Universidade de Aveiro
2021

Bruno Alexandre
Barbosa Pereira

Aprendizagem Profunda por Reforço para Tarefas de Manipulação Robótica

Deep Reinforcement Learning for Robotic Manipulation Tasks

Universidade de Aveiro
2021

Bruno Alexandre Barbosa Pereira

Aprendizagem Profunda por Reforço para Tarefas de Manipulação Robótica

Deep Reinforcement Learning for Robotic Manipulation Tasks

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Filipe Miguel Teixeira Pereira da Silva, Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

o júri / the jury

presidente / president	**Professor Doutor José Nuno Panelas Nunes Lau** Professor Associado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro
vogais / examiners committee	**Doutor Manuel Fernando Santos Silva** Professor Coordenador do Instituto Superior de Engenharia do Porto (arguente principal)
	Professor Doutor Filipe Miguel Teixeira Pereira da Silva Professor Auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro (orientador)

agradecimentos / acknowledgements

Gostaria de agradecer ao orientador Professor Filipe Silva pela oportunidade de trabalhar nesta dissertação, pelas suas sugestões e a sua constante disponibilidade em ajudar. Agradeço também ao IRISLab por me oferecer acesso aos seus recursos computacionais. Por fim, quero agradecer à minha família e amigos, em particular aos meus pais e irmão, por todo o apoio.

Palavras-Chave

Aprendizagem Profunda por Reforço, Controlo Contínuo, Ator-Crítico, Métodos de Gradiente de Política, Manipulação Robótica, Tarefas de Alcance

Resumo

Os avanços recentes na Inteligência Artificial (IA) demonstram um conjunto de novas oportunidades para a robótica. A Aprendizagem Profunda por Reforço (DRL) é uma subárea da IA que resulta da combinação de Aprendizagem Profunda (DL) com Aprendizagem por Reforço (RL). Esta subárea define algoritmos de aprendizagem automática que aprendem diretamente por experiência e oferece uma abordagem compreensiva para o estudo da interação entre aprendizagem, representação e a decisão. Estes algoritmos já têm sido utilizados com sucesso em diferentes domínios. Nomeadamente, destaca-se a aplicação de agentes de DRL que aprenderam a jogar vídeo jogos da consola Atari 2600 diretamente a partir de pixels e atingiram um desempenho comparável a humanos em 49 desses jogos. Mais recentemente, a DRL em conjunto com outras técnicas originou agentes capazes de jogar o jogo de tabuleiro Go a um nível profissional, algo que até ao momento era visto como um problema demasiado complexo para ser resolvido devido ao seu enorme espaço de procura. No âmbito da robótica, a DRL tem vindo a ser utilizada em problemas de planeamento, navegação, controlo ótimo e outros. Nestas aplicações, as excelentes capacidades de aproximação de funções e aprendizagem de representação das Redes Neuronais Profundas permitem à RL escalar a problemas com espaços de estado e ação multidimensionais. Adicionalmente, propriedades inerentes à DRL fazem a transferência de aprendizagem útil ao passar da simulação para o mundo real. Esta dissertação visa investigar a aplicabilidade e eficácia de técnicas de DRL para aprender políticas de sucesso no domínio das tarefas de manipulação robótica. Inicialmente, um conjunto de três problemas clássicos de RL foram resolvidos utilizando algoritmos de RL e DRL de forma a explorar a sua implementação prática e chegar a uma classe de algoritmos apropriada para estas tarefas de robótica. Posteriormente, foi definida uma tarefa em simulação onde um agente tem como objetivo controlar um manipulador com 6 graus de liberdade de forma a atingir um alvo com o seu terminal. Esta é utilizada para avaliar o efeito no desempenho de diferentes representações do estado, hiperparâmetros e algoritmos do estado da arte de DRL, o que resultou em agentes com taxas de sucesso elevadas. O foco é depois colocado na velocidade e restrições de tempo do posicionamento do terminal. Para este fim, diferentes sistemas de recompensa foram testados para que um agente possa aprender uma versão modificada da tarefa anterior para velocidades de juntas superiores. Neste cenário, foram verificadas várias melhorias em relação ao sistema de recompensa original. Finalmente, uma aplicação do melhor agente obtido nas experiências anteriores é demonstrada num cenário simplificado de captura de bola.

Keywords Deep Reinforcement Learning, Continuous Control, Actor-Critic, Policy Gradient Methods, Manipulation Robotics, Reaching tasks

Abstract The recent advances in Artificial Intelligence (AI) present new opportunities for robotics on many fronts. Deep Reinforcement Learning (DRL) is a sub-field of AI which results from the combination of Deep Learning (DL) and Reinforcement Learning (RL). It categorizes machine learning algorithms which learn directly from experience and offers a comprehensive framework for studying the interplay among learning, representation and decision-making. It has already been successfully used to solve tasks in many domains. Most notably, DRL agents learned to play Atari 2600 video games directly from pixels and achieved human comparable performance in 49 of those games. Additionally, recent efforts using DRL in conjunction with other techniques produced agents capable of playing the board game of Go at a professional level, which has long been viewed as an intractable problem due to its enormous search space. In the context of robotics, DRL is often applied to planning, navigation, optimal control and others. Here, the powerful function approximation and representation learning properties of Deep Neural Networks enable RL to scale up to problems with high-dimensional state and action spaces. Additionally, inherent properties of DRL make transfer learning useful when moving from simulation to the real world. This dissertation aims to investigate the applicability and effectiveness of DRL to learn successful policies on the domain of robot manipulator tasks. Initially, a set of three classic RL problems were solved using RL and DRL algorithms in order to explore their practical implementation and arrive at class of algorithms appropriate for these robotic tasks. Afterwards, a task in simulation is defined such that an agent is set to control a 6 DoF manipulator to reach a target with its end effector. This is used to evaluate the effects on performance of different state representations, hyperparameters and state-of-the-art DRL algorithms, resulting in agents with high success rates. The emphasis is then placed on the speed and time restrictions of the end effector's positioning. To this end, different reward systems were tested for an agent learning a modified version of the previous reaching task with faster joint speeds. In this setting, a number of improvements were verified in relation to the original reward system. Finally, an application of the best reaching agent obtained from the previous experiments is demonstrated on a simplified ball catching scenario.

Contents

Contents **i**

List of Figures **iii**

List of Tables **vi**

Acronyms **vii**

1 Introduction **1**
- 1.1 Motivation . . . 2
- 1.2 Objectives . . . 3
- 1.3 Outline . . . 3

2 Background and Context **5**
- 2.1 Reinforcement Learning . . . 5
  - 2.1.1 Framework and Concepts . . . 6
  - 2.1.2 Solution Methods . . . 8
- 2.2 Deep Reinforcement Learning . . . 12
  - 2.2.1 Deep Learning . . . 12
  - 2.2.2 Value Function Approximation . . . 14
  - 2.2.3 Policy Gradient Methods . . . 15
- 2.3 Related Work . . . 18

3 Materials and Methods **21**
- 3.1 Software Tools . . . 21
  - 3.1.1 OpenAI Gym . . . 21
  - 3.1.2 Robo-gym . . . 23
  - 3.1.3 PyTorch . . . 26
  - 3.1.4 Stable Baselines3 . . . 27
  - 3.1.5 Ray Tune . . . 28
- 3.2 DRL for Simple Problems . . . 28
  - 3.2.1 Gridworld . . . 29
  - 3.2.2 Solving Gridworld with Tabular Q-learning . . . 29
  - 3.2.3 Solving Gridworld with DQN . . . 32
  - 3.2.4 Solving the Pendulum Swing-up with DDPG . . . 35
- 3.3 DRL for Reaching Tasks . . . 37
  - 3.3.1 Reaching Tasks Specification . . . 37

3.3.2 Algorithms 42
3.3.3 Model Training and Evaluation 44
3.3.4 Hyperparameter Optimization 46
3.3.5 Performance Metrics 47

4 Experiments and Results 49
4.1 Solving the Reaching Task 50
4.1.1 Reaching Task with the Base State 50
4.1.2 Reaching Task with the End Effector State 52
4.1.3 Correlation of End Effector State with Reward 54
4.2 Hyperparameters 57
4.2.1 Hyperparameter Search 57
4.2.2 TD3: Replay Buffer 59
4.2.3 PPO: Horizon 60
4.3 Assessing Applicability to Robot Ball Catching 62
4.3.1 Action Cycle Rate 62
4.3.2 Different Reward Systems 63
4.3.3 Demo 69
4.4 Final Remarks 73

5 Conclusion and Future Work 75

References 77

List of Figures

2.1 Diagram of the interaction between the agent and environment, image from [1] 6
2.2 Diagram of an episodic task (top) ending in a terminal state (gray) and continuing task (bottom). 7
2.3 Heatmaps illustrating the effect of the ε parameter 9
2.4 Taxonomy of Deep RL algorithms, image from [2] 12
2.5 Diagram of an artificial neural network with a single hidden layer, image from [3] 13
2.6 The actor-critic architecture using the advantage estimate in the optimization of both the actor and critic networks. 17
2.7 Architecture of JAiLeR, image from [4] 19

3.1 Four default environments provided by OpenAI Gym. 22
3.2 Robo-gym architecture, image from [5] 25
3.3 Robo-gym interaction between agent and robot server, image from [5] 25
3.4 UR10 robot in Gazebo simulation 26
3.5 Computational graph of example expression 27
3.6 A Gridworld board 29
3.7 Learned policy visualized for a fixed goal 31
3.8 Steps as agent trains for one initial configuration 31
3.9 Vanilla DQN: static agent and goal 33
3.10 Vanilla DQN: random agent and goal 33
3.11 DQN with replay buffer and target network: random agent and goal. 34
3.12 Gym's Pendulum-v0 environment 35
3.13 Moving average of returns (window of 10 episodes) 37
3.14 Random targets generated in the *UR10Reach* environment (in RViz[1]) 38
3.15 Random targets generated in the UR10ReachEval environment (in RViz) . . 39
3.16 Random targets generated in the UR10BallCatch environment (in RViz) . . . 40
3.17 Setup of UR10BallCatch evaluation mode seen from above with the interception of random trajectories with the plane of the base link (in blue), end effector position (in red), initial ball position (in orange), and UR10's base link (in dark green). 41
3.18 Initial parameters defining the trajectory of a ball: the velocity V, elevation angle θ and azimuth angle ϕ. 42
3.19 PPO loss function L^{CLIP} as a function of probability ratio r_t for positive and negative advantages, image from [6]. 44

4.1 Average Return of evaluation episodes during training for TD3 and PPO. . . 51
4.2 Average Time Steps of evaluation episodes during training for TD3 and PPO. 51
4.3 Proportion of episode outcomes during training of TD3 (base state). 51
4.4 Proportion of episode outcomes during training of PPO (base state). 51
4.5 Average Return of evaluation episodes during training for TD3 and PPO. . . 52
4.6 Average Time Steps of evaluation episodes during training for TD3 and PPO. 52
4.7 Proportion of episode outcomes during training of TD3 (end effector state). . 53
4.8 Proportion of episode outcomes during training of PPO (end effector state). . 53
4.9 Mean Correlation between state vector and reward during an episode (considering last step) for multiple agents run on 300 random seed episodes. 55
4.10 Correlation Coefficient between episode rewards and state vector's r component during an episode (considering the last step) for 300 random seed episodes. . 55
4.11 Mean Correlation between state vector and reward during an episode (**ignoring** last step) for multiple agents run on 300 random seed episodes. 56
4.12 Correlation Coefficient between episode rewards and state vector's r component during an episode (**ignoring** the last step) for 300 random seed episodes. . . 56
4.13 Average Return of evaluation episodes during training of best configurations for TD3 and PPO. 58
4.14 Average Time Steps of evaluation episodes during training of best configurations for TD3 and PPO. 58
4.15 Average Return of evaluation episodes for agents trained with different replay buffer sizes. 60
4.16 Average Time Steps of evaluation episodes for agents trained with different replay buffer sizes. 60
4.17 Average Return of evaluation episodes for PPO agents trained with different values for the *horizon*. 61
4.18 Average Time Steps of evaluation episodes for PPO agents trained with different values for the *horizon*. 61
4.19 Success rates on *UR10Reach* with maximum joint velocities for TD3 model trained on 12.5 ACR and slow joint velocities run on different ACRs. 62
4.20 Average Return of evaluation episodes for agents trained at different ACRs. . 63
4.21 Average Time Steps of evaluation episodes for agents trained at different ACRs. 63
4.22 Success rates of TD3 agents trained with different ACRs 64
4.23 Average Return of evaluation episodes for TD3 agent trained with the normal reward. 65
4.24 Average Time Steps of evaluation episodes for TD3 agent trained with the normal reward. 65
4.25 Average Return of evaluation episodes for TD3 agent trained with the joint penalty reward. 65
4.26 Average Time Steps of evaluation episodes for TD3 agent trained with the joint penalty reward. 65
4.27 Log base reward for different coefficient values compared against linear base reward. 66
4.28 Average Return of evaluation episodes for TD3 agent trained with the log reward. 67
4.29 Average Time Steps of evaluation episodes for TD3 agent trained with the log reward. 67

4.30 Average Return of evaluation episodes for TD3 agent trained with the log without success threshold reward. 67
4.31 Average Time Steps of evaluation episodes for TD3 agent trained with the log without success threshold reward. 67
4.32 Average Return of evaluation episodes for TD3 agent trained with the joint penalty and log reward. 68
4.33 Average Time Steps of evaluation episodes for TD3 agent trained with the joint penalty and log reward. 68
4.34 Evaluation grid results 70
4.35 Intercept point of trajectory_6220 with catching plane 70
4.36 Target and End Effector trajectory on a successful attempt 71
4.37 Joint velocities on a successful attempt 72

List of Tables

3.1 UR10 specifications 26
3.2 Hyperparameters used for training the tabular Q-learning agent 31
3.3 Hyperparameters used for training the DQN agent 34
3.4 Hyperparameters used for training the DDPG agent 36
3.5 Hyperparameters used for training the TD3 agents 46
3.6 Hyperparameters used for training the PPO agents 46

4.1 Metrics of episodes with **successful outcomes** for the base state 51
4.2 Metrics of episodes with **failure outcomes** for the base state 52
4.3 Metrics of episodes with **successful outcomes** for the end effector state . . 53
4.4 Metrics of episodes with **failure outcomes** for the end effector state 54
4.5 Metrics of evaluation episodes for the base state 59
4.6 Metrics of evaluation episodes for TD3 agents trained with different replay buffer sizes. 60
4.7 Metrics of evaluation episodes for PPO agents trained with different values for the *horizon*. 61
4.8 Metrics of successful evaluation episodes with a **threshold distance of 0.1 meters** for agents trained with different reward systems. 68
4.9 Metrics of successful evaluation episodes with a **threshold distance of 0.05 meters** for agents trained with different reward systems. 69

Acronyms

A2C Advantage Actor-Critic. 16, 28, 43

A3C Asynchronou Advantage Actor-Critic. 16

ACR Action Cycle Rate. iv, 24, 39, 40, 62–64, 73, 75, 76

AI Artificial Intelligence. 5

ANN Artificial Neural Network. 13

CPU Central Processing Unit. 26, 50, 73

DAG Directed Acyclic Graph. 26

DDPG Deep Deterministic Policy Gradient. 17, 18, 28, 29, 35, 42, 43, 59, 75

DL Deep Learning. 12, 13

DNN Deep Neural Network. 12, 14, 15, 21, 34

DoF Degrees of Freedom. 18, 19, 25, 38

DP Dynamic Programming. 10

DPG Deterministic Policy Gradient. 17

DQN Deep Q-Network. 15, 17, 18, 28, 59

DRL Deep Reinforcement Learning. 12, 14, 18, 19, 21, 28, 37, 42, 43, 49, 73, 75, 76

GPU Graphics Processing Unit. 26, 50

HER Hindsight Experience Replay. 19, 28

MC Monte Carlo. 10

MDP Markov Decision Process. 6, 21, 29, 48, 73, 75

ML Machine Learning. 12, 46

MSE Mean Squared Error. 13, 15, 16

ODE Open Dynamics Engine. 45

PPO Proximal Policy Optimization. vi, 18, 19, 28, 42, 43, 57, 60, 61, 73, 75

RAM Random Access Memory. 50

ReLU Rectified Linear Unit. 13, 32, 35

RL Reinforcement Learning. 5–7, 9–12, 14, 18, 21, 23, 24, 27–29, 31, 34, 47, 48, 59, 76

RNG Random Number Generator. 44, 45

ROS Robot Operating System. 23, 24, 37, 38, 41, 42, 50, 69, 75, 76

SAC Soft Actor Critic. 28

SARSA State–Action–Reward–State–Action. 10, 11

SB3 Stable Baselines3. 28, 42, 44, 46, 58, 60

SGD Stochastic Gradient Descent. 14–16

TCP Tool Center Point. 25, 63

TD Temporal-Difference. 10, 11

TD3 Twin Delayed Deep Deterministic Policy Gradient. vi, 28, 42–44, 57, 59, 60, 62, 63, 65–67, 73, 75

TPE Tree-structured Parzen Estimator. 46, 57

TRPO Trust Region Policy Optimization. 43

Chapter 1

Introduction

The rise in adoption of automation technologies and their benefits are evident across industry and different aspects of life. Automation aims to scale down the need for human intervention in many activities and improve the standards of living for the general population. Back in the mid-18th century, the advent of the industrial revolution brought unprecedented economic, technological and population growth. This is justified partly by the use of machines alongside steam power engines to replace manual labor and the subsequent increase of productivity. Today's automation encompasses not only physical automation but increasingly cognitive automation.

Currently, automating physical tasks relies on the use of machines and particularly on the use of robots. In general, robots are programmable and autonomous machines which make use of sensor input to act in an environment. Robots are widely used in commercial and industrial contexts. For example, Amazon, the well known e-commmerce company, has made heavy use of mobile robots to carry loads across its warehouses to reduce picking time for orders. Automotive factories often use industrial manipulator robots to assemble, paint, weld and glue car parts on a production line. Manipulators or robotic arms are articulated robots which may resemble a human arm. They consist of multiple joints with links connecting them and the tool used to perform a task (like a gripper), referred to as the end effector. More recently, the concept of collaborative robots (cobots) has been growing in popularity. These are robots made to be easily programmed to do simple tasks (such as packaging or lifting weights) while being able to interact directly and safely with human workers in a shared workspace. This makes cobots easily integrated into an existing human pipeline to quickly automate a process.

For many decades now, computers have lead the automation of cognitive labor. However, they are limited to problems for which the solutions can be explicitly programmed by humans. Complex problems such as image classification, natural language processing, speech recognition, self-driving cars and others, have posed a challenge for hand coded solutions. New methods using Artificial Intelligence which leverage great volumes of data, have shown to be extremely effective, specifically Machine Learning algorithms which automatically improve through experience.

Thus, AI plays an important role in modern businesses. Namely, using AI for recommender systems in online platforms (e.g., YouTube, Netflix, Facebook) is now commonplace. For example, a recommender system for a news website tries to increase click through rate by learning a user's profile before using it to suggest the next news article to read. Additionally, in 2016, an AI system was successfully used to improve the energy efficiency of Google's data centers. By using data from thousands of sensors a deep neural network is able to minimize future energy consumption. This system managed to achieve a reduction of up to 40% in the energy used for cooling servers.

Deep Reinforcement Learning (DRL) is the class of Machine Learning algorithms mainly addressed in this dissertation. It is concerned with solving sequential decision making problems and is built from two components that have already independently had a profound impact in many fields: Deep Learning (DL) and Reinforcement Learning (RL). Typically, a DRL system combines a deep neural network to compute a non-linear mapping from perceptual inputs to action-values (or action probabilities) and RL signals that update the weights of the network in order to increase the frequency of highly rewarded actions. Concretely, this dissertation's focus is on the use of state-of-the-art Deep Reinforcement Learning algorithms to solve robot manipulator reaching tasks using the UR10 robot on a simulated environment.

1.1 Motivation

Recently, DRL has been successfully applied to solve problems in a variety of domains. The most notable have been problems involving games, since these provide a level of complexity interesting enough for research and can run faster than real time. In 2015, DeepMind (a London based AI company) proposed a novel artificial agent which achieved a level of performance comparable to humans in 49 Atari 2600 video games after learning by only having access to the pixels and the game score. For a case combining supervised learning from human expert games and DRL by playing against itself, in 2016, DeepMind's AlphaGo agent became extremely proficient at playing the ancient board game of Go, ultimately defeating the considered best player at the time in a match 4-1. Subsequently, in 2017, DeepMind developed an agent named AlphaGo Zero that trained only from playing against itself. It surpassed the previous AlphaGo's performance, arguably becoming the best Go player in the world.

Robotics has also been a domain of focus for the application of Reinforcement Learning and Deep Reinforcement Learning. A great number of problems in robotics are often formulated in ways closely resembling the Reinforcement Learning framework, making its application a natural pursuit. This is best illustrated by the extensive use of control theory in robotics. Here, similarly to RL, the objective is to find a policy (controller) which, using feedback, manipulates a dynamical system through control actions in order to minimize a cost function. However, while control theory relies on perfect knowledge of a system, RL differs in the fact that the dynamics defining a system's response to actions are usually unknown and must be learnt.

At present, the control of robotic manipulators is mostly achieved by solving inverse kinematic equations to position the end-effector with respect to a fixed reference frame. These robots perform repetitive tasks with speeds and accuracies far exceeding those of a human operator. However, they have difficulties in adapting to complex real-world environments subject to generalized and unsystematic variations. The robot is not expected to have, in advance, an accurate model of its environment, the objects it contains and the necessary skills to manipulate them.

In this context, decision-making and learning will be central abilities for such autonomous systems to handle the challenges of real-world scenarios. DRL appears as a promising approach to endow robots with these capabilities since no predefined training dataset is required, which may suit manipulation tasks. The expectation is that a wide range of robotic behaviours can be acquired by combining general-purpose function approximators, such as neural networks, with model-free reinforcement learning algorithms. However, applying DRL to real-world robotic control faces many challenges. For example, the volume of environment samples required is high. As a consequence, some previous works relies on parallelizing learning across multiple robots and the use of simulation before transferring the knowledge to the real world.

1.2 Objectives

This dissertation aims to investigate the applicability and effectiveness of Deep Reinforcement Learning to learn successful policies when applied to the domain of robot manipulator reaching tasks. The objectives defined for this dissertation are the following:

- From Reinforcement Learning to Deep Reinforcement Learning: to obtain a solid background on fundamental concepts of Deep Learning and Reinforcement Learning through practical implementation of DRL algorithms to solve simple problems.
- DRL for robot reaching tasks: to simulate, develop and implement DRL agents capable of controlling a manipulator in reaching tasks.
- Study the influence of different parameters on the learned policies: to choose simulation parameters, learning parameters and state representations with the goal of increasing the performance of the DRL agents.

1.3 Outline

This dissertation is organized into 5 chapters. Chapter 1 introduces the general context for the dissertation, presents the motivation, its objectives and outline. Chapter 2 provides the theoretical background on Reinforcement Learning and Deep Reinforcement Learning necessary to interpret this dissertation and also overviews previous related work. Chapter 3 presents: the main software tools used for developing and evaluating DRL agents; an early application of RL and DRL to simple problems used to become familiarized with different aspects of these algorithms; and the specification of the manipulator reaching tasks addressed in this dissertation along with the methods used to solve them and evaluate the results. Chapter 4 describes the procedures and results of a set of experiments conducted to solve the different versions of the reaching task. Finally, Chapter 5 provides a conclusion reviewing the results, the key ideas drawn from this dissertation and the potential paths for future work.

Chapter 2

Background and Context

In this chapter, the context for this dissertation is presented. The Reinforcement Learning (Section 2.1) and Deep Reinforcement Learning (Section 2.2) frameworks are defined and their main theoretical tools and limitations described. Finally, a description of previous related work is provided (Section 2.3).

2.1 Reinforcement Learning

Artificial Intelligence (AI) is a discipline of science characterized by its goal of achieving human level intelligent behaviour in machines. In the pursue of this goal, the field of AI attended to solve problems such as reasoning, learning and knowledge representation. From this emerged the Machine Learning field which is broadly defined as a set of techniques used to make computers learn to predict and make decisions from experience, in the form of data, without requiring to be explicitly programmed. Depending on the nature of this data, Machine Learning algorithms are commonly categorized into: supervised, unsupervised and reinforcement learning.

In supervised learning the data is labeled (by an external process or entity) with the correct prediction. It consists of solving either a regression problem, where the output is a single value or vector of real numbers; or a classification problem, when predicting one value out of a set discrete values which represent classes. In both cases, a supervised learning algorithm is tasked with finding a function which maps a feature vector into the desired output while optimizing a specific performance metric (often termed *Loss*).

For unsupervised learning the data has no labels and the goal is to learn any patterns and structure present in the data. Common instances of this are clustering algorithms such as *k-means*, which is an iterative process used to determine a possible set of k classes existing within a given dataset. Another example is *representation learning.* This involves automatically learning a representation from the feature vectors which is then is used in classification or regression problems.

Reinforcement learning (RL) [7] is a framework used to model and solve sequential decision-making tasks, while maximizing a reward signal. These tasks consist of deciding what to do given a situation in order to achieve a specific goal. Reward signals are delayed, often sparse and work only as evaluative feedback, thus differing from supervised learning where the correct answer for each example (an action in the case of RL) is explicitly provided. The challenge is being able to learn, through trial and error (experience), to correlate taking an action with the rewards it might produce in the future and act to maximize the total sum of rewards obtained. The remainder of this section overviews a description of this framework, its terminology and the main classes of RL algorithms.

2.1.1 Framework and Concepts

Markov Decision Process

In Reinforcement Learning a problem is formulated as Markov Decision Process (MDP). An MDP is defined by a state space S, an action space A, a state transition probability function P and a reward function R. For a problem where the probability of the next state depends only on the current state information and action, it is said to satisfy the *Markov property*. The actions are performed by an *agent* in an *environment* over discrete time steps. At each time step t an *agent* receives a state $s_t \in S$ from the *environment*, takes an action $a_t \in A$ according to some policy function $\pi(s_t)$ and finally transitions to a new state s_{t+1} and receives evaluative feedback from the *environment* in the form of a numerical reward r_{t+1}. This iterative process is depicted in Figure 2.1.

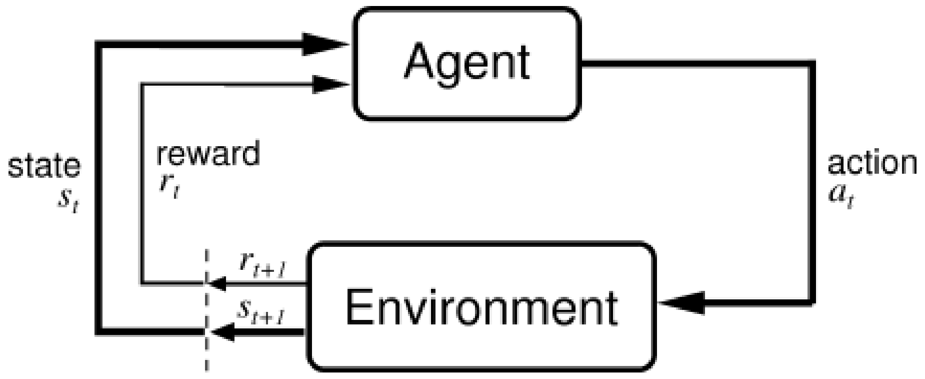


Figure 2.1: Diagram of the interaction between the agent and environment, image from [1]

Some tasks are considered to be *episodic* (see Figure 2.2) due to the sequence, or *trajectory*, of agent-environment interactions being structured as sub sequences which end within a finite number of time steps and reset to an initial state afterwards. In these cases, a state is considered to be a *terminal state* if it occurs at the final time step T of a sub sequence. These sub sequences are named *episodes* and can be represented as a set of ordered tuples containing the state, action and reward respective of each time step:

$$(s_0, a_0, r_1), (s_1, a_1, r_2), (s_2, a_2, r_3), ..., (s_{T-1}, a_{T-1}, r_T) \tag{2.1}$$

The rewards in Equation 2.1 follow a notation where the reward r at a given time step t is considered to be received on the next time step, hence r_{t+1}. In contrast, a task is defined to be a *continuing* task if it is not episodic, this is: a task which is not structured as sub sequences (episodes), it is instead a continuing sequence of agent-environment interactions (final time step $T = \infty$).

An environment's dynamics are characterized by the state transition probability function $P(s_{t+1}|s_t, a_t)$, which provides the probability of transitioning from a current state $s_t \in S$ to another in the next time step $s_{t+1} \in S$ when taking an action $a_t \in A$. In a *deterministic* environment the state-transition probability function defines a probability distribution over S where all the probability is accumulated in a single state: $P(s_{t+1}|s_t, a_t) = 1$. If this isn't the case, an environment is said to be a *stochastic* environment. That is, performing the same action a_t in state s_t can lead to different next states s_{t+1}.

