Universidade do Minho
Escola de Engenharia

Tiago Alcântara Ribeiro

Deep Reinforcement Learning for Robot Navigation Systems

Dissertação de Mestrado
Mestrado Integrado em Engenharia Eletrónica Industrial e Computadores

Trabalho efetuado sob a orientação do
Professor Doutor Agostinho Gil Teixeira Lopes

julho de 2019

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Acknowledgements

This dissertation is the culmination of five years of study, which were only possible with the essential support and incentive of several people, to whom I cannot forget to thank.

A special thanks to my parents Marisa Alcântara and Fernando Ribeiro for all the love, support, and always believing in me. My sister and brothers, Inês, António and Francisco Ribeiro for all the joy and affection, but mostly for being my main source of motivation. Also, to my two grandmothers São and Dina for always helping me in my studies.

A big thanks to my supervisor, Professor Gil Lopes, for the opportunity, the valuable guidance, and tireless support. To the Laboratory of Automation and Robotics members for the experiences and shared knowledge, in particular to Pedro Osório for helping with the simulators and to Fernando Gonçalves for being my dissertation partner.

To all my classmates, with whom I had the pleasure to share great adventures, especially to my close friends, Ana Correia, Alexandre Martins, Dylan Pereira, João Martins, João Passos, João Ribeiro, José Martins, Marcelo Andrade, Pedro Lopes, Romão Castro and Tiago Ferreira. To my hometown friends, especially to Carlos Areias, Joana Trigueiros, Michael Oliveira and Paulo Monteiro, for all the help and always being there for me.

To all the members of botnroll.com, for the knowledge and friendship, especially to José Cruz and Nino Pereira for all the teaching and years working together in robotics.

A special thanks to PhD Martin Riedmiller from Google DeepMind for broadening my view over reinforcement learning, thus providing inestimable advice and direction.

A very special thanks from the bottom of my heart, to Inês Garcia for all the help, the constant support, continuous encouragement, and for being my source of inspiration.

To João Silva, our eternal astronaut, for being the source of so many amazing memories that I will cherish for the rest of my life. Wherever you are, thank you my friend. *"Nothing beats an Astronaut..." – "Nada bate um Astronauta..."*

Lastly, a very special thanks to my grandfather António Ribeiro, you were, are and always will be my greatest example that through hard work and perseverance one can achieve anything one sets his mind to. Thank you for always taking care of us and the incredible example that I will take for the rest of my life.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Resumo

A aprendizagem por reforço na robótica tem sido um tema desafiante dos últimos anos. A capacidade de equipar um robô com uma ferramenta tão poderosa, como permitir a descoberta, de forma autónoma, de um comportamento optimizador a partir de tentativa-erro, tem gerado inúmeros projetos de investigação.

Esta dissertação apresenta os fundamentos teóricos de diferentes algoritmos de aprendizagem por reforço. Destes, três algoritmos distintos, nomeadamente Q-Learning, Monte Carlo Policy Gradient e Deep Deterministic Policy Gradient foram implementados em ambientes de controlo do OpenAI Gym. Os ambientes selecionados são MountainCar, CartPole e Pendulum, que garantem uma variedade de algoritmos implementáveis para diferentes espaços de estados e espaços de ações. De seguida, um agente simulado foi criado no V-REP e configurado via ROS e um nó de controlo em Python. O agente, Bot'n Roll ONE A, é um robô diferencial com sensores de distância embebidos. O objetivo do robô/agente é resolver três labirintos que aumentam de dificuldade utilizando os sensores de distância. Foram desenvolvidos testes com diferentes posições e orientações dos sensores e adicionados sensores Time-of-Flight.

Dois algoritmos, Q-Learning e Monte Carlo Policy Gradient foram implementados no robô simulado. O Q-Learning permitiu a comparação entre dois métodos distintos no que toca a tempos de seleção das ações, em que um dos métodos conseguiu resolver os três labirintos utilizando os sensores embebidos. O método Monte Carlo Policy Gradient permitiu uma análise detalhada de como o sistema de recompensas influencia a política de ações aprendida. O Deep Deterministic Policy Gradient, ainda que não implementado no robô simulado, demonstrou um enorme potencial e vantagens essenciais, tais como a política de comportamento estocástica aliada a uma política alvo determinística, o método Actor-Crítico e a controlo de ações continuo.

