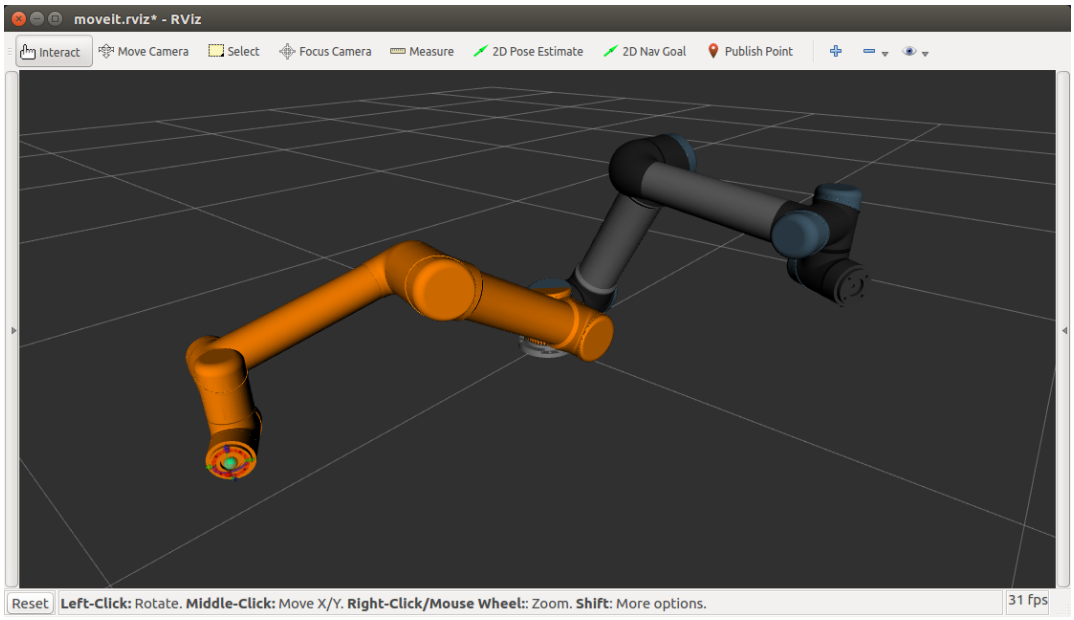


Figure 7.4: (orange) The starting position and **(gray)** goal position for all stability tests.

Both positions are close to the boundaries of the robot's reach, and the Euclidean distance between them is 1.08 m. A DQN agent was set to solve the discrete URReach environment within 100,000 timesteps with a maximum of 100 timesteps per episode, and a relative TCP step size of 0.02 m. In Figure 7.5 the mean time per timestep is shown along with a linear trend line and below is the accumulated crash episodes. It can be seen that around episode 1000, where the robot has crashed 400 times the instability is noticeable. Some episodes have a mean time per timestep of over 10 seconds. Moreover, the increase in execution time also happens in the first 1000 timesteps, where initially the mean time was around 0.1 seconds and around 0.7 seconds after 1000 episodes. The complete executing time of the

100,000 timesteps was 91,020 seconds, which is ≈ 25 hours and 17 minutes, and the environment was not solved.

To validate that the stability problems are related to the deleting and re-spawning of the robot, a second test was carried out. The second test had the same start/goal position, along with all the other parameters. The difference is that the robot was hard-coded to override the agent's action and only moved around in a square, thus avoiding crashes. In Figure 7.6, the result for the second test is shown. It can be seen that the mean execution time is around 0.11 seconds and is close to being constant throughout the test. The total execution time was 10,592 seconds ≈ 2 hours and 56 minutes. Hence it can be concluded that the operation of deleting and re-spawning the robot is the cause of the instability and increased execution time. This also confirms what some participants of the survey by Ivaldi et al. 2014 stated about the stability of robot collision. Since it is in the essence of an RL agent to explore the environment, random movement causing it to crash cannot be avoided. Nonetheless, an attempt to avoid crash situations and accelerate training time is made in the following section.

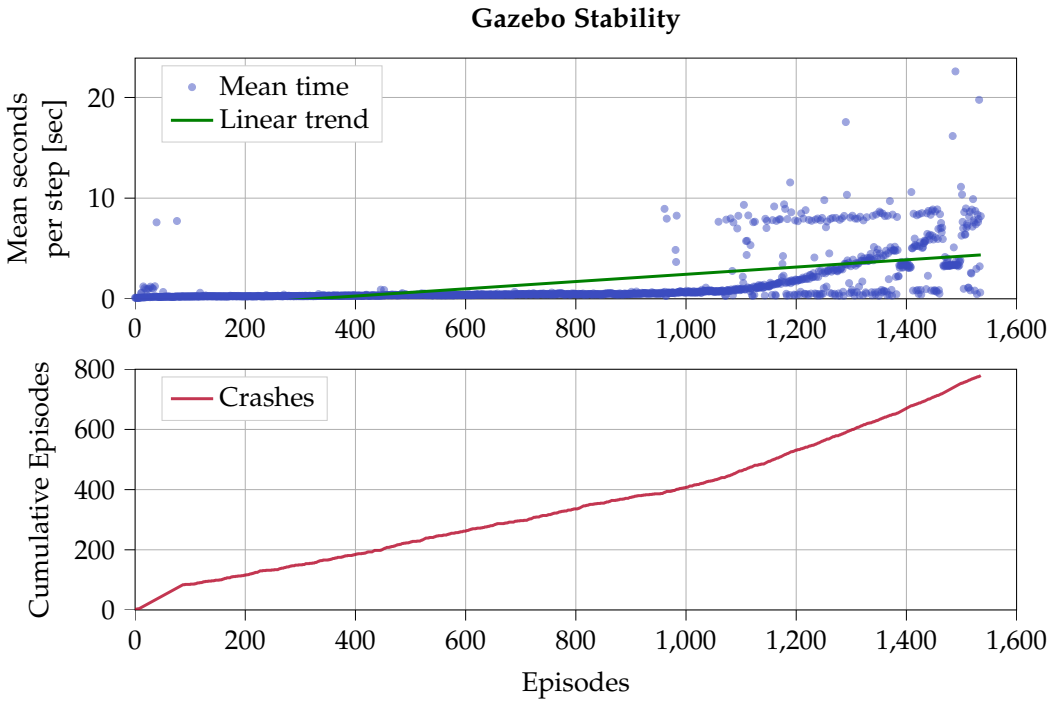


Figure 7.5: (top) The mean timestep time per episode in seconds, along with a linear trend line. **(bottom)** The accumulated crash throughout all the episodes.

7.3 TCP Simulation Environment

This thesis is only concerned with relative TCP movements, and since the physics simulation suffers from instability in performance, a different approach was investigated. This approach is referred to as a *TCP Simulation* environment and focuses only on TCP simulation. Thus all kinematics, dynamics and other physical prop-

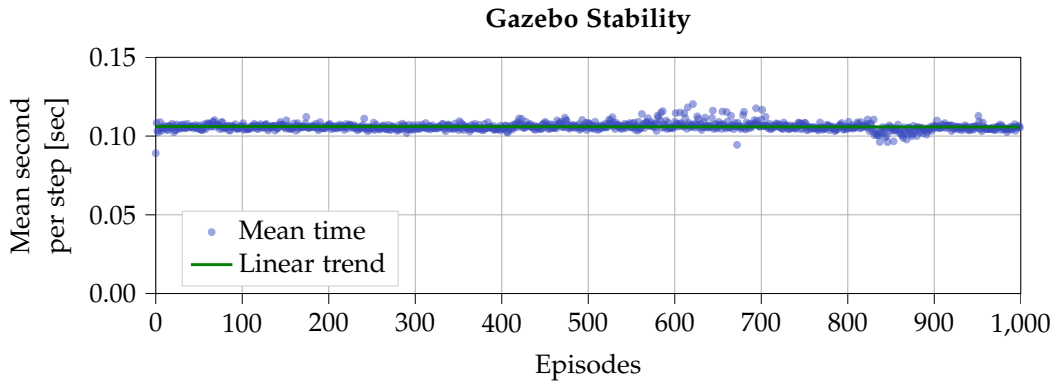


Figure 7.6: The mean timestep time per episode in seconds, where the UR5's movements were restricted.

erties are left out. The environment is built in ROS and only uses transformation matrices to describe the goal and current TCP position. It is not built with a GUI in mind, but it is possible to visualise the environment in the program called RViz, as seen in Figure 7.7. Different *TCPEnv* Gym environments are built to train and test different algorithms. Examples of these are *ContinuousTCPEnv* which makes the environment continuous and *IRLContinuouesTCP3DVR* is built for IRL with different viapoints from the expert data.

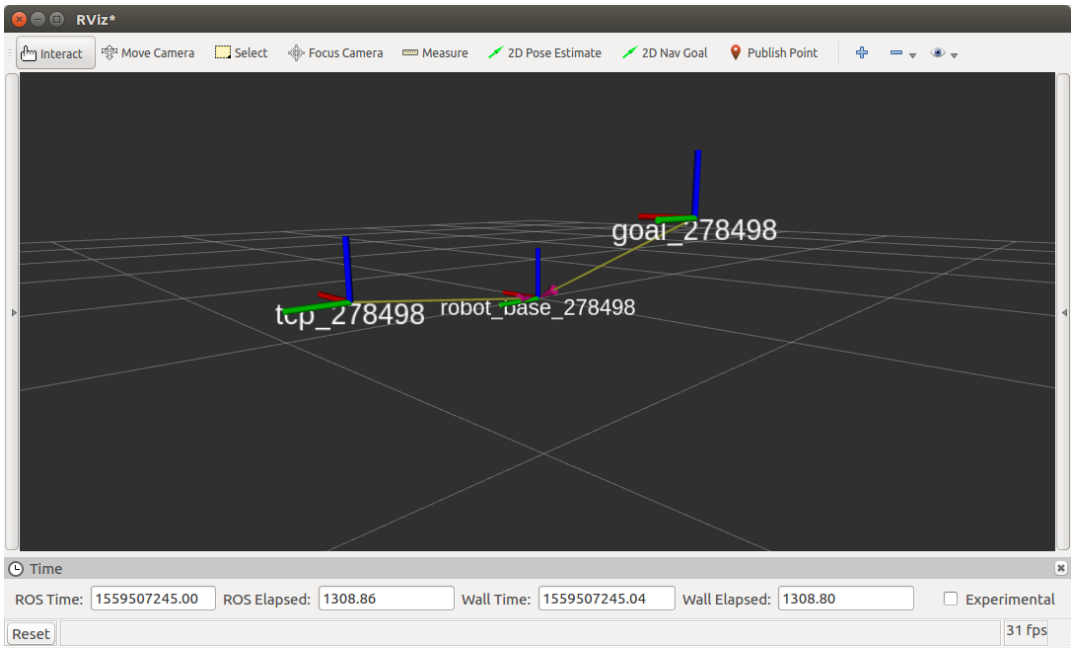


Figure 7.7: The TCP Simulation environment visualised in RViz. The nodes are the TCP, base and goal, where the number trailing the names are unique identifiers for each environment if multiple agents are running.

7.3.1 Stability & Performance

To investigate the stability and performance of the TCP Simulation environment, the stability test performed on Gazebo was repeated. Moreover, the TCP Simulation was also benchmarked using the continuous RL method DDPG. In Figure 7.8 the results are shown. It can be seen that the DQN version has a mean time at around 0.003 seconds per timestep, and it is stable throughout the episodes. The total executing time is 278 seconds $\approx$ 4.6 minutes. The DDPG also had 100,000 timesteps with a maximum of 100 timesteps per episode. It had a mean time around 0.005 seconds throughout the test, with a total time of 501 seconds $\approx$ 8.35 minutes. The reason that the DDPG takes almost double the time compared to DQN is that the DDPG is an actor-critic method and thus has two NN to back-propagate compared to the DQN, which only has a single NN.