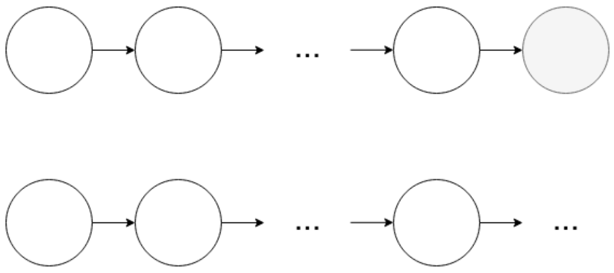


Figure 2.2: Diagram of an episodic task (top) ending in a terminal state (gray) and continuing task (bottom).

When learning from trial and error, a goal is defined for an RL agent by using a *reward* signal. A reward $r_t \in \mathbb{R}$ is a number computed by the environment, using the reward function $R(s, a)$, where $s \in S$ and $a \in A$, which evaluates the agent's previous action in a state. While trying to maximize this reward signal, an agent learns to achieve the desired goal. An agent's objective at any time step t is to choose an action that maximizes the reward, or the long-term cumulative reward, which results from the sequence of rewards received along an MDP after that time step t:

$$r_{t+1}, r_{t+2}, r_{t+3} ..., r_T \tag{2.2}$$

This cumulative reward is called *return* and is defined as a sum of the sequence of rewards. It can be paired with a parameter $\gamma \in [0, 1]$, named the *discount factor*, to define the *discounted return*:

$$\begin{aligned} G_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... + \gamma^{T-t-1} r_T \\ &= r_{t+1} + \gamma(r_{t+2} + \gamma^2 r_{t+3} + ... + \gamma^{T-t-1} r_T) \\ &= r_{t+1} + \gamma G_{t+1} \end{aligned} \tag{2.3}$$

The discount factor γ is used to parameterize how much an agent should discount future rewards. For values of γ closer to 1, the agent becomes more farsighted and gives more importance to rewards distant from the current time step t. On the other hand, for values close 0, the agent focuses more on the immediate rewards, where in the limit $\gamma = 0$ the discounted return is $G_t = r_{t+1}$.

Policy

How an agent behaves in an environment is specified by its *policy*. A policy, denoted π, maps states to a probability distribution over the set of possible actions (defined in 2.4). The probability of each action in this distribution is usually (for a good policy) related to the probability it will yield a reward given a state. Higher probability actions yield greater returns than lower probability actions.

$$\pi(s) \mapsto Pr(A|s), \forall s \in S \tag{2.4}$$

An agent can follow multiple policies, each yielding different long-term rewards. The *optimal policy*, denoted $\pi*$, is the policy which when followed produces maximum long-term rewards. Thus, the optimal policy can be defined as the policy which maximizes the expected return:

$$\pi* = \underset{\pi}{\operatorname{argmax}} \mathbb{E}[G_t|\pi] \tag{2.5}$$

Value Function

Most RL algorithms can be split into two categories: prediction and control. Prediction algorithms have the objective of estimating the value of an environment's feature. Typically, the predicted features are the reward or return. For control, an algorithm is tasked with improving an existing policy to find the optimal policy.

Prediction algorithms, also called *policy evaluation algorithms*, are used to assess a policy's performance. This involves determining a *value function* for a particular policy π. Value functions compute the expected return from state or state-action pair inputs. This is a measure of how good it is to be in given state or how good it is to choose an action while being in a given state. The *state-value function* $v_\pi(s)$ is a value function which outputs the expected return the agent will produce during its interaction with the environment when starting from state s and subsequently continuing to follow the policy π. This expectation is defined in Equation 2.6.

$$V_\pi(s) = \mathbb{E}_\pi[G_t|s], \forall s \in S \tag{2.6}$$

The *action-value function* $Q_\pi(s, a)$ outputs the expected return after taking an action a in state s and following policy π (see Equation 2.7). Action-value functions are useful in learning problems without a model of the environment because in these scenarios it is not possible to predict the next state and plan to choose the action which leads to highest return. Thus a policy can be indirectly learned without a model of the environment by only learning the Q values for all states.

$$Q_\pi(s, a) = \mathbb{E}_\pi[G_t|s, a], \forall s \in S, \forall a \in A \tag{2.7}$$

2.1.2 Solution Methods

Off-Policy vs On-Policy

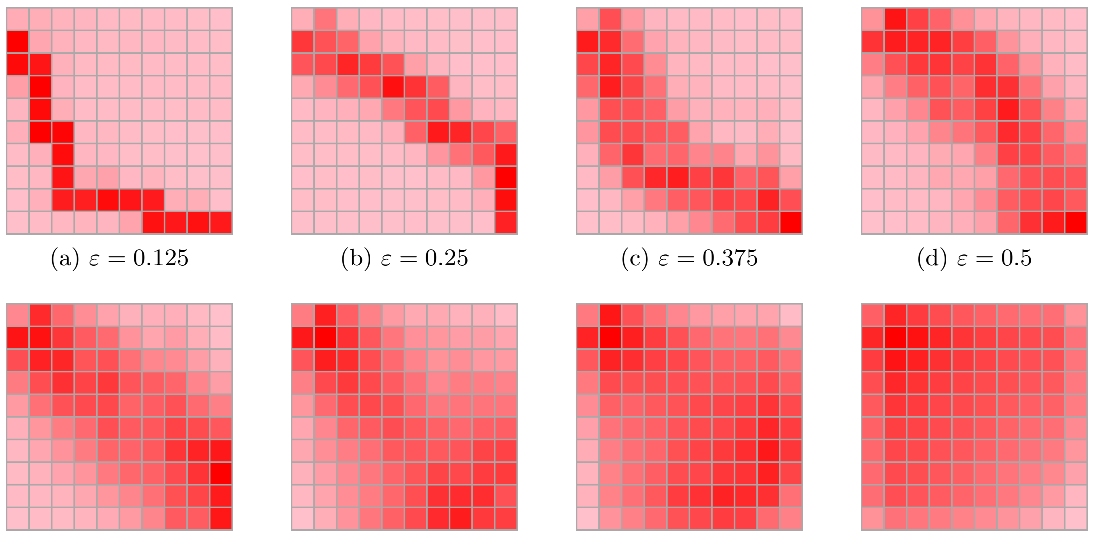
Prediction and control methods are also distinguished depending on the use of a policy to generate the experience data. If an algorithm learns a policy while simultaneously using it to collect the experience data, it is considered to be *on-policy.* On the other hand, an algorithm is said to be *off-policy* if it learns independently of the policy used to act.

Model-Based vs Model-Free

The state-transition probability function P together with the reward function R define a *model* of the environment. This allows an agent to predict the behavior of an environment and plan according to possible future events. RL algorithms which rely on environment models are called *model-based*, whereas algorithms which learn purely by experience are *model-free*.

Exploration vs Exploitation

The trade-off between *exploration* and *exploitation* is a central problem in RL: at any given moment an agent must decide between: attempting to leverage its current knowledge (exploitation) by greedily choosing actions to maximize short-term rewards; and randomly sampling the action space (exploration) in order to improve action-value estimates and consequently find alternative strategies which may lead to greater rewards in the long-term. Achieving the correct balance between the two is crucial when attempting to estimate the optimal policy. The ε-*greedy* (Epsilon-greedy) strategy is one common method employed to balance exploration and exploitation. It consists of choosing a random action from the action space with a probability ε and with a probability $(1-\varepsilon)$ choosing the action which maximizes returns. The resultant behaviour of using ε-greedy strategy is controlled by the ε parameter, an agent explores more often the closer ε is to 1 and explores less (exploits) if it is closer 0. This effect is illustrated in Figure 2.3 where 8 different values of ε are tested for an agent that was set to learn a single goal in a Gridworld environment. Gridworld is an RL problem consisting of a 2D board where an agent navigates to reach a goal position. In this case, an agent learns to move towards (9,9) (bottom right) starting from a fixed position (0,0) (top left) in a 10x10 board with no obstacles. The figure displays the cells that are visited more often in darker color as the probability of the agent acting randomly, controlled by ε, increases.



(a) $\varepsilon = 0.125$ (b) $\varepsilon = 0.25$ (c) $\varepsilon = 0.375$ (d) $\varepsilon = 0.5$

(e) $\varepsilon = 0.625$ (f) $\varepsilon = 0.75$ (g) $\varepsilon = 0.875$ (h) $\varepsilon = 1.0$

Figure 2.3: Heatmaps illustrating the effect of the ε parameter

Dynamic Programming

Dynamic programming (DP) is an optimization method used to solve MDPs given a perfect model of the environment. This is done by iterative algorithms which use the *Bellman* equation to update a value function estimate for a policy π. Despite most RL problems not having a perfect model of the environment, the concepts covered in this method are present in other RL algorithms. In the case of *finite* MDPs (finite state and action spaces), an environment's model is defined by $p(s', r|s, a)\ \forall s, s' \in S, \forall a \in A$ and $\forall r \in R$. This p is the probability of s' and r occurring at a given time step t given the previous state s and action a. With the definition of V_π and p, the following equality named the Bellman equation , is derived for the state-value function:

$$V_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma V_\pi(s')] \tag{2.8}$$

Using an iterative *policy evaluation* algorithm, the value-state function V_π for a policy π can be estimated. This policy evaluation in conjunction with a *policy improvement* step allows for estimating the optimal policy π^* in an algorithm named *Policy Iteration.* The policy improvement consists in updating the policy with a greedy strategy where for each state $s \in S$ $\pi(s) = a$ where a is the action that maximizes $V_\pi(s')$ and $s' \in S$ is the next state.

Monte Carlo Methods

Monte Carlo (MC) methods involve averaging over many samples of actual returns to estimate state-value and action-value functions. Similarly to DP, the prediction results are used to then solve the control problem of approximating optimal policies. These are methods that fully use experience without any model of the world. The samples of returns are obtained from experience rollouts in episodic tasks, hence the estimates are updated episode-by-episode rather than step-by-step.

Temporal-Difference

The two methods previously described present some limitations for RL problems: DP requires a model of the environment (the state-transition probability function and reward function) to compute the expectation of $V(s)$ (model-based) and hence its direct applications are narrow; MC methods although improving over DP by being model-free and relying on experience samples, requires an episode rollout to complete before being able to learn and compute its estimates. Temporal-Difference (TD) learning methods combine aspect of both DP and MC. Like MC, it learns from experience samples and, similarly to DP, also *bootstraps* by iteratively updating an estimate based on a previously learned estimate. The difference in experience sampling between TD and MC is that the former only waits until the next time step to receive a reward and update its estimate instead of waiting for an episode to end to obtain the return G_t. This results in TD being less computationally intensive during training, since updates are done iteratively along time rather than all at the end.

State–Action–Reward–State–Action (SARSA) is an on-policy RL algorithm, where a Q action-value function is learned by minimizing a *TD error* for observed state-action pairs using the Bellman equation iteratively. This error is calculated using the action derived from the current policy for the next state (see Equation 2.9). The complete procedure for learning with SARSA is listed in Algorithm 1.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{2.9}$$

This algorithm results in a Q function accurately describing the state-action value of the policy used during learning.

Algorithm 1 SARSA, adapted from Sutton and Barto (2018)

```
1:  Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
2:  Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, .) = 0$
3:  for each episode do
4:      Initialize $S_t$
5:      Choose $A_t$ from $S_t$ using policy derived from $Q$          ▷ e.g., $\varepsilon - greedy$
6:      for each step of episode do
7:          Take action $A_t$, observe $R_{t+1}, S_{t+1}$
8:          Choose $A_{t+1}$ from $S_{t+1}$ using policy derived from $Q$          ▷ e.g., $\varepsilon - greedy$
9:          $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$
10:         $S_t \leftarrow S_{t+1}$
11:         $A_t \leftarrow A_{t+1}$
12:     until $S_t$ is terminal
```

Q-learning is an instance of a TD method used to learn the optimal action-value function Q_*. It is a model-free RL algorithm that differs from SARSA for being off-policy. Instead of using the action derived from the policy to calculate the *TD error* it uses the action which maximizes the Q value for the next state (see Equation 2.10). The algorithm for Q-learning is listed in Algorithm 2.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.10}$$

Algorithm 2 Q-learning, adapted from Sutton and Barto (2018)

```
1:  Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
2:  Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, .) = 0$
3:  for each episode do
4:      Initialize $S_t$
5:      for each step of episode do
6:          Choose $A_t$ from $S_t$ using policy derived from $Q$          ▷ e.g., $\varepsilon - greedy$
7:          Take action $A_t$, observe $R_{t+1}, S_{t+1}$
8:          $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$
9:          $S_t \leftarrow S_{t+1}$
10:     until $S_t$ is terminal
```

2.2 Deep Reinforcement Learning

Deep Reinforcement Learning [8][9] is the combination of Deep Learning with Reinforcement Learning. In this context, Deep Neural Networks are used to approximate the state value function or state-action value function, the policy itself or the model of the environment. Deep Learning brings two main advantages to RL: generalization and representation learning. Generalization is the ability of a model to correctly predict the values of data inputs different then those used in training. This allows to solve the limitations of tabular RL algorithms in complex problems where tables would have to be exceedingly large, requiring more memory to store them and more visits per state to obtain good estimates. Representation learning consists in a model learning from experience a transformation of the input which makes it easier to solve a classification or prediction problem. In other words, its automatic feature engineering as a result of training. It allows for the direct application of RL to more interesting problems with complex inputs, such as pixels from a camera, without requiring hand-engineered features, which leads to relying less on specific domain knowledge.

This section provides a theoretical overview of Deep Learning (Section 2.2.1) and the primary classes of solutions methods for DRL: Value Function Approximation (Section 2.2.2) and Policy Gradient Methods (Section 2.2.3). These will focus on model-free methods and the algorithms presented are discerned based on the functions approximated using DNNs. An incomplete yet useful classification of DRL algorithms, some of which are addressed in this section, is summarized in Figure 2.4.

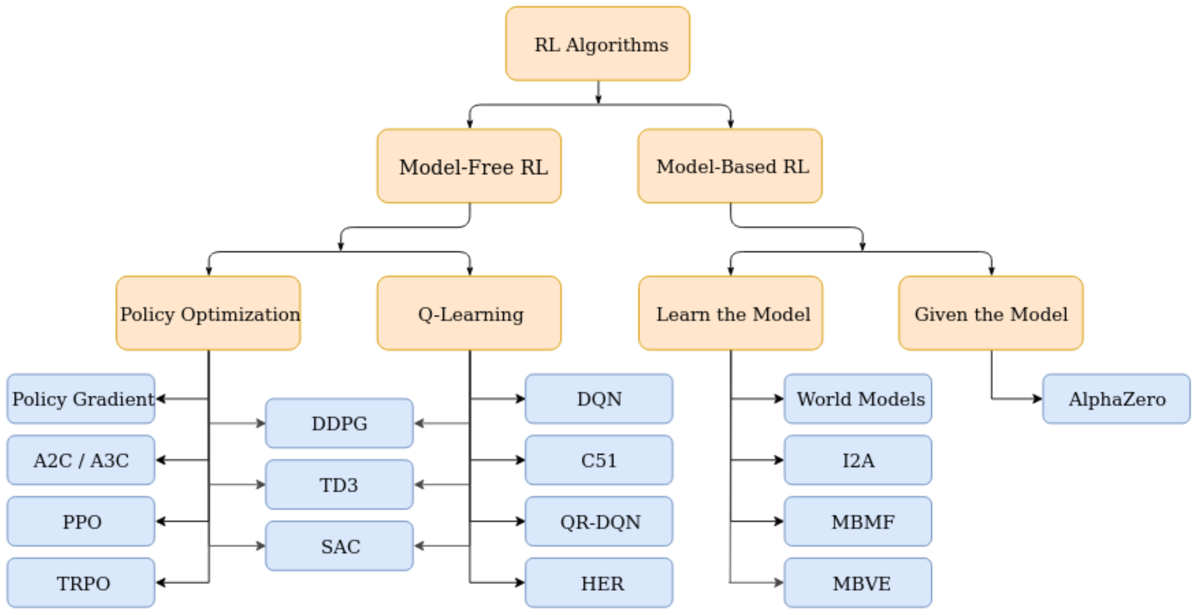


Figure 2.4: Taxonomy of Deep RL algorithms, image from [2]

2.2.1 Deep Learning

Deep Learning (DL) [10] is a class of ML techniques used for prediction and classification tasks which, unlike ”shallow” learning algorithms (e.g., SVM, Logistic Regression), have more than two hidden layers between the input and output layers. These hidden layers learn a hierarchy of representations where complex concepts are made out of simpler ones allowing high dimensional inputs, which may not have a linear relation with the output, to be used where classical ML algorithms would require hand-engineered features. The most common type of DL model is an Artificial Neural Network with multiple hidden layers, termed *Deep Neural Network*.

The *Artificial Neural Network* ANN is a biologically inspired Machine Learning model consisting of a number of neurons with weighted connections between them arranged in a network topology (Figure 2.5). This network acts as a function approximator $f(x;\theta)$ which performs a computation on inputs x to return outputs parameterized by the weights θ. The main goal is to optimize the parameters θ to obtain the best approximation of a given function. These models are also often referred to as a *feedforward neural network* for the simplest architecture where a computation begins at the inputs and flows to the outputs. This procedure is named a *forward pass*.

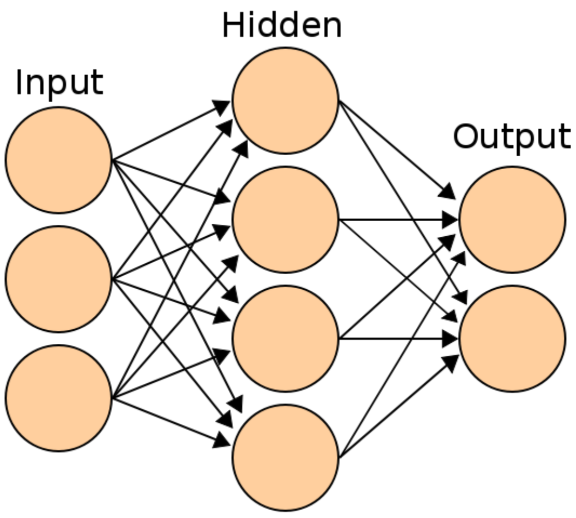


Figure 2.5: Diagram of an artificial neural network with a single hidden layer, image from [3]

The neuron is the basic computation unit composing an ANN. It is a function of a vector inputs $x \in \mathbb{R}^n$ and single output $y \in \mathbb{R}$, such as a logistic regression unit. In this case output y is a linear function of the input defined as:

$$y = \theta^T x \tag{2.11}$$

where $\theta \in \mathbb{R}^n$ is a vector of parameters with the same size as the input. Thus, an ANN can be interpreted as a combination of smaller functions for which the output of some neurons is the input to others. To allow for more complex non linear functions to be learned, an additional operation is applied on the output of a neuron $h = g(y)$, where g is an activation function such as the *rectified linear unit* (ReLU). The loss function representing the accuracy of the neural network in a task like regression is usually the Mean Squared Error (MSE) function. Given a dataset of N inputs x and corresponding outputs y to learn from, the MSE loss function is computed with the following definition:

$$J(\theta) = \frac{1}{N}\sum_{x\in\mathbb{X}}(y - f(x;\theta))^2 \tag{2.12}$$

To decrease the loss, which is equivalent to increasing performance of the model, three main methods exist: *batch gradient descent*, using the entire dataset to compute the loss; *minibatch gradient descent*, computing the loss before each update using a batch smaller than the dataset; and *stochastic gradient descent* (SGD), computing a loss and performing an update for each element of a dataset. These consists of computing the gradient of the loss function with respect to the trainable parameters θ and updating these same parameters in the negative direction of the gradient $\nabla_\theta J(\theta)$.

To compute these gradients a method termed *backpropagation* is used. This method computes the gradients using the chain rule, where the error observed in the loss function is propagated in a reverse order to that of a forward pass.

2.2.2 Value Function Approximation

Value Function Approximation is a class of DRL methods which involve using a DNN to approximate either a state-action value function $Q(s,a)$ or state-value function $V(s)$. The optimization of these DNNs is made by defining a loss using the Bellman update equation on a batch of environment transitions.

Deep Q-learning is the archetypal instance of a value function approximation algorithm. It is an algorithm proposed by DeepMind in 2013 [11] where a Q-table is approximated by a Deep Neural Network, termed Deep Q-network (DQN). This intends to leverage the feature learning strengths of DNNs to apply Q-learning to RL tasks with high-dimensional state spaces. Later in 2015, DeepMind presented the results [12] of applying a DQN agent to learn how to play 49 different Atari 2600 games directly from image pixels, where the agent performed comparably to humans.

The DQN architecture implemented by DeepMind is composed of 3 convolutional layers with the input being a $84 \times 84 \times 4$ tensor (the 4 most recent 84×84 sized frames of a game) followed by 1 fully connected layer and finally an output layer which the size (number of actions) varies according to the game. This network architecture differs from the Q-function definition $Q(s,a) \to \mathbb{R}$ where both the state s and action a are the input. Instead, in order to avoid running a forward pass on the network for every action at each time step, DeepMind's solution redefines the Q-functions as $Q(s) \to \mathbb{R}^k$, where $k = |A|$ is the number of actions. This is more efficient by computing the Q-values for all actions simultaneously with a single forward pass. The optimization of the network is done using minibatch gradient descent to minimize a mean-squared error loss function, defined in (2.13), where the target $y = r + \gamma \max_{a'} Q(s', a'; \theta^-)$ is computed using the reward received after a step, similarly to the Bellman update in (2.10).

$$L(\theta) = \mathbb{E}_{(s,a,r,s')\ U(D)}[(y - Q(s,a;\theta)^2] \tag{2.13}$$

The loss function L is computed using mini batches of training data uniformly sampled from a fixed size queue D named *replay buffer* which stores experience tuples (s,a,r,s') of the most recent agent-environment interactions and allows the agent to learn on past experiences. This is a technique named *experience replay* and is the solution found to solve the *catastrophic forgetting* phenomenon, where a neural network learning from new data with minibatch gradient descent (or SGD) corrupts old learned parameters, resulting in the network becoming stuck in poor local minima or diverging. Additionally, DeepMind also observed instabilities during training due to correlations between the Q-value and the target y. This network is called the target because when minimizing the MSE loss in Equation 2.13 the Q approaches the y. Problematically, the target y depends on the same parameters θ being trained (using Q to calculate it), which makes the optimization of this loss function unstable. This was solved in Deep Q-learning by computing y using a second *target network* parameterized by θ^-, which is a copy of network Q periodically updated at a fixed number of iterations.

The Q-learning algorithm is known to overestimate action values due to the max operation performed to compute the target y, which may negatively affect its performance relating to the convergence and the resulting policies. This same problem was also observed in the DQN algorithm. A solution, originally proposed in a tabular setting, is shown to work with arbitrary function approximation and applied to DQN in [13]. The authors termed this new algorithm *Double DQN*. In the standard DQN, the max operator uses the same value to select and evaluate an action. Instead, in Double DQN two Q-value functions are used to decouple the selection from the evaluation when computing the target y. The second value function is often the already available target network. This works by using one network to greedily select the action and another to determine its value, as expressed in Equation 2.14.

$$y = r + \gamma Q(s', \operatorname*{argmax}_{a'} Q(s', a'; \theta); \theta^-) \tag{2.14}$$

2.2.3 Policy Gradient Methods

In value function approximation methods the learned function is used to derive the policy, for example, by greedily choosing the action with the highest Q value. In Policy Gradient Methods, the policy π is explicitly represented as a DNN. Through various techniques, this policy network is then optimized, using gradient descent, to increase the probability of highly rewarded actions. This has an interesting property: it is applicable to tasks with continuous action spaces by outputting either a single value deterministically or the mean and variance of a continuous probability distribution. This solves the problem of having to learn the Q values for an infinite number of actions. The latter case, taking actions according to a probability distribution, can also be beneficial in problems where a complete state representation is not available and the optimal policy is stochastic.

REINFORCE

The REINFORCE algorithm is often considered the simplest implementation of a policy gradient method. It is a model-free, on-policy algorithm and since all updates are done with respect to a previous episode rollout, it is also classified as a Monte Carlo algorithm. It consists of using a model, such as DNN, to represent the policy π_θ which is differentiable with respect to its parameters θ. For this policy the performance objective function is defined as:

$$J(\theta) = v_{\pi_\theta}(s) \tag{2.15}$$

where v_{π_θ} is the true value function of π_θ. The objective is to maximize the performance of the policy, which means increasing the expected return v_{π_θ} for a given state. To this end, the parameters are updated through gradient *ascent* using the following gradient vector:

$$\nabla J(\theta) = \mathbb{E}_\pi[G_t \nabla ln(\pi(A_t|S_t, \theta))] \tag{2.16}$$

The expression in brackets is a quantity that can be sampled at each time step whose the expectation is equal to the gradient $\nabla J(\theta)$. After an episode rollout obtained following π, for each time step the returns G_t are calculated using a given discount factor γ to be used in the following update rule:

$$\theta \leftarrow \theta + \alpha G_t \nabla ln(\pi(A_t|S_t, \theta)) \tag{2.17}$$

These updates are done with SGD (or minibatch gradient descent for one or multiple rollouts) using a gradient vector that points to a direction (in parameter space) that increases the probability of taking an action A_t when in a given state S_t. This gradient vector has a magnitude proportional to the return G_t observed, hence reinforcing actions that experienced positive subsequent rewards and penalizing actions that lead to negative rewards (by gradient *descent*).

Advantage Actor-Critic

A generalization of the update rule in Equation 2.17 is to replace the term G_t by a comparison to an arbitrary *baseline* $b(S_t)$. This takes the form expressed in Equation 2.18.

$$\theta \leftarrow \theta + \alpha(G_t - b(S_t)) \nabla ln(\pi(A_t|S_t, \theta)) \tag{2.18}$$

Baselines can have a great effect on reducing the variance of the update value and result in better stability and speed ups in learning. A baseline should be a function of the state in order to discern high value actions in states where all actions are high value and the same for low value states. A commonly used baseline is a learned estimate of the value function $b(S_t) \approx V^\pi(S_t)$. The quantity $G_t - b(S_t)$ is then an estimate of the *advantage* function $A(A_t, S_t) = Q(A_t, S_t) - V(S_t)$ where G_t can be seen as an estimate of the Q^π and b and estimate of V^π. The advantage expresses how much better or worse an action was compared to the expected value V for a particular state.

The Advantage Actor-Critic (A2C) algorithm is a synchronous, deterministic implementation of the earlier Asynchronous Advantage Actor-Critic (A3C) [14]. These algorithms are on-policy and use the *Actor-Critic* method (summarized in Figure 2.6) where a value function approximator parameterized by θ^V (*critic*) is trained alongside the policy parameterized by θ^μ (*actor*), often sharing some network parameters. The critic, as the name suggests, criticizes the policy's selection of actions by means of learning the state value function which is then used to compute the advantage. The resulting parameter update is expressed as:

$$\theta^\mu \leftarrow \theta^\mu + \alpha A(A_t, S_t, \theta^V) \nabla ln(\pi(A_t|S_t, \theta^\mu)) \tag{2.19}$$

On the other hand, the critic is optimized using a MSE loss function with the returns G_t as a target. A2C performs synchronous updates by waiting for all the parallel agents to run a predefined number of environment time steps before using the collected experience to compute the gradient, unlike A3C where agents asynchronously update the shared parameters. The latter was thought to have a regularization or exploratory effect but A2C has shown to be better in many aspects.

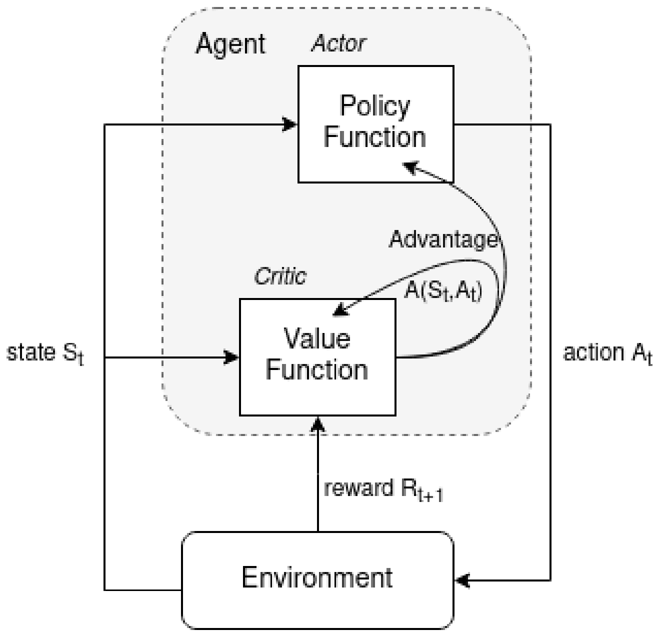


Figure 2.6: The actor-critic architecture using the advantage estimate in the optimization of both the actor and critic networks.

Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient (DDPG) [15] is a model-free, off-policy Actor-critic algorithm which simultaneously learns a Q-function and a policy function using Deep Neural Networks. It is suited for tasks with continuous actions spaces and high dimensional states. The *Actor-critic* method used in DDPG is composed of: a policy network known as *actor*; and a Q-function network known as *critic*. The actor μ is a deterministic policy $\pi(s) \mapsto A$ which outputs a single action value given a state (as opposed to the stochastic policy which returns a probability distribution over the actions $\pi(s) \mapsto Pr(A|s)$). The critic Q evaluates the actor's actions and learns to estimate the respective policy's Q-values. DDPG is based on the Deterministic Policy Gradient (DPG) presented in [16]. In this work, the authors derived the policy gradient with respect to the actor's parameters θ^μ:

$$\begin{aligned}\nabla_{\theta^\mu} J &\approx \mathbb{E}_{s_t \sim \rho^\beta}[\nabla_{\theta^\mu} Q(s,a|\theta^Q)|_{s=s_t, a=\mu_{(s_t,\theta^\mu)}}] \\ &= \mathbb{E}_{s_t \sim \rho^\beta}[\nabla_a Q(s,a|\theta^Q) \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s=s_t, a=\mu_{(s_t)}}]\end{aligned} \tag{2.20}$$

where the critic function is parameterized by θ^Q and the actor function by θ^μ. By adjusting the actor's parameters following this gradient one will increase the critics Q-value. Similarly to DQN, the networks in DDPG make use of an experience replay buffer and target networks (one for each: policy and Q) to address the issues of convergence and instability during training. The critic network Q is trained to approximate $max_a Q(s,a) \approx Q(s, \mu(s))$ using the policy μ and the reward together with the Bellman equation to define targets for optimization, as in DQN. The deterministic policy network μ is trained with the goal of choosing the action which maximizes the Q network output. To this end, the policy network is trained by performing gradient ascent using the gradient of the following loss function with respect to θ^μ:

$$J = \frac{1}{|B|} \sum_{s \in B} Q(s, \mu(s|\theta^\mu)|\theta^Q) \tag{2.21}$$

which is Equation (2.20) where the expectation is approximated by computing the mean across a sampled minibatch B. Instead of the target networks' parameters θ' being copied at fixed rate, like DQN, they are updated continuously after processing each minibatch by *polyak averaging*:

$$\theta' \leftarrow \tau\theta + (1 - \tau)\theta', \;\; \texttt{for } \tau \ll 1 \tag{2.22}$$

To address the problem of exploration during training which arises from using a deterministic policy, noise is added to the actions. In the DDPG paper this noise is sampled from a Ornstein-Uhlenbeck process, but other options such as zero mean Gaussian noise have been shown to work.

2.3 Related Work

Continuous control tasks are regularly used to benchmark new DRL algorithms due to their complexity and often high state and action space dimensions [6][17][15]. Some commonly used benchmarks are the OpenAI Gym's [18] 2D and 3D robots (simulated using the MuJoCo physics engine [19]) and RLLab [20], a framework that includes a wide variety of continuous control tasks. The tasks in these benchmarks range from bipedal locomotion of humanoids-like agents to in-hand object manipulation [21].

Given the application of DRL to continuous control, robotic manipulation presents many opportunities for these algorithms to be employed in the real world. An overview of DRL for robotic manipulation can be found in [22] and [23]. These robotic manipulation tasks have been the focus of previous work [24][25][26][27] and are distinguished based on: the sensory input, which can be proprioceptive and contain visual information; the robot and the number of controllable degrees of freedom (DoF); and whether it is done in simulation or with a real robot (or both).

Lopez et al. [28] proposed a new version of an RL toolkit named gym-gazebo2, that is based on ROS2 and Gazebo. The first version had already been successfully used by multiple research laboratories and with this work the authors intended to improve it. Gym-gazebo2 consists of a real or simulated robot which is controlled by an agent via an OpenAI Gym environment layer with ROS2 as a middleware for communication. To showcase its application, the Proximal Policy Optimization (PPO) algorithm was applied to 4 different versions of a reaching task for a single goal position, using the 6 DoF Modular Articulated Robotic Arm (MARA) from Acutronic Robotics, to replace traditional path planning techniques. The 4 versions are differentiated by what the reward is based on: the distance to the target of the end effector; the distance and orientation of the end effector; distance of the end effector and collision; distance and orientation of the end effector and collision. In this work, only the position of the end effector is specified to be the state used by the agent and the actions are joint positions between $[-\pi, \pi]$. In all versions of the task the agent successfully learned a reaching motion consistently achieving average errors below 10 mm. However, in this work the goal positions are limited to the one the agent was trained on.

A more generic application of DRL to reaching tasks is presented in [4]. Here, the authors apply DRL in order to learn a joint space controller to map a target position to joint angles. The proposed *Joint Action-space Learned Reacher* (JAiLeR) approach consists of a policy network trained using PPO which uses a state containing the joint positions q, velocities $\dot{q}$ and the Cartesian space error δx between the end effector and the target position. The policy then outputs actions in joint space in the form of joint velocities $\dot{q}_d$ which are subsequently integrated to joint positions q_d and sent to a proportional-derivative (PD) controller mapping it to joint torques τ. An overview of this approach is depicted in Figure 2.7.

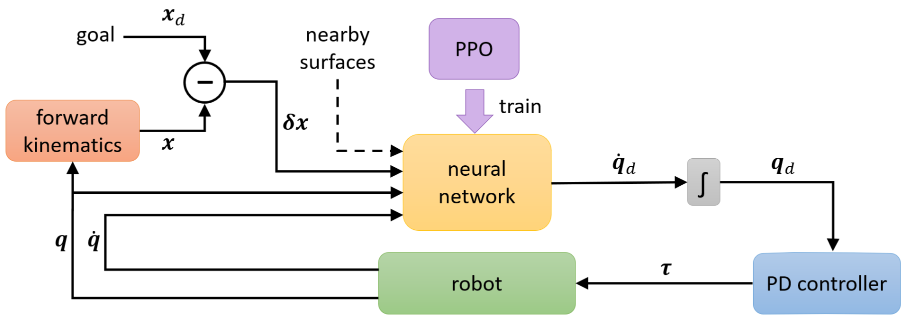


Figure 2.7: Architecture of JAiLeR, image from [4]

Additionally, Kumar et al extend the reaching task to have an obstacle avoidance element. To this end, the state representation is enhanced to also have n vectors, one for each of the n links, representing the vector made of the two closest points between the surface of a link and any object in the world. The reward is shaped to reduce the end effector to goal error, penalize both high effort solutions and distances between the links and objects bellow a defined threshold of 5 cm. A reaching policy was then trained for the Kinova Jaco 6 DoF manipulator with 40 parallel actors and afterwards tested on randomly sampled goals, achieving distances to the goal of less than 1.0 cm for 96.5% of evaluations and an average error of 0.4 cm, which is comparable to other velocity-based controllers.

In [29] Aumjaud et al. defined an experimental procedure to benchmark model-free DRL algorithms and applied it in the context of robotic reaching tasks. The authors trained a set of state-of-the-art DRL algorithms in two simulated environments where an agent controls the WidowX MKII 6 DoF manipulator with joint position commands. Afterwards the resulting policies are tested for average return over 100 episodes, train time, the success ratio and reach time for different thresholds (50 mm, 20 mm, 10 mm and 5 mm) both in simulation and with a real robot to evaluate its generalization to physical systems. These environments define two variations of a reaching task where the robot has the same initial joint positions and the goal is either constant across all episodes (fixed goal) or randomly sampled for every new episode (random goal), both always within reach of the robot. The environment is simulated in the PyBullet physics engine and implemented with OpenAI Gym. An increase in difficulty is observed from the fixed goal to the random goal settings. In the environment with the fixed goal multiple agents achieved 100% success rate. For the random goal fewer agents managed to learn a successful policy and the best performing were the off-policy algorithms which used a sample augmentation technique termed Hindsight Experience Replay (HER) [30] achieving a maximum success rate of 67%.

Chapter 3

Materials and Methods

This chapter presents the materials and methodologies used in this dissertation to solve the proposed robotic manipulation tasks. The chapter is composed of three parts: Section 3.1 justifies the choice of software tools and describes their functionalities; Section 3.2 discusses the implementation details and results obtained in developing DRL agents to solve three classic RL problems; and Section 3.3 presents the definition of the chosen robot manipulator reaching tasks, the algorithms used to solve them and the metrics used in evaluating the resulting agents.

3.1 Software Tools

The following section presents the software tools used for the implementation of the reaching tasks, the DRL agents and their respective training: Section 3.1.1 presents the OpenAI Gym RL framework; Section 3.1.2 describes the operation of the robo-gym toolkit used to simulate the reaching tasks; Section 3.1.3 describes the PyTorch tensor library for Deep Learning which contains automatic gradient calculation features used to implement and train DNNs; Section 3.1.4 presents the Stable Baselines3 DRL library; Section 3.1.5 presents the Ray Tune hyperparameter optimization library used to improve the agent's learning.

3.1.1 OpenAI Gym

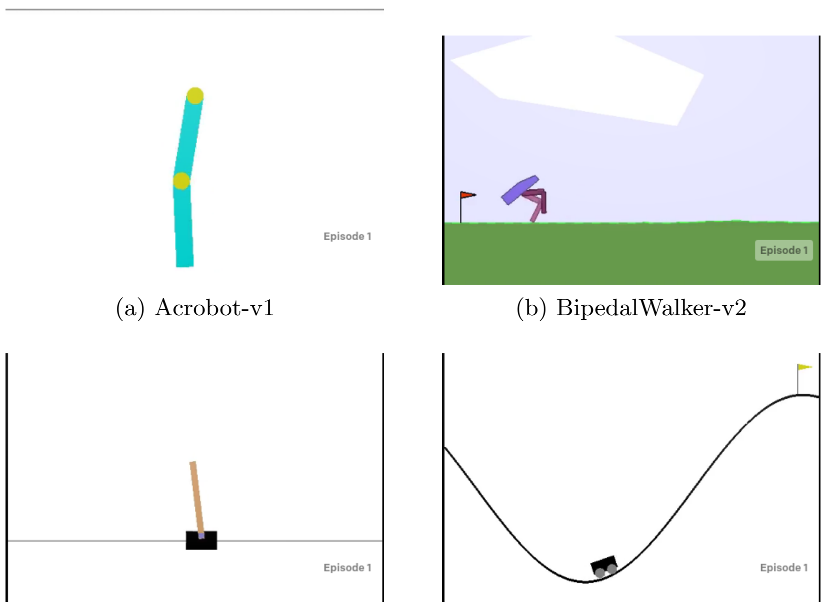
OpenAI Gym [18] is an open source RL research toolkit which standardizes an interface for Reinforcement Learning tasks. It offers a set of default tasks, referred to as *environments* (some are depicted in Figure 3.1), used for developing and benchmarking RL algorithms. These environments follow commonly defined interface of interaction and behave as an episodic MDP: at each time step the *agent* performs an *action* and receives a *reward* and a new *observation* (state). The following snippet of Python code illustrates the usage of the Gym's environment (*env*) interface alongside an *agent* object:

```
import gym
env = gym.make('CartPole-v0')
observation = env.reset()
done = False
while not done:
    action = agent.predict(observation)
```

```
        observation, reward, done, info = env.step(action)
    env.close()
```

Initially, the *env.reset* function is called in order to setup the environment on a new episode and returns its initial observation. In this case, the environment runs a single episode until it reaches some terminal state. While the episode is not over, the agent (implemented by the user) chooses an *action* based on the previous *observation*. The *env.step* function is then called to progress the environment to the next time step by running the *action* chosen by the agent. The *env.step* function then returns 4 values:

- *observation* - an object representing the next state
- *reward* - a float number rewarding the previous action
- *done* - a boolean flag indicating the termination of the episode
- *info* - a python *dictionary* containing diagnostic and debugging information



(a) Acrobot-v1 (b) BipedalWalker-v2

(c) CartPole-v1 (d) MountainCar-v0

Figure 3.1: Four default environments provided by OpenAI Gym.

An abstraction exists only for the environment and not for the agent, as long as the agent produces an action compatible with the *env.step* function. This constraint is defined for each *environment* in the form of a *Space*. An environment's implementation defines an action space and observation space describing the format of valid actions and observations. A space can be *discrete*, in which case it is specified by an integer number (e.g., 5 different actions), or *continuous* defined as an $n-$dimensional real space $\mathbb{R}^n$ where each dimension can have different bounds or be unbounded. Other space representations exist derived from these two.

The following code listing contains an example of creating these two type of spaces where the discrete *space1* defines the set $\{0,1\}$ of possible values and the continuous *space2* defines a two dimensional space where the first dimension takes values in $[-1.0, 2.0]$ and the second dimension in $[-2.0, 4.0]$.

```
import gym
import numpy as np
space1 = gym.spaces.Discrete(2)
l = np.array([-1.0,-2.0])
h = np.array([2.0,4.0])
space2 = gym.spaces.Box(low=l, high=h)
```

3.1.2 Robo-gym

The simulation of the reaching tasks' environment is based on the open source toolkit *robo-gym* [5]. Robo-gym defines a framework to help create OpenAI Gym environments for real or simulated robots, while being language agnostic regarding the backend used to develop them. This was the simulation tool chosen for this dissertation due to: (i) being flexible and extensible; (ii) using Gym as a top level interface for the agent's interaction with the environment; (iii) providing already existing implementations of environments for several tasks; (iv) using the ROS (Robot Operating System) [31] framework for the robot's programming and Gazebo [32] as a robotics simulator (both popular throughout the robotics community). ROS is an open source middleware suite containing a set of software libraries and tools useful to build robot applications. Although not a real operating system, it includes functionalities usually present in one, such as: hardware abstraction; low-level device control; package management; and message-passing using its communication infrastructure. The communication is done by remote procedure call services, asynchronous streaming of data over *topics* using a *Publish-Subscribe* pattern and data storage on a *Parameter Server*. The following is a summary of the main features of robo-gym:

- Integration of multiple commercially available industrial robots.
- Built-in distributed capabilities, which allow the use of distributed algorithms and hardware, making it scalable.
- Easier transfer from training in simulation to real world robots, in part because of ROS.
- Use of gRPC[1] as a communication layer between Gym's front end and the robot's backend implementation, allowing for the development of robots using different programming languages (supported by gRPC).
- Using the OpenAI Gym interface, which is the standard for modern RL research.
- Open source project[2]: it is easily extended and can be used without licensing costs.

[1]A remote procedure call (RPC) framework, `https://grpc.io/docs/what-is-grpc/faq/`
[2]`https://github.com/jr-robotics/robo-gym`

Framework Architecture

Robo-gym is structured in a distributed architecture where the agent interacts with a standard Gym environment which then communicates, using gRPC messages, with a *Robot Server* which may be running in a separate machine. This architecture, shown in Figure 3.2, is made of seven main components:

1. **Real or Simulated Robot** - This includes the robots and sensors implemented in ROS, and the simulated Gazebo scenes.

2. **Command Handler** - ROS node responsible for publishing command messages to the robot's actuator topics at a defined *robot-actuation* rate. Receives command messages from the *ROS Bridge* on a queue of size one. When the queue is empty, it publishes a command to interrupt the movement. The implementation is specific to each robot.

3. **Robot Server** - An instance of a gRPC server that contains a single *ROS Bridge* Python object through which it interacts with a single robot. It handles the gRPC messages received from the *Environment* and calls functions in the *ROS Bridge* accordingly.

4. **ROS Bridge** - Python Class specific to each robot responsible for mediating the interaction between the *Robot Server* and the real or simulated robot, through ROS messages. It gathers sensor information using various callbacks and stores it for when requested (e.g., get the current state of the robot). It also publishes command messages to the *Command Handler*. All this through function calls made by the *Robot Server*.

5. **Server Manager** - A gRPC server (usually one per machine) responsible for managing the Robot Servers and the simulations. This includes instantiating, termination and error handling for automatic restarts.

6. **Environment** - OpenAI Gym *Env* class implementing the hidden dynamics required to expose the standard environment interface. It works by sending/receiving information (through gRPC messages) to/from a *Robot Server* or a *Server Manager*.

7. **Agent** - The RL algorithm interacting with the *Environment*.

In order to understand the agent-environment interactions at each time step and describe the real time behaviour of robo-gym, it is necessary to discern the different time intervals of each component. The *action cycle time* is the time between any two consecutive actions sent by the Agent (referred in this dissertation more often as *action cycle rate* (ACR)). The *robot-actuation cycle time* is the time between commands sent to the robot's controller by the *command handler*, which can be smaller than the *action cycle time*. The *action generation time* is the time required for the agent to produce an action upon receiving a new state. The *sleep time* is the difference between the *action cycle time* and the *action generation time*, and constrains the rate at which the agent can send actions. The relation between these intervals and the ordering of events is depicted in Figure 3.3.

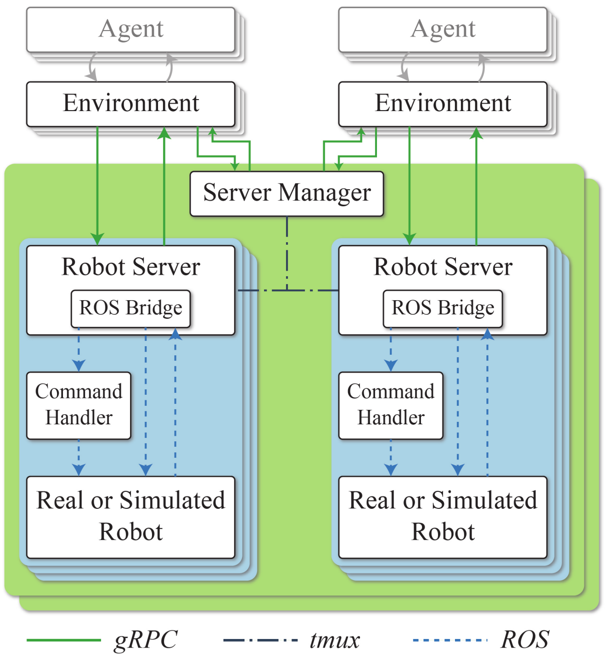


Figure 3.2: Robo-gym architecture, image from [5]

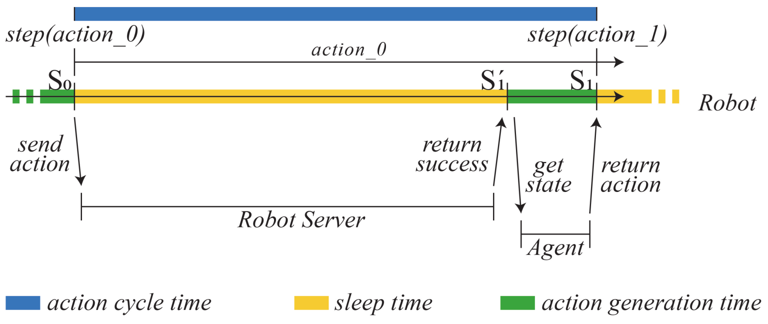


Figure 3.3: Robo-gym interaction between agent and robot server, image from [5]

UR10

Amongst the available robots in this version of robo-gym (MiR 100 and some Universal Robots) the one used in this dissertation is the UR10[3] collaborative robot (see Figure 3.4). Known for its ease of use and robustness, the UR10 is a 6 DoF robotic arm intended to work besides humans on more repetitive tasks which has experienced an increase of use in research and industrial applications. It has a maximum range of 1300 mm, repeatability of 0.1 mm and a Tool Center Point (TCP) speed limit of 1 m/s. Remaining relevant specifications are listed in Table 3.1.

[3]`https://www.universal-robots.com/media/50880/ur10_bz.pdf`

Attribute	Value
Weight	28.9 kg
Payload	10 kg
Reach	1300 mm
Repeatability	0.1 mm
Joint ranges	+/- 360º
Joint	**Max Velocity**
Base	120º/s
Shoulder	120º/s
Elbow	180º/s
Wrist 1	180º/s
Wrist 2	180º/s
Wrist 3	180º/s

Table 3.1: UR10 specifications

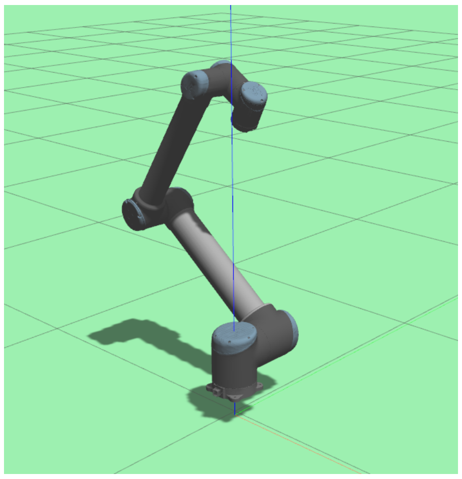


Figure 3.4: UR10 robot in Gazebo simulation

3.1.3 PyTorch

PyTorch is an open source tensor library used mostly for Deep Learning. In this context, *tensors* are a type of data structure generally defined as n-dimensional matrices where n is referred to as the *rank* or, in other words, the number of indices required to index a number in a tensor. At the core of Deep Learning are neural networks which are often implemented as matrices and matrix operations. This, along with the formulation of many Deep Learning problems, makes training neural networks highly parallelizable, and consequently faster by using specialized hardware or Graphics Processing Units (GPU). PyTorch is one of multiple libraries which provide a high-level abstraction to perform tensor operations optimized both for GPUs and Central Processing Units (CPU). More interestingly, it also provides an array of functionalities to define neural networks and train them using its automatic differentiation system named *autograd* together with various implementations of gradient based optimization algorithms (*torch.optim*).

Computational graph

Similarly to other Deep Learning libraries, PyTorch makes use of *computational graphs* for computing gradients. A computational graph is a Directed Acyclic Graph (DAG) used to store a history of all operations performed on input data and its intermediate results. It represents a function (mathematical expression) as a sequence of operations done on input tensors, which are the leaves of this graph, resulting in output tensors, which are the roots. For example, consider the following expression for the case of *0-rank* tensors (scalar values):

$$z(a,b) = c.a^2 + b \tag{3.1}$$

This function takes two real numbers, a and b, and outputs a single real number parameterized by a constant c. The computational graph of this expression can be represented by the graph in Figure 3.5, where each node is a tensor. Some nodes are tensor variables while others represent tensor results of operations.

PyTorch defines computational graphs in run time as a forward pass (sequence of operations) is executed, which makes it a *dynamic* computational graph. In contrast, *static* computational graphs are those which are defined previously before execution (as used in TensorFlow[4]).

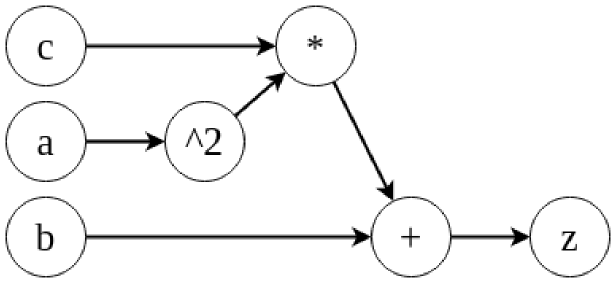


Figure 3.5: Computational graph of example expression

Autograd

As an expression is being executed, the PyTorch's automatic differentiation system (autograd) builds a computational graph by creating a *Function* object for each operation. This Function object defines how to compute an operation's tensor outputs from tensor inputs (*forward* method) and how to compute its gradient with respect to the tensor inputs (*backward* method) from the gradient of some other function with respect to the output tensors.

After the forward pass, calling the *backward* method on a root node will compute the gradient of this root tensor with respect to the graph leaves. The graph is then processed in reverse order by calling the *backward* method of each *Function* object and passing the resulting gradient to the next *Function* objects. This is an implementation of the back-propagation algorithm, which relies on the chain rule. The resulting gradients can then be used in conjunction with an optimizer to, for example, minimize a loss function in a supervised learning problem.

3.1.4 Stable Baselines3

Different implementations of the same Reinforcement Learning algorithm (and often the same implementation) can have vastly distinct performances in a given task. This becomes an issue when attempting to evaluate a new algorithm and comparing it with existing ones. An improvement in performance of the new algorithm might be due to the use of unreliable and poorly optimized baselines in the comparison.

In an attempt to solve this problem and allow for easier replication of results and easier development of new algorithms, OpenAI Baselines[5] was released in 2017. This is a Python framework containing a set of high-quality TensorFlow implementations of RL algorithms whose performance was matched with previously published results.

Subsequently, in 2018 the Stable Baselines[6] fork of OpenAI Baselines was released. This version, now referred to as SB2, is focused on simplicity of use and consistency by providing

[4]TensorFlow is another machine learning library like PyTorch (`https://www.tensorflow.org/`)
[5]`https://github.com/openai/baselines`
[6]`https://github.com/hill-a/stable-baselines`

new algorithms, improved code, bug fixes, a common interface for every algorithm, and other additional functionalities.

The SB2's dependency on OpenAI Baselines and the release of TensorFlow 2 deprecating some of its initial codebase, motivated a rewrite of Stable Baselines into a new version using PyTorch: Stable Baselines3 (SB3) [33] which had its first major release in early 2021.

This is the library used in this dissertation for the implementation and training of DRL algorithms. It contains implementations of state-of-the-art on-policy and off-policy algorithms such as: Advantage Actor Critic (A2C), Deep Deterministic Policy Gradient (DDPG), Deep Q Network (DQN), Hindsight Experience Replay (HER), Proximal Policy Optimization (PPO), Soft Actor Critic (SAC) and Twin Delayed DDPG (TD3). Its functionalities include saving and loading models; callbacks for monitoring learning and evaluating the models; TensorBoard[7] support for plotting training curves and other custom metrics; custom network architectures; and vectorized environments support for parallelized training. More importantly, SB3 is fully compatible with the OpenAI Gym's environment interface.

3.1.5 Ray Tune

Hyperparameter optimization is an important step in model selection to increase performance in Machine Learning problems. Ray Tune [34] is an open source library for automatic hyperparameter optimization which offers many cutting-edge black-box optimization algorithms and allows them to be executed distributedly across multiple machines.

Using this framework consists of defining a hyperparameter *Search Space* and choosing a *Search Algorithm* and/or *Trial Scheduler*. The Search Algorithm provides a configuration of hyperparameters, taken from the Search Space, to run on the next trial. This configuration can be random, be part of a brute-force grid search or based on previous observed performance. The Trial Schedulers can be paired with a Search Algorithm and provide functionality to decrease search time based on performance metrics. Namely, it allows for early termination of bad trials, pausing trials, cloning trials and altering hyperparameters of a running trial.

Additionally, Ray Tune provides many checkpointing (within experiments and trials) and logging functionalities, making the search easily monitored and resistant to hardware or software failures.

3.2 DRL for Simple Problems

In order to become more familiarized with the tools and details of developing RL agents, three algorithms were implemented and explored more in depth: Q-learning, Deep Q-learning and DDPG. For each of these algorithms, an example problem was used to evaluate its application. Each successive algorithm tries to solve the limitations of the previous one when applied to a more complex environment. This section presents the implementation details and results obtained in the process of developing these agents with the goal of illustrating the use of each algorithm and learning about some practical aspects of DRL. Initially, tabular Q-learning, a classic RL algorithm, is applied to a Gridworld problem (Section 3.2.2). Afterwards, the state representation of the same Gridworld problem is changed to be high dimensional and is solved using an agent based on the Deep Q-Learning DRL algorithm (Section 3.2.3). Finally, the continuous control pendulum swing-up problem is solved using the

[7]TensorBoard is a metrics visualization tool (`https://www.tensorflow.org/tensorboard`)

DDPG algorithm (Section 3.2.4). This allows to arrive at an appropriate class of algorithms capable of solving problems with high dimensional and continuous state/action spaces.

3.2.1 Gridworld

Gridworld is often used as a toy environment to illustrate the application of RL algorithms since it can be easily modeled as a finite MDP. A simple engine for a version of the Gridworld game was developed using Python. The Gridworld environment used in the next examples (depicted in Figure 3.6) is composed of a 10×10 board with two obstacles (black) and the origin (0,0) at the top left corner. The agent (blue) and the goal (yellow) are placed in different positions on the available free space. An agent's objective is to navigate the board avoiding the obstacles and reaching the goal through the shortest path possible within a fixed number of steps (the maximum episode length is 1000 steps).