Palavras-Chave: Aprendizagem Máquina, Aprendizagem por reforço, Aprendizagem Profunda, Robótica, Sistemas de Navegação

Abstract

Reinforcement Learning in robotics has been a challenging topic for the past few years. The ability to equip a robot with a tool powerful enough to allow an autonomous discovery of optimal behaviour through trial-and-error interactions with the environment, has been a motive for numerous in-depth research projects.

This dissertation presents a thorough theoretical foundation that supports different reinforcement learning algorithms. Three different algorithms namely Q-Learning, Monte Carlo Policy Gradient and Deep Deterministic Policy Gradient were selected and implemented on OpenAI Gym control environments. The selected environments were MountainCar, CartPole and Pendulum. These granted a wide variety of applicable algorithms for different action-space and state-space. For each implemented algorithm, a detailed hyperparameter configuration is analysed and compared. A simulated agent was also created in V-REP and configured via ROS and a Python control node. The agent is a Bot'n Roll ONE A robot, which is a differential robot with embedded distance sensors. The goal of the robot/agent is to surpass three levels of increasing complexity mazes using its distance sensors. Tests with different sensor topologies using the embedded distance sensors and additional Time-of-Flight sensors were carried out.

Q-Learning and Monte Carlo Policy Gradient algorithms were implemented in the simulated robot. Q-Learning allowed a comparison between two different methods regarding different action selection timings. One of the methods was able to solve the three mazes using the embedded discrete distance sensors. With the Monte Carlo Policy Gradient algorithm, a thorough analysis of how reward functions influence the robot learned policies is presented. The Deep Deterministic Policy Gradient, even though not implemented on the simulated robot, demonstrated a significant potential with several essential advantages such as the stochastic behaviour policy associated with a deterministic target policy, the Actor-Critic method and continuous control.

Keywords: Machine Learning, Reinforcement Learning, Deep Learning, Robotics, Navigations Systems

Table of Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

AC – Actor-Critic

A2C –Advantage Actor-Critic

A3C – Asynchronous Advantage Actor-Critic

AGI – Artificial General Intelligence

ANN – Artificial Neural Network

CHARMIE – Collaborative Home Assistant Robot by Minho Industrial Electronics

CPU – Central Processing Unit

DDPG – Deep Deterministic Policy Gradient

DPG – Deterministic Policy Gradient

ES – Exploring Starts

GPI – generalised policy iteration

GPU – Graphics Processing Unit

HDD – Hard Disk Drive

LAR – Laboratory of Automation and Robotics

LIDAR – Light Detection And Ranging

MC – Monte Carlo

ML – Machine Learning

MNIST – Mixed National Institute of Standards and Technology

MSL – Middle Size League

PG – Policy Gradient

PPO – Proximal Policy Optimization

RBF - Radial Basis Function

RAM – Random-Access Memory

RL – Reinforcement Learning

ROS – Robot Operating System

SAC – Soft Actor-Critic

SAR – State-Action-Reward

SARSA – State-Action-Reward-State'-Action'

SGD – Stochastic Gradient Descent

SSD – Solid-State Drive

TD – Temporal Difference

ToF – Time-of-Flight

V-REP – Virtual Robot Experimentation Platform

Notation

A – Action-space

G – Return

J – Gradient of a scalar performance measure

N – Noise process

Q – Action-value function

Q_* – Optimal action-value function

R – Reward Signal

S – State-space

V – Value function

V_* – Optimal value function

a and A_t – Action

c – Centre state

h – Linear feature

p – Dynamics function

s and S_t – State

t – Time step

w – State-value weights

$\mathbb{E}$ – Expected value

α – Learning rate

γ – Discount Factor.

δ – Temporal difference error

ε – Exploration variable

θ – Policy parameter vector

μ - Parameterised actor function

π – Policy

ρ^{β} – Discounting in the state-visitation distribution

σ – Deviation

τ – Learned network influence

1

Introduction

Machine Learning (ML) is a data science subject that teaches machines to do what comes naturally to humans and animals: learn from experience. Learning by interacting with the environment is probably the first idea about the nature of learning that comes to mind. When an infant plays, looks around, waves its arms or even smells its surroundings, he has no explicit teacher, but a direct sensor-motor connection to its environment. This connection, when exercised, can produce information about cause and effect, actions consequences, and which actions to perform, to achieve a specific goal. Throughout one's life, such interactions, are a great source of knowledge about both the environment as well as one's self. An agent interacts with an environment performing actions that alter its state, whereas, the environment responds with a numeric value for the action taken (Figure 1.1). An agent is deeply aware of the environment's responses to its actions, regardless of whether it is learning how to walk, how to keep a conversation or how to drive a car. Learning from interaction is an elementary idea underlying a significant number of learning and intelligence theories.