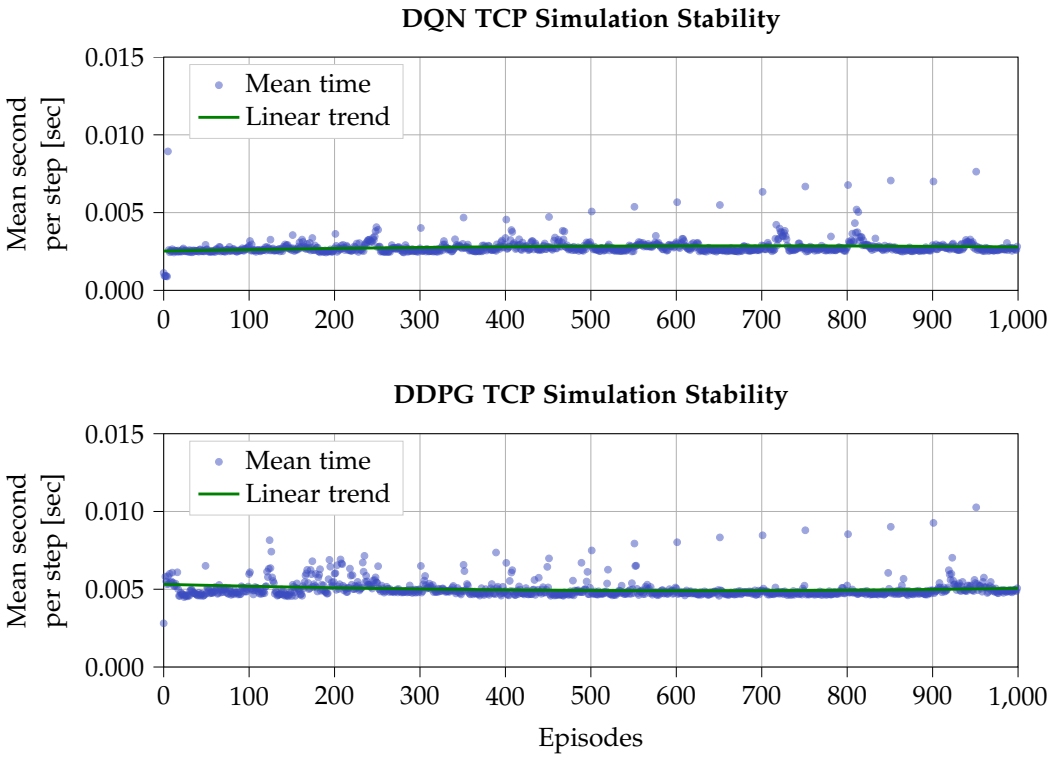


Figure 7.8: The first graph is a DQN network, and the last is the DDPG network. The mean timestep time per episode in seconds is plotted on both graphs along with a linear trend.

Compared with the execution time of physics simulation, the TCP Simulation environments enables the agent to train at a significantly increased speed. However, the agent is not exposed to all the information that a physics engine and simulation provides, and some vital information might be lost.

7.4 Training the Agent for the Real Robot

Instead of taking the learned agent directly from the TCP Simulation environment to the real robot and potentially cause a hazardous situation, a different approach

is recommended. This approach involves pre-training the agent in the TCP Simulation environment. After this initial training, the trained weights are loaded in the Gazebo simulation, and the training continues. Here it is crucial to set the right hyper-parameters in the agent, such that it does not explicitly take random actions. When the training in the simulation is done, the trained agent can then be used at the physical robot. This sequence is also illustrated in Figure 7.9. The gazebo code architecture has been set up such that the switch between the simulation environment and the real UR5 robot works seamlessly. When using the Gazebo simulation, a ROS launch file called *Robot_Demostration.launch* launches all the necessary tools and programs (i.e. Gazebo and MoveIt). After all tools and programs are running the agent program can be executed. If it is desired to use the agent on the real UR5 then *Robot_Demostration.launch* is **not** launched. Instead, the *ur_modern_driver* (which is the ROS driver that can control the UR robots (Andersen 2015)) is launched along with a MoveIt control. The UREnv automatically detects that it is the real robot, and all the specific Gazebo commands are ignored.

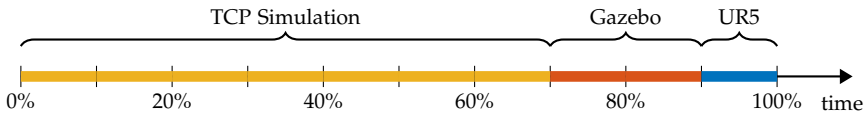


Figure 7.9: Proposed time distribution when training an agent.

A problem with the TCP Simulation environment is that it only works for TCP position. However, since transformation matrices can describe the robot kinematics, it is possible to create a TCP Simulation environment which handles joint movements by manually creating a kinematic model of the robot.

Chapter 8

Trajectory Learning

The following chapter is a description of the RL tests done in this thesis. The last research question from Chapter 5 surrounds the topic of IRL. With the ability to capture expert data and efficiently simulate relevant RL environments, the last step is to implement IRL and test it on the DMRI use case. The tests include several smaller examples of both RL, Linear IRL and an imitation approach to RL. Section 8.1 presents the results of two implementations of the Linear IRL algorithms introduced in Section 3.4.2 on three simple environments that serve as a benchmark of how the methods handle discrete and continuous action and state spaces. These environments are delivered by OpenAI (Brockman et al. 2016) and include CliffWalking, MountainCar, and Inverted Pendulum. Afterwards, in Section 8.2, the same methods are applied to the use case described in Chapter 4. Finally, a discretised version of the use case is presented in Section 8.3 where an imitation approach is made using the Generative Adversarial Imitation Learning approach described in Section 3.4.3.

8.1 Linear Inverse Reinforcement Learning

The following section shows the result of multiple implementations of RL and IRL algorithms. Simulation environments have been selected from existing OpenAI (Brockman et al. 2016) implementations which already have defined the observations and given an ability to capture them. A regular RL method has been used on each of the environments, which then serves as an expert for the IRL giving the ability to record expert observations at will. The results of the expert and the recovered policy are then compared. The IRL method used in this chapter is limited to the two linear versions described in Section 3.4.2. In order to determine the suitability of these methods, four environments are tested all with different properties, as shown in Table 8.1. The same IRL algorithm is applied to all of the environments with the only difference being what features they use ($\phi(s)$ from Equation 3.26).

8.1.1 CliffWalking

The CliffWalking environment is a simple form of a grid world where the agent traverses a discrete grid. Since both observations and actions are discrete, according

Environment	Action space			Observation space		
	Min	Max	Domain	Min	Max	Domain
CliffWalking	1	4	$\in \mathbb{N}$	1	48	$\in \mathbb{N}$
MountainCar	1	2	$\in \mathbb{N}$	-1.2 -0.07	0.6 0.07	$\in \mathbb{R}$ $\in \mathbb{R}$
Inverted Pendulum	-2	2	$\in \mathbb{R}$	-1 -8	1 8	$\in \mathbb{R}^2$ $\in \mathbb{R}$

Table 8.1: The size and shape of the action- and observation space of the tested environments.

to Table 3.1, Monte Carlo, Sarsa, and DQN are suitable for this type of environment, from which a DQN has been used for the following tests. The DQN contains three hidden layers, each with 16 nodes and relu activation functions. The agent uses a constant ϵ-greedy exploration policy with $\epsilon = 0.1$ along with a discount factor of $\gamma = 0.99$. In the original environment, the agent receives a reward of -1 for every timestep, except if it walks off a cliff for which it will receive a reward of -100. An outline of the original environment can be seen in Figure 8.1.

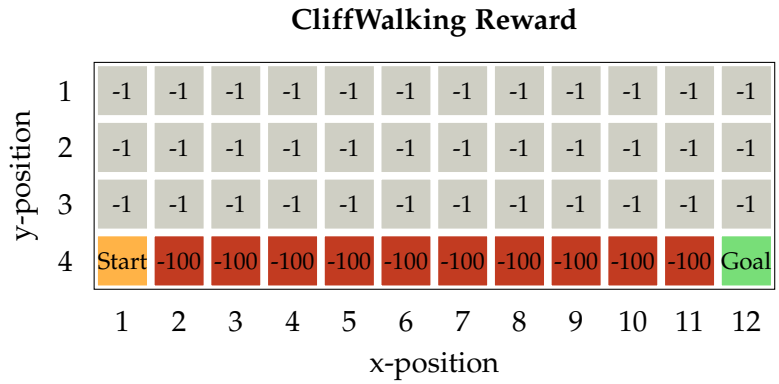


Figure 8.1: Original CliffWalking environment. Selectable actions are moving in the four directions (up, down, left, right).

Since this environment is discrete, the features are simply chosen to be the weight of each state (i.e. $\phi(s) = 1$), by representing the position of the agent as a 1 and all other positions in the grid as 0. Thus the state is a vector with 48 elements, and the weights have the same size. The goal is then to learn how much each state (or agent position) should be weighted. Running the implementation of Algorithm 5 gives the results shown in Figure 8.2 where it can be seen how the goal has been captured. Note the arrow representing the path the agent found optimal is equal the optimal path for the original reward function in Figure 8.1.

Training a new agent with this recovered reward function gives a convergence rate faster than using the original reward, as shown in Figure 8.3. One reason for this can be the recovered reward being more descriptive than the original reward since all states do not have the same reward and thereby, it is easier to tell them apart.

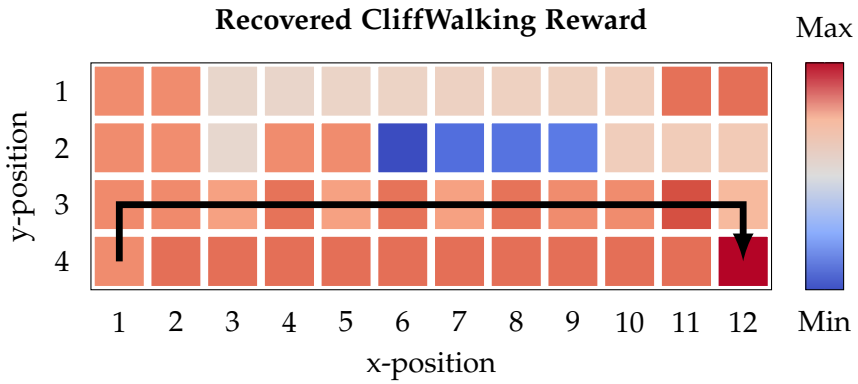


Figure 8.2: Recovered reward function from CliffWalking.

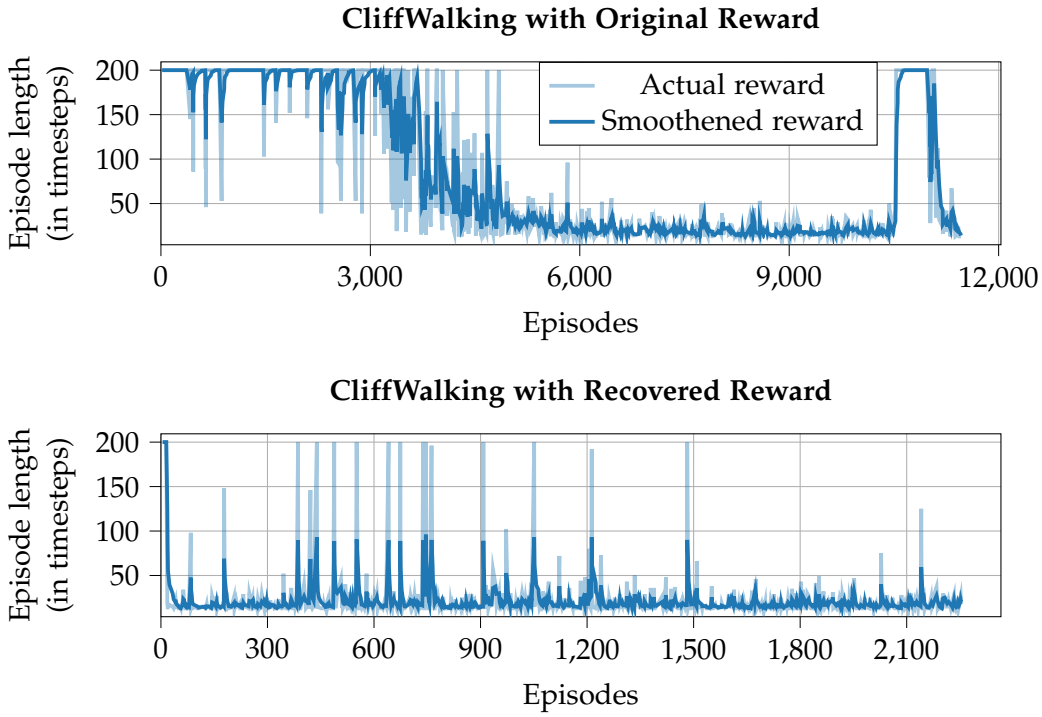


Figure 8.3: The episode length when training using the original reward function **(top)** and the policy trained from the recovered reward in Figure 8.2 **(bottom)**. Note that the values have been smoothed with 0.6 and the scale of the two x-axes.

8.1.2 MountainCar

The MountainCar environment has a continuous observation space in the form of the position and velocity of the car and discrete actions. Therefore, according to Table 3.1, a DQN agent is suitable for this type of environment. The DQN has three layers, each with 16 nodes and relu activation functions. The agent uses a linearly annealed ϵ-greedy policy starting with $\epsilon = 1$ and ending with $\epsilon = 0$ after $100,000$ timesteps. Similar to the CliffWalking environment, the MountainCar environment gives a reward of -1 for each timestep and an episode is terminated either if more than 200 timesteps have passed, or the car reaches a position ≥ 0.5. A layout of the

environment can be seen in Figure 8.4.