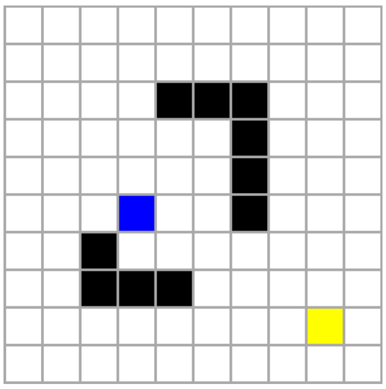


Figure 3.6: A Gridworld board

Assuming a board with no obstacles and a static goal, the state space dimension is $|S| = 10 \times 10 - 1 = 99$, where each state can be represented as tuple $s = (x_{agent}, y_{agent})$ corresponding to the position of the agent on the board. With a random goal position, the state is a tuple with 4 components, which results from adding the position of the goal to the previous tuple $s = (x_{agent}, y_{agent}, x_{goal}, y_{goal})$. In this case, and with no obstacles, the state space dimension is $|S| = (10 \times 10) \times (10 \times 10 - 1) = 9900$. For the board used in the next examples (Figure 3.6) the obstacles occupy $k = 10$ cells, thus the resulting state space dimension is $|S| = (10 \times 10 - k) \times (10 \times 10 - 1 - k) = 8010$. At each time step of the Gridworld game loop, the agent receives an observation of the environment as a state $s \in S$, takes an action $a \in A$, where $A = \{up, down, left, right\}$ is the action space, and receives a reward r. The reward r is computed according to the following function:

$$R(x_{agent}, y_{agent}) = \begin{cases} 10, & \text{if } (x_{agent}, y_{agent}) = (x_{goal}, y_{goal}) \\ -1, & otherwise \end{cases}$$

3.2.2 Solving Gridworld with Tabular Q-learning

In Chapter 2, Q-learning is presented as a TD control algorithm that learns the optimal policy π^* by learning the optimal Q-value for each state-action pair. The following example aims to train an agent to solve Gridworld by learning a tabular Q-function. This Q-function is implemented as a lookup table of values with an entry for each state-action pair which are updated, according to the rule in Equation 2.10, as the agent explores. Solving the Gridworld problem means solving one of three different scenarios regarding the position of the agent and goal: a) static agent and goal; b) random agent and static goal (or vice versa); c) and random agent and goal. This example solves the third scenario c), hence the values in the lookup table are indexed using the state representation $(x_{agent}, y_{agent}, x_{goal}, y_{goal})$ which is enough to uniquely identify all configurations of an agent and goal. The algorithm used for training the agent is shown below (Algorithm 3).

Considering the objective of learning to move from the initial agent position to the goal for all possible starting states, the agent is set to learn in $M = 10000$ episodes. After selecting a random starting state, the episode advances in the while loop where the Q table is updated. The episode ends if the maximum number of steps for an episode is reached or the agent finds the goal. The learning rate α was chosen empirically: if it is too small the convergence time increases, but it must be small enough to converge. The λ parameter must only be greater than 0, otherwise it is only learning to predict the -1 reward for taking a step. The values for all hyperparameters used are listed in Table 3.2.

Algorithm 3 Algorithm for training the agent with tabular Q-learning

```
1: Initialize $Q$ table with 0 values
2: for episode = 1,$M$ do
3:     Randomly sample a starting state $s_0 = (x_{agent}, y_{agent}, x_{goal}, y_{goal})$
4:     $t \leftarrow 0$                                        ▷ Episode step
5:     $win \leftarrow false$
6:     while $t < L$ and $win =$ false do
7:         With probability $\varepsilon$ select a random action $a_t$
8:         Else select action with $a_t = \text{argmax}_a Q(s_t, a)$
9:         Take action and observe reward $r_{t+1}$ and next state $s_{t+1}$ from Gridworld engine
10:        if $s_{t+1}$ is the win state then
11:            $win \leftarrow true$
12:        Update the $Q$ table according to the rule
13:        $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$
14:        $t \leftarrow t + 1$
15:    if $\varepsilon > \varepsilon_{min}$ then
16:        $\varepsilon \leftarrow \varepsilon - 1/M$
```

A variation of *ε-greedy* strategy is used in Algorithm 3 which includes one additional factor: ε is decremented linearly at the end of each episode by $1/M$ until a minimum value ε_{min} is reached. This has the effect of having the agent explore earlier in the training process and, eventually, turn to exploiting near the end. As a result, this decreases training time as episodes become shorter (due to acting greedily) at the cost of potentially losing accuracy of the Q-value estimates by not visiting every state as often (which was observed not to be a problem in this case). After the 10000 episodes of training the agent was tested on all board configurations which resulted in two performance measures: 94.0% win rate and 75.4% optimal path rate. A win is defined as reaching the goal within 50 steps. The optimal path rate measures the ratio of winning games in which the total number of steps are equal to the length of the optimal path. This optimal path length is computed using the Manhattan distance which, given the obstacles, is not correct for some episodes but is still useful for comparing different policies. The resulting learned policy, considering a goal in position (8,8), is depicted in Figure 3.7 where the arrows represent the action with the highest Q-value for each cell. To illustrate the learning progress, the graph in Figure 3.8 shows the of number of steps per episode taken by an agent as it learns to move from position (0,0) (top left) to (9,9) (bottom right) in a 10×10 board with no obstacles. Initially, with a high value of ε the agent mainly explores, often reaching the maximum number of steps in the first 100 episodes. As the agent learns and ε decreases, the learning becomes clear as it approaches the optimal path length of 18 steps.

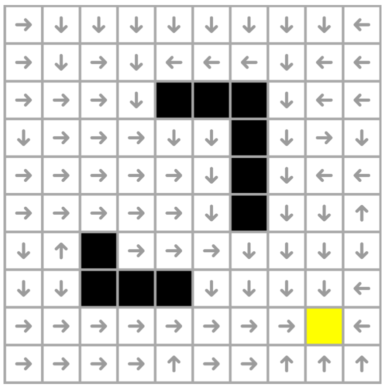


Figure 3.7: Learned policy visualized for a fixed goal

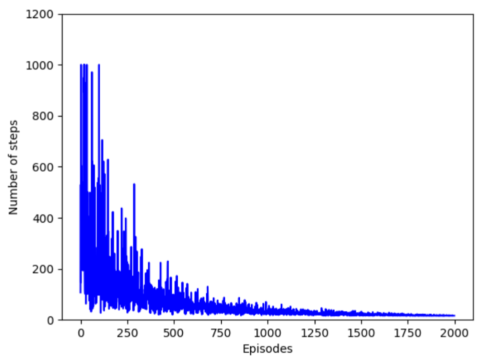


Figure 3.8: Steps as agent trains for one initial configuration

Although the trained agent performed reasonably well given the number of episodes, tabular Q-learning is limited to MDPs such as this version of Gridworld where the state and action spaces are discrete and small. Tabular methods are generally not applicable to RL tasks where state and action spaces are big. For example, tasks involving images as state representations with hundreds of pixels and continuous actions spaces where the range is too big to be discretized without significant loss of resolution. The consequent problems that arise from applying tabular methods to these tasks are: tables which occupy large memory space and the impossibility of accurately updating all of the possible state values. The next section presents RL methods that leverage supervised learning to replace tables by function approximation techniques. This allows for the application of RL to tasks with high-dimensional state spaces and continuous action spaces.

Hyperparameter	Value	Description
α	0.6	learning rate
γ	0.99	discount factor
ε	1.0	initial exploration parameter
ε_{min}	0.1	minimum exploration parameter
M	10000	number of episodes
L	1000	maximum episode length

Table 3.2: Hyperparameters used for training the tabular Q-learning agent

3.2.3 Solving Gridworld with DQN

To exemplify the application of Deep Q-learning to tasks with high-dimensional state spaces, the previous Gridworld problem described in Section 3.2.1 was changed to have the state representation be a $2 \times 10 \times 10$ tensor. This tensor is two 10×10 matrices with one-hot encoding (all zeros and a single 1) of the position of the agent and goal to which random noise was added. An DQN agent was then implemented using PyTorch and trained to solve this new version of the Gridworld problem with a random agent and goal. Similarly to DeepMind's Atari 2600 agent [12], this implementation also makes use of a replay buffer and a target network to improve the DQN performance.

The two one-hot encoding matrices are flattened and stacked to produce a 200 dimensional vector which consists of the input layer. The hidden layers are two fully connected layers with ReLU activation of 150 and 100 neurons. The output layer is a fully-connected linear layer with 4 neurons: one for each action. The algorithm implementation is summarized in Algorithm 4. The hyperparameters observed to have the most weight in the agent's performance and learning were the replay buffer size N (the maximum number of stored past experience tuples) and target network update frequency C (the rate at which the target network is updated). The set of hyperparameters used for the following results are in Table 3.3.

Algorithm 4 Algorithm for training the DQN agent

```
1: Initialize $Q$ network with random parameters $\theta$
2: Initialize target $\hat{Q}$ network by copying parameters $\theta^- \leftarrow \theta$
3: Initialize replay buffer $D$ with size $N$
4: for episode = 1,$M$ do
5:     Randomly sample a starting state $s_0 = (x_{agent}, y_{agent}, x_{goal}, y_{goal})$
6:     $t \leftarrow 0$                                                  ▷ Episode step
7:     $win \leftarrow false$
8:     while $t < L$ and $win =$ false do
9:         With probability $\varepsilon$ select a random action $a_t$
10:        Else select action with $a_t = \text{argmax}_a Q(s_t, a)$
11:        Take action and observe reward $r_{t+1}$ and next state $s_{t+1}$ from Gridworld engine
12:        if $s_{t+1}$ is the win state then
13:            $win \leftarrow true$
14:        Add experience $(s_t, a_t, r_{t+1}, s_{t+1})$ to replay buffer $D$
15:        if number of stored experiences in $D \geq k$ then
16:            Sample minibatch of $k$ experience tuples $(s_j, a_j, r_{j+1}, s_{t+1})$ from $D$
17:            Set respective targets $y_j = r_{j+1} + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-)$
18:            Perform gradient descent with $(y_j - Q(s_j, a_j; \theta))^2$ and gradient clipping [-1,1]
19:            if $t \mod C = 0$ then
20:                Copy $Q$ parameters to $\hat{Q}$ by $\theta^- = \theta$
21:        $t \leftarrow t + 1$
22:    if $\varepsilon > \varepsilon_{min}$ then
23:        $\varepsilon \leftarrow \varepsilon - 1/M$
```

To illustrate the effect of using the replay buffer and the target network, different versions of the DQN agent were set to learn different Gridworld scenarios regarding the position of the agent and goal. Firstly, a vanilla DQN (without replay buffer or target network) agent in a static version of the same Gridworld board: agent starting position at (0,0) and goal at (9,9). The resulting loss after only 3000 episodes of learning is depicted in Figure 3.9, where the loss is clearly approaching zero.

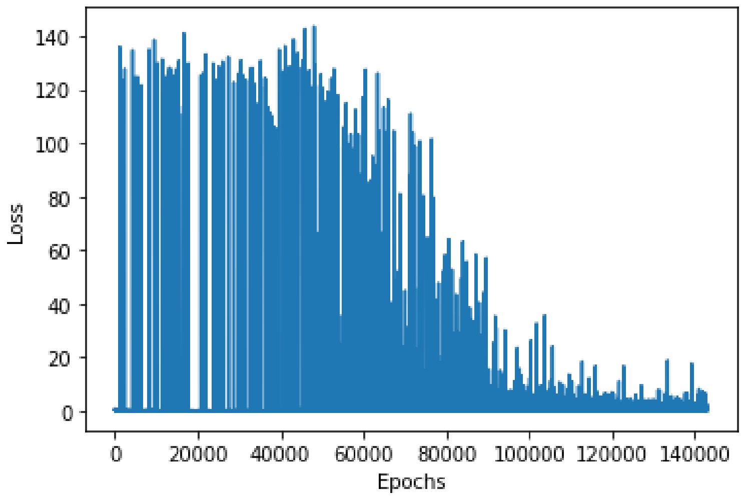


Figure 3.9: Vanilla DQN: static agent and goal

The instability and divergence problems described in DeepMind's work become obvious when attempting to train the same vanilla DQN agent on the scenario of a random agent and goal position. As shown in Figure 3.10, the loss does not decrease and the agent displays the *catastrophic forgetting* phenomenon (detailed in Section 2.2.2) resulting in not learning.

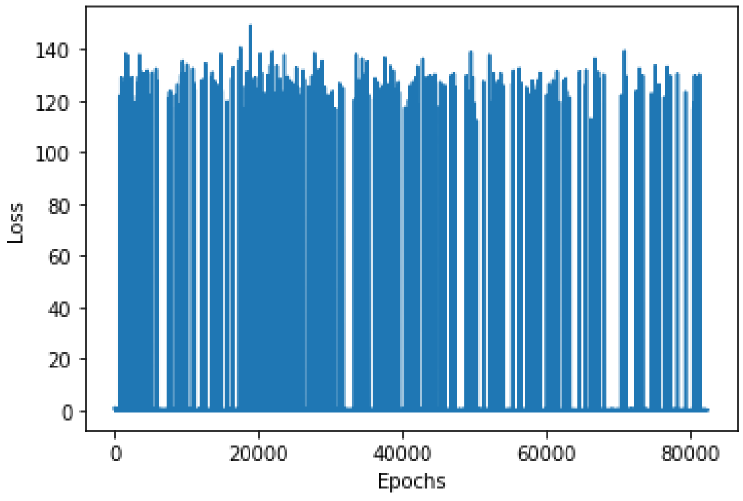


Figure 3.10: Vanilla DQN: random agent and goal

After implementing the experience replay buffer and the target network, the agent began to learn as seen by the downward trend of the loss function in Figure 3.11 (although not as prominent as in Figure 3.9). This final agent was evaluated with the same performance tests as the tabular version in Section 3.2.2 resulting in: 97.8% win rate and 57.4% optimal path rate.

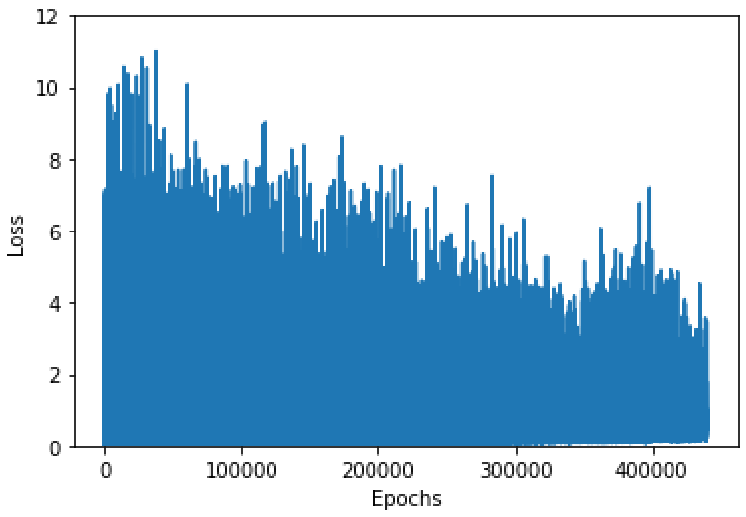


Figure 3.11: DQN with replay buffer and target network: random agent and goal.

As described previously, Deep Learning enables the application of RL algorithms, such as Q-learning, to tasks with high-dimensional state spaces. Nevertheless, the action space is still limited to low-dimensional and discrete actions spaces. Some problems require having continuous control where the actions are real valued and high-dimensional. These problems cannot be directly solved with a DQN without first discretizing the action space, which often leads to an enormous number of actions. The next subsection presents a different approach to learning a policy without directly using action-value functions. This approach still leverages DNNs for feature learning of the state space and uses many of the strategies employed to achieve stability in a DQN, and can also be applied to problems requiring continuous actions.

Hyperparameter	Value	Description
α	0.0001	learning rate
γ	0.8	discount factor
ε	1.0	exploration parameter
ε_{min}	0.1	minimum exploration parameter
k	32	minibatch size
N	50000	replay buffer size
C	10000	target network update frequency
M	10000	number of episodes
L	100	maximum episode length

Table 3.3: Hyperparameters used for training the DQN agent

3.2.4 Solving the Pendulum Swing-up with DDPG

The pendulum swing-up is a classic control problem. The goal is to swing up a frictionless pendulum using a single actuated joint in order for it to stay upright. The version used here (depicted in Figure 3.12) is an implementation part of the default set of environments provided in OpenAI Gym.

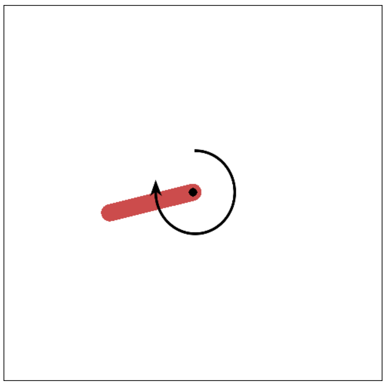


Figure 3.12: Gym's Pendulum-v0 environment

The pendulum environment is characterized by a state vector with 3 components: $\cos(\theta) \in [-1, 1]$; $\sin(\theta) \in [-1, 1]$; and the angular velocity $\omega \in [-8, 8]$. The action vector is a single component: joint effort (torque) $x \in [-2, 2]$. The reward function is:

$$R(\theta, \omega, x) = -(\theta^2 + 0.1 \times \omega^2 + 0.001 \times x^2) \tag{3.2}$$

where θ is the angle of rotation. Each pendulum episode begins with random starting angle $\theta \in [-\pi, \pi]$ and a random angular velocity $\omega \in [-1, 1]$ and ends within a configured number of steps.

A DDPG agent was implemented using PyTorch to solve the Gym's Pendulum-v0. The Algorithm 5 summarizes the agent training. Due to the noise added for exploration, the resulting action is clipped to the environment's maximum and minimum action values. The experience tuples stored in the replay buffer have one additional element: the "done" flag d, which is used to compute the targets such that $y = r$ for transitions in which the agent reaches a terminal state.

This implementation is structured into four networks: actor μ and critic Q and their respective target networks. The actor is composed of a single hidden linear layer of 50 neurons with ReLU activation followed by a linear output layer with 3 neurons (action). The output of this network is passed to a tanh activation function, which maps it to $[-1, 1]$, and is then multiplied by the maximum action value of 2. The critic network has two hidden linear layers of: 50 and 53 neurons with ReLU activation. The second hidden layer takes as input the activations of the first hidden layer (50) and the action from the actor (3). This allows the critic to learn a state representation separately from the Q-function using the action. Finally, the output layer is a single neuron with the Q-value. The hyperparameters used in training are listed in Table 3.4. The angle of rotation of the pendulum takes values between $[-\pi, \pi]$, hence the reward function (3.2) can yield a minimum reward of -16.274 or maximum of 0. An agent trained using this reward function will try to learn a policy which minimizes the angle, rotational velocity and joint effort. Figure 3.13 graphs the moving average of returns of the agent during training. This graph shows the agent is learning, as the returns increase over the 200 training episodes, and that it was able to balance the pendulum.

Algorithm 5 Algorithm for training the DDPG agent

```
1: Initialize critic network $Q$ with parameters $\theta^Q$
2: Initialize actor network $\mu$ with parameters $\theta^\mu$
3: Set parameters for target $\hat{Q}$ network by copying $\theta^{Q-} = \theta^Q$
4: Set parameters for target $\hat{\mu}$ network by copying $\theta^{\mu-} = \theta^\mu$
5: Initialize a replay buffer $D$ with max of size $N$
6: for episode = 1,$M$ do
7:     Randomly sample a starting state $s_0 = (x_{agent}, y_{agent}, x_{goal}, y_{goal})$
8:     $done \leftarrow false$
9:     $t \leftarrow 0$                                                        ▷ Episode step
10:    while $t < L$ and $done =$ false do
11:        Sample $noise \sim \mathcal{N}(0, 1)$                                 ▷ Gaussian noise
12:        Select action $a_t = clip(\mu(s_t) + noise, a_{min}, a_{max})$
13:        Take action $a_t$
14:        Observe reward $r_{t+1}$, next state $s_{t+1}$ and "done" flag $d_t$ from the environment
15:        Add experience $(s_t, a_t, r_{t+1}, s_{t+1}, d_{t+1})$ to replay buffer $D$
16:        if number of stored experiences in $D \geq k$ then
17:            Sample minibatch $B$ of $k$ experience tuples $(s_j, a_j, r_{j+1}, s_{t+1}, d_{j+1})$ from $D$
18:            Set respective targets $y_j = r_{j+1} + (1 - d_{j+1})\gamma\hat{Q}(s_{j+1}, \hat{\mu}(s_{j+1}))$
19:            Use $L_Q = \frac{1}{|B|}\sum_j (y_j - Q(s_j, a_j))^2$ as a loss for the critic $Q$
20:            Perform a gradient descent step with $L_Q$ on $\theta^Q$
21:            Use $L_\mu = \frac{1}{|B|}\sum_j Q(s_j, \mu(s_j))$ as a loss for the actor $\mu$
22:            Perform a gradient ascent step with $L_\mu$ on $\theta^\mu$
23:            Update target networks with polyak averaging:
24:            $\theta^{Q-} \leftarrow \tau\theta^Q + (1 - \tau)\theta^{Q-}$
25:            $\theta^{\mu-} \leftarrow \tau\theta^\mu + (1 - \tau)\theta^{\mu-}$
26:        $done \leftarrow d_{t+1}$
27:        $t \leftarrow t + 1$
```

Hyperparameter	Value	Description
α_μ	0.0001	Actor learning rate
α_Q	0.001	Critic learning rate
γ	0.99	discount factor
τ	0.001	polyak parameter
k	64	minibatch size
N	10000	replay buffer size
M	200	number of episodes
L	200	maximum episode length

Table 3.4: Hyperparameters used for training the DDPG agent

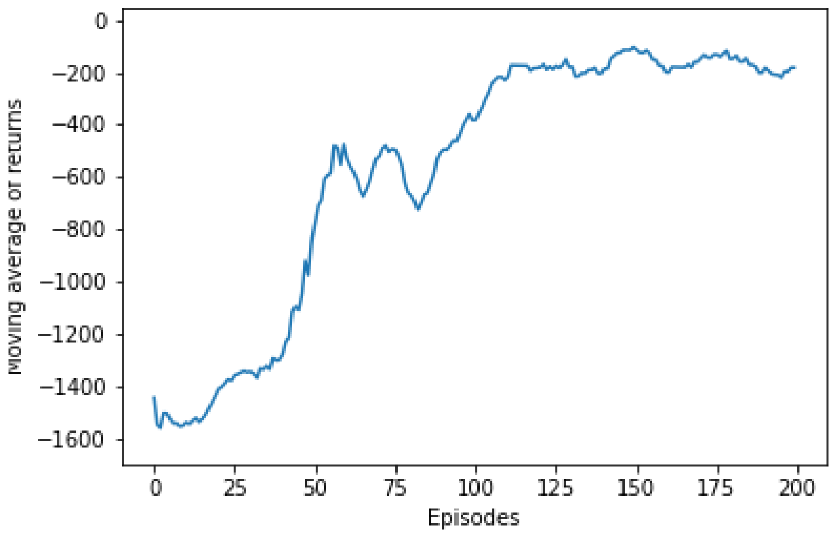


Figure 3.13: Moving average of returns (window of 10 episodes)

3.3 DRL for Reaching Tasks

This section presents and discusses the methods adopted in this dissertation to solve the proposed robot manipulator reaching tasks with Deep Reinforcement Learning. Concretely, three reaching tasks are defined and their respective Gym environment implementations detailed (Section 3.3.1). Afterwards, the two state-of-the-art DRL algorithms used to train the agents (Section 3.3.2), their respective training procedure, the hyperparameter choice and its optimization (when applicable) are described (Section 3.3.3). Finally, the different metrics used in evaluating the performance of each algorithm and trained agent are presented (Section 3.3.5).

3.3.1 Reaching Tasks Specification

The reaching tasks considered in this dissertation consist of an agent choosing the optimal sequence of actions to position a robot manipulator's end effector within some threshold distance of a target position. These actions are a result of a mapping between a desired position in *task space* (often Cartesian space) and a position in *joint space*. The desired joint position is then ensured by a low-level controller. The control commands performed by the agent can be either *position* or *velocity* based. In robo-gym, the UR10 simulation is controlled by position, this means the control commands are joint positions in radians. These commands are then sent to a Command Handler which, for a defined number of robot-actuation cycles, sends *JointTrajectory* commands (with a single joint trajectory point) through the '/pos_traj_controller/command' ROS topic to the robot's controller.

UR10Reach environment

This environment defines a task consisting of reaching a random target within the UR10's workspace given fixed initial joint positions. The workspace (depicted in Figure 3.14) is a semi sphere around the base link of the UR10 which is 0.1 meters above the ground. This environment is implemented as a *gym.Env* class which inherits from robo-gym's URBaseEnv class and is required to implement the *env.reset*, *env.step*, *env.reward*, *env._get_observation_space* and *env._get_action_space* Gym methods. The environment's state s is represented as a 13-dimensional vector containing the target's spherical coordinates (size 3), the joint positions (angles) normalized between $[-1, 1]$ (size 5) and the joint velocities (angular velocities) (size 5), where the last joint is ignored:

$$s = [r, \theta, \phi, \theta_{base}, \theta_{shoulder}, \theta_{elbow}, \theta_{wrist1}, \theta_{wrist2}, \omega_{base}, \omega_{shoulder}, \omega_{elbow}, \omega_{wrist1}, \omega_{wrist2}] \quad (3.3)$$

Additionally, two modes of defining the target's spherical coordinates are available: in the reference frame of the base link (0.1 meters above the world's origin) which makes it constant throughout an episode; or in the reference frame of the end effector which changes as the manipulator moves. An assumption is made that an existing sensor (e.g., RGB-D camera) is used to obtain the target's position in both these reference frames.

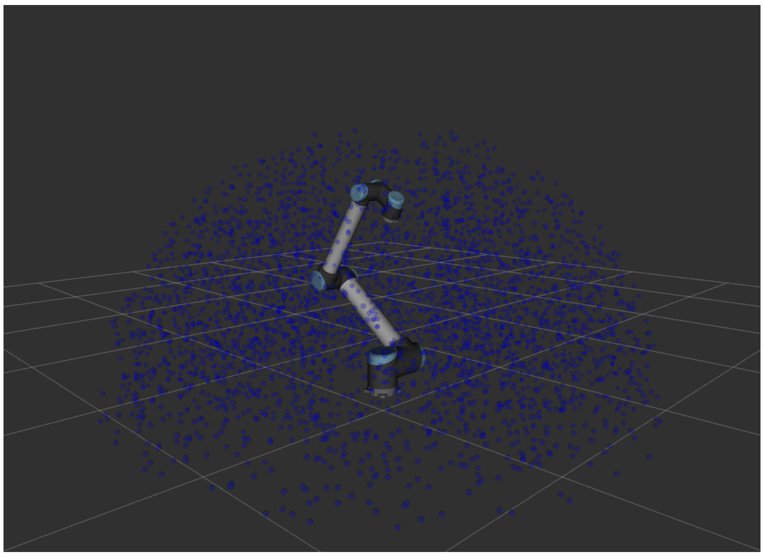


Figure 3.14: Random targets generated in the *UR10Reach* environment (in RViz[8])

The UR10 has 6 DoF, however, for this task no tool is attached to the manipulator, so the 6th joint (wrist3) is not controlled and it is fixed. The actions generated by the agent at each time step are vectors of 5 joint positions normalized between $[-1, 1]$:

$$a = [\theta_{base}, \theta_{shoulder}, \theta_{elbow}, \theta_{wrist1}, \theta_{wrist2}] \quad (3.4)$$

An episode has a maximum of 300 time steps but it may end before. There are 3 possible outcomes which lead to episode termination: collision of the robot against itself or the ground; exceeding the maximum number of time steps; or successfully reaching the target position with the end effector within a tolerance of 0.1 meters (referred to as threshold distance).

A dense reward signal is used, as it is often faster to learn in comparison to a sparse reward signal (such as -1 if not at the target, 0 if otherwise). This means that at each time step a

[8]RViz is a 3D visualizer for the ROS framework (`http://wiki.ros.org/rviz`)

base reward is given to the agent which is linearly proportional to the negative of the distance between the end effector and the target. Smaller the distance, the greater the reward:

$$reward_{base} = -||target\_coord - ee\_coord|| \tag{3.5}$$

A constant is added to this base reward depending if the episode ended on the last time step or not. The complete reward definition favours fast success and penalizes collision:

$$reward = \begin{cases} reward_{base} + 100, & \text{if } Success \\ reward_{base} - 400, & \text{if } Collision \\ reward_{base}, & otherwise \end{cases} \tag{3.6}$$

For this environment, the ACR is 12.5 Hz and the maximum velocity of each joint is decreased to 1/5 of the true maximum ($max\_velocity\_scale\_factor = 0.2$). An episode of 300 steps is then 24 seconds in simulation time. The maximum velocity is robo-gym's default and the ACR was chosen empirically (early results proved to work for this task definition) with guidance of [26]. In this work, the trade off in choosing the ACR is well summarized by the authors: a greater rate requires the agent to learn to gain momentum by repeating similar actions which makes learning harder; a smaller rate limits performance by reducing precision and increases time for data collection.

UR10ReachEval environment

This environment is the same as the *UR10Reach* in Section 3.3.1 except the initial configuration for the joint position is different and the targets appear randomly in the $z = 0.35$ meters plane closer to the end effector (as depicted in Figure 3.15). This change in configuration makes the task easier to learn and reduces the training time which is why it was used to quickly test different combinations of ACRs and maximum joint velocities.