Figure 1.1 - Reinforcement learning interactions between an agent and its specific environment.

By using computational methods, it is possible to "learn" directly from data without relying on a predetermined equation as a model. Machine learning algorithms create mathematical models based on

sample data to perform decisions or predictions without being directly programmed to perform a specific task. These algorithms find patterns in data that generate insights and help to make better decisions and predictions. The approach explored throughout this dissertation, named reinforcement learning (RL), is significantly more focused on goal-oriented learning than other machine learning approaches.

1.1 Motivation

Reinforcement learning refers to goal-oriented algorithms, which learn how to achieve a complex goal or to maximise along a particular dimension over many steps. It learns how to plot situations to actions to maximise a reward signal. The actions to take, are not stated to the learner, but instead, it must discover which actions profit higher reward by trying them out. This trial-and-error strategy offers a different approach where the agent alone must develop ways to solve a problem without human intervention. An agent can start from a null intelligence state, and under the right conditions, develop innovative strategies, achieving superhuman performance.

One of the greatest achievements in reinforcement learning history is AlphaGo Zero [1]. This algorithm was designed to play the strategy board game Go. Go is a highly-complex game since it is estimated that the lower bound on the number of legal board positions is 2×10^{170}. The game of Go consists of a 19 by 19 board, and it is played by placing the playing pieces, trying to surround the opposing stones.

AlphaGo Zero is a purely reinforcement learning algorithm that outperformed all previous versions of AlphaGo. Specifically, it defeated by a hundred to none the previous version of AlphaGo that won against the world champion. The essential idea in AlphaGo Zero is that it learns completely *tabula rasa*[1] which means, it starts entirely from a blank knowledge slate and figures out for itself, only from self-playing, without any human knowledge, data, examples, features, or interventions from humans. By achieving this learning process, it is possible to transplant an agent from the board game of Go to almost any other domain. It was noticed, during the training, that AlphaGo Zero not only rediscovered the common patterns and openings that human players tend to play, it also learned, discovered and ultimately discarded them in preference for its own variances, which humans never managed to discover (Figure 1.2). The fact that this algorithm can achieve a significantly high-performance level in a domain as complicated and challenging as Go should mean that now the most challenging and impactful problems for humanity can start to be tackled, such as robotics.

[1] Epistemological theory that individuals are born without built-in mental content and therefore all knowledge comes from experience or perception.

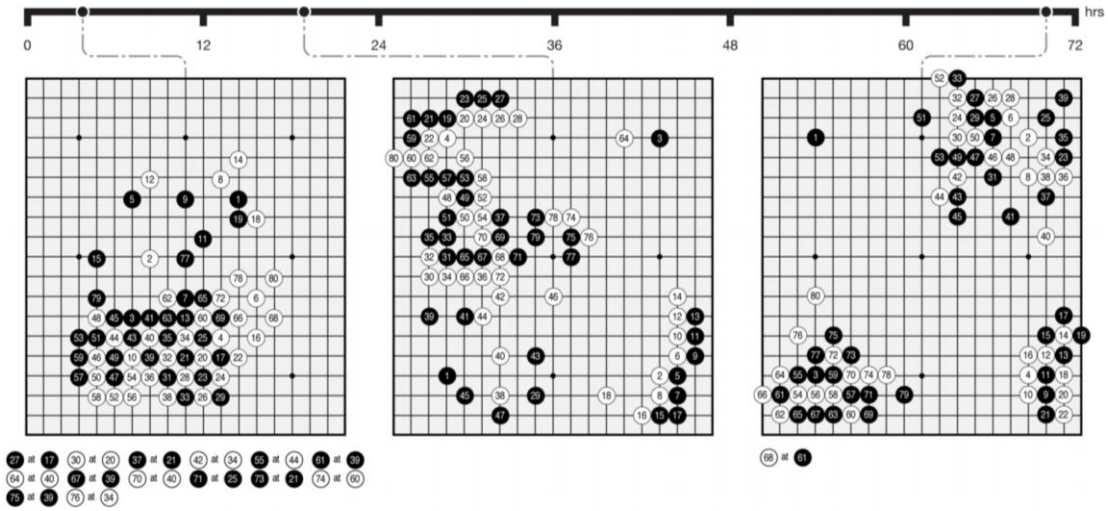


Figure 1.2 - Three Go games, that represent the first eighty moves of three self-play games at different training stages. At three hours, the game focuses on greedily capturing stones. At nineteen hours, the game exhibits the fundamentals of life-and-death, influence and territory. At seventy hours, the game is beautifully balanced, involving multiple battles and a complicated KO fight (image adapted from [1], with permission from David Silver and Springer Nature).