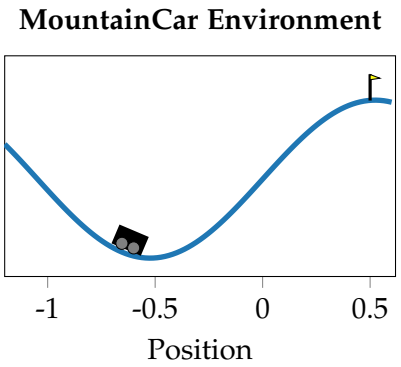


Figure 8.4: Original MountainCar environment.

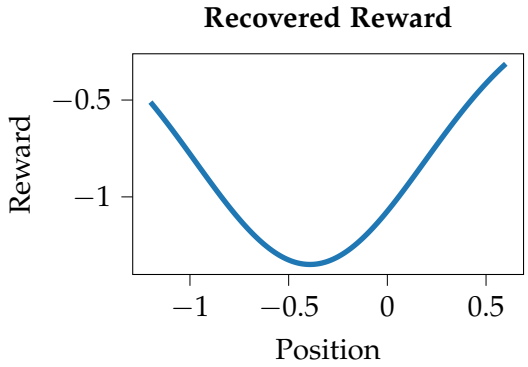


Figure 8.5: Recovered reward for the MountainCar game.

Since this environment contains continuous observations, the feature maps from Algorithm 4 should map a continuous value into the reals $\phi_i : s \to \mathbb{R}$. Similar to the original paper, (Ng and Russell 2000), the feature maps are here chosen to be N normal distributions spread evenly across the position interval of $[-1.2, 0.6]$ and a standard deviation of 0.5. Equation 8.1 shows the definition of each feature function for $i = \{0, \ldots, N+1\}$.

$$\phi_i(s) = \mathcal{N}\left(s \mid \mu = \frac{0.6 - (-1.2)}{N} i - 1.2,\ \sigma^2 = 0.5\right) \tag{8.1}$$

Where μ is a mean, σ^2 is a standard deviation as mentioned earlier and $\mathcal{N}$ is the corresponding normal distribution. Thus the weights computed in Algorithm 4 becomes a weight of which of these normal distributions are important and which are not. The results of running Algorithm 4 when $N = 50$ gives the reward function shown in Figure 8.5. As it can be seen, the reward contains a higher reward on both sides of the hill, encouraging the car to gain momentum.

Training an agent using the recovered reward function gives a similar performance as when training on the original reward. A learning curve to show this is shown in Figure 8.6 where it can be seen how the recovered policy converges to at least the same performance as the expert policy or better.

8.1.3 Pendulum

The third environment benchmark is the standard pendulum environment which contains both a continuous state-space as well as a continuous action-space. Here the goal is to balance a pendulum to an upright position by applying a torque to the centre of the rod. Since this environment contains both continuous observations and actions, according to Table 3.1, either a DDPG, a SAC, or one of the policy gradient agents is necessary, from which a DDPG agent has been selected.

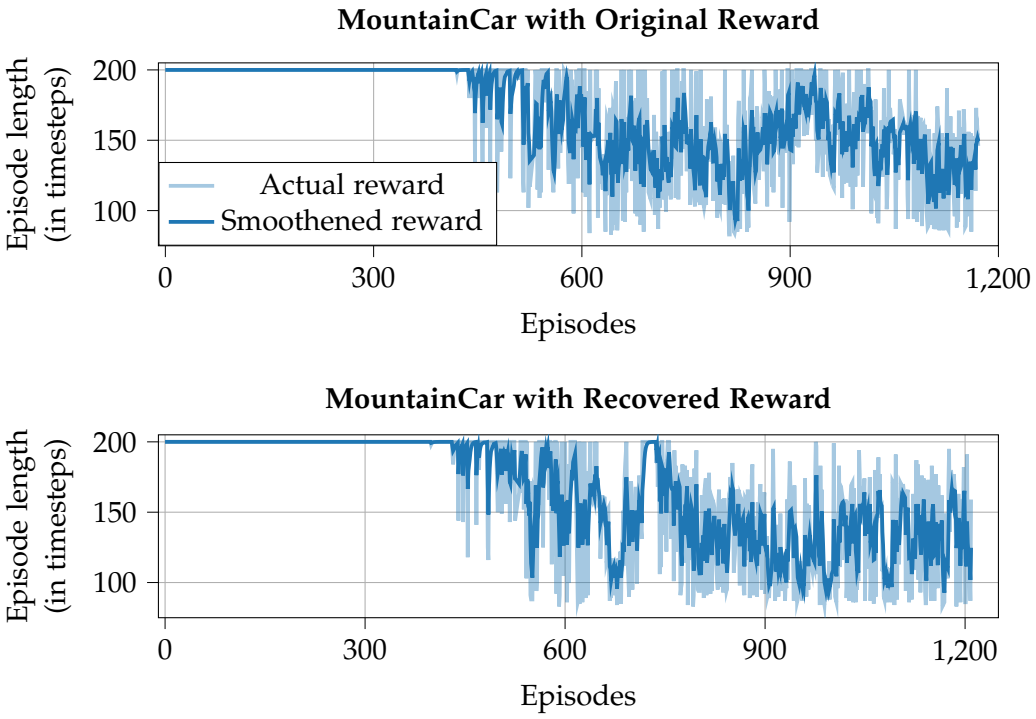


Figure 8.6: The episode length when training using the original reward function **(top)** and the policy trained from the recovered reward in Figure 8.5 **(bottom)**. Note that the values have been smoothed with 0.6.

Both the actor and the critic of the agent contains three hidden layers. The actor has 16 nodes and the critic 32 nodes in each layer. The agent exploration was defined by an Ornstein-Uhlenbeck process (Uhlenbeck and Ornstein 1930) with default parameters. The challenge in this environment is that the torque alone is not enough to force the pendulum to be upright. Instead, the agent has to utilise the momentum of the pendulum. The state-space of the environment is defined to be $[\cos\theta,\ \sin\theta,\ \dot{\theta}]^T$ where θ is the angle of the pendulum from the *upright* position. The standard environment uses the reward shown in Equation 8.2.

$$R_t(s_t) = -\left(\theta^2 + 0.1\dot{\theta}^2 + 0.001\mu^2\right) \tag{8.2}$$

Where μ is the torque set by the agent. A contour map of Equation 8.2 is also shown on the left in Figure 8.7. The reason why this environment is included is due to the structure of the reward function in Equation 8.2 since it already consists of three linearly weighted features. By selecting features $\phi(s) = [\theta^2,\ \dot{\theta}^2,\ \mu^2]^T$, the goal is then to create a reward function with a similar outline as the true reward in Figure 8.7. The recovered reward can be seen in the same figure on the right, and shows how a very similar structure to the true reward is achieved.

The angle over an entire trajectory should, according to Equation 8.2, converge to zero. Figure 8.8 shows the results of running an expert policy trained using Equation 8.2 and a policy returned from Algorithm 4. The results is averaged over 1000 trajectories and show how the policy learned from the recovered reward is

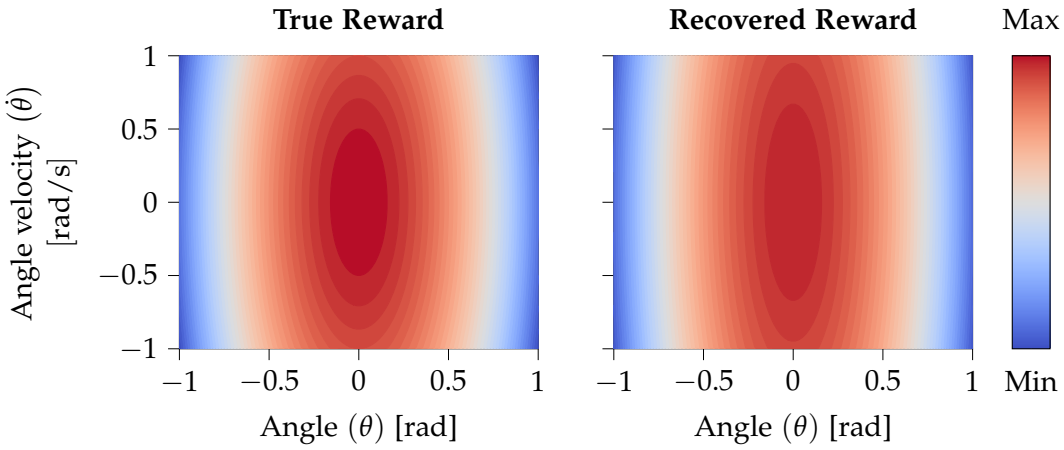


Figure 8.7: Contour map of the true reward (left) and the recovered reward (right) for the pendulum environment when the actions $A = 0$.

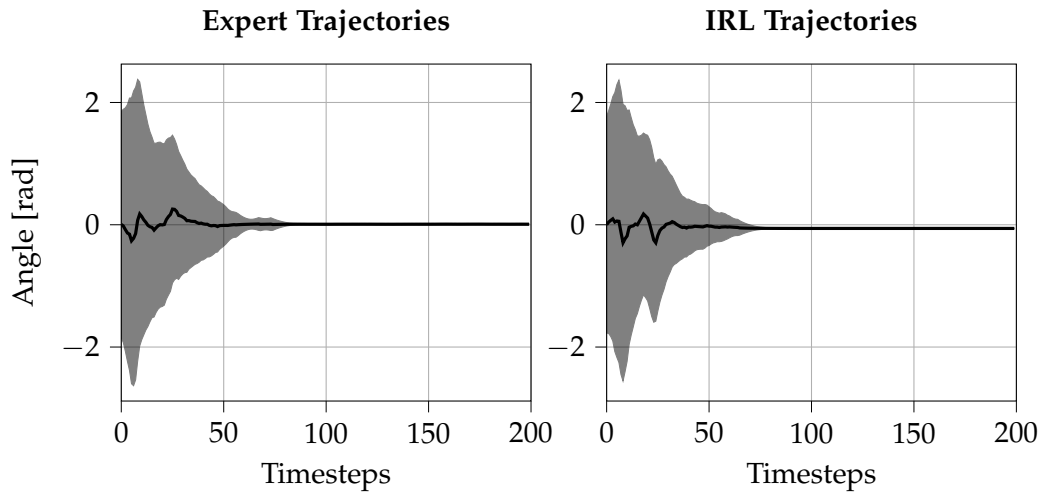


Figure 8.8: The average trajectory of 1000 runs in the pendulum environment including a 1 standard deviation confidence bound. Note that the starting position of the pendulum is random, thus it is expected to have large fluctuations in the first timesteps.

close to having the same performance of the expert policy. It should be noted that the starting position of the pendulum is always random, and thus, it is expected to have large fluctuations in the first timesteps before the policies have time to stabilise the pendulum.

8.2 Robot Trajectory Learning with Linear Inverse Reinforcement Learning

The following section is a description of the result obtained from implementing both RL and IRL on the use case described in Section 4.6. A part of the observations that will be used for the agents was obtained from Chapter 6 that gave a way of recording observations for both an expert and a robot manipulator. Since the

observations recorded with the VR-system is limited to points in 3D space (i.e. x, y, z), the actions, observations, and features ϕ used by any agent has to be a function of 3D points. The observations and actions used for all tests are listed in Table 8.2. Note that the unit of all observations and actions are meters.

Parameter	Description	Definition
Observations	Current TCP Position	$P_t \in \mathbb{R}^3$
	Minimum observed distance to each viapoint	$\mathbf{d}_v(P_t) = \min \parallel P_t - v_i \parallel_2 \in \mathbb{R}^N_{>0}$, $i = \{1, \ldots, N\}$
	Distance to goal	$d_g(P_t) = \parallel P_t - G \parallel_2 \in \mathbb{R}_{>0}$
Actions	Relative TCP position	$a \in [-0.02, 0.02]^3$

Table 8.2: Observations and actions for the environment used to solve the use case from Section 4.6.