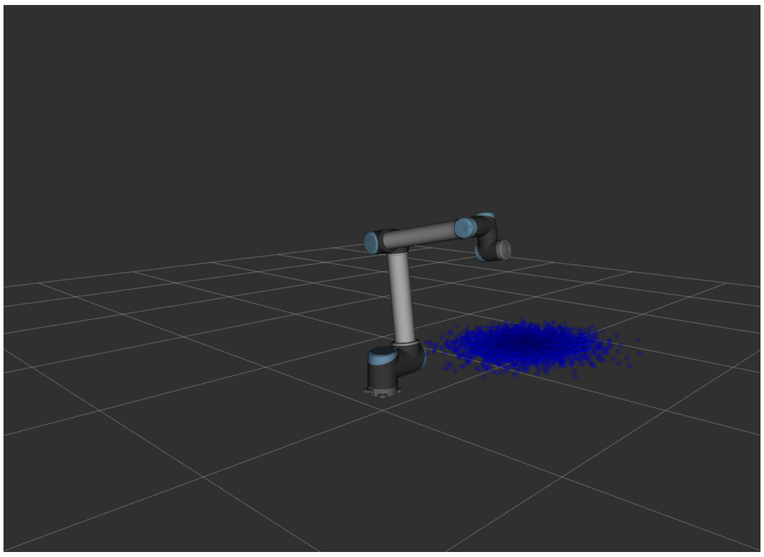


Figure 3.15: Random targets generated in the UR10ReachEval environment (in RViz)

UR10BallCatch environment

Fast manipulator movements are necessary for robotic tasks involving strict time requirements. The UR10BallCatch environment models one of those tasks: ball catching. The ball catching task considered here is defined as reaching a catching position within a time window and does not involve any type of grasping using a tool in the end effector. This catching position is the estimated interception point of a ball following a ballistic trajectory with a given plane. The success is defined not by the distance from the end effector to the catching position but rather to the ball position.

This task involves solving two problems: estimating a catching position during the trajectory of a flying ball; and quickly moving the end effector to the estimated catching position. In this context of reaching tasks, the agent is only set to solve the latter problem, while the former is solved by a classic method. Consequently, this environment has two modes of operation: **train** and **evaluation**.

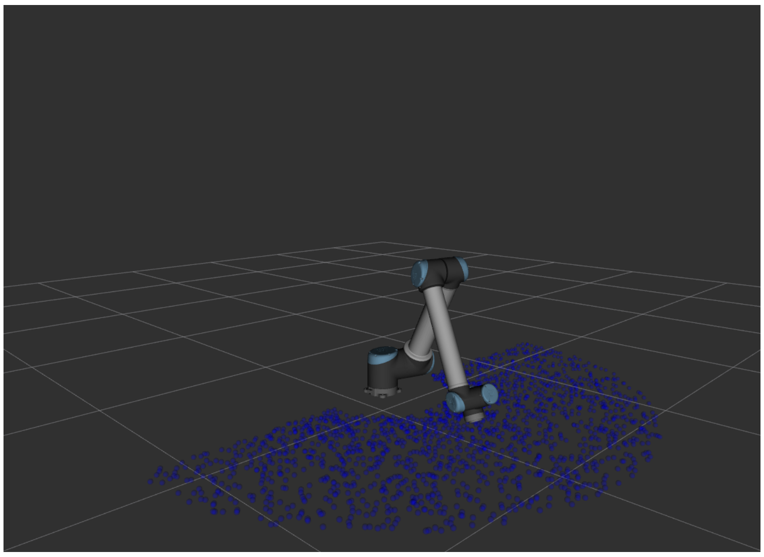


Figure 3.16: Random targets generated in the UR10BallCatch environment (in RViz)

The **train** mode is concerned with reaching random catching positions without the flying ball. Most of its implementation is shared with the *UR10Reach* environment, but the main differences are:

- Targets appear within an annular sector of the horizontal plane at the height of the base link in front of the manipulator (see Figure 3.16);
- The initial joint configuration was chosen so that the end effector is at the centroid of the annular sector (≈ 0.57 meters from the center of the base link), which is the point where the average distance to any other point is minimum;
- The base of the UR10 is 1 meter above ground, thus collisions with the ground are less frequent
- $max\_velocity\_scale\_factor = 1$, thus not limiting maximum joint velocities;
- Parameterizable ACR;

After learning a policy in **train** mode, a trained agent can then be tested in the **evaluation** mode. This consists in a scenario with the specifications of the **train** mode, adding a flying ball (for visualization purposes it is only 0.05 m in diameter) and a method to estimate the catching position.

For the flying ball, a set of 10 thousand random trajectories with an average velocity of ≈ 6.94 m/s were generated following these constraints:

- Positions during a trajectory were calculated for a sampling rate of 100 Hz
- Trajectories' duration before intercepting z=1 plane (base link plane) are between $[1.0, 1.2]$ seconds but the ball travels in the z axis for an additional 0.1 meters
- Trajectories' begin 5.3 meters away from UR10's base link (4 meters from the workspace) and 1 meter above the ground in a point aligned with the base link and the end effector (as depicted in Figure 3.17)

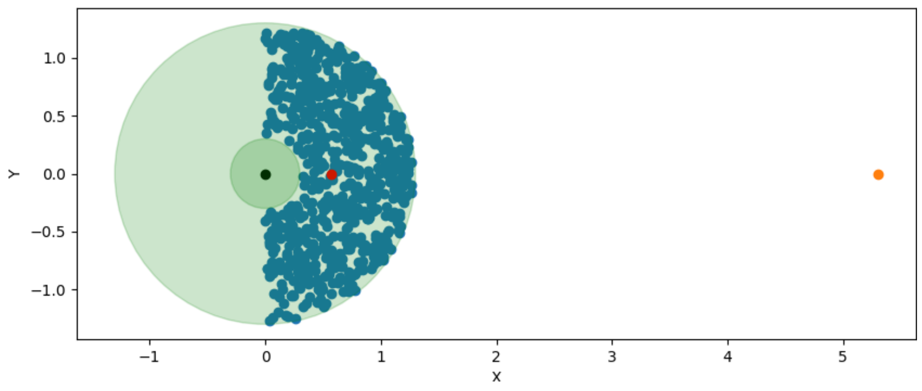


Figure 3.17: Setup of UR10BallCatch evaluation mode seen from above with the interception of random trajectories with the plane of the base link (in blue), end effector position (in red), initial ball position (in orange), and UR10's base link (in dark green).

The trajectories are computed by first choosing a random (x, y) position where $x \in [0, 1.3]$ and $y \in [-1.3, 1.3]$ and filtering by keeping those within the UR10's workspace. Afterwards, a random trajectory duration $t \in [1.0, 1.2]$ is chosen and the velocity, elevation angle θ and azimuth angle ϕ (as depicted in Figure 3.18) required to achieve the desired position within the specified time are computed using projectile motion equations (listed in 3.7) and solving for these variables.

$$\begin{aligned} v_x &= Vsin(\theta)cos(\phi) \\ v_y &= Vsin(\theta)sin(\phi) \\ v_z &= Vcos(\theta) \\ x(t) &= v_x t \\ y(t) &= v_y t \\ z(t) &= v_z t + gt^2/2, \ \texttt{for} \ g = 9.8m/s^2 \end{aligned} \tag{3.7}$$

These trajectories are stored in a file and randomly chosen during the evaluation, where a ROS node named *objects_controller* publishes the position of the ball at a rate of 100 Hz until the trajectory or episode ends. To this trajectory no noise is added given that the purpose of this environment is not to assess the estimation method's noise robustness.

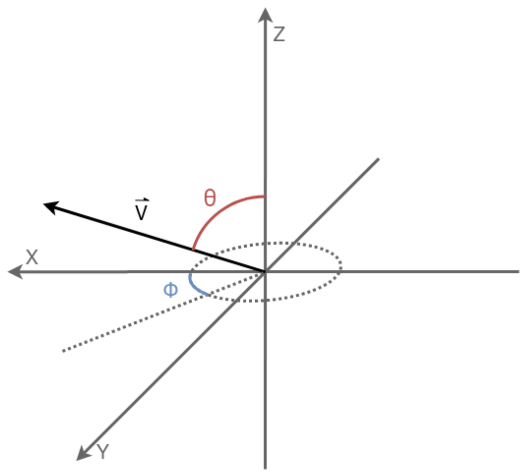


Figure 3.18: Initial parameters defining the trajectory of a ball: the velocity V, elevation angle θ and azimuth angle ϕ.

This ball position is then used by a *trajectory_prediction* ROS node which waits for a parameterized minimum number of ball position samples before using these in a polynomial approximation method to estimate the ball's interception point with the plane of the base link (1 meter above the ground). This plane was chosen due to being the cross section of UR10's workspace with the greatest area. The polynomial approximation method consists in using the *numpy.polyfit* function to obtain the coefficients for the 1-degree polynomials $x(t)$ and $y(t)$, and 2-degree polynomial $z(t)$. Afterwards, using the roots of $z(t)$ in $x(t)$ and $y(t)$ allows to calculate the catching position in the plane $z = 0$ of the base link's frame (or $z = 1$ of the world frame).

In the start of an episode, it may take more time to obtain the catching position than $1/ACR$, so the agent may wait longer in the first action cycle. This catching position is then returned to the agent as the target to reach.

3.3.2 Algorithms

The two model-free DRL algorithms used to solve the reaching tasks are TD3 and PPO. These algorithms were chosen for being state-of-the-art DRL algorithms representing the off-policy (TD3) and on-policy (PPO) classes allowing for these two to be compared. Both support n-dimensional continuous action and state spaces required for these tasks. The SB3's implementation[9] was used for training agents on the OpenAI Gym environments of the reaching tasks described previously.

Twin Delayed DDPG

Twin Delayed Deep Deterministic Policy Gradient (TD3) [35] is an Actor-Critic DRL algorithm which improves upon its predecessor DDPG (detailed in Section 2.2.3). The lat-

[9] `https://stable-baselines3.readthedocs.io/en/v1.0/modules/ppo.html`
`https://stable-baselines3.readthedocs.io/en/v1.0/modules/td3.html`

ter, suffers from a known problem of overestimation of value estimates also present in Deep Q-learning methods (detailed in Section 2.2.2). This problem results from both function approximation errors and the maximization performed in Q-learning which leads to suboptimal policies in the Actor-Critic setting. In [35] Fujimoto et al. present *Clipped Double Q-Learning* as a solution for the overestimation problem and two additional techniques used to reduce variance: *delayed policy updates*; and *target policy smoothing* regularization.

TD3 learns two Q-functions simultaneously. Similarly to DDPG with a single Q-function, both networks optimize a loss function L_{Q_i} using minibatch gradient descent to minimize the Bellman error:

$$L_{Q_i} = \frac{1}{|B|} \sum_{j} (y_j - Q_i(s_j, a_j))^2 \tag{3.8}$$

The learning targets y_j for each sample j are computed by taking the minimum of the two Q-functions' target networks (rather than the opposite target network as in Double Q-learning):

$$y_j = r_{j+1} + \gamma \min_{i=1,2} \hat{Q}_i(s_{j+1}, \hat{\mu}(s_{j+1})) \tag{3.9}$$

Taking the minimum acts as an upper-bound on the Q value used in the learning targets. Thus when optimizing Q_1, the target y_j is computed using the less biased $\hat{Q}_2$ if $\hat{Q}_2 < \hat{Q}_1$ otherwise the more biased $\hat{Q}_1$ is used (clipping $\hat{Q}_2$ to $\hat{Q}_1$) at the benefit of avoiding overestimation. Instead of performing a policy update at every time step, the policy is only updated every $d = 2, 3, 4, ...$ time steps, hence delaying updates to the policy network compared to the Q-networks. Additionally, the target networks of both critics and actor are also updated using *polyak averaging* at a rate of d. According to Fujimoto et al., the less frequent policy updates will use a value estimate from the Q-networks with less variance since the Q-networks were updated more often before the policy changed, resulting in lower value estimation error. Doing a policy update using a lower variance value estimate should result in higher quality policy updates. Additionally, a regularization method which adds noise to the target action when updating the critic network is used.

Proximal Policy Optimization

Proximal Policy Optimization PPO [6] is a policy gradient DRL algorithm with a similar goal to that of the Trust Region Policy Optimization (TRPO) [36]. These methods are sensitive to the update step size because of their on-policy nature, where a very large policy update can lead to catastrophic drops in performance. Similarly to A2C (detailed in Section 2.2.3), PPO uses a critic network to compute the advantage. However, it differs in how the policy gradient is computed by attempting to restrict the size of these update steps by various methods. PPO works by collecting a set of fixed length experience rollouts (e.g., 512, 1024, 2048 time steps) across multiple (or one) parallel actors and combine them into a single buffer to perform gradient descent on a loss function. The objective is to maximize the loss function (gradient ascent) by using the gradients with respect to the policy network's parameters.

If the probability ratio $r_t(\theta)$ (Equation 3.10), where $r_t(\theta_old) = 1$, is larger than 1, the given action is more probable under the new policy compared to old one. If r_t is less than 1, the action is less probable than before.

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{3.10}$$

The method used for constraining the updates consists of a clipping mechanism in the loss function which prevents the new policy from moving too far from the old policy. This clipping is parameterized by $\epsilon \in \mathbb{R}$ (e.g., $\epsilon = 0.2$) and constraints updates that change r_t out of the interval $[1-\epsilon, 1+\epsilon]$. The complete loss function is expressed in Equation 3.11.

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{3.11}$$

As illustrated in Figure 3.19, when the advantage is positive, the action will become more likely (the probability $\pi(a|s)$ will increase) and the loss function is clipped when $r_t \geq 1+\epsilon$, setting a ceiling for the update. Similarly, when the advantage is negative, the action will become less likely and the loss function is clipped for $r_t \leq 1-\epsilon$.

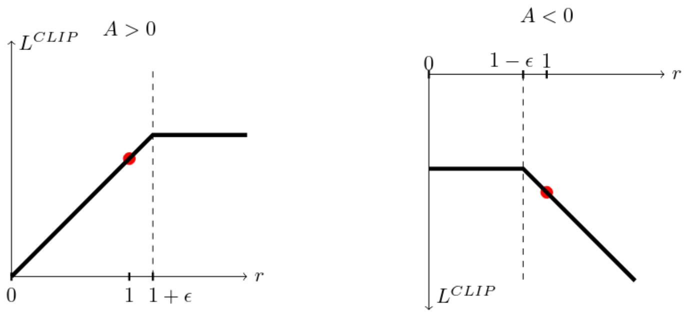


Figure 3.19: PPO loss function L^{CLIP} as a function of probability ratio r_t for positive and negative advantages, image from [6].

The loss function can be improved by adding a term $c_2 S[\pi_\theta](s_t)$ that ensure sufficient exploration. The function S is an entropy estimate of the current policy π_θ in the given state s_t. Increasing the loss function with this term included reinforces exploration by increasing entropy. This happens because the entropy of a probability distribution is greater the more uniformly distributed it is (more random). More random behaviour leads to more exploration, however this parameter must be carefully tuned.

3.3.3 Model Training and Evaluation

For a given experiment, an algorithm is trained on a single random seed for 2 million time steps (in some cases 1 million time steps) and every 50K time steps the current model (and replay buffer for TD3) is stored as a checkpoint.

A seed is a number used to initialize a random number generator (RNG). For the reaching task environments, it initializes the RNG used to choose the next episode's target, such that two environments with the same seed will see the same sequence of episodes. In SB3 it is used to initialize the RNG for Python, PyTorch and Numpy. This may have an effect on some aspects of an algorithm's learning, leading to changes in convergence and initial exploration (due to things such as weight initialization of a neural network).

To evaluate the agent during learning, the training is stopped every 20K time steps (1% of total time steps) and the current agent is set to run a deterministic version of its policy (e.g., without exploration noise) for 10 episodes with random seeds. The resulting episode returns and episode lengths are then stored. Due to the computational cost of Gazebo and in order to avoid having multiple simulations running on the same machine, two environment functions were implemented to be used in the evaluation: *get_state* which returns the state of the RNG, the current joint positions and the target's position; and *set_state* which resets the environment to the same previous values. This makes it possible to reuse the same simulation for training and evaluation, saving significant amounts of time in evaluating the agent by not having to start new simulations and running them in parallel (which also slows down each simulation). The process for evaluation during training consists of the following sequence:

1. Current training episode is stopped at the evaluation steps.
2. Episode state is stored.
3. Simulation is used in new episodes for evaluating the agent.
4. Initial episode state is restored.
5. Training is resumed.

The simulation time for the physics engine used in Gazebo (ODE in this case) is controlled in part by the *max_step_size* parameter (the time resolution of the simulation i.e changes in the simulator occur in steps of *max_step_size* simulation seconds). In conjunction with the *real_time_update_rate* parameter (real time rate at which the simulation time steps are advanced) the *real_time_factor* can be calculated with $max\_step\_size \times real\_time\_update\_rate$.

Across all experiments the *max_step_size* is 0.001 seconds. For training, the simulations were made to run as fast as they can (e.g., 6000 Hz) by having the *real_time_update_rate* parameter set to 0. With the hardware used, the training simulations were observed to run at a *real_time_factor* of ≈ 6. This means 6 simulation seconds would elapse every 1 second of real time. By running the simulation $6\times$ faster it is natural to expect the algorithm should also run $6\times$ faster. Recalling what is summarized in Figure 3.3, the *sleep time* would be shorter (in real time) but the action generation time would stay the same. Given the dynamic nature of this task, it seems possible that with the simulation $6\times$ faster the agent would learn differently (by sending actions with a greater delay) which could hinder the evaluations of the resulting policy made in real time ($\times 1$). However, it was empirically verified to have no significant impact in the evaluations after training.

To verify the ratio of the action generation time in the action cycle time, the action cycle time was measured 100 times in two tests both with $max\_velocity\_scale\_factor = 0.2$ and $action\_cycle\_rate = 12.5$. First an estimation of the action generation time was obtained by running *model.predict* for 1000 samples, resulting in 0.000538 seconds. Then one test of 100 samples was run for *real_time_factor* of ≈ 1 resulting in an average action cycle time of 0.0777 (close to $1/12.5 = 80$ ms) seconds and another for *real_time_factor* of ≈ 6 resulting in an average action cycle time of 0.0129 seconds (about 6.0233 smaller). The action generation time represents, respectively, 0.69241% and 4.1705% of the action cycle time.

3.3.4 Hyperparameter Optimization

Tables 3.5 and 3.6 contain the non optimized hyperparameters used in training. Some are the SB3's default (marked with *), the remaining were obtained either empirically or from other work using these algorithms. The networks' architecture was chosen to be two dense layers of 256 neurons. With early experiments it was observed slower convergence for greater network sizes (such as SB3's [400,300] default). Given the unknown complexity of the problem and some initial results obtained, this architecture was kept constant for all experiments.

The most common approach for systematic hyperparameter optimization in ML is *grid search*. This is a brute force method to find the best performing hyperparameters that consists in testing all possible configurations of a set of values for the chosen hyperparameters. *Random search* is a version of grid search more scalable to a greater number of hyperparameters by testing a fixed budget of random configurations.

Hyperparameter	Value
learning_rate	0.0003
buffer_size*	1000000
learning_starts	10000
batch_size	128
tau*	0.005
gamma	0.995
train_freq	100
gradient_steps	100
action_noise	0.2
policy_delay*	2
target_policy_noise*	0.2
target_noise_clip*	0.5

Table 3.5: Hyperparameters used for training the TD3 agents

Hyperparameter	Value
learning_rate	0.0003∗ $(1 - \frac{current\_timestep}{total\_timesteps})$
n_steps*	2048
batch_size*	64
n_epochs*	10
gamma	0.995
gae_lambda*	0.95
clip_range*	0.2
clip_range_vf*	None
ent_coef*	0
vf_coef*	0.5
max_grad_norm*	0.5
use_sde*	False
sde_sample_freq*	-1
target_kl*	None

Table 3.6: Hyperparameters used for training the PPO agents

However, for computationally expensive objective functions, such as the one considered in this dissertation where an agent takes 1-2 million time steps (and up to 14 hours of time) to fully train, the class of Sequential Model-Based Global Optimization (SMBO) algorithms provides a more feasible framework.

These optimization algorithms use results of past configurations to inform the choice of the next configuration of hyperparameters to test. They work by building a surrogate model that approximates the objective function, but it is significantly cheaper to evaluate than the objective function itself. The hyperparameters are chosen based on which combination yields better results on the surrogate model before being tested on the objective function. As more configurations are tested, the more accurate the surrogate model becomes in predicting the objective function's outcome. Algorithms based on this framework differ in how the surrogate model is computed.

The SMBO algorithm chosen for hyperparameter search is the Tree-structured Parzen Estimator (TPE) [37]. It consists of using a Bayesian method to estimate the probability distribution of the objective function's outcome y given a hyperparameter x: $p(y|x)$ and sampling the most likely to yield a greater y. The implementation used is the one provided as a Search Algorithm in Ray Tune's library (*tune.suggest.hyperopt.HyperOptSearch*). It works by defining a hyperparameter space, discrete or continuous, to sample trials from, along with an initial configuration. Additionally, to avoid using computation time on unpromising trials, a median pruner Trial Scheduler (*tune.schedulers.MedianStoppingRule*) is used to terminate trials early if their performance at a given evaluation step is below the median.

3.3.5 Performance Metrics

To compare the performance of algorithms and assess possible effects of different parameters on the experiment results, each algorithm was evaluated using a set of metrics. The metrics used for evaluating the performance aim to capture different aspects of an algorithm and are categorized into 2 classes:

- **Reinforcement Learning metrics**.
- **Task specific metrics**.

They are also differentiated by the moment when they are computed: **during** or **after** training. The measurements required to compute these metrics were obtained in evaluation episodes with randomized seeds. Evaluation episodes were run on the current model during training at fixed rate of time steps (e.g., every 10k-20k steps) or on the final model after training.

Measurements performed at the episode level are averaged across all evaluation episodes. Given N evaluation episodes and a respective measurement X_n for $n \in [1, N]$, the *Average* $\overline{X}$ is defined as:

$$\overline{X} = \frac{1}{N} \sum_{n} X_n \tag{3.12}$$

To better describe the measurement's distribution, an *Average* is always presented together with the *Standard Deviation* σ_X of the measurement across the evaluation episodes, defined as:

$$\sigma_X = \sqrt{\frac{\sum_n (X_n - \overline{X})^2}{N}} \tag{3.13}$$

The *Average* and *Standard Deviation* are computed using the *Numpy* library[10] functions: *numpy.mean* and *numpy.std*, respectively.

Reinforcement Learning metrics

These are the typical RL metrics used to measure the performance of an algorithm and its resulting model.

[10]Scientific computing library for the Python programming language (`https://numpy.org/doc/stable/user/whatisnumpy.html`)

- **Return** - the sum of collected rewards $R_n = \sum_t r_t$ along time steps t of an episode. The primary goal of any RL agent is to maximize this cumulative reward, which maybe discounted with time. The assumption is: maximizing the cumulative reward is expected to coincide with learning the optimal policy for a particular task.
- **Episode Length** - the total number of time steps in an episode. Depending on the task, an agent's performance maybe correlated with this value. In the case of the reaching task, a longer episode indicates the agent is able to avoid collision and a shorter one might indicate the agent reaches a target faster or quickly collides.

Task specific metrics

The RL metrics measure performance only relative to a tasks' MDP definition. In order to further evaluate an agent on the reaching task the following metrics are used:

- **Success Rate** - proportion of episodes with a success outcome: having the end effector position within the defined threshold distance from the target position. Given N evaluation episodes, the success rate is defined as $(\#success\_outcomes)/N$.
- **Final Distance** - Euclidean distance in *meters* between the end effector position P_{ee} and target position P_{target} at the end of an episode, defined as $||P_{ee} - P_{target}||$. Used mostly for error analysis where two agents might fail to reach a target but with vastly different distances.
- **Elapsed Time** - simulation time (not real time) measured between start and end of an episode, reported in *milliseconds*.

In order to show the reliability of the *Success Rate* estimate, it is reported with a *Confidence Interval* calculated for a confidence level of 95% ($\alpha = 0.05$). Each episode can be interpreted as Bernoulli trial, hence this interval is a binomial confidence interval. The interval is calculated with the *Clopper–Pearson* method utilizing the *scipy.stats.beta.ppf* function from the *SciPy*[11] Python library.

[11] Library of algorithms and mathematical tools for the Python programming language (`https://scipy.org/scipylib/index.html`)

Chapter 4

Experiments and Results

This chapter presents a set of experiments (regarding the reaching tasks described in Section 3.3), the procedures used to conduct them and the results obtained. The experiments consist of training and evaluating DRL agents in different scenarios and are organized into three main parts: (i) solving a reaching task with lower maximum joint velocities (Section 4.1); (ii) assessing effects of hyperparameters on performance (Section 4.2); and (iii) solving a reaching task in the context of ball catching with maximum joint velocities (Section 4.3). The state vectors used in these reaching tasks are distinguished by the encoding of the target's position (as mentioned in Section 3.3.1). The target position as spherical coordinates in the base reference frame is referred to as **base state** and in the end effector reference frame is referred to as **end effector state**. The complete set of conducted experiments are:

- Comparison between TD3 and PPO on a reaching task with the **base state** (Section 4.1.1).
- Comparison between TD3 and PPO on a reaching task with the **end effector state** (Section 4.1.2).
- Correlation of the state with reward along episodes on a reaching task with the **end effector state** (Section 4.1.3).
- Attempt at hyperparameter optimization for both TD3 and PPO on a reaching task with the **base state** (Section 4.2.1).
- Assessing the effect of the *Replay Buffer* size for TD3 on a reaching task with the **base state** (Section 4.2.2).
- Assessing the effect of the *Horizon* (n_steps) for PPO on a reaching task with the **base state** (Section 4.2.3).
- Assessing the effect of different *action cycle rates* in achieving better performance on a reaching task with the maximum joint velocity ($max\_velocity\_scale\_factor = 1$) and the **end effector state** (Section 4.3.1).
- Assessing the effect of different *reward systems* in achieving better performance on a reaching task with the maximum joint velocity ($max\_velocity\_scale\_factor = 1$) and the **end effector state** (Section 4.3.2).

- Application of the best agent to a robot ball catching task with the maximum joint velocity ($max\_velocity\_scale\_factor = 1$) and the **end effector state** (Section 4.3.3)

For all results, the graphs plotting the *Average Return* and *Average Time Steps* were done using a moving average with a window of 10 on the results obtained in the evaluation episodes. The shaded areas represent $+/-$ half of the standard deviation ($\sigma/2$). The evaluation episodes done on agents **after** training are done with a $real\_time\_factor \approx 1$. The training and evaluation of agents were done on a machine with the following specifications: Ubuntu 18.04.5 OS; 4 Core Intel i5-7500 @ 3.800GHz CPU; 8GB of Random Access Memory (RAM); NVIDIA GeForce GTX 1050 Ti GPU. The versions of the software tools used are: Gym 0.18.3; robo-gym 1.0.0; ROS Melodic; Gazebo 9.0.0; PyTorch 1.8.1; Stable-baselines3 1.0; Ray 1.4.0.

4.1 Solving the Reaching Task

This section presents the experiments conducted in order to solve a reaching task specified by the *UR10Reach* Gym environment (detailed in Section 3.3.1) with slow maximum joint velocities ($max\_velocity\_scale\_factor = 0.2$).

4.1.1 Reaching Task with the Base State

An initial experiment was conducted as an attempt to obtain early results, using the set of hyperparameters in Table 3.5 and Table 3.6. Both algorithms were set to train for 2 million time steps on the *UR10Reach* environment using the base state and only 1 worker (not running multiple environments in parallel). The models were evaluated every 20k time steps, on 10 episodes with random seeds to obtain the *Average Episode Return* and *Average Episode Time Steps*, totaling 100 evaluation moments during training. As a baseline for comparison a random agent, randomly sampling actions from the action space, was set to run the same number of evaluations. Figures 4.1 and 4.2 plot these metrics evolving during the model's training. In addition, every 100K time steps the current model was saved to be used afterwards to plot the proportions of episode outcomes across time steps by evaluating each saved model on 100 episodes with random seeds, as presented in Figure 4.3 and Figure 4.4.

The training time was approximately 14 hours for TD3 and 12 hours for PPO. The lower training time for PPO is mostly due to it taking more time steps (2048 steps) between network updates as compared to TD3 (100 steps). Only TD3 achieved some type of convergence by the end of training. Both TD3 and PPO begun with a biased model: TD3's initial model ends most episodes with a collision outcome either from colliding against the ground or itself; on the other hand, PPO's initial model ends most episodes exceeding the maximum number of episode steps. As training progresses, TD3 learns to avoid collision shown both by the declining proportion of collision outcomes as well by the increase of *Average Time Steps*. Near the 1 million time steps the number of *Average Time Steps* begins to decline faster and the proportion of successes increases leading to the eventual convergence.

Within the first 100K time steps PPO learns a policy consisting of high collision rate which persists, possibly due to its on-policy nature, for approximately 1.1 million time steps leading to worse performance and returns than the random agent. Afterwards, the *Average Return* begins to increase along with the *Average Time Steps* and the decrease of collision rate. However, PPO ends training without achieving the same level of *Average Return* or Success Rate as TD3. Tables 4.1 and 4.2 present a summary of RL and task specific metrics

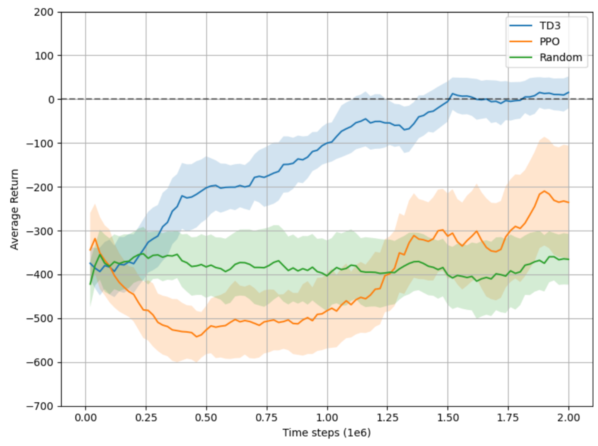


Figure 4.1: Average Return of evaluation episodes during training for TD3 and PPO.