Reinforcement learning offers to robotics a framework and set of tools for the design of sophisticated and hard-to-engineer behaviours. As described in [2], reinforcement learning enables a robot to discover optimal behaviours through trial-and-error interactions with its environment autonomously. Instead of clearly specifying the solution to a problem, in reinforcement learning, the developer of the control task must only provide feedback in terms of a scalar objective function that measures the one-step performance of the robot. This strategy allows robots to learn on their own, developing new and unbiased tactics to solve problems. Many times, these robots develop never seen solutions, thus achieving a performance level impossible for humans.

Google's Deepmind [3] has been developing crucial work on this topic. In [4], the principle of locomotion on a diverse set of simulated bodies is studied. Virtual bodies are trained on a diverse set of complex terrains and obstacles. In Figure 1.3, some examples of simulated agents like quadrupeds (a), humanoids (b) and planar walkers (c) must overcome hurdles, gaps, variable terrains, moving walls and even platforms.

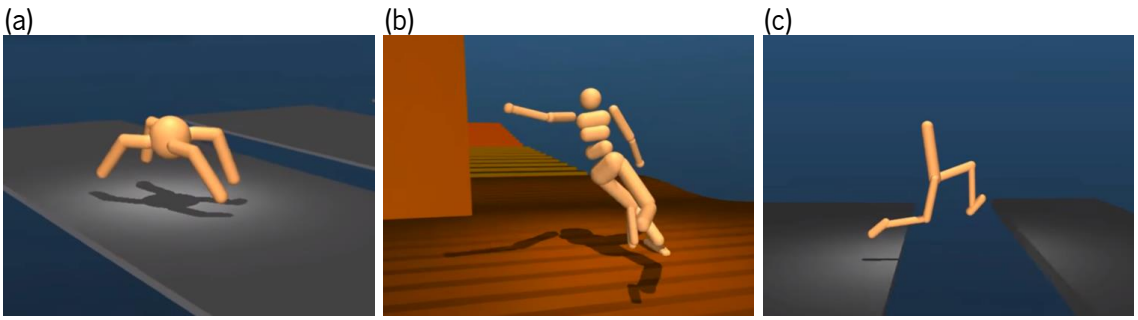


Figure 1.3 – Quadruped (a), humanoid (b) and planar walker (c) bodies overcoming different obstacles.

The aftermath results in the discovery of non-trivial locomotion strategies from all agents which manage to solve numerous different obstacles laid out on the training and testing courses. As the studied agents resemble a great variety of robots, the adaptation to physical robots is an achievable reality. These simulated agents are the launching pad for the introduction of reinforcement learning on physical robots into day-to-day human environments.

1.2 Objectives

This dissertation goals reside on the following significant topics:

- Study and development of an introductory literature review on the basics of machine learning and reinforcement learning to be used as a foundation in further works on this project and the MinhoTeam Robotics Laboratory, known as Laboratory of Automation and Robotics (LAR);
- Step-by-step practical implementation of all the algorithms on multiple simulation platforms reviewed on the previous topic;
- Creation of a simulation environment for a specific simulated robot. Design of a control script and architecture for different reinforcement learning algorithms;
- Implementation of different reinforcement learning algorithms on the developed environment for navigation and obstacle avoidance;
- Results demonstration, comparison, description and analysis from the implemented algorithms.

The introduction to reinforcement learning strategies as well as an in-depth explanation of the mathematics behind the principles of this topic is not so trivial. The purpose of the first topic is to leave a profound study to the coming generations of LAR teams. The second goal is associated with the first as open practical implementations cement a thorough understanding of the theoretical part. The examples provided will allow hands-on experience on the addressed algorithms.

A robot from the laboratory is developed in a simulation framework as well as the environment on which it will perform. This process will give a thorough knowledge of how a simulation environment must be set up in order to provide an effective reinforcement learning implementation. Bonded to the simulation environment, the control script and communication architecture will also be studied and developed.