This naturally lends itself to an Euclidean minimisation problem, i.e. there is a final position to reach, and along the way there are N viapoints to pass through (or at least near). Figure 8.9 illustrates how the minimum distances develop through a trajectory with two viapoints. This representation simplifies the tasks to; move in the direction of a goal and to pass through the vicinity of a set of viapoints on the way, where each viapoint has some weight of importance associated with it. Thus the reward function becomes a linear combination of d_g and $\mathbf{d}_v$ from table Table 8.2 as shown in Equation 8.3, where w are the weights of the distances to the goal and each of the viapoints.

$$\phi(P_t) = w^T \begin{bmatrix} d_g(P_t) \\ \mathbf{d}_v(P_t) \end{bmatrix} \tag{8.3}$$

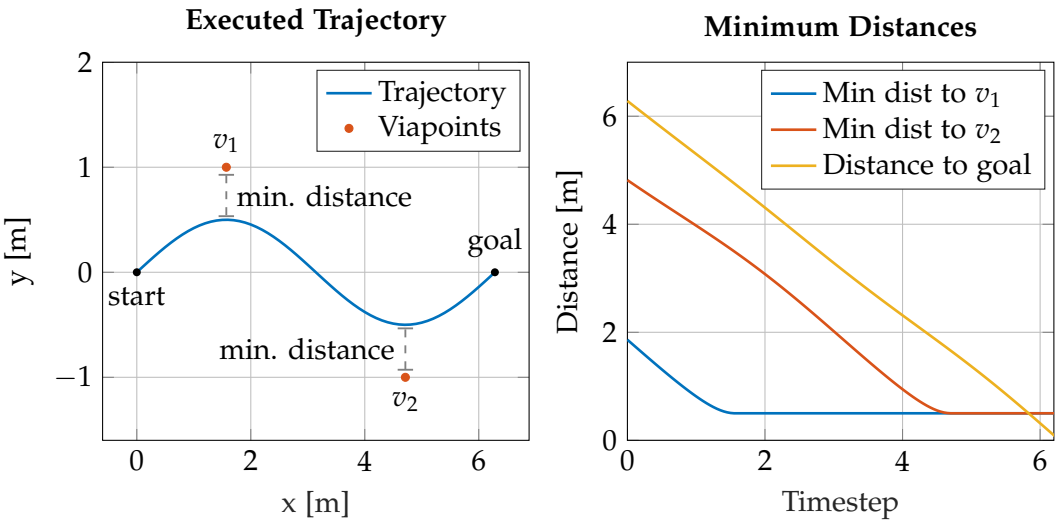


Figure 8.9: (left) A trajectory with two viapoints weighted with the minimum observed distance to the trajectory. **(right)** The minimum observed distance of the viapoints and the goal at each timestep.

The viapoints can be selected arbitrarily - they are ideally only used to guide the trajectory. Therefore three sets of viapoints are used for the tests containing a different amount of viapoints (N):

- **Two viapoints ($N = 2$):** The edge of the aluminium table and the top hook of the Christmas tree, from Figure 6.2.
- **Four viapoints ($N = 4$):** The edge of the aluminium table, top of the Christmas Tree hook, floating between table and Christmas tree, and a distraction point raised above the table.
- **100 viapoints ($N = 10$x10):** 10 viapoints are sample uniformly from 10 expert trajectories.

Since both the observation space- and the action-space is in the continuous domain, a DDPG agent is used as an agent for all tests (as defined in Table 3.1). In order to be able to compare the performance, the same hyper-parameters are used for each agent. Two NNs are used for the ACN with three layers. Each layer of the actor contains 16 nodes, each layer of the critic contains 32, and all nodes have a relu activation function as seen in Figure 3.2. The output layer of the critic has a single node with a linear activation (i.e. no activation: $x = y$). The output of the actor is three nodes - one for each dimension (x, y, z), and with a tanh activation that binds the actions to $[-1, 1]$ which can easily be scaled to a step size of 0.02 m.

The DDPG agents are first run on the TCP Simulation environment from Chapter 7 in order to speed up the training. This gives a pre-trained agent that is biased towards a solution once the Gazebo simulation is run and thus the agent will not crash the simulation more than necessary which has significant performance consequences as was shown in Chapter 7. Algorithm 6 summarises the steps taken when implementing a DDPG agent. All tests in this chapter with RL uses 1000 episodes to learn. This value is hardcoded, so the output policy may have overfitted, or it may not have converged yet.

Algorithm 6 Summary of steps to take for solving an MDP using RL.

```
for e = 0 → E do                                   ▷ Number of episodes to run
    s_0 = env.reset()                              ▷ Observe initial state
    for t = 0 → T do                               ▷ Number of timesteps in the episode
        a_t = π(s_t)                               ▷ Select action following policy π
        s_{t+1}, r_t = env.step(a_t)               ▷ execute action, observe next state & reward
        Update policy using s_t, a_t, s_{t+1}, r_r ▷ See DDPG in Section 3.3.1
    end for
end for
```

The first IRL approach selected to test is Linear IRL set up as a linear programming problem. Algorithm 7 summarises the necessary steps taken to run a Linear IRL algorithm and corresponds to the steps taken in this section.

Algorithm 7 Summary of steps to take for solving an MDP using IRL.

```
for I = 0 → I do                                   ▷ Number of iterations to run
    Compute reward function                        ▷ e.g. by using Algorithm 4
    Solve MDP with computed reward function using Algorithm 6
end for
```

It should be noted that the trajectories produced by both the generated policy and the expert are clipped to a length of 75 timesteps to ensure the expected trajectory is produced over the same interval. Additionally, the discount factor used to compute the expected trajectory is $\gamma = 0.99$. Since **line 3** in Algorithm 7 includes solving a complete MDP problem, which has to be done multiple times, the number of episodes to run for all IRL tests are reduced from the initial 1000 to 750. Otherwise, the same settings are applied, i.e. the number of episodes is constant, and thus the agent may not have converged yet.

The results of running regular RL on the first test consisting of two viapoints are shown in Figure 8.10. On the left is the learned trajectory which clearly shows how the trajectory goes through the first viapoint, but only passes near the second viapoint and the goal. The learning curve on the right indicates that the agent has converged. Similarly, the IRL results shows how the agent no longer goes directly to the first viapoint, and still do not visit the second viapoint directly. However, it does achieve reaching the goal position. Comparing the two trajectories with the expert data also shown, it is clear that while none of them successfully replicate the expert trajectory, the regular RL achieves a simpler and closer fit. While the learning curve for RL has converged to a stable signal, the learning curve for the IRL is more fluctuating. The IRL curve also indicates that 750 episodes were not enough since it looks like the curve is upwards going.

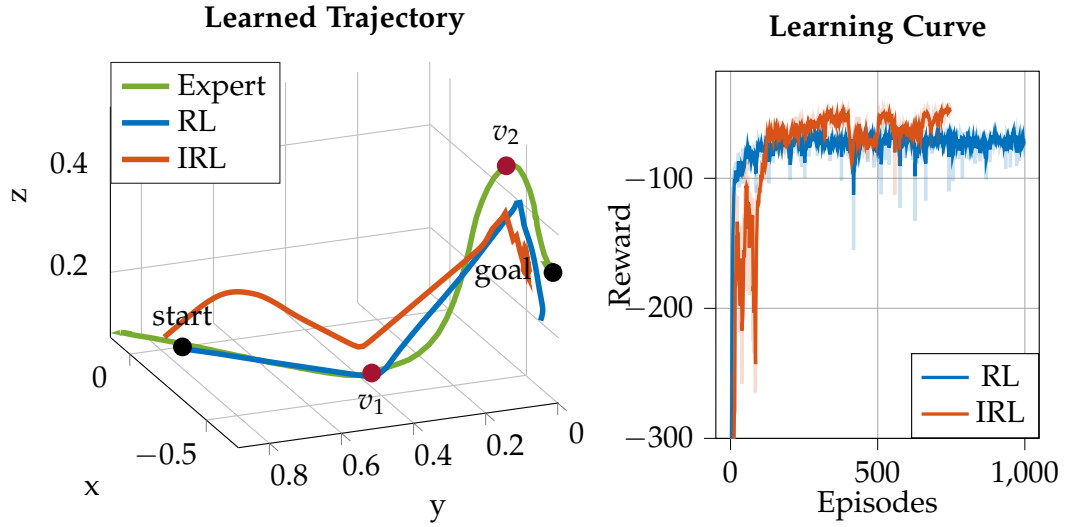


Figure 8.10: A learned trajectory with two viapoints. Note that the reward function has been smoothened with a value of 0.6.

The second test indicates that the agent is not able to differentiate between relevant viapoints and noise when using regular RL. As it can be seen in the figure to the right in Figure 8.11, the reward does not converge as indicated by the dip in the end. The produced trajectory in the same figure is also far from the goal, meaning the task is not solved. An explanation for this behaviour can be found in that since all the viapoints are weighted equally, the distraction point is pulling the agent out of course, and thus the optimum is to stay in between the goal and the distraction point. Differently is it for the IRL test, where a rough structure of the expert data

can still be identified. Though the trajectory deviates from the expert trajectory from the beginning, it partly recovers the structure of the arc in the last half of the trajectory. This also means that the learned trajectory reaches the goal. The dip at the end of the learning curve for the RL test indicates that the agent stopped at a suboptimal configuration. If the agent had been stopped earlier or had more episodes to train, it may have solved the task, however, this was not investigated further.

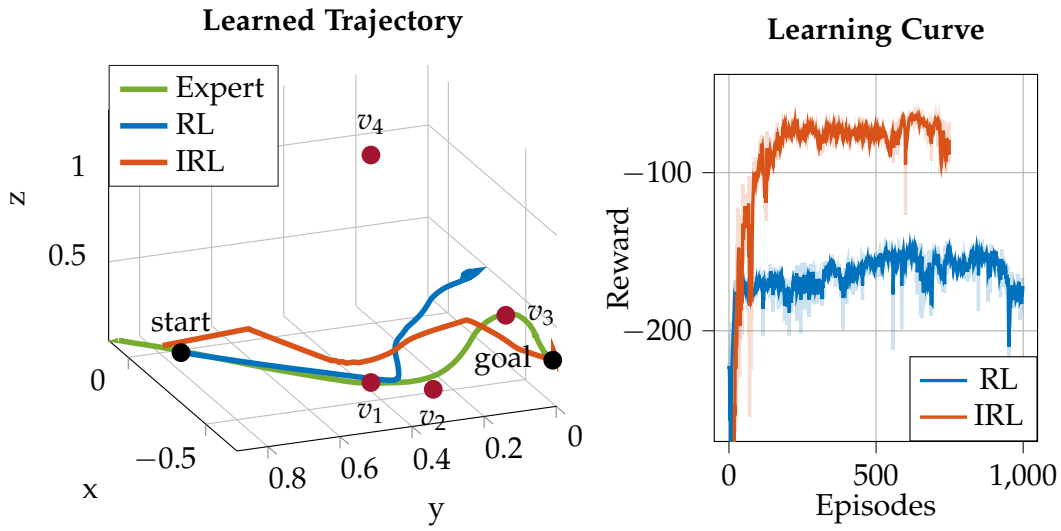


Figure 8.11: A learned trajectory with four viapoints. Note that the reward function has been smoothened with a value of 0.6.

The third test is done to investigate the behaviour of the agent when the trajectory is well defined with many viapoints. The optimal trajectory is to follow the viapoints since the distance only has to be minimised once, and this is also what the agent learns from the first half of the trajectory as shown in Figure 8.12. After the first half, the produced trajectory starts deviating from the path of viapoints. This deviation coincides with when the upwards motion is starting, and a possible reason for this behaviour can be that the many viapoints create a complex reward function that is difficult for the agent to optimise. Since all the viapoints only contribute with a minimum distance to the total reward function, the goal gets overshadowed. When the agent starts moving away from the goal, all the viapoints stay constant, and the only encouragement the agent has is the goal. As shown in the reward plot, the scale of the reward is significantly higher than the encouragement of the goal, and thus the gradient of the reward function is close to zero when moving away from the goal. A near constant reward function gives no information to the agent, and the agent will not be able to learn from it. A simple analogy is 100 people humming noise that gives no information about what direction to go, and a single person giving general directions - the humming will simply drown the single person.

One possible solution to this problem could be to increase the exploration of the agent such that it may "stumble" more on optimal directions. By exploring, the actions predicted by the agent becomes less relevant, and thus, even though the agent

may have no clue of where to go, there is a probability it will get back on track. This, coupled with more training time, may improve performance. Additionally using fewer viapoints will make each viapoint have a more significant influence on the reward function and thereby, the agent may have a better chance of learning. The learning curve for the RL test also has large fluctuations in the end, which indicate that the agent may have become unstable, and thus not capable of solving the environment.