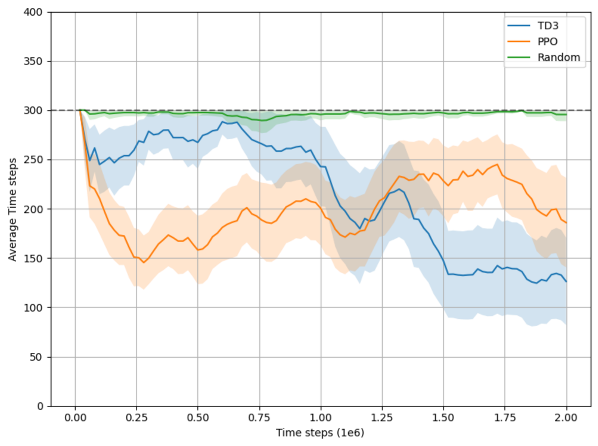


Figure 4.2: Average Time Steps of evaluation episodes during training for TD3 and PPO.

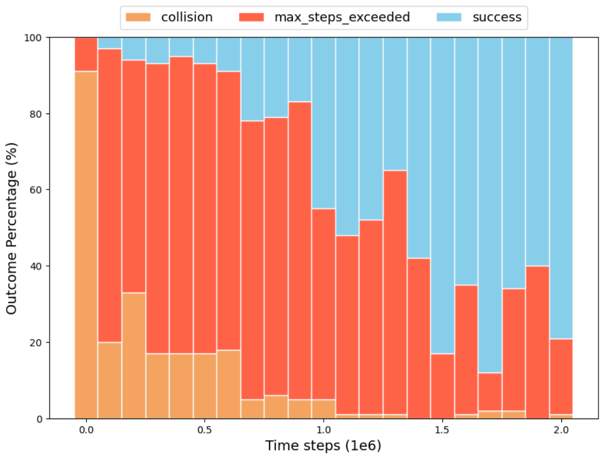


Figure 4.3: Proportion of episode outcomes during training of TD3 (base state).

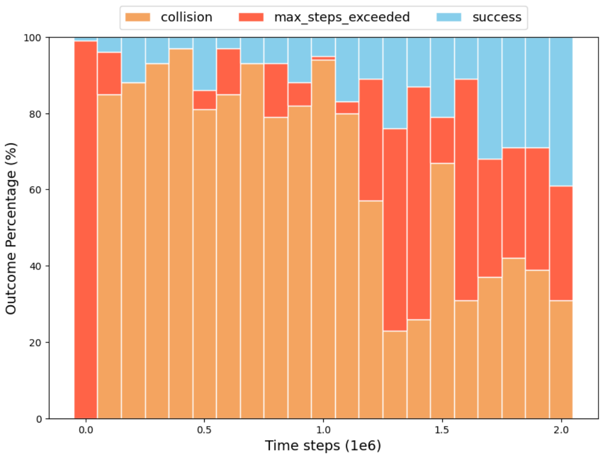


Figure 4.4: Proportion of episode outcomes during training of PPO (base state).

---

obtained from evaluating the final models on 100 episodes with random seeds. All metrics were computed with respect to successful episodes (except the Success Rate, *SR*) in case of Table 4.1 and with respect to failed episodes in case of Table 4.2.

Agent	SR	SR CI	R	FD	TS	ET
Random	0.02	(0.00,0.07)	78.34±13.74	0.10±0.00	81.00±36.00	5930.52±2642.51
PPO	0.38	(0.28,0.48)	43.90±30.06	0.09±0.01	113.61±43.00	8529.55±3259.74
TD3	0.84	(0.75,0.91)	52.66±26.28	0.09±0.01	82.26±33.12	6082.73±2483.54

Agent - Agent model; *SR* - Success Rate; *SR CI* - Success Rate 95% Confidence Interval; *R* - Mean Return; *FD* - Mean Final Distance to target (m); *TS* - Mean Time Steps; *ET* - Mean Elapsed Simulation Time (ms);

Table 4.1: Metrics of episodes with **successful outcomes** for the base state

Both TD3 and PPO significantly outperform the Random agent with TD3 achieving the highest Success Rate and fastest policy on successful episodes. Additionally, TD3 is faster than the PPO agent resulting in greater Average Return. For episodes with failed outcomes, the Random agent mostly ends due to exceeding the maximum number of steps and with greatest distance to the target; TD3 accumulates the highest return and fails to reach the target's threshold by only 0.15 meters on average; and PPO failed mostly due to collision.

Agent	CR	MSR	R	FD	TS	ET
Random	0.04	0.96	-381.33±142.74	1.38±0.50	299.27±4.42	22123.33±335.09
PPO	0.68	0.32	-387.74±151.28	0.59±0.24	204.74±94.28	15503.91±7222.85
TD3	0.44	0.56	-256.59±194.19	0.25±0.16	204.62±108.28	15309.88±8160.48

Agent - Agent model; *CR* - Collision Rate; *MSR* - Max Steps exceeded Rate; *R* - Mean Return; *FD* - Mean Final Distance to target (m); *TS* - Mean Time Steps; *ET* - Mean Elapsed Simulation Time (ms);

Table 4.2: Metrics of episodes with **failure outcomes** for the base state

4.1.2 Reaching Task with the End Effector State

In the *UR10Reach* environment the end effector frame is available as a translation and rotation relative to the base frame. Consequently, to assess if having the target defined relative to the end effector would have an effect on training and performance of the algorithms, the previous experiments in Section 4.1.1 were replicated now using the **end effector state**. The resulting *Average Return* and *Average Time Steps* graphs are shown in Figures 4.5 and 4.6.

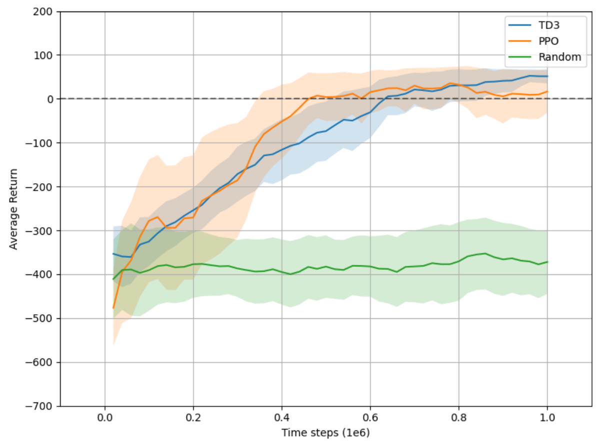


Figure 4.5: Average Return of evaluation episodes during training for TD3 and PPO.

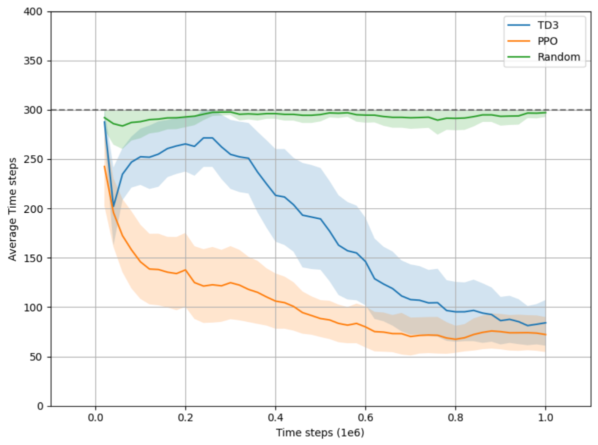


Figure 4.6: Average Time Steps of evaluation episodes during training for TD3 and PPO.

Compared to the previous experiments both algorithms converged early, hence the agents' training was stopped at 1 million time steps. The training time was approximately 7.2 hours for TD3 and 6.5 hours for PPO. The plots were done with new performance samples for the Random agent. Similarly to the **base state** experiment, TD3 quickly learns to avoid collision, as it can be seen in Figure 4.6 by the number of steps approaching the maximum more often in the beginning and in Figure 4.7 where the proportion of collision outcomes drop significantly after the 100K time steps. PPO on the other hand, starts with an initial policy which often exceeds the maximum number of steps and then begins to learns a policy with high collision rate and finally converges in successful policy. Despite converging faster, agents learning on this state present learning patterns similar to the **base state** experiments, as portrayed in outcome proportion graphs in Figures 4.7 and 4.8. However, in the Average Return graph the PPO agent has a distinct evolution as its performance is above the Random agent more often than the previous experiment. Figure 4.6 illustrates well how PPO differs from TD3 by not first learning to increase *max_steps_exceeded* outcomes in order to avoid collisions and, instead, going directly from collision to success outcomes.

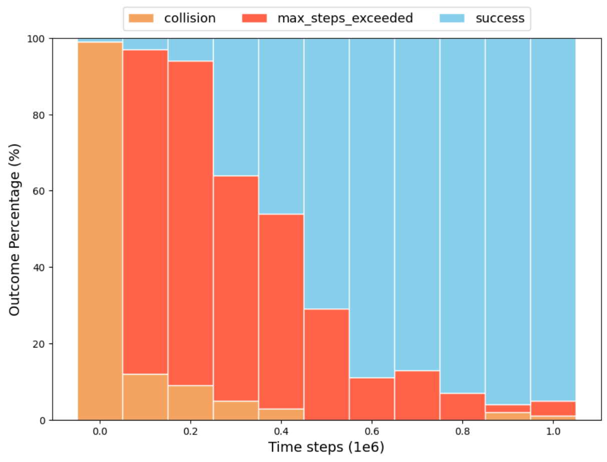


Figure 4.7: Proportion of episode outcomes during training of TD3 (end effector state).

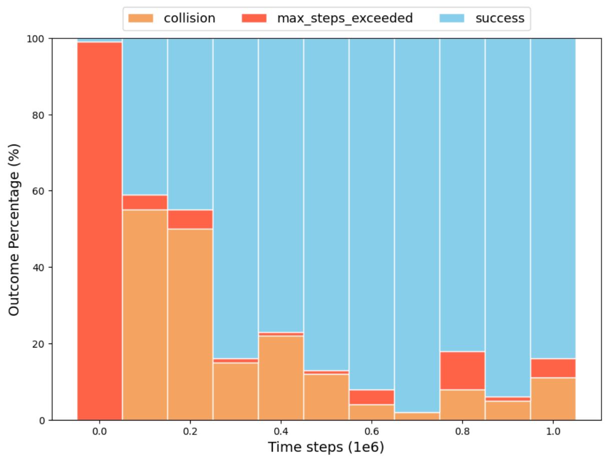


Figure 4.8: Proportion of episode outcomes during training of PPO (end effector state).

Since a different state representation will not affect the Random agent's performance, the results reported in Tables 4.3 and 4.4 are compared against the same previous results of the Random Agent in this reaching task. For these results, the final models were evaluated with the same previous procedure of 100 episodes with random seeds.

Agent	SR	SR CI	R	FD	TS	ET
Random	0.02	(0.00,0.07)	78.34±13.74	0.10±0.00	81.00±36.00	5930.52±2642.51
PPO	0.85	(0.76,0.91)	62.63±28.16	0.09±0.01	61.04±33.86	4584.21±2611.22
TD3	0.97	(0.91,0.99)	59.95±26.89	0.09±0.01	69.47±30.84	5168.26±2336.49

Agent - Agent model; *SR* - Success Rate; *SR CI* - Success Rate 95% Confidence Interval; *R* - Mean Return; *FD* - Mean Final Distance to target (m); *TS* - Mean Time Steps; *ET* - Mean Elapsed Simulation Time (ms);

Table 4.3: Metrics of episodes with **successful outcomes** for the end effector state

Again, TD3 and PPO significantly outperform the Random agent with TD3 achieving the highest Success Rate. Compared to the previous experiment, PPO more than doubled the success rate. Additionally, PPO is now the fastest policy on successful episodes and both TD3 and PPO see a decrease in *Mean Time Steps* and *Mean Elapse Simulation Time* on the evaluation episodes with PPO's having the most significant decrease by 52.57 time steps and 3945.33 milliseconds respectively. For episodes with failed outcomes, the TD3 agent ends only due to exceeding the maximum number of steps and with smallest distance to the target. TD3 accumulates the highest return and fails to reach the target's threshold by 0.24 meters on average. Unlike TD3, PPO has a closely balanced distribution of failed outcomes. Due to the high success rate these results for failed outcomes have fewer samples and might not fully describe the resulting policies for TD3 and PPO in failing episodes.

Agent	CR	MSR	R	FD	TS	ET
Random	0.04	0.96	-381.33±142.74	1.38±0.50	299.27±4.42	22123.33±335.09
PPO	0.47	0.53	-448.99±199.07	1.03±0.43	282.80±20.04	21737.74±1506.20
TD3	0.00	1.00	-109.07±23.34	0.34±0.09	300.00±0.00	22878.33±114.73

Agent - Agent model; *CR* - Collision Rate; *MSR* - Max Steps exceeded Rate; *R* - Mean Return; *FD* - Mean Final Distance to target (m); *TS* - Mean Time Steps; *ET* - Mean Elapsed Simulation Time (ms);

Table 4.4: Metrics of episodes with **failure outcomes** for the end effector state

4.1.3 Correlation of End Effector State with Reward

The results in Section 4.1.2 prompted the following question: why would training with the **end effector state** be faster in converging and produce better performing policies than training with the **base state**? Although not initially clear, the hypothesis that this state vector has a greater correlation with the reward is explored in this section. To that end, the PPO agent trained in the previous section with the **end effector state**, two of its intermediate models (100K and 500K time steps models) and a random agent were set to run 300 evaluation episodes with the same random seeds, during which the reward and state vector at each time step were recorded. Afterwards, for each agent and episode the Pearson correlation coefficient between the state vector and the reward was calculated using Numpy's *numpy.corrcoef* function.

Figure 4.9 shows the correlation coefficient between the state vector's elements and the reward averaged for all evaluation episodes. Here the random agent has the greatest negative correlation with the reward on the r component of the target's coordinates, which is expected given that this is the distance to the target: a smaller distance increases the reward and the chance of being within the threshold for success. For the remaining agents this r component also has highest correlation among the state vector although not to the same degree of the random agent. Some other elements display some correlation, however this might be specific to the learned policy.

To better visualize the correlation between the r component and the reward, it was plot in Figure 4.10 for the 300 episodes using the final PPO agent (trained on 1000K time steps) together with the correlation between random samples of r and the reward as a reference for comparison. The random samples of the r component (computed by randomly sampling between the observed minimum and maximum across all episodes) shows no visible correlation across episode given it is centered around 0. On the other hand, the observed r component shows some negative correlation by having it fluctuate around ≈ -0.25 for most episodes. Occasionally, there are peaks of correlation very close to -1 which indicates episodes ending without collision or success and instead exceed the maximum number of steps that returns no reward penalty or bonus.

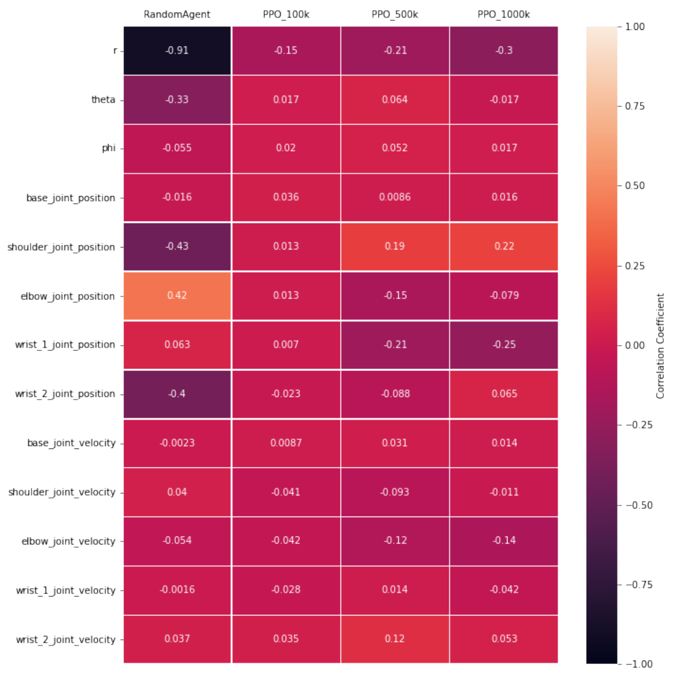


Figure 4.9: Mean Correlation between state vector and reward during an episode (considering last step) for multiple agents run on 300 random seed episodes.

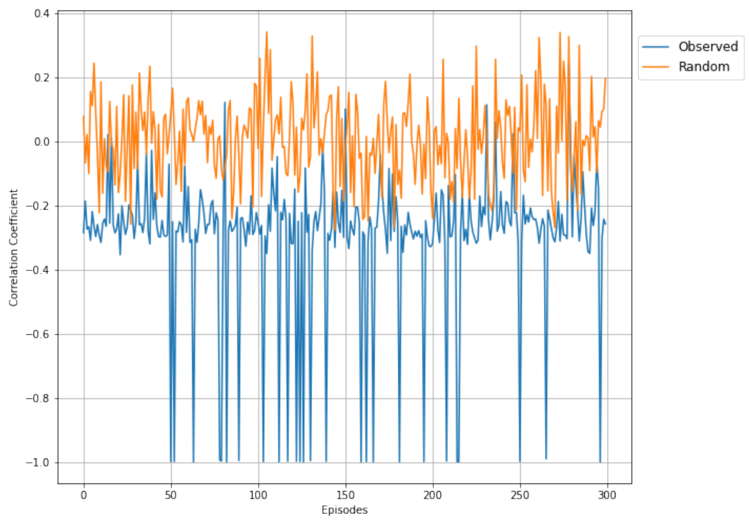


Figure 4.10: Correlation Coefficient between episode rewards and state vector's r component during an episode (considering the last step) for 300 random seed episodes.

Computing the correlation and plotting the previous graphs without the last time step, where a reward penalty or bonus is provided, results in a clearer negative correlation as depicted in Figures 4.11 and 4.12. Here the negative correlation of ≈ -1 is constant across episodes and models, which is obvious when considering the reward is based on distance to the target. More interestingly, for PPO model at 500K time steps and 1000K time steps the *shoulder_joint_position* and *wrist1_joint_position* components of the state have high positive and negative correlation respectively, which could describe the policy learned where the first wrist joint is extended when reaching for targets.

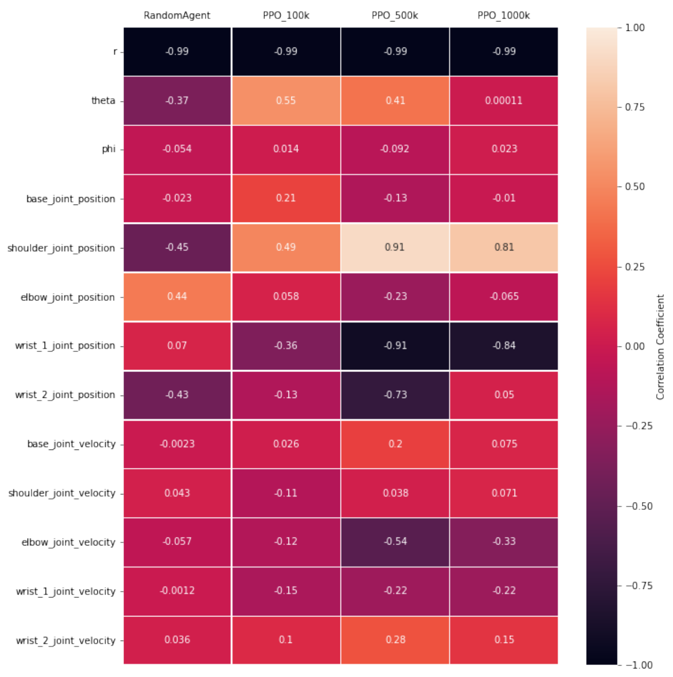


Figure 4.11: Mean Correlation between state vector and reward during an episode (**ignoring** last step) for multiple agents run on 300 random seed episodes.

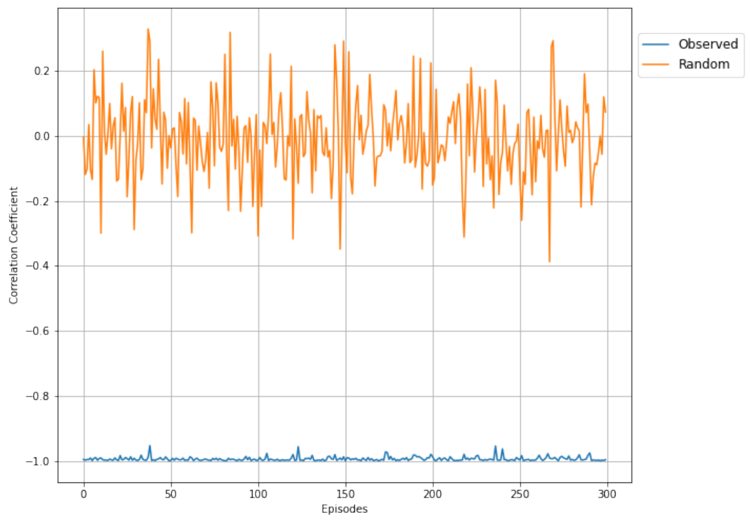


Figure 4.12: Correlation Coefficient between episode rewards and state vector's r component during an episode (**ignoring** the last step) for 300 random seed episodes.

4.2 Hyperparameters

This section presents the approach used to improve the agents' performance, by hyperparameter tuning, on the *UR10Reach* environment with the **base state**. The environment with the **base state** is considerably harder to learn compared to the one with the **end effector state** given that both TD3 and PPO had worst results in this setting when training with the same configurations. This is possibly due to the **base state** requiring the agent to learn a more complex transformation of the state in order to estimate its value. Conversely, the **end effector state**, as shown in Section 4.1.3, has a strong correlation with the reward making it easier to predict a state's value. This is well illustrated in the case of the PPO agent trained using the **base state**, where the drop in performance is dramatic compared to the **end effector state** version. The goal is to improve performance with better hyperparameter configurations without requiring more environment samples. In order to fairly compare the algorithms in this setting, for each of them it was decided to firstly attempt a systematic hyperparameter search and then a more focused evaluation of one hyperparameter related with sample efficiency.

4.2.1 Hyperparameter Search

The algorithm used for hyperparameter search is Ray Tune's implementation of TPE. It was chosen in place of a grid search or random search approach due to it performing an informed search over the hyperparameter space (as detailed in Section 3.3.4) which is appropriate for the low computational budget available. For each algorithm, the search consisted of 25 trials with an initial parameter suggestion using the hyperparameters in Tables 3.5 and 3.6. These are parameters known to work and help the search algorithm make better suggestions for future parameters. Each trial runs sequentially for 300K time steps instead of the full training length of 2 million. This was an assumption made for this experiment which consists of: 300K time steps is enough for an algorithm to make measurable progress in learning (the previous results support this) and the best hyperparameters at 300K time steps is expected to also be the best at 2 million time steps. This decision is justified again by the limited computational resources available.

The hyperparameter search for TD3 was done over the following parameters:

- learning_rate: {0.003, 0.001, 0.0006, 0.0003, 0.0001, 0.00003}
- batch_size: {32, 64, 128, 256}
- tau: {0.001, 0.005, 0.01, 0.02, 0.05}
- gamma: {0.99, 0.995, 0.999}
- train_freq: {50, 100, 150}
- gradient_steps: {50, 100, 200}
- noise_std: {0.0, 0.1, 0.2, 0.3, 0.4}

The hyperparameter search for PPO was done over the following parameters:

- learning_rate: {0.003, 0.001, 0.0006, 0.0003, 0.0001, 0.00003}

- n_steps: {256, 512, 1024, 2048}
- batch_size: {32, 64, 128, 256}
- n_epochs: {5, 10, 20}
- gamma: {0.99, 0.995, 0.999}
- gae_lambda: {0.95, 0.98}
- clip_range: {0.1, 0.2, 0.3}
- max_grad_norm: {0.3, 0.5, 0.7}

For the duration of each trial there are a total of 20 evaluation moments (every 15K time steps). An evaluation is the average return resulting from running 20 episodes with random seeds on the current model's deterministic policy. The results of these evaluations are used together with a *median pruner* Trial Scheduler which compares the current's trial evaluation results at a particular evaluation step with the median of previous trials and terminates if it is bellow. However, this only happens after a period of 7 evaluations (105K time steps) in order to avoid ending possibly good trials which perform bad initially due to random factors such as exploration.

Although the weight of the initialization seed (for the environment and SB3) in an algorithm's results is not known, in order to mitigate any noise this might introduce in the attempt to find the best hyperparameters each trial is run on the same random seed. Additionally, to prevent any type of overfitting the seeds used for the evaluation episodes are randomly chosen but, again to reduce noise introduced by an agent getting a set of "easier" episodes, it is the same across all trials.

The search time was approximately 55 hours for TD3 and 47 hours for PPO. The resulting configurations for the best trials were then trained for 2 million time steps. Figures 4.13 and 4.14 show the *Average Return* and *Average Time Steps* of evaluation episodes during training.

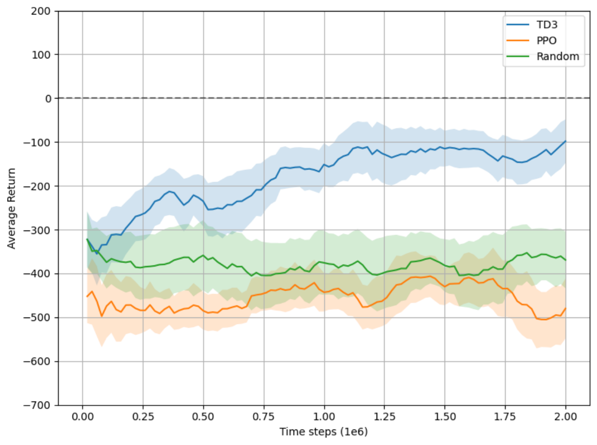


Figure 4.13: Average Return of evaluation episodes during training of best configurations for TD3 and PPO.

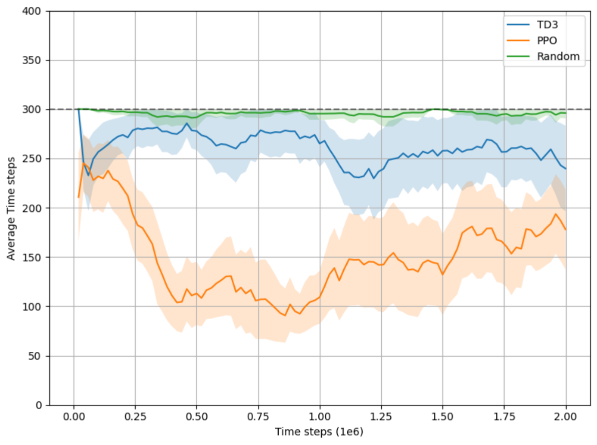


Figure 4.14: Average Time Steps of evaluation episodes during training of best configurations for TD3 and PPO.

The resulting models were then evaluated like previously on 100 episodes, the results are summarized for all episodes in Table 4.5. This hyperparameter search for both algorithms proved to not be an effective method to improve performance and given it is very time consuming it was not pursued further.

Agent	SR	SR CI	R	FD	TS	ET
Random	0.02	(0.00,0.07)	-372.13±155.28	1.36±0.53	294.90±31.29	21799.47 ±2321.41
PPO	0.02	(0.00,0.07)	-514.91±123.63	1.59±0.49	127.49±85.00	9609.67 ±6491.35
TD3	0.46	(0.36,0.56)	-69.37±100.80	0.24±0.18	211.12±100.29	15772.56 ±7546.11

Agent - Agent model; *SR* - Success Rate; *SR CI* - Success Rate 95% Confidence Interval; *R* - Mean Return; *FD* - Mean Final Distance to target (m); *TS* - Mean Time Steps; *ET* - Mean Elapsed Simulation Time (ms);

Table 4.5: Metrics of evaluation episodes for the base state

4.2.2 TD3: Replay Buffer

The replay buffer in TD3 works just like the ones in DQN and DDPG. It stores the last agent-environment interactions and is used for experience replay to achieve stability. Every fixed number of episodes or steps a batch of experiences is randomly sampled from the replay buffer and used to minimize the loss function. This is a way of casting the problem of RL closely into supervised learning. Similarly to this, the amount data in the replay buffer and its quality will influence learning. The replay buffer is of a fixed size, less than the number of time steps to train on, where memories are inserted and removed with a *first-in-first-out* method. Choosing this size defines how old is the oldest memory. A small replay buffer will have only the most recent memories, which towards the end of training is made of experience of a converging policy but initially might lead to some type of overfitting. On the other hand, large replay buffers will contain older memories of more off-policy data due to older more random policies and will take longer to be refreshened with good trajectories. In [38], the authors provide an analysis of experience replay and observe, for a set of environments, that performance improves when trained on data from more recent policies.