The task proposed to solve is the navigation of a differential robot, Figure 1.4. It must solve mazes by avoiding and overcoming different obstacles in the environment. The algorithm must not be entirely specific to this robot as the learning transplantation to other robot's navigations systems, must be fulfilled with ease. Different input data will be tested for an analysis of different sensor configurations, different

types of sensors and different sensor specifications. For all the configurations previously described, an in-depth analysis will be described, explaining all the results and conclusions from the implemented strategies. The comparison between the implemented algorithms will allow a deep study of how different configurations and paradigms of reinforcement learning simulations are implemented to work with robotics.

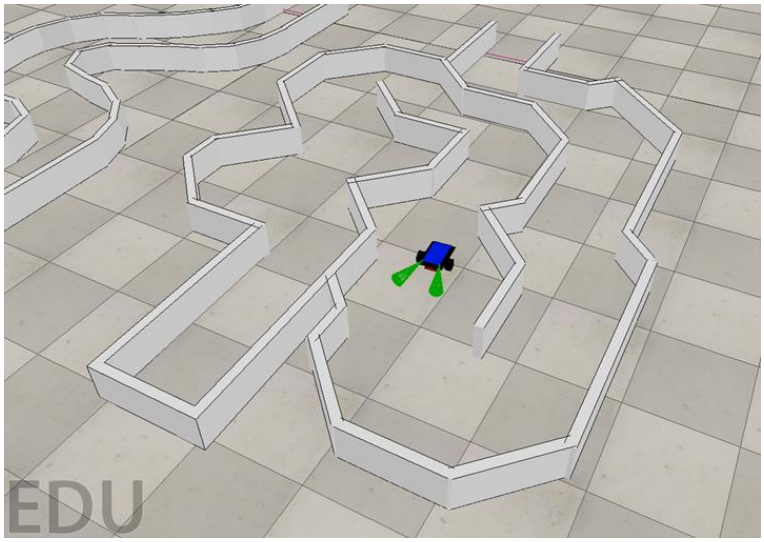


Figure 1.4 - Simulated robot as the agent and the maze as the environment.

1.3 Scientific Contributions

Over the years, LAR has been part of over a hundred robotics projects, over a vast sort of topics, from industrial, educational, research, to competitive. MinhoTeam (Figure 1.5), the competitive team of LAR has been highly active in the last 20 years, participating in robotics competitions, such as [5]–[7].

Figure 1.5 - MinhoTeam's Logo.

CHARMIE (Collaborative Home Assistant Robot by Minho Industrial Electronics) (Figure 1.6) is an anthropomorphic domestic service robot under development by MinhoTeam@Home [8] branch. The primary purpose of this robot is to perform daily chores and help humans with difficulties performing daily tasks. Some tasks such as opening cupboards, or wardrobes, establishing a dialogue with a human, object manipulation, items transportation, such as grocery bags, remote controllers, meals, beverages,

among others. By granting easiness in day-to-day tasks by collaboration or completely solving some of these chores, this project will ease the workload and independence of humans.

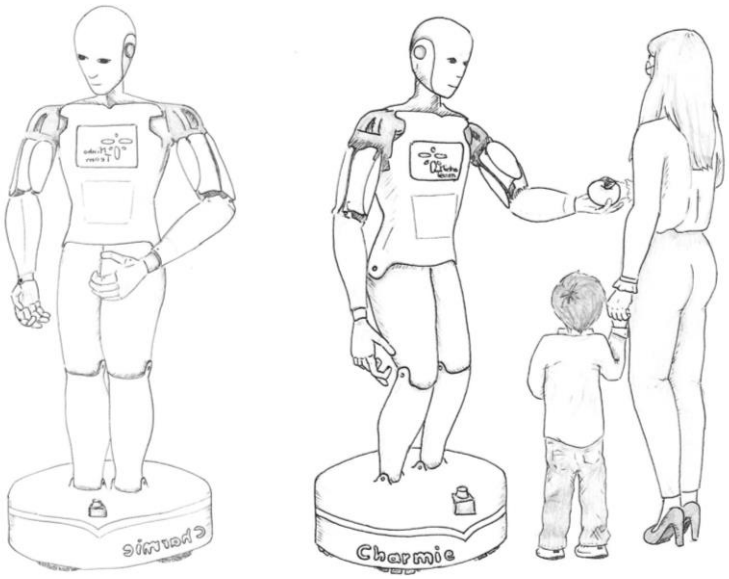


Figure 1.6 - Sketches from the expected CHARMIE design.