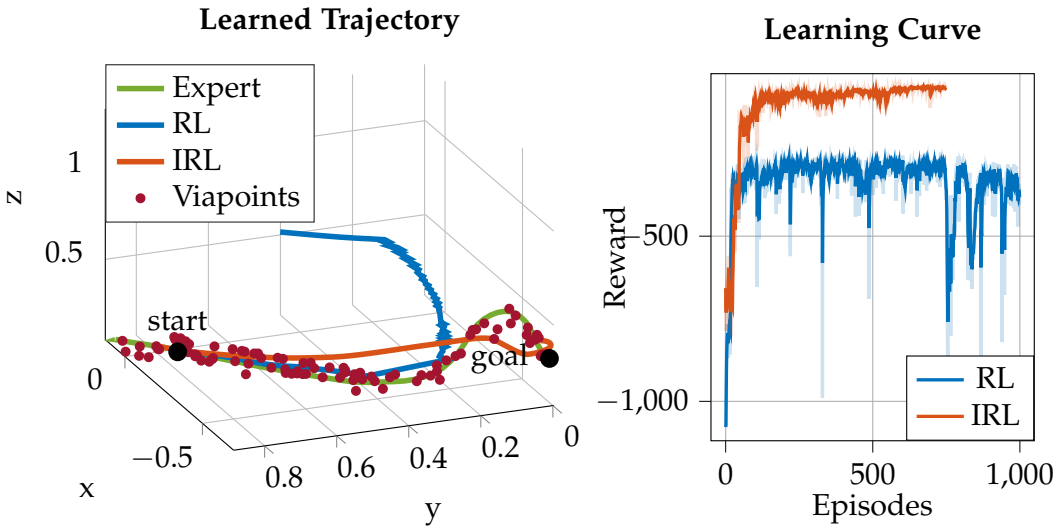


Figure 8.12: A learned trajectory with 100 viapoints. Note that the reward function has been smoothened with a value of 0.6.

The IRL results in Figure 8.12, on the other hand, shows how the agent is capable of approaching the goal. While the agent do not follow the expert data, it does produce a similar structure in that it rises slowly from the table into an arc before descending to the goal again. The learning curve of the IRL test has clearly converged and even has a slightly upward direction, which means the final curve may improve if given more training episodes.

One of the potential problems with the results obtained in this chapter thus far is due to the optimisation in Equation 3.28 (repeated in Equation 8.4) which sums an error between expert data and the generated trajectories.

$$\begin{aligned} \text{maximise} \quad & \sum_{i=1}^{k} \left(V^{\pi^*}(s_0) - V^{\pi}(s_0) \right) \\ \text{s.t.} \quad & |w_i| \leq 1, \quad i = \{1, \ldots, k\} \end{aligned} \tag{8.4}$$

The generated policy will generally have a higher reward than the expert policy on the same reward function because it was fitted to the specific reward function. Combined with the observations used that are strictly positive (since they are Euclidean distances), the error will be decreasing resulting in scenarios where the weights from Equation 8.4 will be optimised to have the same values. One such example is the results of running Algorithm 4 with two viapoints which find the

optimal weights to be $[-0.577, -0.577, -0.577]$. Even though the weights converge to the same value, the plots produced still vary as shown in Figure 8.13 where the first 10 iterations of the algorithm have been plotted for two viapoints. However, this behaviour was only observed in the test with two viapoints. For both four and 100 viapoints, the weights all had different values. The learned weights for all three sets of viapoints can be found in Appendix B.

Another source of errors is the random starting position of the expert trajectories, i.e. the state at timestep t can for one trajectory be at viapoint two, whereas another trajectory is only halfway to viapoint two. This becomes relevant when the expected trajectory is computed, as shown in Figure 8.14. Even though each single expert trajectory solves the task, the expected expert trajectory when averaged over timesteps do not. This behaviour may also explain why all the trajectories generated by IRL in this chapter does not reach the second viapoint (v_2 in Figure 8.14).

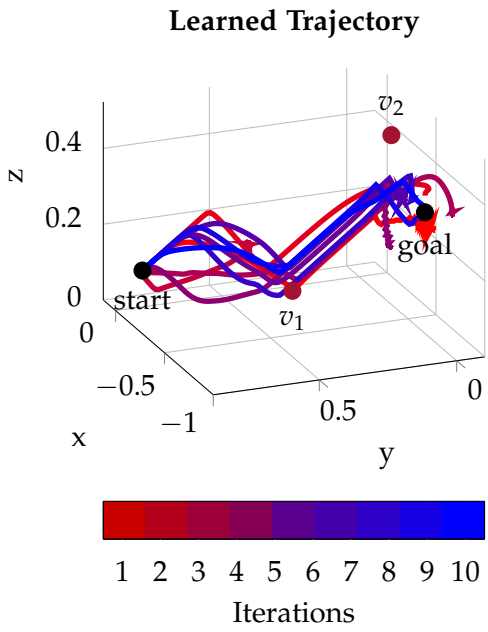


Figure 8.13: 10 iterations of Algorithm 4.

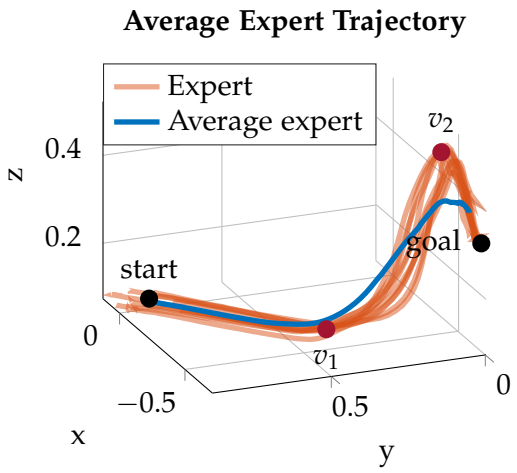


Figure 8.14: The average trajectory used for IRL does not follow the trajectory of the individual expert trajectories due to the random starting position.

Apprenticeship Learning

Since Algorithm 5 sets up the optimisation problem as a quadratic problem, the weights are dependent on each other by a norm constraint unlike with Algorithm 4. This means that there is a lower probability that the optimal combination of features has to be weighted equally. To test this, Algorithm 5 was used in Algorithm 7. The same tests with 2, 4, and 100 viapoints are then run which produced the trajectories in Figure 8.15. From this figure, it can be seen that this IRL approach does not produce better fitting trajectories compared to the previous approach. The only trajectory that partly produces a similar trajectory is with two viapoints, though this trajectory fluctuates in the beginning and completely omits the lifting motion in the end and instead goes directly to the goal.

A pitfall of quadratic programming in combination with the used distance features in this thesis is a significant sensitivity toward positive weights. If a positive weight is assigned to d_g, the optimal policy related to the constructed reward function will move as far away from the goal as possible. Thus the feature expectation of the returned policy and the expert will be significantly different, breaking the intention of apprenticeship learning. A possible solution to this might be to construct features which are not as sensitive to positive weights as Euclidean distances.

Even though some of the tests showed a visual better result for IRL than traditional RL, the set of features used were designed through a process with similarities to constructing a traditional reward function. This was similar for the two implemented Linear IRL methods.

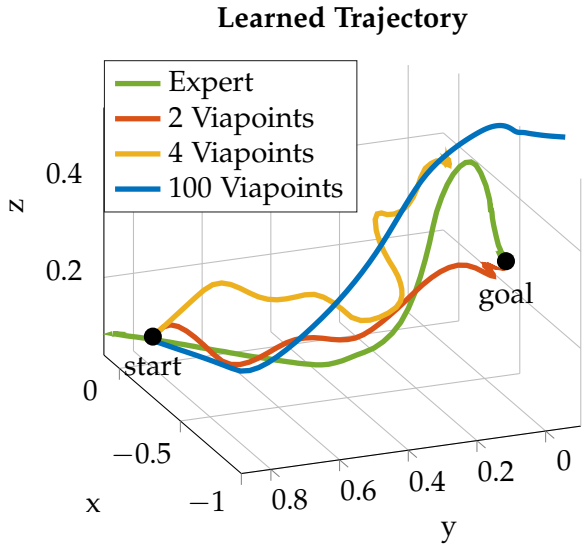


Figure 8.15: An expert trajectory along with learned trajectories for 2, 4, 100 viapoints using quadratic programming IRL from Algorithm 5.

8.3 Robot Trajectory Learning with Deep Imitation Learning

As mentioned in Section 3.5, a different approach to shaping a reward function is to imitate the expert data. One example of doing this was using Generative Adversarial Imitation Learning. In this section, a discrete version of PPO is used to generate policies that a discriminator will learn to classify trajectories from. Since the expert data also has to be discretised, the trajectory will look different than the actual trajectory due to downsampling and rounding errors. Since this is an imitation problem, there is no need for defining external features such as the viapoints used in the previous implementations. The final trajectory can be seen in Figure 8.16 where a single expert trajectory is shown along with ten policy generated trajectories.

Discretising the trajectory heavily influences the shape of the expert trajectories,

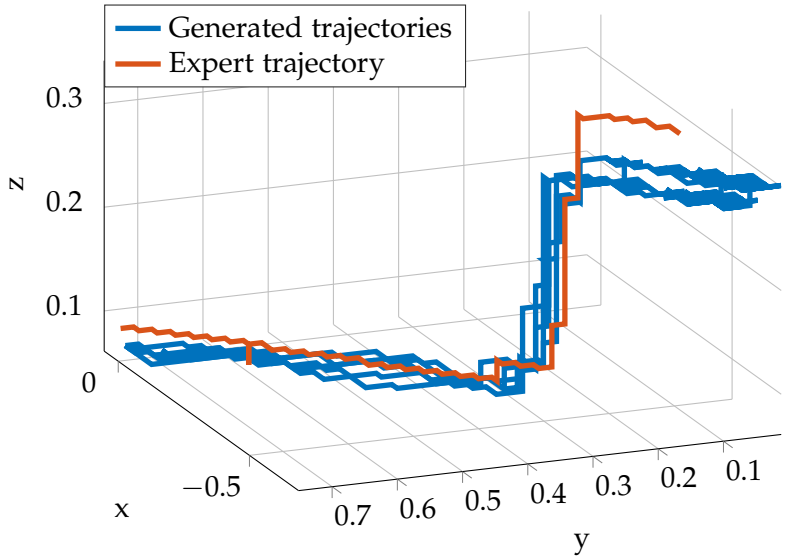


Figure 8.16: An imitated trajectory based on discretised expert data.

and in reality, this trajectory would not be able to solve the task. A solution to this could be using a continuous version of GAIL. Such version is available in Stable Baseline; however, this implementation collides with the software architecture described in Chapter 7. Nonetheless, this test indicates that an imitation approach may be better at fitting to a static trajectory.

Chapter 9

Discussion

Chapter 4 presented a method called Reinforcement Learning Complexity (RLC) for evaluating the complexity of introducing RL in 15 use cases. It appeared while the expert panel evaluated a use case, subjects such as safety, reward shaping, and partially observable physical aspects were discussed. While scoring the use cases, it was noticed that the expert panel did not fully have the know-how of the use cases. Therefore, this method should be used together with experts with process specific knowledge within the use case to get a complete outcome. It is not possible for an RL expert panel to give a clear, trustworthy score of a topic such as healthcare and safety as the use case with Life Science Robotics. For these reasons the RLC score might not give the full picture.

The scoring of the weights can be an issue when giving a high weight (rating it important) to a small scored variable, resulting in the RLC score being reduced unless the other general variables are weighted high.

When the expert panel scored the use cases, it was noticed that although the final RLC score does not show the whole picture, it still functioned as a discussion on both how to solve the task at hand but also notice the use case shortcomings. This gives the ability to plan the RL task, and thus potentially avoid future issues.

As described in Chapter 6-8, the process of using IRL contains multiple steps all of which have a significant impact on the performance of the final results. These steps are summarised in Procedure 1 and serves as a general guide to how IRL was implemented in this thesis.

Procedure 1 Steps necessary for IRL.

1: Identify a task where RL is to be used.
2: Identify observations and actions for the task (e.g. positions, forces, etc.).
3: Apply sensory equipment to record desired observations and actions.
4: Record observations and actions from expert performing the task.
5: Select an RL approach from Table 3.1 to be used with IRL.
6: Select an IRL approach from Table 3.2.
7: Run IRL until desired outcome has been achieved.

This procedure is an attempt at generalising the steps needed to perform IRL and does therefore not convey the difficulty of each step. The first steps of the procedure are in some cases simple to prepare and perform, while the later steps requires significant machine learning know-how.

Chapter 6 presented different approaches to collecting expert data of which the direct expert learning was chosen in the form of a VR-system. This leaves a limited amount of observable features in the form of Cartesian positions. Since this sparse information gives no information about dynamics of the system, the results presented in Section 8.1 may have suffered from an incomplete set of information. None of the simulations incorporate this kind of dynamics. While Gazebo does have a physics simulator, the timesteps are defined to be relative Cartesian movements and does not inform any learning agent of the dynamics other than collisions. This means the agent will take large steps from the very first timestep making the feature matching more inaccurate. In all test performed in this thesis, the timestep was run deterministically meaning the step was not over until the robot had finished moving (e.g. the velocity will always be zero at each timestep). Running each timestep of the environment at a fixed frequency would possibly alleviate the problem of lacking dynamics since the simulation can affect the observations.