In order to improve the performance of TD3 on the *UR10Reach* environment with the **base state**, three agents were trained with the same previous configuration, but varying the replay buffer sizes: 250K, 500K and 2000K (Figures 4.15 and 4.16). The respective train times were approximately 14.9 hours, 15.2 hours and 14.7 hours. These agents were evaluated on 100 episodes with random seeds and compared against the one in Section 4.1.1 trained with a replay buffer size of 1 million (TD3_1000K). The results are summarized for all evaluation episodes in Table 4.6. The agent with the best Success Rate is the one trained with a 500K replay buffer, improving by 4% in relation to TD3_1000K. The smallest and biggest sizes performed significantly worst.

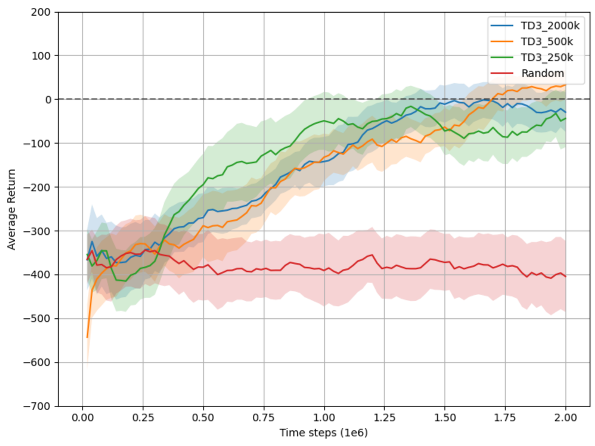


Figure 4.15: Average Return of evaluation episodes for agents trained with different replay buffer sizes.

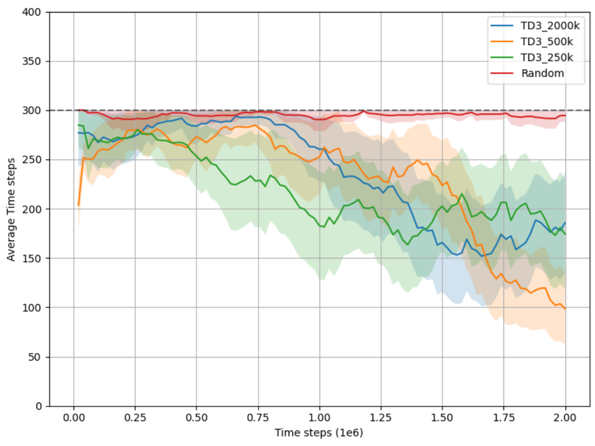


Figure 4.16: Average Time Steps of evaluation episodes for agents trained with different replay buffer sizes.

Agent	SR	SR CI	R	FD	TS	ET
TD3_250K	0.59	(0.49,0.69)	-33.05±105.95	0.25±0.26	177.14±106.72	13225.90 ±8045.87
TD3_500K	0.88	(0.80,0.94)	23.61±112.06	0.11±0.10	89.14±70.61	6620.00 ±5320.45
TD3_1000K	0.84	(0.75,0.91)	3.18±139.52	0.12±0.09	101.84±69.35	7559.07 ±5222.92
TD3_2000K	0.55	(0.45,0.65)	-34.29±108.98	0.22±0.18	179.43±113.15	13405.39 ±8524.14

Agent - Agent model; *SR* - Success Rate; *SR CI* - Success Rate 95% Confidence Interval; *R* - Mean Return; *FD* - Mean Final Distance to target (m); *TS* - Mean Time Steps; *ET* - Mean Elapsed Simulation Time (ms);

Table 4.6: Metrics of evaluation episodes for TD3 agents trained with different replay buffer sizes.

4.2.3 PPO: Horizon

The *horizon* in PPO, referred to *n_steps* in SB3's implementation, is a learning hyperparameter controlling the number of steps an agent runs on the environment before updating its policy network. This has a similar function to the replay buffer in the case of TD3, which is to accumulate a rollout of trajectories to use for optimizing a loss function. However, for PPO these trajectories must be obtained in on-policy way and are discarded after each policy network update. A smaller horizon adds more noise when learning, possibly leading to overfitting. This is more problematic for on-policy algorithms since this overfitting will affect the policy used on the next rollout of trajectories, resulting in an accumulative effect and great drops in performance. In contrast, longer horizons reduce overfitting, but make learning less sample efficient by having to run for longer before updates. Often, PPO is trained with multiple parallel agents running the current version of a policy to add experiences in a bigger rollout buffer where the total now is *horizon* × *number_of_agents*.

To assess the effect of the *horizon* on PPO's performance on the *UR10Reach* environment with **base state**, three agents were trained with the same previous configuration but varying the *horizon*: 512, 1024 and 4096 (Figures 4.17 and 4.18) with respective times of approximately 14.8 hours, 13.84 hours and 12.95 hours. The agents were then evaluated on 100 episodes with random seeds and compared against the agent trained with 2048 in Section 4.1.1. The results summarized in Table 4.7 show that no improvement was made. The observed drop in performance from changing this hyperparameter is significant, suggesting this is already well tuned for the given configuration.

Agent	SR	SR CI	R	FD	TS	ET
PPO_512	0.02	(0.00,0.07)	-326.53±126.01	1.06±0.48	297.74±9.49	22657.26 ±754.44
PPO_1024	0.05	(0.02,0.11)	-343.76±134.31	0.99±0.47	269.53±72.81	20466.09 ±5565.04
PPO_2048	0.38	(0.28,0.48)	-223.72±241.72	0.40±0.30	170.11±90.39	12853.65 ±6916.86
PPO_4096	0.09	(0.04,0.16)	-419.26±237.62	0.81±0.35	239.84±84.18	18215.28 ±6431.41

Agent - Agent model; *SR* - Success Rate; *SR CI* - Success Rate 95% Confidence Interval; *R* - Mean Return; *FD* - Mean Final Distance to target (m); *TS* - Mean Time Steps; *ET* - Mean Elapsed Simulation Time (ms);

Table 4.7: Metrics of evaluation episodes for PPO agents trained with different values for the *horizon*.

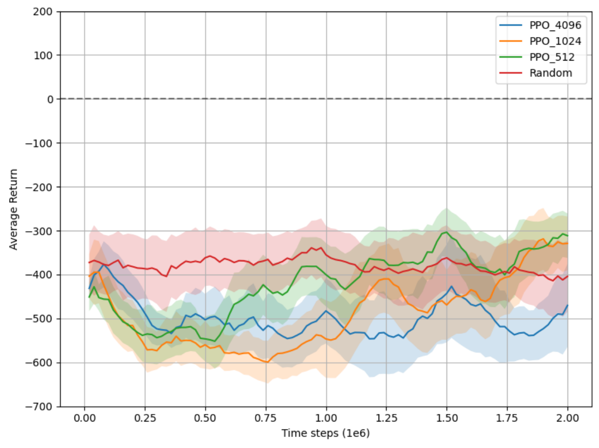


Figure 4.17: Average Return of evaluation episodes for PPO agents trained with different values for the *horizon*.

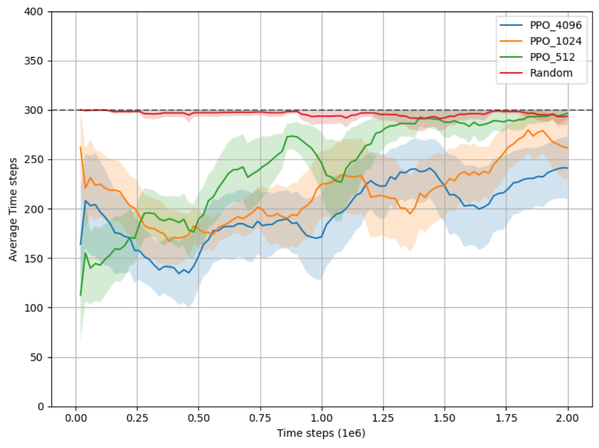


Figure 4.18: Average Time Steps of evaluation episodes for PPO agents trained with different values for the *horizon*.

4.3 Assessing Applicability to Robot Ball Catching

This section presents the experiments conducted in order to solve a ball catching task specified by the *UR10BallCatch* Gym environment (detailed in Section 3.3.1) with the maximum joint velocities. As mentioned before, this problem is defined in a way the agent is only tasked with learning to quickly reach a target where the ball intercepts a plane. The **end effector** state and the algorithm TD3 are used since this was the best performing combination obtained from the previous experiments. The approach consists, firstly, in tuning the *action cycle rate* to the faster joint velocities and given success threshold. Afterwards, agents are trained on different reward systems. The one with the best performance on the **train** mode is applied to a complete scenario, along side a polynomial trajectory approximation method on the **evaluation** mode.

4.3.1 Action Cycle Rate

In order to maintain a given level of precision for greater maximum joint velocities it is necessary to increase the *action cycle rate*. However, this increase may create a harder environment by requiring an agent to learn to gain momentum by repeating similar actions, as explained in [26]. This section presents the experiments conducted for tuning the ACR to a task with faster maximum joint velocities.

Initially, an experiment was made where the TD3 model trained on the *UR10Reach* environment with the **end effector state**, $max\_velocity\_scale\_factor = 0.2$ and $ACR = 12.5$ (from section 4.1.2) was run on the same environment for 100 evaluation random seed episodes with $max\_velocity\_scale\_factor = 1$ for the following set of ACRs: 12.5 (original ACR), 20, 30, 40, 50, 62.5 (5× original ACR). The goal is to observe if a model trained at lower maximum joint velocities could be transferred to faster speeds by simply adjusting the ACR. The resulting success rates for the evaluation episodes are depicted in Figure 4.19.

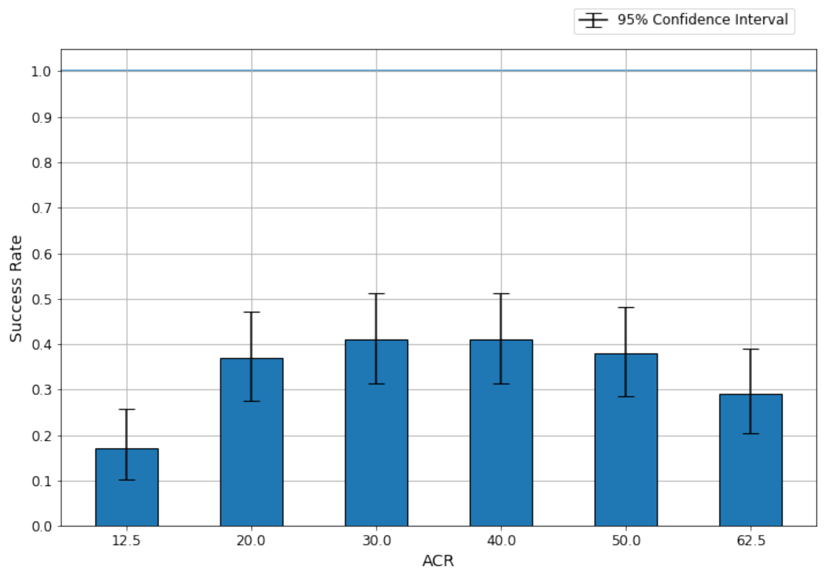


Figure 4.19: Success rates on *UR10Reach* with maximum joint velocities for TD3 model trained on 12.5 ACR and slow joint velocities run on different ACRs.

All scenarios show a great drop in performance compared to the 97% Success Rate of this same model in its original slower environment (the best performance is 41% by ACR of 30 and 40). This demonstrates that the information of the dynamics and/or ranges of the joint velocities are valuable to the model and adjusting ACR alone does not result in good transfer to faster maximum joint velocities. One other important observation is that the Success Rate initially increases with the ACR, achieves a peak and eventually decreases, as suggested previously.

These results informed that indeed the ACR has a role in the precision, but the joint velocities and its dynamics specific to an environment are also important in learning. In a further attempt to choose an appropriate ACR for fast joint velocities, four TD3 agents were trained sequentially, with the same configurations as the one of Section 4.1.2, in a simple reaching task specified by *UR10ReachEval* environment (detailed in Section 3.3.1). This reduced version of the previous reaching task was used to have the agent's performance depend less on the complexity of the task and more on the variable evaluated, the ACR. The agents were trained for 2 million time steps on the same first four ACRs as the previous experiment. The *Average Return* and *Average Time Steps* graphs are plotted in Figures 4.20 and 4.21 respectively, with the random agent baseline run on ACR of 12.5.

The trained agents were then evaluated on 100 random seed episodes and the results of this are summarized using the Success Rate in Figure 4.22. In this figure a similar pattern to the previous experiment can be observed, an increase followed by a decrease in Success Rate for greater ACRs. The best agent was the one trained for $ACR = 30$. For the maximum TCP speed of 1 m/s the end effector will move $0.033m$ in between actions which is less than the success threshold. Given this result, values of $ACR > 40$ were not tested.

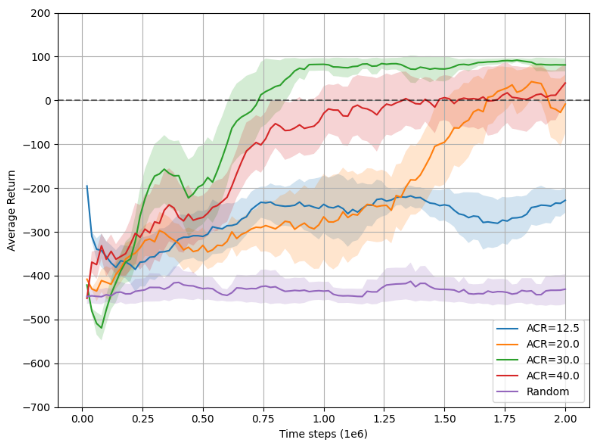


Figure 4.20: Average Return of evaluation episodes for agents trained at different ACRs.

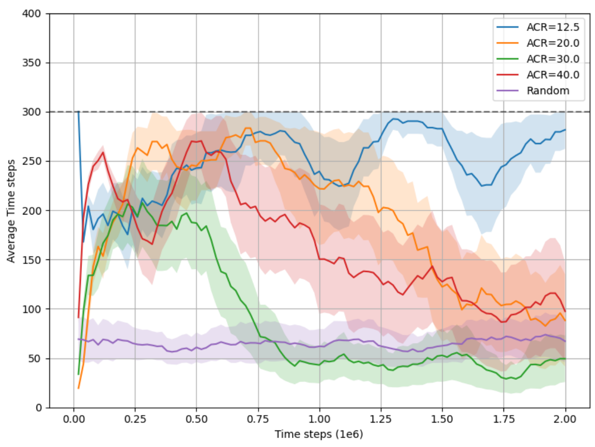


Figure 4.21: Average Time Steps of evaluation episodes for agents trained at different ACRs.

4.3.2 Different Reward Systems

In previous results the reward definition for the *UR10Reach* environment proved to be effective for slow maximum joint velocities and a threshold distance of 0.1 meters. For the ball catching scenario addressed here, the Success Rate is not the primary measure of performance, the time taken to reach the target is also important. With the goal of improving performance

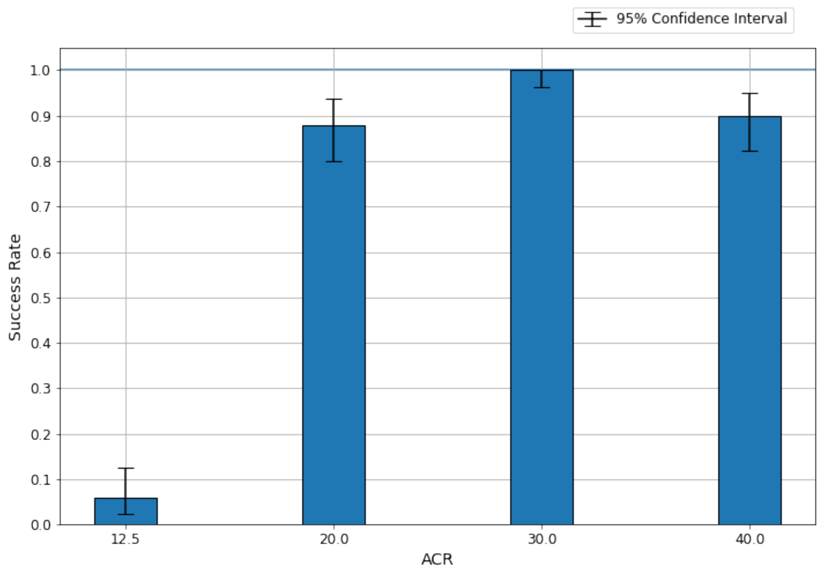


Figure 4.22: Success rates of TD3 agents trained with different ACRs

---

in **train** mode of the *UR10BallCatch* environment, five agents were trained and evaluated for different reward systems: normal, joint penalty, log, log w/o threshold, joint penalty and log. The *Average Return* graphs are plot separately and are not compared based on absolute values due to the different rewards systems. Additionally, the baseline random agents were run for each reward system. The results of evaluating the trained agents are presented and described in the final subsection. All agents were trained for a *real_time_factor* of ≈ 3.

Normal reward

The normal reward is the original described in Section 3.3.1, consists of a base reward computed from the distance between the end effector and the target to which a bonus or penalty is added depending on success or collision. As a baseline for comparison an agent was trained in this environment with the normal reward for 2 million time steps ($\approx$13.6 hours). The resulting graphs plotting the *Average Return* and *Average Time Steps* during training are shown in Figures 4.23 and 4.24.

Joint penalty reward

When reaching a target, the manipulator's joints change position over time. Depending on the change in each joint, the movement of the manipulator could be smooth (e.g., a straight line from end effector to a target) or jerky (increasing reach time). With the normal reward system, an agent receives a penalty, for every step, linearly proportional to the negative of the distance, incentivizing smooth movements. However, this may not be explicit enough: if the end effector reaches the threshold for success, the received bonus is much greater than optimizing for smooth movement, hence an agent may achieve high success with a jerky policy.

This joint penalty reward system is a crude attempt at explicitly reinforcing smooth movements. It is defined as the sum of the delta between the current joint positions θ normalized between $[-1, 1]$ and the action of the agent a, at a given time step (Equation

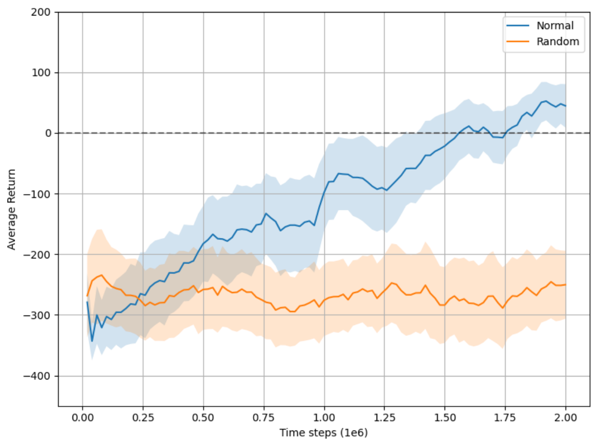


Figure 4.23: Average Return of evaluation episodes for TD3 agent trained with the normal reward.

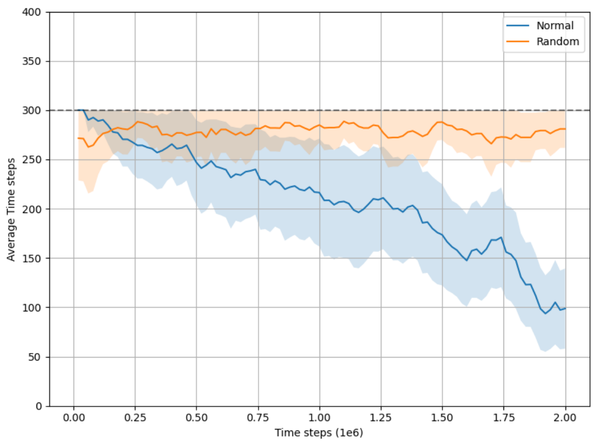


Figure 4.24: Average Time Steps of evaluation episodes for TD3 agent trained with the normal reward.

4.1). This penalty value is scaled by a coefficient and subtracted from the normal reward. The penalty coefficient is a hyperparameter that can be tuned. A coefficient too big will be unproductive by reinforcing no movement and too small will have no effect on jerky movement.

The *Average Return* and *Average Time Steps* graphs resulting from training a TD3 agent for 2 million time steps (≈14.6 hours) on this reward system with the penalty coefficient = 1 are plotted in Figures 4.25 and 4.26.

$$\begin{aligned} joint\_penalty &= \sum_{joint} |\theta_{joint} - a_{joint}| \\ reward_{new} &= reward - penalty\_coef \times joint\_penalty \end{aligned} \tag{4.1}$$

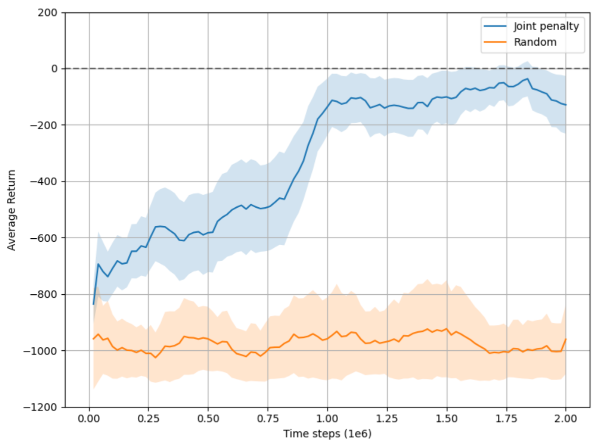


Figure 4.25: Average Return of evaluation episodes for TD3 agent trained with the joint penalty reward.

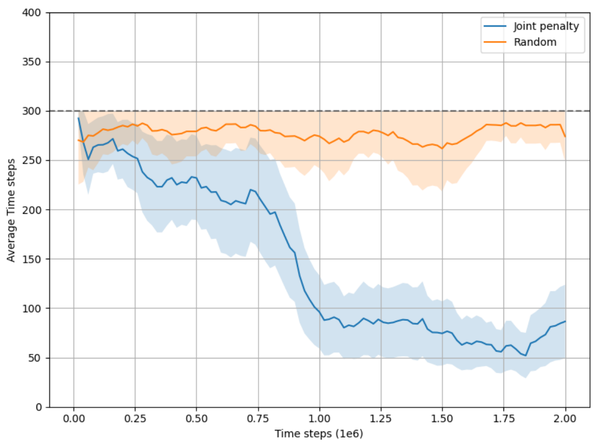


Figure 4.26: Average Time Steps of evaluation episodes for TD3 agent trained with the joint penalty reward.

Log reward

The base for the normal reward is linearly proportional to the negative of the distance between the target and end effector. A reward system with a logarithmic (natural logarithm) base value is used here to assess if faster convergence and improved performance could be obtained within the time step limit. This log reward gives increasingly more reward (or less penalty) for a change in distance the closer this distance approaches 0 as compared to the normal reward (depicted in Figure 4.27). The base for the log reward defined in Equation 4.2 has 3 constants: *log_coef* which is parameterizable and used to choose curve's the steepness and height; the 0.1 value offsets the curve in x axis so the reward doesn't go to infinity; the 1 value offsets the curve in y axis so its start is closer to the linear base of the normal reward.

$$reward_{base} = -log\_coef \times log(||target\_coord - ee\_coord|| + 0.1) + 1 \tag{4.2}$$

A TD3 agent was trained for 2 million time steps (≈14.0 hours) with log_coef = 1. The resulting *Average Return* and *Average Time Steps* graphs are plotted in Figures 4.28 and 4.29.

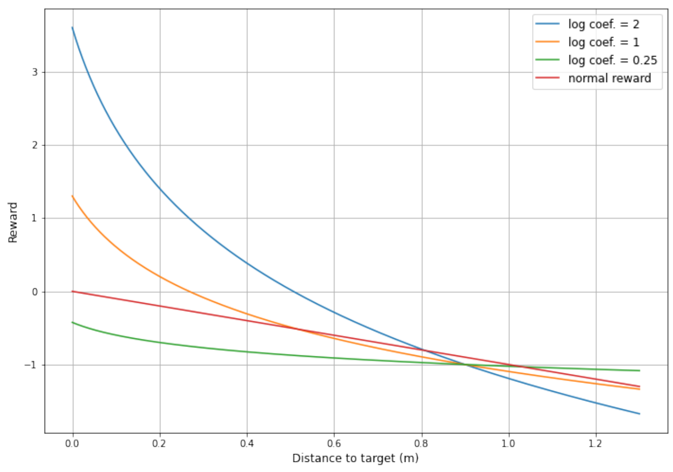


Figure 4.27: Log base reward for different coefficient values compared against linear base reward.

Log reward without success threshold

The previous log reward base is positive starting from a distance smaller than $1/e - 0.1 \approx 0.26788$ meters. This together with removing the success threshold and its reward bonus was intended to reinforce precision and smooth motion when optimizing for accumulated reward by having to place the end effector at the target position for longer. Without a success threshold an episode may extend to its fullest duration if the manipulator does not collide. A TD3 agent was trained in this reward system for 2 million time steps (≈12.9 hours). The resulting *Average Return* and *Average Time Steps* graphs are in Figures 4.30 and 4.31.

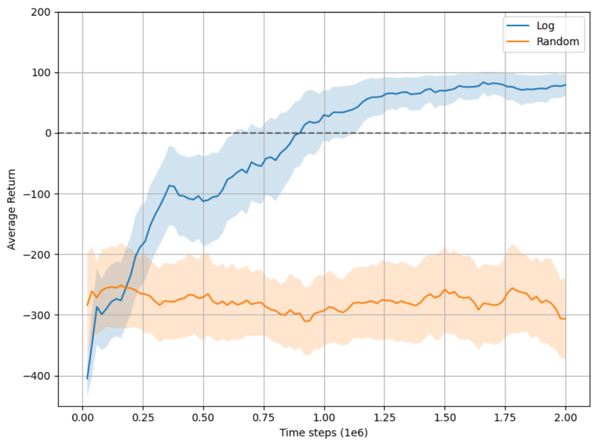


Figure 4.28: Average Return of evaluation episodes for TD3 agent trained with the log reward.

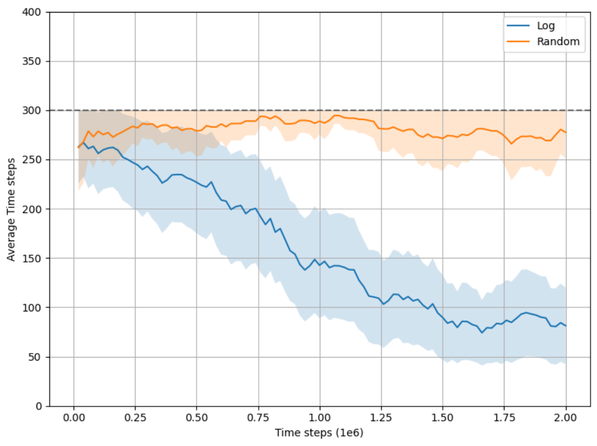


Figure 4.29: Average Time Steps of evaluation episodes for TD3 agent trained with the log reward.

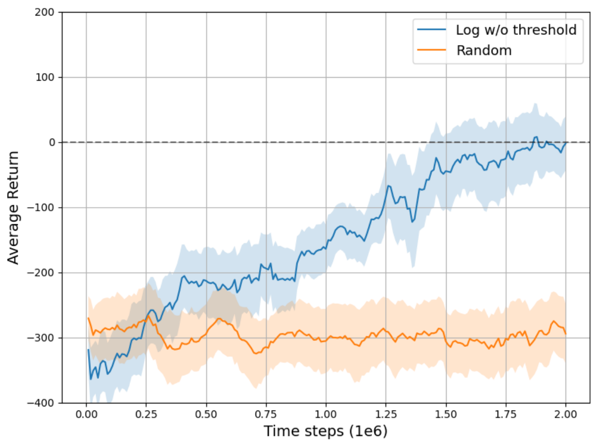


Figure 4.30: Average Return of evaluation episodes for TD3 agent trained with the log without success threshold reward.

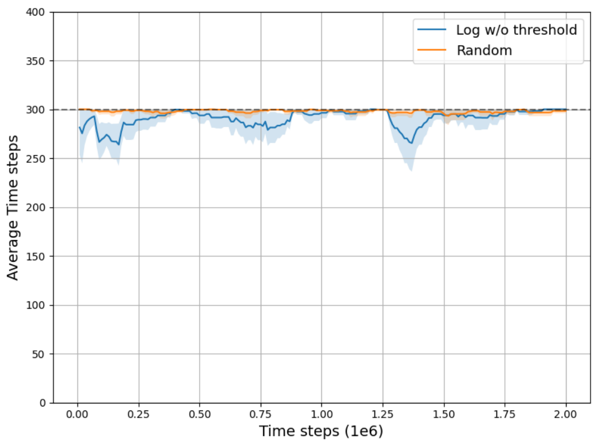


Figure 4.31: Average Time Steps of evaluation episodes for TD3 agent trained with the log without success threshold reward.

Joint penalty and log reward

A TD3 agent was trained for 2 million time steps (≈13.7 hours) in a reward system consisting of the combination of the previous log base reward (Section 4.3.2) and joint penalty (Section 4.3.2). The resulting *Average Return* and *Average Time Steps* graphs are in Figures 4.32 and 4.33.