As a secondary purpose, it fits the team's intentions to compete in RoboCup@Home [9], a competition for multi-purpose domestic service robots held within a house. The main goals are to perform a set of domestic tasks and to demonstrate capabilities to do innovative chores as discussed in [10] and [11]. In [12]–[15], some of the most successful robot's team description papers are presented. In Figure 1.7, CHARMIE's first proof of concept [16] was developed for necessary robot module testing. At the time, the second version is under development, and a great part of this project's goals will be achieved.

(a) (b)

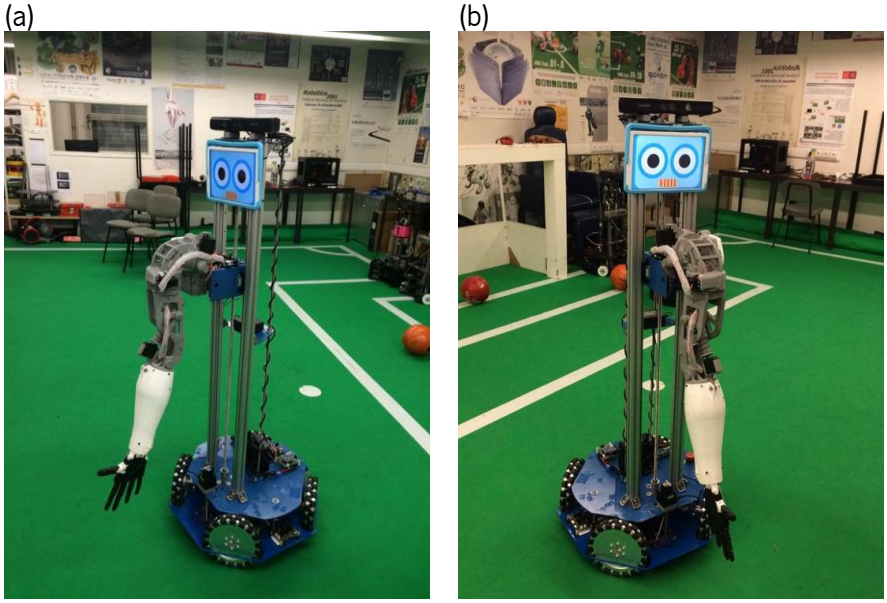


Figure 1.7 - CHARMIE's first proof of concept developed in 2017.

CHARMIE's innovations are the similarities with the human body and senses, and that all algorithms from this robot are based on machine learning algorithms. The robot's shape and degrees of freedom are in all ways similar to the human body, except for a four-wheeled omnidirectional locomotion base. The robot inputs are also similar to human senses: vision, touch and hearing, with the addition of some Light Detection And Ranging (LIDAR) sensors. The robot must interact with its environment the same way a humans does, by speaking, hearing, object manipulation, vision, among others. CHARMIE's principal innovation is set by entire machine learning strategies. This dissertation is the first step of the navigation strategies to be implemented on CHARMIE.

1.4 Document Structure

To better understand and plan the several steps and features essential to the progression of the project, this dissertation is divided into several phases. Firstly, it was necessary to do a theoretical study of reinforcement learning algorithms to understand the specificities of each learning system. Moreover, practical implementation on OpenAI Gym environments regarding the three selected RL algorithms, namely Q-Learning, Policy Gradient (PG) and Deep Deterministic Policy Gradient (DDPG). Next, the creation of a new simulation environment regarding a selected robot. Along with the environments, the Robot Operating System (ROS) structure and the control scripts were also designed for customised reinforcement learning implementations. The first two algorithms, Q-Learning and Policy Gradient, were also tested on the created agent. In Q-Learning, the comparison between two different action selection methods is detailed, whereas, in Policy Gradient, a thorough reward function comparison was made. A detailed hyperparameters search and analyses are performed on all simulation environments and for all algorithms.

This dissertation is organised in seven chapters. The first one is the Introduction chapter, where the motivation and objectives are described, followed by the Literature Review, where state of the art is present and analysed. The third chapter, Algorithm-Architecture: Theoretical Foundations, is an essential chapter of this dissertation since all the algorithms mathematical foundations are described and studied. The following chapter, Simulation Framework: Development introduces the environments and simulated parts integrated into this work. The chapters, Q-Learning, Policy Gradient, Deep Deterministic Policy Gradient, refer to the algorithms selected to be implemented throughout this dissertation. All methods will be detailed as well as their hyperparameter configurations. Finally, in the Conclusions chapter, the focal outcomes are enumerated as well as further work.