A more rich information space could be obtained for the robot manipulator as presented in Section 3.4.1 by doing kinesthetic teaching. By doing so, the convergence of the RL algorithms (for both regular RL and IRL) may have been faster, which ultimately could have resulted in trajectories that are more similar to the expert data. Testing this is left as future work due to time constraints, but the results of comparing the approach in this thesis with the results of an equivalent setup with kinesthetic teaching could show which of the approaches presented in Section 3.4.1 is more suited for the task solved in this thesis. Using the same line of thought, the actions done by the agent could be either relative or absolute joint angles of the robotic manipulator. Thus there are no risks associated with the planner (in this case MoveIt) encountering singularities since it can be circumvented completely. An additional benefit of this is that it would no longer be necessary to plan a Cartesian trajectory which would speed up the simulation. If the Gazebo simulation could be sped up sufficiently, the TCP Simulation environment would also be redundant making the training process simpler and more realistic.

The solutions in Section 8.1 only covered linear approaches with the exception of a discrete version of GAIL. Since there were no physical test to compare the expert trajectory and the generated trajectory other than a visual comparison, it is difficult to say if the generated trajectories would be able to solve the task. However, in some of the cases there are a clear discrepancy between the expert trajectories and the generated trajectories. This could indicate that the approach in this chapter is not well suited for the kind of use case selected in this thesis. Other work, such as Guided Cost Learning (Finn et al. 2016) and Generative Adversarial Imitation Learning (Ho and Ermon 2016), has shown successful implementations of different similar tasks.

Chapter 10

Conclusion

As presented in Chapter 1, production systems are required to adapt to a high variety of customer demand, customisation, and globalisation. This requires models that are adaptable to these factors and can self-adjust depending on the current state of the manufacturing company. One approach to adaptability is using machine learning in the form of reinforcement learning. However, few industrial applications utilise this kind of adaptable models. Chapter 4 proposed a method called Reinforcement Learning Complexity (RLC) for evaluating the complexity of using reinforcement learning in different industrial cases. RLC served as a simple scoring to identify tasks which contains the least complexity if reinforcement learning was to be applied. However, The scoring of the 15 different use cases might serve best as a platform for discussion between industry experts, with the tasks specific knowledge, and experts within machine learning.

One of the aspects of reinforcement learning that often requires significant engineering is the shaping of a reward function that accurately describes the true goal of a task. One approach to this is using expert data to model the reward by inverse reinforcement learning. To investigate inverse reinforcement learning for a pick-and-place task at DMRI, the following problem formulation was defined: *How can an IRL algorithm with a robot manipulator be set up and used to solve a pick-and-place task at DMRI with the use of recorded expert data?*. This was then refined into three research questions.

The first research question of this thesis was; *How can expert data be collected and utilised for the task at DMRI?*. This was answered in Chapter 6 by using an imitation data collection approach. This approach was then further limited such that the teacher and the learner (i.e. human and robot) would share the same observation space meaning that the only mapping between human and robot is a static transformation. The shared observation space was captured using an HTC VIVE virtual reality system. The static transformation between human and robot was found using a world to base calibration. The accuracy of this calibration depended on the number of calibration points used where 4, 6, 8, 10, and 12 were tested. While 6 calibration points produced the best norm error, any of the tested numbers of calibration points would have been sufficient for this thesis.

An integral part of reinforcement learning is the time-consuming training. The second research question of this thesis was therefore; *How should a training and simulation environment be structured for the task at DMRI?*. The standard for defining simulation environments already defined by OpenAI Gym was used to structure the developed environments in this thesis. By following this standard, a rich se-

lection of freely available tools becomes accessible. The simulation engine called Gazebo was used to simulate a robotic manipulator at an increased speed. However, this simulation suffered from instability caused by the robot crashing during the exploration of the environment. To address this, a simpler TCP Simulation environment was used to bias the agent towards a partly optimal solution. By doing so, the reinforcement learning agent would not have to explore the entire state-space in the Gazebo environment.

The third and last research question; *How can an IRL algorithm be used for a pick-and-place task as the use case and does the performance differ from an RL approach?*, was then answered based on the observation data and the simulation environments presented in Chapter 6 and 7. Two linear inverse reinforcement learning algorithms were implemented, each producing different trajectories. Additionally, the trajectories were compared to a naive reinforcement learning implementation, that in some cases produced visually closer trajectories to the expert trajectories. A discrete implementation of GAIL was furthermore briefly tested, which indicated the potential for inverse reinforcement learning.

The answer to the problem formulation *How can an IRL algorithm with a robot manipulator be set up and used to solve a pick-and-place task at DMRI with the use of recorded expert data?* is thus that expert data for inverse reinforcement learning has to capture rich observations of the task. In linear inverse reinforcement learning the complexity of designing a reward function for traditional reinforcement learning is not entirely omitted, but only decomposed to engineering a set of features directly impacting the complete reward function. All of the implemented solutions were only evaluated visually, and thus, it is not known if any of the solutions actually solve a real industrial task.

Chapter 11

Future Work

In Section 3.4.1 different strategies for collecting expert data were discussed. The strategy chosen for collecting expert data used a principle referred to as direct task space learning. This strategy caused some problems when the expert data was applied on the implemented IRL algorithms. For future work it should be investigated how the expert data can be mapped, such that the feature expectation from the generated policy and the expert data will match better. This could e.g. be achieved by using kinesthetic teaching or securing that the already collected trajectories fits the trajectories generated by the polices in length and density.

Instead of collecting expert data in the experimental setup presented in Chapter 6 it should be done in the real environment at a slaughterhouse by an experienced employee. The task of moving a piece of meat from a start position to a hook, furthermore consists of more steps than following a trajectory, which has been the focus in this thesis. An investigation of how to solve the other steps in the process should therefore be investigated as future work, e.g. pick up the meat and place the meat on the hook.

Other IRL and behaviour cloning algorithms should for future work be investigated on the existing expert data. These algorithms could include Guided Cost Learning (GCL) and continuous Generative Adversarial Imitation Learning (GAIL). A continuous version of GAIL is especially interesting, since the freely available RL framework Stable Baseline (Hill et al. 2018) have an implementation. This library, however, requires Python 3, which is different from the python version currently supported by the simulation environment presented in Chapter 7. The simulation environment should thus be converted or replaced by an environment which can run Python 3, e.g. ROS 2.

Bibliography

Abbeel, Pieter and Andrew Y. Ng (2004). "Apprenticeship Learning via Inverse Reinforcement Learning". In: *Proceedings of the Twenty-first International Conference on Machine Learning*. ICML '04. New York, NY, USA: ACM, pp. 1–. ISBN: 1-58113-838-5. DOI: 10.1145/1015330.1015430. URL: http://doi.acm.org/10.1145/1015330.1015430.

Andersen, Thomas Timm (2015). *Optimizing the Universal Robots ROS driver.* Tech. rep. Technical University of Denmark, Department of Electrical Engineering. URL: http://orbit.dtu.dk/en/publications/optimizing-the-universal-robots-ros-driver(20dde139-7e87-4552-8658-dbf2cdaab24b).html.

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

Argall, Brenna D., Sonia Chernova, Manuela Veloso, and Brett Browning (2009). "A Survey of Robot Learning from Demonstration". In: *Robot. Auton. Syst.* 57.5, pp. 469–483. ISSN: 0921-8890. DOI: 10.1016/j.robot.2008.10.024. URL: http://dx.doi.org/10.1016/j.robot.2008.10.024.

Asis, Kristopher De, J. Fernando Hernandez-Garcia, G. Zacharias Holland, and Richard S. Sutton (2017). "Multi-step Reinforcement Learning: A Unifying Algorithm". In: *CoRR* abs/1703.01327. arXiv: 1703.01327. URL: http://arxiv.org/abs/1703.01327.

Bishop, Christopher M (2006). *Pattern recognition and machine learning*. springer.

Bøgh, Simon, Mads Hvilshøj, Morten Kristiansen, and Ole Madsen (2012). "Identifying and evaluating suitable tasks for autonomous industrial mobile manipulators (AIMM)". In: *The International Journal of Advanced Manufacturing Technology* 61.5, pp. 713–726. ISSN: 1433-3015. DOI: 10.1007/s00170-011-3718-3. URL: https://doi.org/10.1007/s00170-011-3718-3.

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

Calinon, Sylvain and Aude Billard (2007). "Incremental Learning of Gestures by Imitation in a Humanoid Robot". In: *Proceedings of the ACM/IEEE International Conference on Human-robot Interaction*. HRI '07. Arlington, Virginia, USA: ACM, pp. 255–262. ISBN: 978-1-59593-617-2. DOI: 10.1145/1228716.1228751. URL: http://doi.acm.org/10.1145/1228716.1228751.

Casler Jr, Richard J (1986). *Method and apparatus for manipulator welding apparatus with improved weld path definition*. US Patent 4,568,816.

Chen, H., T. Fuhlbrigge, and X. Li (2008). "Automated industrial robot path planning for spray painting process: A review". In: *2008 IEEE International Conference*

on Automation Science and Engineering, pp. 522–527. DOI: 10.1109/COASE.2008.4626515.

Chen, Y. F., M. Everett, M. Liu, and J. P. How (2017). "Socially aware motion planning with deep reinforcement learning". In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 1343–1350. DOI: 10.1109/IROS.2017.8202312.

Choset, Howie M, Seth Hutchinson, Kevin M Lynch, George Kantor, Wolfram Burgard, Lydia E Kavraki, and Sebastian Thrun (2005). "Principles of Robot Motion: Theory, Algorithms, and Implementation". In: MIT Press. Chap. 4 Potential Functions.

Compare, Michele, Luca Bellani, Enrico Cobelli, and Enrico Zio (2018). "Reinforcement learning-based flow management of gas turbine parts under stochastic failures". In: *The International Journal of Advanced Manufacturing Technology* 99.9, pp. 2981–2992. ISSN: 1433-3015. DOI: 10.1007/s00170-018-2690-6. URL: https://doi.org/10.1007/s00170-018-2690-6.

Cosío, F. Arambula and M.A. Padilla Castañeda (2004). "Autonomous robot navigation using adaptive potential fields". In: *Mathematical and Computer Modelling* 40.9, pp. 1141 –1156. ISSN: 0895-7177. DOI: https://doi.org/10.1016/j.mcm.2004.05.001. URL: http://www.sciencedirect.com/science/article/pii/S0895717704003097.

DMRI (2019). *DMRI Homepage*. URL: https://www.dti.dk/dmri.

Doerr, Andreas, Nathan D Ratliff, Jeannette Bohg, Marc Toussaint, and Stefan Schaal (2015). "Direct Loss Minimization Inverse Optimal Control." In: *Robotics: Science and Systems*.

E. Rohmer S. P. N. Singh, M. Freese (2013). "V-REP: a Versatile and Scalable Robot Simulation Framework". In: *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*.

EU (2018). *A brief refresher on Technology Readiness Levels (TRL)*. URL: http://www.cloudwatchhub.eu/exploitation/brief-refresher-technology-readiness-levels-trl.

Finn, Chelsea, Sergey Levine, and Pieter Abbeel (2016). "Guided Cost Learning: Deep Inverse Optimal Control via Policy Optimization". In: *CoRR* abs/1603.00448. arXiv: 1603.00448. URL: http://arxiv.org/abs/1603.00448.

Fortunato, Meire et al. (2017). "Noisy Networks for Exploration". In: *CoRR* abs/1706.10295. arXiv: 1706.10295. URL: http://arxiv.org/abs/1706.10295.

Gabel, T. and M. Riedmiller (2007). "Scaling Adaptive Agent-Based Reactive Job-Shop Scheduling to Large-Scale Problems". In: *2007 IEEE Symposium on Computational Intelligence in Scheduling*, pp. 259–266. DOI: 10.1109/SCIS.2007.367699.

Geniar, Mattias (2016). *Why do we automate?* URL: https://ma.ttias.be/why-do-we-automate/.

Ghalamzan, Amir and Matteo Ragaglia (2017). "Robot learning from demonstrations: Emulation learning in environments with moving obstacles". In: *Robotics and Autonomous Systems* 101. DOI: 10.1016/j.robot.2017.12.001.

Grand, Stephen, Dave Cliff, and Anil Malhotra (1997). "Creatures: Artificial Life Autonomous Software Agents for Home Entertainment". In: pp. 22–29. DOI: 10.1145/267658.267663.