Summarized results

The trained agents were each evaluated in a total of 200 random seed episodes in the *UR10BallCatch* environment using the respective reward system they were trained. The exception is the *log w/o thr* agent that was evaluated on the *log* reward system in order to

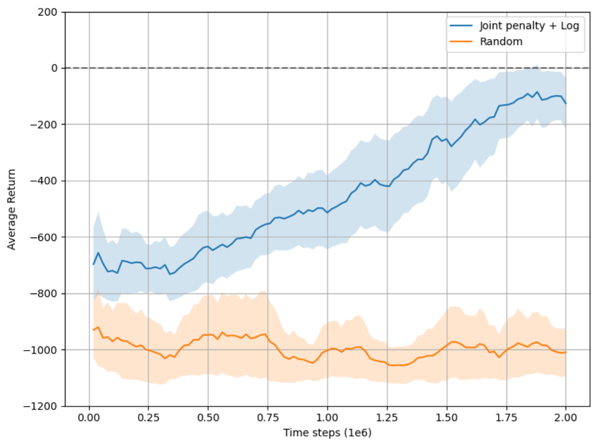


Figure 4.32: Average Return of evaluation episodes for TD3 agent trained with the joint penalty and log reward.

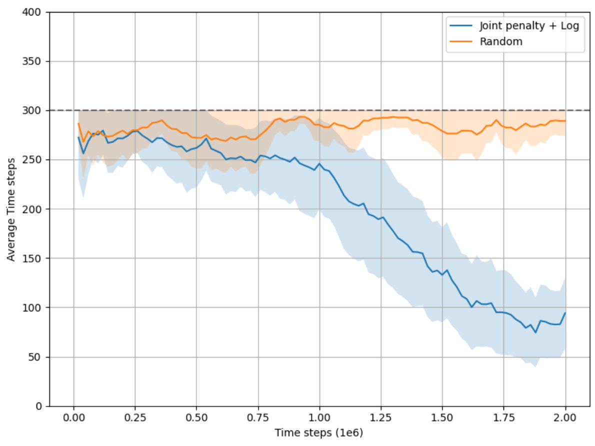


Figure 4.33: Average Time Steps of evaluation episodes for TD3 agent trained with the joint penalty and log reward.

compute the success rate. Initially, only 100 evaluation episodes were run. However, after observing very similar performance in the *Success Rate* metric between the *normal*, *joint*, and the *log* reward agents (having a maximum difference of 1%), 100 additional episodes were run with new random seeds, resulting in no significant change but smaller confidence intervals.

The results for successful episodes, excluding the *Mean Return* metric, are listed in Table 4.8. The best agent was the one trained with the *log* reward, but only improving the *Success Rate* by 1% relative to the *normal* reward. All new rewards systems lowered (improved) the *Mean Time Steps* and *Mean Elapsed Simulation Time*. These metrics experienced a minimum decrease, relative to the *normal* reward, of 10.84 time steps and 304.42 ms respectively.

Agent	SR	SR CI	FD	TS	ET
normal	0.985	(0.957,0.997)	0.08±0.02	60.55±51.02	1626.27±1432.46
joint	0.985	(0.957,0.997)	0.08±0.02	49.71±45.41	1307.56±1212.43
log	0.995	(0.972,1.000)	0.08±0.02	49.66±41.14	1319.66±1129.19
log wo thr	0.725	(0.658,0.786)	0.08±0.01	47.94±43.99	1321.85±1261.17
joint + log	0.925	(0.879,0.957)	0.08±0.02	48.73±42.32	1296.23±1171.46

Agent - Agent model; *SR* - Success Rate; *SR CI* - Success Rate 95% Confidence Interval; *FD* - Mean Final Distance to target (m); *TS* - Mean Time Steps; *ET* - Mean Elapsed Simulation Time (ms);

Table 4.8: Metrics of successful evaluation episodes with a **threshold distance of 0.1 meters** for agents trained with different reward systems.

To better distinguish the most similarly performing agents, 200 additional evaluation episodes were run for a success threshold distance of = 0.05 meters. The results are listed in Table 4.9. Here, all agents experience a drop in *Success Rate*, increase in *Mean Time Steps* and increase in *Mean Elapsed Simulation Time*. This was expected given they were not trained on this threshold, suggesting that the end effector was staying close to the target position for

Agent	SR	SR CI	FD	TS	ET
normal	0.650	(0.580,0.716)	0.04±0.01	95.88±71.33	2589.09±1977.85
joint	0.880	(0.827,0.922)	0.04±0.01	80.77±66.34	2163.90±1801.42
log	0.765	(0.700,0.822)	0.04±0.01	96.93±73.01	2620.16±2005.28
log wo thr	0.525	(0.453,0.596)	0.04±0.01	85.18±70.65	2395.63±2035.22
joint + log	0.750	(0.684,0.808)	0.04±0.01	74.41±47.98	2002.79±1328.65

Agent - Agent model; *SR* - Success Rate; *SR CI* - Success Rate 95% Confidence Interval; *FD* - Mean Final Distance to target (m); *TS* - Mean Time Steps; *ET* - Mean Elapsed Simulation Time (ms);

Table 4.9: Metrics of successful evaluation episodes with a **threshold distance of 0.05 meters** for agents trained with different reward systems.

longer before crossing the threshold. The *joint* reward agent experience the smallest drop in *Success Rate*. The remaining have a drop of at least 17.5% and the *normal* reward agent of 33.5%. Hence, the best performing agent was the one trained with the *joint penalty reward*.

4.3.3 Demo

This section presents the results of applying the best agent trained in the *UR10BallCatch*'s **train** mode on its **evaluation** mode. This mode runs randomly chosen ball trajectories which start at a height of 1 meter, land within reach of the manipulator and have a flight time between $[1.0, 1.2]$ seconds before crossing $z = 1$ meters. The estimated catch position is then used in the state for the agent to reach. The details of these trajectories and the **evaluation** mode are in Section 3.3.1. Before running the agent in **evaluation** mode, it was run on 300 episodes, with targets evenly spaced in a grid discretizing the catching plane, to get an estimate of the learned policy. As depicted in Figure 4.34, 298 targets were successfully reached, but only 154 (51%) were under the maximum limit of the trajectories' flight time of 1.2 seconds. The remaining targets are the ones at the extremity of the workspace.

In order to visualize the movement of the end effector during an episode in **evaluation** mode, the best agent was run on an episode whose intercept point can be seen in Figure 4.35). The trajectory has a total flight time of approximately 1.33 seconds (1.16 seconds before intercepting the catching plane). Figure 4.36 shows the Cartesian coordinates of the end effector, the flying ball and the estimated catching position (obtained from the *trajectory_prediction* ROS node) changing during this episode. It is shown only the final 0.97 seconds of the episode since the agent has to wait (at the start) for a number of ball position samples to act. The number of samples is a parameter that can be adjusted and has a trade-off between better initial catching position estimate and a longer wait time to start moving.

For the purposes of this demonstration 30 initial samples were used. The end effector did not move in a perfectly straight line towards the catching position, but it was still able to reach it in time and be within the required distance to the ball for success. Additionally, the velocities in rad/s for each controlled joint was measured and is plotted in Figure 4.37. In the first 0.4 seconds the base joint stays at its velocity limit, resulting in the fast displacement along the positive Y axis seen in Figure 4.36, while the elbow joint stays mostly static. The base and shoulder joints seem to be responsible for the majority of the movement, given they reach their maximum velocity limits more often then remaining joints.

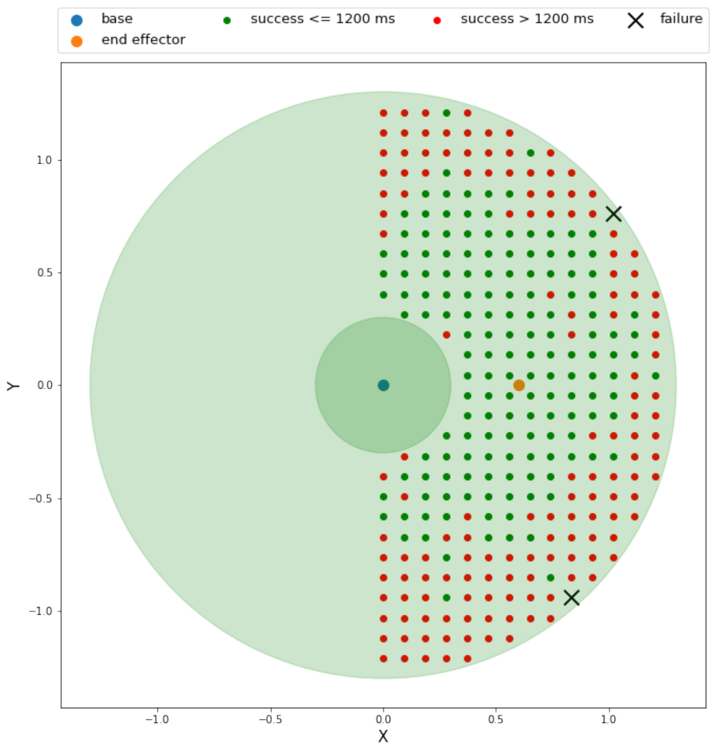


Figure 4.34: Evaluation grid results

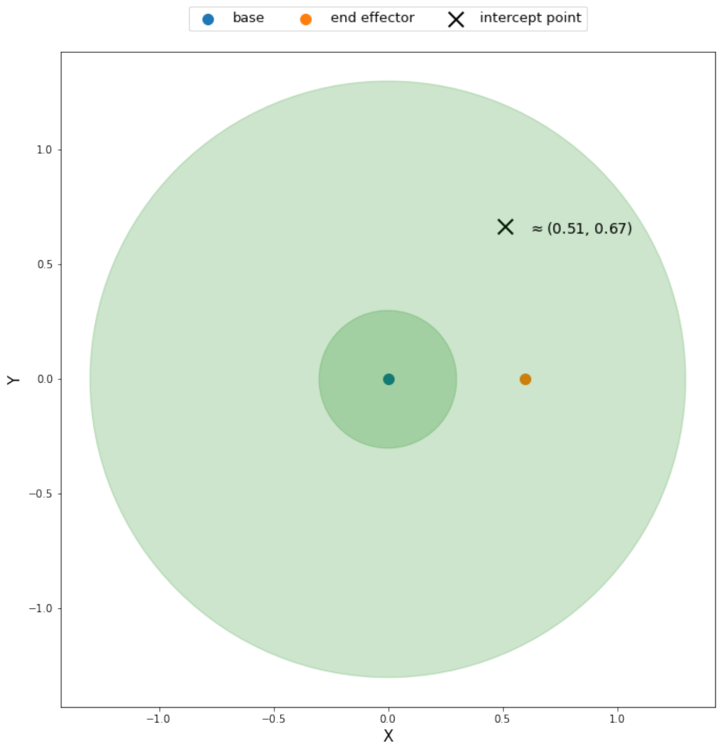


Figure 4.35: Intercept point of trajectory_6220 with catching plane

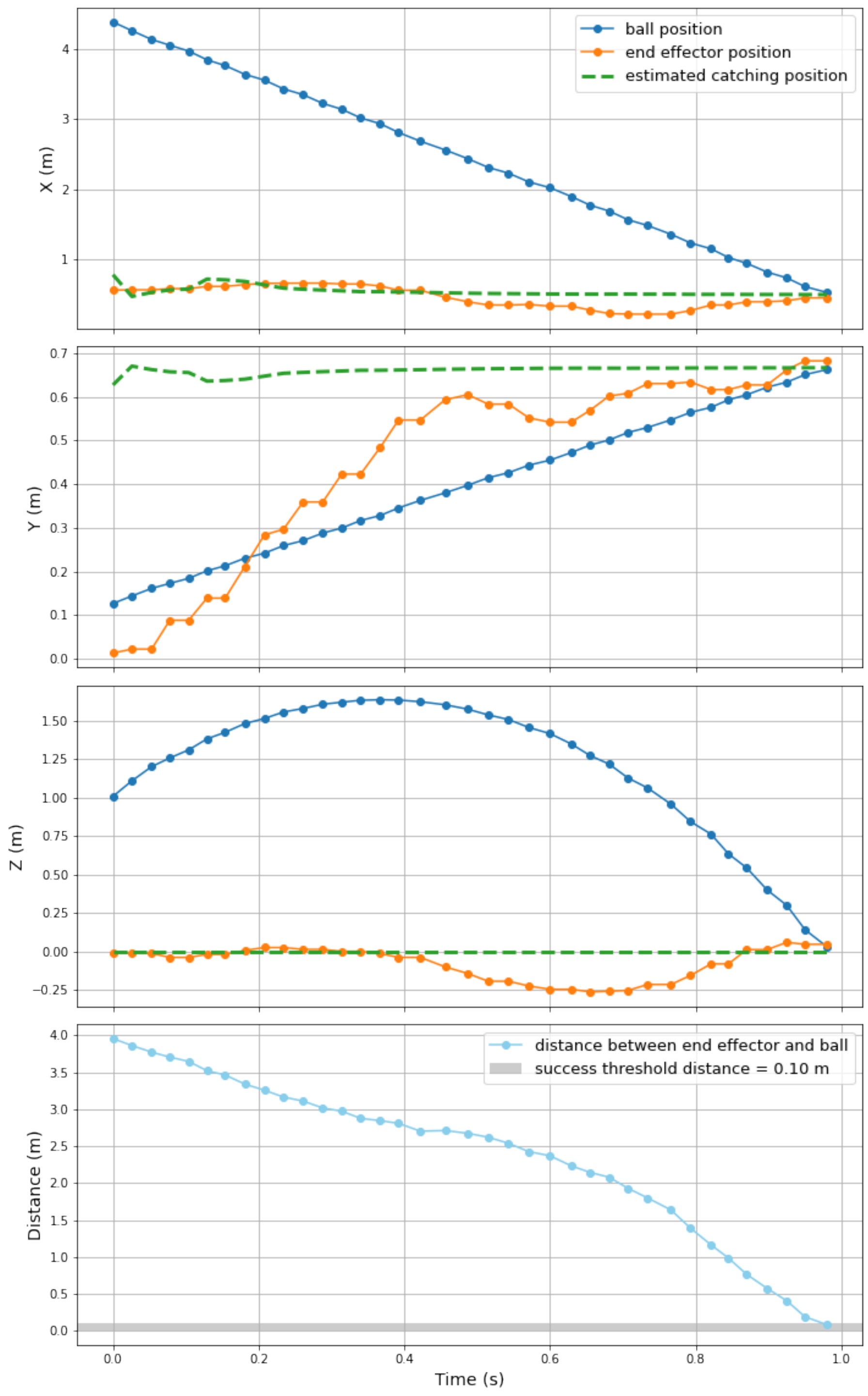


Figure 4.36: Target and End Effector trajectory on a successful attempt

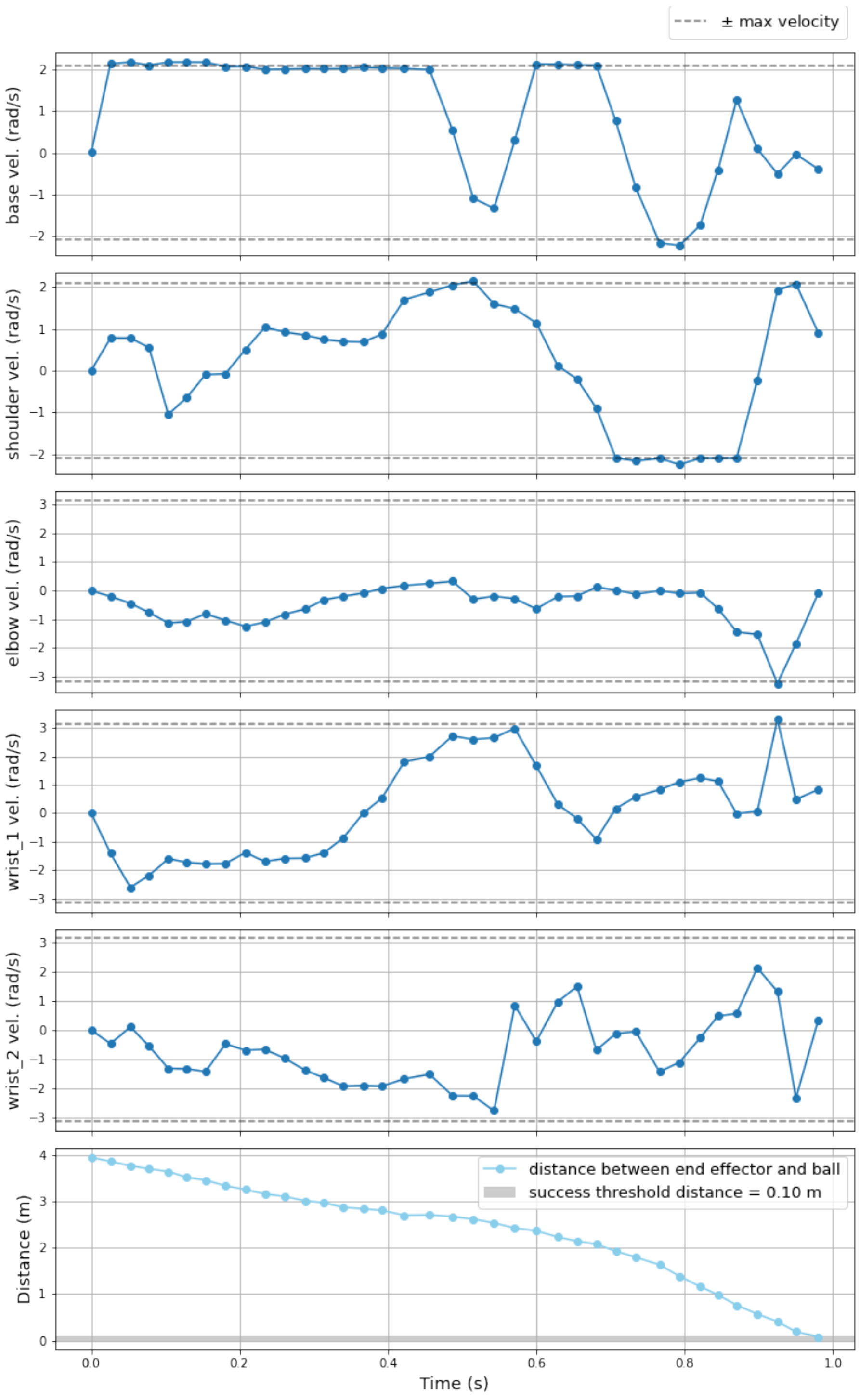


Figure 4.37: Joint velocities on a successful attempt

4.4 Final Remarks

In this chapter, a set of DRL agents were trained and evaluated in different reaching tasks. Initially, these tasks were defined with 1/5th of the maximum joint velocities, thus requiring less tuning of the robo-gym's MDP parameters (e.g., ACR). The **end effector state** proved to be the best solution for encoding the target, resulting in faster convergence and better policies. However, this comes at a cost: while in this simulated environment the end effector reference frame is easily available, in real world applications this would require specialized sensors or performing forward kinematics. Thus, despite the **base state** being harder to learn, an agent using this state solves the full problem by not requiring a kinematic model of the robot and would be more applicable to a real world scenario.

Often, in DRL research, algorithms are benchmarked in simple, computationally cheap environments. This offers the opportunity to use hyperparameter search in great volumes to obtain improvements. In this work, the hyperparameter search performed was constrained by the available computational resources, in part due to the CPU dependent nature of Gazebo. Hence, the number of configuration trials and the time steps per trial executed during the search may possibly have been smaller than necessary to find a better set of hyperparameters. On the other hand, the more focused approach on some parameters yielded better policies. It also highlighted key differences in the off-policy and on-policy algorithms used, specifically, in the case of learning with the **base state**. TD3 was observed to be more reliable and sample efficient than PPO, due to being off-policy and leveraging the replay buffer.

Tuning the ACR and handling the trade off between precision and complexity of the environment was shown to be essential in obtaining better performance for greater joint velocities. Additionally, the reward system using a penalty for actions too far from the current joint positions was observed to be a useful approach in decreasing reach times and increasing success rate for smaller thresholds.

The ball catching demonstration using a reaching agent was able to meet the speed requirements in reaching the targets in its workspace. Nevertheless, for targets it could reach in time in the train mode, in evaluation mode it would often reach inside the threshold and move away from the catching position before the ball being close to the end effector. This shows a need for the agent to stabilize at the catching position, which can be achieved by additional reward shaping or stopping the joints' motion when inside the threshold.

Chapter 5

Conclusion and Future Work

The main objective of this dissertation was to apply Deep Reinforcement Learning to the domain of robotic manipulation tasks and evaluate its effectiveness in obtaining successful policies. To this end, the concrete class of manipulation tasks addressed in this work were robot reaching.

Initially, a set of classic Reinforcement Learning problems of varying complexity were solved and used as a practical approach to obtain a solid background in DRL. This consisted of implementing agents to solve: Gridworld using tabular Q-learning; Gridworld with a tensor state representation using Deep Q-learning and PyTorch; and the pendulum swing-up continuous control task using DDPG and PyTorch.

Afterwards, a number of OpenAI Gym environments for the different versions of the reaching task were implemented based on the robo-gym toolkit. This allowed the simulation of the manipulator, that has a backend in ROS and Gazebo, to have an MDP abstraction for these tasks. This follows the Gym's standard interface supported by the Stable Baselines3 DRL library, which provides implementations of state-of-the-art DRL algorithms.

In simulation, the UR10 collaborative robot was used together with two DRL algorithms (TD3 and PPO), representative of approaches using off-policy and on-policy learning, to train agents capable of successfully controlling the manipulator's joints to solve two versions of a reaching task at slower speeds (*UR10Reach*). This resulted in an agent capable of replacing the function of methods using trajectory planning and inverse kinematics. Additionally, it was observed that defining the target in the end effector reference frame produced faster convergence and better resulting policies.

A systematic approach was then employed to further improve performance of the DRL agents and an assessment of learning hyperparameters was made. Here, an improvement was verified for TD3. The PPO algorithm demonstrated to perform worst in the harder version of the *UR10Reach* when run with a single actor collecting experiences. This is most likely due to the lower sample efficiency of PPO and the unstable aspects of on-policy learning. Because of this, most work using this algorithm uses multiple parallel actors (e.g., 8,16,32) to collect more experience by combining rollouts, which requires much greater computational resources than the ones used for this dissertation.

Subsequently, the TD3 algorithm was chosen to solve a faster version of the reaching task (*UR10BallCatch*) because of its better performance in previous experiments. This required tuning the *action cycle rate*, a parameter that is part of the MDP definition for robotic tasks in robo-gym, for the faster maximum joint velocities. Greater ACRs are necessary to achieve high precision for a given threshold. However, beyond an ACR value, performance was observed to decrease with the increase of the ACR. As explained in [21], RL algorithms are extremely sensitive to the frequency of taking actions and the performance goes to zero as the frequency of taking actions goes to infinity. This is solved in [12] by using a technique known as *frame skipping*: repeating the previous agent's actions on the environment for a number of states in between every new action. This was not used in this work since the *robot-actuation cycle time* was equal to *action cycle time* $(1/ACR)$, but it would be interesting to attempt shorter *robot-actuation cycle times* which would increase the rate at which actions are sent to the joints controller.

With an appropriate ACR for this faster scenario, the focus was turned to increase performance of agents in the *UR10BallCatch* environment with regards to the success rate and the time to reach a target. A set of different reward systems were tested which resulted in significantly lower reach times and greater success rates for a smaller success threshold. Finally, the agent with the best performance was applied to a ball catching scenario where a polynomial approximation method is used to estimate the catching position for a flying ball.

For future work, different lines of research could be explored. For example, in this dissertation only 5 out of the UR10's 6 joints are controlled, since no tool is used, but for other tasks the orientation of end effector with an object could be used as a reward. For goal oriented tasks, sparse rewards where the agent only receives a reward when achieving a goal may take longer to converge, but can offer the agent more freedom to explore different strategies which would be penalized in hand-engineered rewards. In the context of reaching, further improvements could be obtained from redefining the task and optimizing for precision or achieving a specific pose. It would also be interesting to transfer the learned policies from simulation to a physical system, which is feasible given the environment's implementation using ROS, and to use pixels from a camera as a state together with Convolutional Neural Network (CNN) architectures. Additionally, other classes of manipulations tasks could be attempted, for example: reaching with obstacles avoidance, full ball catching, stacking, assembly, pushing and inserting. For tasks requiring additional sequential context, such as the ball catching problem, enhancing the state to include multiple samples of time changing properties of an environment could be applied to better encode these dynamics. In supervised learning contexts, the Recurrent Neural Network (RNN) [39] architecture is used for problems with sequential inputs. Consequently, using RNNs to represent the policy when applying DRL to these tasks could also be beneficial.

References

[1] Richard S. Sutton and Andrew G. Barto. The agent-environment interaction in reinforcement learning. `http://www.incompleteideas.net/book/first/ebook/node28.html`, 1998.

[2] OpenAI. Part 2: Kinds of rl algorithms. `https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html#a-taxonomy-of-rl-algorithms`, 2018.

[3] Wikimedia Commons. Illustration of the topology of a generic artificial neural network (ann). `https://commons.wikimedia.org/wiki/File:Artificial_neural_network.svg`, 2020.

[4] Visak Kumar, David Hoeller, Balakumar Sundaralingam, Jonathan Tremblay, and Stan Birchfield. Joint space control via deep reinforcement learning. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3619–3626, 2021.

[5] Matteo Lucchi, Friedemann Zindler, Stephan Mühlbacher-Karrer, and Horst Pichler. robo-gym–an open source toolkit for distributed deep reinforcement learning on real and simulated robots. *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020.

[6] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[7] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, second edition, 2018.

[8] Yuxi Li. Deep reinforcement learning: An overview, 2018.

[9] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau. An introduction to deep reinforcement learning, 2018.

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016. `http://www.deeplearningbook.org`.

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[12] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig

Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–33, 02 2015.

[13] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, page 2094–2100. AAAI Press, 2016.

[14] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016.

[15] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019.

[16] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. *31st International Conference on Machine Learning, ICML 2014*, 1, 06 2014.

[17] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1861–1870. PMLR, 2018.

[18] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

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

[20] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control, 2016.

[21] Matthias Plappert, Marcin Andrychowicz, Alex Ray, Bob McGrew, Bowen Baker, Glenn Powell, Jonas Schneider, Josh Tobin, Maciek Chociej, Peter Welinder, Vikash Kumar, and Wojciech Zaremba. Multi-goal reinforcement learning: Challenging robotics environments and request for research. *CoRR*, abs/1802.09464, 2018.

[22] Rongrong Liu, Florent Nageotte, Philippe Zanne, Michel de Mathelin, and Birgitta Dresp-Langley. Deep reinforcement learning for the control of robotic manipulation: A focussed mini-review. *Robotics*, 10(1):22, Jan 2021.

[23] Hai Nguyen and Hung La. Review of deep reinforcement learning for robot manipulation. In *2019 Third IEEE International Conference on Robotic Computing (IRC)*, pages 590–595, 2019.

[24] Andrea Franceschetti, Elisa Tosello, Nicola Castaman, and Stefano Ghidoni. Robotic arm control and task training through deep reinforcement learning, 01 2021.

[25] Sha Luo, Hamidreza Kasaei, and Lambert Schomaker. Accelerating reinforcement learning for reaching using continuous curriculum learning. *2020 International Joint Conference on Neural Networks (IJCNN)*, Jul 2020.

[26] A. Rupam Mahmood, Dmytro Korenkevych, Brent J. Komer, and James Bergstra. Setting up a reinforcement learning task with a real-world robot. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4635–4640, 2018.

[27] Shixiang Shane Gu, Ethan Holly, Timothy P. Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3389–3396, 2017.

[28] Nestor Gonzalez Lopez, Yue Leire Erro Nuin, Elias Barba Moral, Lander Usategui San Juan, Alejandro Solano Rueda, Victor Mayoral Vilches, and Risto Kojcev. gym-gazebo2, a toolkit for reinforcement learning using ROS 2 and gazebo. *CoRR*, abs/1903.06278, 2019.

[29] Pierre Aumjaud, David McAuliffe, Francisco Javier Rodríguez Lera, and Philip Cardiff. Reinforcement learning experiments and benchmark for solving robotic reaching tasks. *CoRR*, abs/2011.05782, 2020.

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

[31] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. volume 3, 01 2009.

[32] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2149–2154, Sendai, Japan, Sep 2004.

[33] Antonin Raffin, Ashley Hill, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, and Noah Dormann. Stable baselines3. `https://github.com/DLR-RM/stable-baselines3`, 2019.

[34] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

[35] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. *CoRR*, abs/1802.09477, 2018.

[36] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2017.

[37] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24, 2011.

[38] William Fedus, Prajit Ramachandran, Rishabh Agarwal, Yoshua Bengio, Hugo Larochelle, Mark Rowland, and Will Dabney. Revisiting fundamentals of experience replay, 2020.

[39] Zachary C. Lipton, John Berkowitz, and Charles Elkan. A critical review of recurrent neural networks for sequence learning, 2015.