Gu, S., E. Holly, T. Lillicrap, and S. Levine (2017). "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates". In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3389–3396. DOI: 10.1109/ICRA.2017.7989385.

Günther, Johannes, Patrick M. Pilarski, Gerhard Helfrich, Hao Shen, and Klaus Diepold (2016). "Intelligent laser welding through representation, prediction, and control learning: An architecture with deep neural networks and reinforcement learning". In: *Mechatronics* 34. System-Integrated Intelligence: New Challenges for Product and Production Engineering, pp. 1 –11. ISSN: 0957-4158. DOI: https://doi.org/10.1016/j.mechatronics.2015.09.004. URL: http://www.sciencedirect.com/science/article/pii/S0957415815001555.

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

Hafner, Roland and Martin Riedmiller (2011). "Reinforcement learning in feedback control". English. In: 84, pp. 137–169. ISSN: 0885-6125. DOI: 10.1007/s10994-011-5235-x.

Hasselt, Hado van, Arthur Guez, and David Silver (2015). "Deep Reinforcement Learning with Double Q-learning". In: *CoRR* abs/1509.06461. arXiv: 1509.06461. URL: http://arxiv.org/abs/1509.06461.

Hessel, Matteo, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Daniel Horgan, Bilal Piot, Mohammad Gheshlaghi Azar, and David Silver (2017). "Rainbow: Combining Improvements in Deep Reinforcement Learning". In: *CoRR* abs/1710.02298. arXiv: 1710.02298. URL: http://arxiv.org/abs/1710.02298.

Hill, Ashley et al. (2018). *Stable Baselines*. https://github.com/hill-a/stable-baselines.

Ho, Jonathan and Stefano Ermon (2016). "Generative Adversarial Imitation Learning". In: *CoRR* abs/1606.03476. arXiv: 1606.03476. URL: http://arxiv.org/abs/1606.03476.

HTC VIVE (2019). *HTC VIVE*. URL: https://www.vive.com/us/.

Hu, Yazhou and Bailu Si (2018). "A reinforcement learning neural network for robotic manipulator control". In: *Neural computation* 30.7, pp. 1983–2004.

Huang, X., F. Naghdy, H. Du, G. Naghdy, and C. Todd (2015). "Reinforcement learning neural network (RLNN) based adaptive control of fine hand motion rehabilitation robot". In: *2015 IEEE Conference on Control Applications (CCA)*, pp. 941–946. DOI: 10.1109/CCA.2015.7320733.

Indiamart Webpage (2019). *Upvc Window Frame*. URL: https://www.indiamart.com/proddetail/upvc-window-frame-15124921612.html.

Inropa (2016). *Inropa Homepage*. URL: https://www.inropa.com/da/.

Irving, Geoffrey and Amanda Askell (2019). "AI Safety Needs Social Scientists". In: *Distill*. https://distill.pub/2019/safety-needs-social-scientists. DOI: 10.23915/distill.00014.

Ivaldi, Serena, Vincent Padois, and Francesco Nori (2014). "Tools for dynamics simulation of robots: a survey based on user feedback". In: *CoRR* abs/1402.7050. arXiv: 1402.7050. URL: http://arxiv.org/abs/1402.7050.

Jin, Zeshi, Haichao Li, and Hongming Gao (2019). "An intelligent weld control strategy based on reinforcement learning approach". In: *The International Journal of Advanced Manufacturing Technology* 100.9, pp. 2163–2175. ISSN: 1433-3015. DOI: 10.1007/s00170-018-2864-2. URL: https://doi.org/10.1007/s00170-018-2864-2.

Julia (2013). *Creatures Deluxe*. URL: https://fangirled.wordpress.com/2013/07/16/fangirl-flashback-creatures-deluxe/.

Kaneko, T., H. Kameoka, N. Hojo, Y. Ijima, K. Hiramatsu, and K. Kashino (2017). "Generative adversarial network-based postfilter for statistical parametric speech synthesis". In: *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 4910–4914. DOI: 10.1109/ICASSP.2017.7953090.

Khodayari, S. and M. J. Yazdanpanah (2005). "Network routing based on reinforcement learning in dynamically changing networks". In: *17th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'05)*, 5 pp.–366. DOI: 10.1109/ICTAI.2005.91.

Kim, B., Y. Zhang, M. van der Schaar, and J. Lee (2016). "Dynamic Pricing and Energy Consumption Scheduling With Reinforcement Learning". In: *IEEE Transactions on Smart Grid* 7.5, pp. 2187–2198. ISSN: 1949-3053. DOI: 10.1109/TSG.2015.2495145.

Kim, Beomjoon and Joelle Pineau (2016). "Socially Adaptive Path Planning in Human Environments Using Inverse Reinforcement Learning". In: *International Journal of Social Robotics* 8, pp. 51–66. DOI: 10.1007/s12369-015-0310-2.

Koenig, Nathan and Andrew Howard (2004). "Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. Sendai, Japan, pp. 2149–2154.

Koskinopoulou, Maria, Stylianos Piperakis, and Panos E. Trahanias (2016). "Learning from Demonstration facilitates Human-Robot Collaborative task execution". In: *2016 11th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, pp. 59–66.

Ledig, Christian, Lucas Theis, Ferenc Huszar, Jose Caballero, Andrew P. Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, and Wenzhe Shi (2016). "Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network". In: *CoRR* abs/1609.04802. arXiv: 1609.04802. URL: http://arxiv.org/abs/1609.04802.

Lee, Donghun, TaeWon Seo, and Jongwon Kim (2011). "Optimal design and workspace analysis of a mobile welding robot with a 3P3R serial manipulator". In: *Robotics and Autonomous Systems* 59.10, pp. 813 –826. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2011.06.004. URL: http://www.sciencedirect.com/science/article/pii/S0921889011001011.

Life Science Robotics (2019). *Life Science Robotics Webpage*. URL: http://www.lifescience-robotics.com/.

Lillicrap, Timothy P., Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra (2015). "Continuous control with deep reinforcement learning". In: *CoRR* abs/1509.02971. arXiv: 1509.02971. URL: http://arxiv.org/abs/1509.02971.

Lin, S., I. F. Akyildiz, P. Wang, and M. Luo (2016). "QoS-Aware Adaptive Routing in Multi-layer Hierarchical Software Defined Networks: A Reinforcement

Learning Approach". In: *2016 IEEE International Conference on Services Computing (SCC)*, pp. 25–33. DOI: 10.1109/SCC.2016.12.

Lipnevicius, Geoff M (2005). *Robotic cylinder welding*. US Patent 6,942,139.

Madsen, Ole, Henrik Schøiler, Charles Møller, Torben Bach Pedersen, Brian Vejrum Wæhrens, Arne Remmen, and Anders Vestergaard (2014). *Smart Production Et forskningprogram under AAU Production*. Online. Danish. URL: https://www.smartproduction.aau.dk/digitalAssets/221/221365_aau-smart-production.pdf?fbclid=IwAR2xgDFLshxnIu1vuylvIk2YwMNu3kL96Q7VJSeKkURkZIM3kgjdV3lJsrc.

Mankins, John C (1995). "Technology readiness levels". In: *White Paper, April* 6, p. 1995.

Meyes, Richard, Hasan Tercan, Simon Roggendorf, Thomas Thiele, Christian Büscher, Markus Obdenbusch, Christian Brecher, Sabina Jeschke, and Tobias Meisen (2017). "Motion Planning for Industrial Robots using Reinforcement Learning". In: *Procedia CIRP* 63. Manufacturing Systems 4.0 – Proceedings of the 50th CIRP Conference on Manufacturing Systems, pp. 107 –112. ISSN: 2212-8271. DOI: https://doi.org/10.1016/j.procir.2017.03.095. URL: http://www.sciencedirect.com/science/article/pii/S221282711730241X.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. (2015). "Human-level control through deep reinforcement learning". In: *Nature* 518.7540, p. 529.

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

Mogren, Olof (2016). "C-RNN-GAN: Continuous recurrent neural networks with adversarial training". In: *CoRR* abs/1611.09904. arXiv: 1611.09904. URL: http://arxiv.org/abs/1611.09904.

Moveit Webpage (2019). *Moving robots into the future*. URL: https://moveit.ros.org/.

Muelling, Katharina, Abdeslam Boularias, Betty Mohler, Bernhard Schölkopf, and Jan Peters (2014). "Learning Strategies in Table Tennis Using Inverse Reinforcement Learning". In: *Biol. Cybern.* 108.5, pp. 603–619. ISSN: 0340-1200. DOI: 10.1007/s00422-014-0599-1. URL: http://dx.doi.org/10.1007/s00422-014-0599-1.

Nair, Ashvin, Bob McGrew, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel (2017). "Overcoming Exploration in Reinforcement Learning with Demonstrations". In: *CoRR* abs/1709.10089. arXiv: 1709.10089. URL: http://arxiv.org/abs/1709.10089.

Ng, Andrew Y. and Stuart J. Russell (2000). "Algorithms for Inverse Reinforcement Learning". In: *Proceedings of the Seventeenth International Conference on Machine Learning*. ICML '00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp. 663–670. ISBN: 1-55860-707-2. URL: http://dl.acm.org/citation.cfm?id=645529.657801.

"Autonomous Inverted Helicopter Flight via Reinforcement Learning" (2006). English. In: *Experimental Robotics IX*. Ed. by Andrew Y. Ng, Adam Coates, Mark

Diel, Varun Ganapathi, Jamie Schulte, Ben Tse, Eric Berger, and Eric Liang. Vol. 21, pp. 363–372. ISBN: 978-3-540-28816-9. DOI: 10.1007/11552246_35.

Park, Jung-Jun, Ji-Hun Kim, and Jae-Bok Song (2007). "Path planning for a robot manipulator based on probabilistic roadmap and reinforcement learning". In: *International Journal of Control, Automation, and Systems* 5.6, pp. 674–680.

Pascual, Santiago, Antonio Bonafonte, and Joan Serrà (2017). "SEGAN: Speech Enhancement Generative Adversarial Network". In: *CoRR* abs/1703.09452. arXiv: 1703.09452. URL: http://arxiv.org/abs/1703.09452.

Pehlivan, A. U., F. Sergi, and M. K. O'Malley (2015). "A Subject-Adaptive Controller for Wrist Robotic Rehabilitation". In: *IEEE/ASME Transactions on Mechatronics* 20.3, pp. 1338–1350. ISSN: 1083-4435. DOI: 10.1109/TMECH.2014.2340697.

Peng, Xue Bin, Glen Berseth, Kangkang Yin, and Michiel Van De Panne (2017). "DeepLoco: Dynamic Locomotion Skills Using Hierarchical Deep Reinforcement Learning". In: *ACM Trans. Graph.* 36.4, 41:1–41:13. ISSN: 0730-0301. DOI: 10.1145/3072959.3073602. URL: http://doi.acm.org/10.1145/3072959.3073602.

Pinciroli, Carlo et al. (2012). "ARGoS: a Modular, Parallel, Multi-Engine Simulator for Multi-Robot Systems". In: *Swarm Intelligence* 6.4, pp. 271–295.

Plappert, Matthias (2016). *keras-rl*. https://github.com/keras-rl/keras-rl.

Purnell, G. (2013). "13 - Robotics and automation in meat processing". In: *Robotics and Automation in the Food Industry*. Ed. by Darwin G. Caldwell. Woodhead Publishing Series in Food Science, Technology and Nutrition. Grimsby Institute of Further & Higher Education (Gifhe). Woodhead Publishing, pp. 304 –328. ISBN: 978-1-84569-801-0. DOI: https://doi.org/10.1533/9780857095763.2.304. URL: http://www.sciencedirect.com/science/article/pii/B978184569801050013X.

Radford, Alec, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever (2019). *Language Models are Unsupervised Multitask Learners*. Tech. rep. OpenAI, San Francisco, California, United States.

RobNor (2019a). *RobNor Homepage*. URL: https://www.robnor.se/.

— (2019b). *RobNor Projects*. URL: https://www.robnor.se/projekt.

RoboCluster Webpage (2019). *RoboCluster*. URL: https://www.robocluster.dk/.

RoboDK (2019). *RoboDK*. Website. URL: https://robodk.com/.

RoboSavvy (2019). *vive_ros*. Github. URL: https://github.com/robosavvy/vive_ros.

Russell, Stuart J (1998). "Learning agents for uncertain environments". In: *COLT*.

Rüßmann, Michael, Markus Lorenz, Philipp Gerbert, Manuela Waldner, Jan Justus, Pascal Engel, and Michael Harnisch (2015). "Industry 4.0: The future of productivity and growth in manufacturing industries". In: *Boston Consulting Group* 9.1, pp. 54–89. URL: http://www.inovasyon.org/pdf/bcg.perspectives_Industry.4.0_2015.pdf.

Schaul, Tom, John Quan, Ioannis Antonoglou, and David Silver (2016). "Prioritized Experience Replay". In: *CoRR* abs/1511.05952.

Schulman, John, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz (2015). "Trust Region Policy Optimization". In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by Francis Bach and David Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, pp. 1889–1897. URL: http://proceedings.mlr.press/v37/schulman15.html.

Schulman, John, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov (2017). "Proximal Policy Optimization Algorithms". In: *CoRR* abs/1707.06347.

Silver, David, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. (2016). "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587, p. 484.

Silver, David, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. (2017). "Mastering the game of go without human knowledge". In: *Nature* 550.7676, p. 354.

SPARC (2016). *Robotics 2020 Multi-Annual Roadmap*. Tech. rep. SPARC The Partnership for Robotics in Europe. URL: https://www.eu-robotics.net/cms/upload/topic_groups/H2020_Robotics_Multi-Annual_Roadmap_ICT-2017B.pdf.

Sutton, Richard S and Andrew G Barto (2018). *Reinforcement learning: An introduction*. MIT press.

Sutton, Richard S., David McAllester, Satinder Singh, and Yishay Mansour (1999). "Policy Gradient Methods for Reinforcement Learning with Function Approximation". In: *Proceedings of the 12th International Conference on Neural Information Processing Systems*. NIPS'99. Denver, CO: MIT Press, pp. 1057–1063. URL: http://dl.acm.org/citation.cfm?id=3009657.3009806.

A model of shared grasp affordances from demonstration (2007). English, pp. 27–35. ISBN: 978-1-4244-1861-9. DOI: 10.1109/ICHR.2007.4813845.

Takadama, K., K. Hajiri, T. Nomura, K. Shimohara, M. Okada, and S. Nakasuka (1998). "Learning model for adaptive behaviors as an organized group of swarm robots". In: *Artificial Life and Robotics* 2.3, pp. 123–128. ISSN: 1614-7456. DOI: 10.1007/BF02471168. URL: https://doi.org/10.1007/BF02471168.

Todorov, E., T. Erez, and Y. Tassa (2012). "MuJoCo: A physics engine for model-based control". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033. DOI: 10.1109/IROS.2012.6386109.

Tsai, Roger Y. and Reimer K. Lenz (1988). "A New Technique for Fully Autonomous and Efficient 3D Robotics Hand-eye Calibration". In: *Proceedings of the 4th International Symposium on Robotics Research*. Univ. of California, Santa Clara, California, USA: MIT Press, pp. 287–297. ISBN: 0-262-02272-9. URL: http://dl.acm.org/citation.cfm?id=57425.57456.

Uhlenbeck, G. E. and L. S. Ornstein (1930). "On the Theory of the Brownian Motion". In: *Phys. Rev.* 36 (5), pp. 823–841. DOI: 10.1103/PhysRev.36.823. URL: https://link.aps.org/doi/10.1103/PhysRev.36.823.

Vallés, Marina, José Cazalilla, Ángel Valera, Vicente Mata, Álvaro Page, and Miguel Díaz-Rodríguez (2017). "A 3-PRS parallel manipulator for ankle rehabilitation: towards a low-cost robotic rehabilitation". In: *Robotica* 35.10, 1939–1957. DOI: 10.1017/S0263574715000120.

Vinyals, Oriol et al. (2019). *AlphaStar: Mastering the Real-Time Strategy Game StarCraft II*. https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/.

Wang, Ziyu, Nando de Freitas, and Marc Lanctot (2015). "Dueling Network Architectures for Deep Reinforcement Learning". In: *CoRR* abs/1511.06581. arXiv: 1511.06581. URL: http://arxiv.org/abs/1511.06581.

Wang, Ziyu, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Rémi Munos, Koray Kavukcuoglu, and Nando de Freitas (2016). "Sample Efficient Actor-Critic with Experience Replay". In: *CoRR* abs/1611.01224. arXiv: 1611.01224. URL: http://arxiv.org/abs/1611.01224.

Wulfmeier, Markus, Peter Ondruska, and Ingmar Posner (2015). "Deep Inverse Reinforcement Learning". In: *CoRR* abs/1507.04888. arXiv: 1507.04888. URL: http://arxiv.org/abs/1507.04888.

Xanthopoulos, A. S., A. Kiatipis, D. E. Koulouriotis, and S. Stieger (2018). "Reinforcement Learning-Based and Parametric Production-Maintenance Control Policies for a Deteriorating Manufacturing System". In: *IEEE Access* 6, pp. 576–588. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2017.2771827.

Yang, Li-Chia, Szu-Yu Chou, and Yi-Hsuan Yang (2017). "MidiNet: A Convolutional Generative Adversarial Network for Symbolic-domain Music Generation using 1D and 2D Conditions". In: *CoRR* abs/1703.10847. arXiv: 1703.10847. URL: http://arxiv.org/abs/1703.10847.

Yip, Geeman (2018). *What, why, and when do we automate?* URL: https://www.infoworld.com/article/3251068/what-why-and-when-do-we-automate.html.

Zhang, Han, Tao Xu, Hongsheng Li, Shaoting Zhang, Xiaolei Huang, Xiaogang Wang, and Dimitris N. Metaxas (2016). "StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks". In: *CoRR* abs/1612.03242. arXiv: 1612.03242. URL: http://arxiv.org/abs/1612.03242.

Zhang, Jingwei and Lei Tai (2017). *jingweiz/pytorch-rl*. URL: https://github.com/jingweiz/pytorch-rl.

Ziebart, Brian D., Andrew Maas, J. Andrew Bagnell, and Anind K. Dey (2008). "Maximum Entropy Inverse Reinforcement Learning". In: *Proc. AAAI*, pp. 1433–1438.

Åström, Karl Johan (1980). "Why Use Adaptive Techniques for Steering Large Tankers?" eng. In: *International Journal Of Control* 32, pp. 689–708. ISSN: 0020-7179.

Appendix A

UML Diagram

A class UML diagram of the whole Python software can be found in the repository. Figure A.1 shows a simplified version of the reach environment.

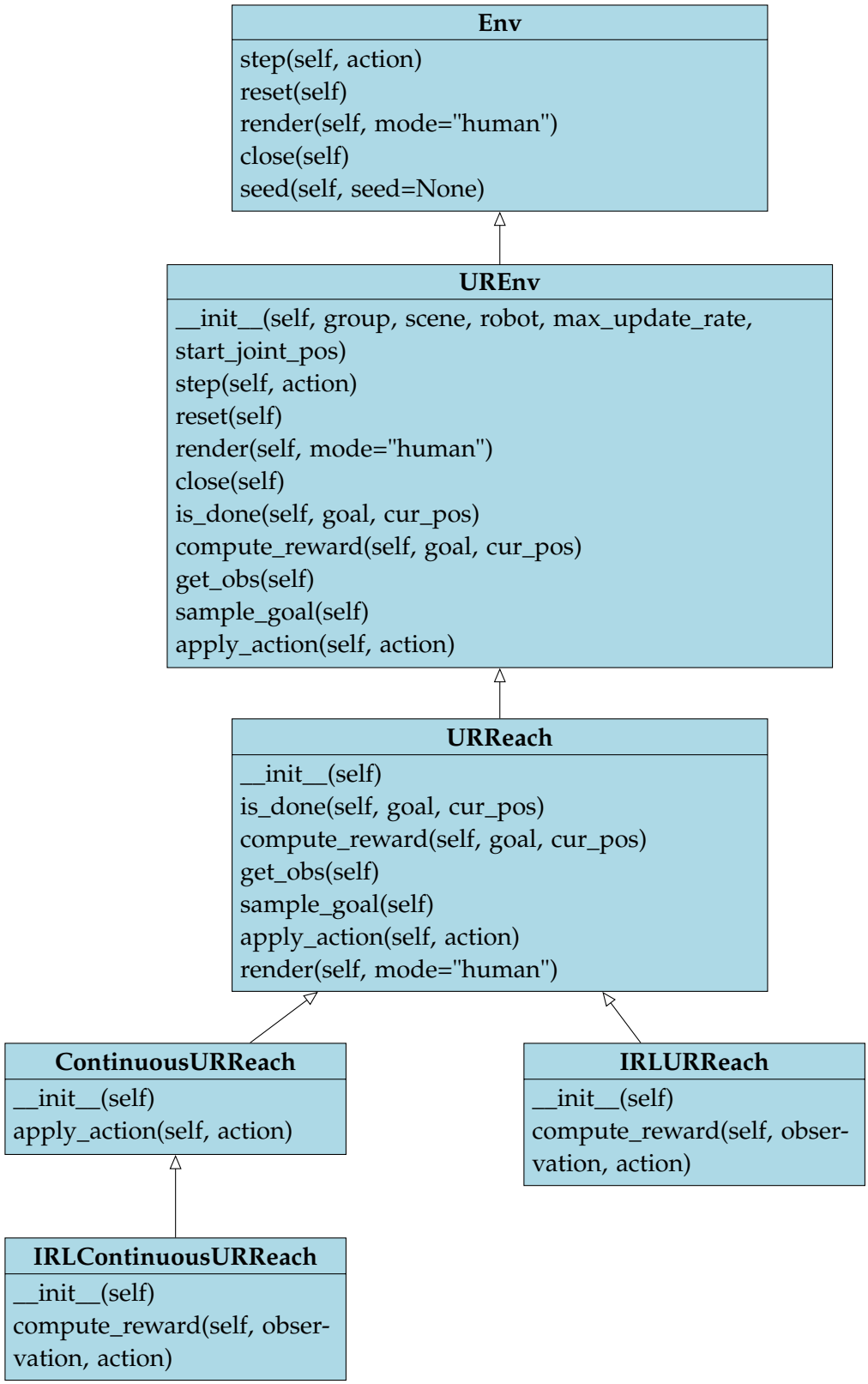


Figure A.1: A simplified class UML diagram of the custom built UR environment. Class attributes has been omitted in this diagram.

Appendix B

Linear Inverse Reinforcement Learning Weights

		Viapoints	
	Goal	1	2
Weights	-0.577	-0.577	-0.577

Table B.1: The weights for the minimum distance to 2 viapoints.

		Viapoints			
	Goal	1	2	3	4
Weight	-0.955	-0.160	-0.156	-0.140	-0.138

Table B.2: The weights for the minimum distance to 4 viapoints.

Goal weight	-0.9921									
Viapoints	1	2	3	4	5	6	7	8	9	10
Weights	-0.0119	-0.0119	-0.0117	-0.0118	-0.0132	-0.0127	-0.0118	-0.0130	-0.0123	-0.0129
Viapoints	11	12	13	14	15	16	17	18	19	20
Weights	-0.0133	-0.0126	-0.0122	-0.0118	-0.0118	-0.0117	-0.0119	-0.0126	-0.0128	-0.0127
Viapoints	21	22	23	24	25	26	27	28	29	30
Weights	-0.0117	-0.0129	-0.0124	-0.0131	-0.0121	-0.0119	-0.0117	-0.0131	-0.0133	-0.0132
Viapoints	31	32	33	34	35	36	37	38	39	40
Weights	-0.0135	-0.0127	-0.0117	-0.0125	-0.0127	-0.0119	-0.0117	-0.0129	-0.0129	-0.0131
Viapoints	41	42	43	44	45	46	47	48	49	50
Weights	-0.0119	-0.0127	-0.0122	-0.0117	-0.0117	-0.0119	-0.0130	-0.0118	-0.0124	-0.0117
Viapoints	51	52	53	54	55	56	57	58	59	60
Weights	-0.0123	-0.0117	-0.0117	-0.0132	-0.0131	-0.0128	-0.0129	-0.0130	-0.0119	-0.0131
Viapoints	61	62	63	64	65	66	67	68	69	70
Weights	-0.0119	-0.0129	-0.0133	-0.0128	-0.0128	-0.0126	-0.0118	-0.0129	-0.0124	-0.0128
Viapoints	71	72	73	74	75	76	77	78	79	80
Weights	-0.0129	-0.0117	-0.0135	-0.0120	-0.0127	-0.0128	-0.0135	-0.0129	-0.0122	-0.0136
Viapoints	81	82	83	84	85	86	87	88	89	90
Weights	-0.0123	-0.0131	-0.0127	-0.0128	-0.0130	-0.0123	-0.0128	-0.0119	-0.0129	-0.0117
Viapoints	91	92	93	94	95	96	97	98	99	100
Weights	-0.0132	-0.0130	-0.0130	-0.0136	-0.0133	-0.0136	-0.0119	-0.0117	-0.0128	-0.0117

Table B.3: The weights for the minimum distance to 100 viapoints.