2

Literature Review

As presented in the previous chapter, Introduction, the application of reinforcement learning to robots is a novel topic on robotics scientific community. As so, an extensive literature review is presented to introduce all the topics in the forthcoming chapters properly. The state-of-the-art analysis and study were approached on four perspectives:

1. Machine learning and respective branches
2. Reinforcement learning in robotics
3. Simulators and frameworks
4. In-depth mathematical study on reinforcement learning

2.1 Machine Learning

Artificial Intelligence is the area of computer science which emphasises the creation of intelligent machines that work and react like humans. Machine Learning is a branch of Artificial Intelligence that allows computers to learn how to perform new tasks without being pre-programmed. It consists of a study of algorithms and statistical models that computers use to accurately perform a specific task with no use of explicit instructions. A mathematical model is built based on sample data, known as “training data”, to make predictions or decisions without any direct programming to perform the task. These types of algorithms teach themselves and grow when new data is provided for the machine to learn from. Machine learning can be divided into three branches (Figure 2.1): supervised learning, unsupervised learning and reinforcement learning, which can respectively be divided into specific sub-branches.

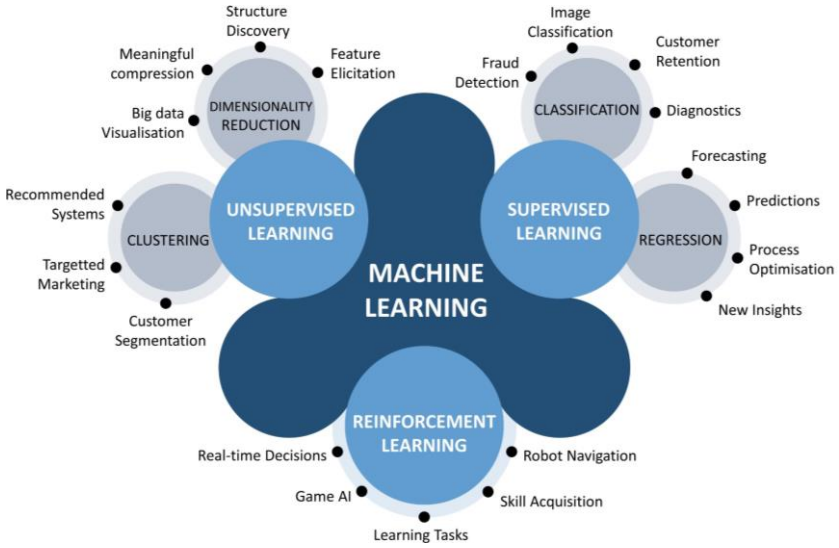


Figure 2.1 - Machine learning branches and respective sub-branches.

2.1.1 Supervised Learning

Supervised learning [17] consists of learning a function that maps an input to an output based on example, input-output pairs. It deduces a function from labelled training data that includes a set of training samples which comprises a pair of both the input object and the desired output value. A supervised learning algorithm scrutinises the training data and generates a function, which is used to classify or regress new examples. The focus of this algorithm is the correct classification/regression for unseen instances. This requires the supervised learning algorithm the capability of generalising from training data to never previously seen samples.

The MNIST [18] dataset is an extensive database of handwritten digits used for training various image processing systems. The database is also broadly used for training, testing and even benchmarking in the field of machine learning, more specifically, supervised learning. MNIST is commonly used as an introductory dataset to learn the main principles of machine learning as well as supervised learning, neural networks, image processing, classification, among others. The dataset contains $60\,000$ training images and $10\,000$ testing images. A significant number of scientific papers have been published on attempts to achieve the lowest classification error rate. At the time of writing this dissertation, the published lowest test error percentage used a 35 layer convolutional neural network with width normalisation, achieving a 0.23% error rate [19]. In Figure 2.2, an example of the MNIST classification is presented, developed as a preparation for this dissertation. It displays both the cross-entropy (a) and the accuracy (b) for the algorithm developed. Figures (d) and (e) represent the evolution over the number of episodes of the weights and biases, respectively. Graph (c) represents the 100 images used for training in the last iteration and (d) represents the results of the last algorithm testing.

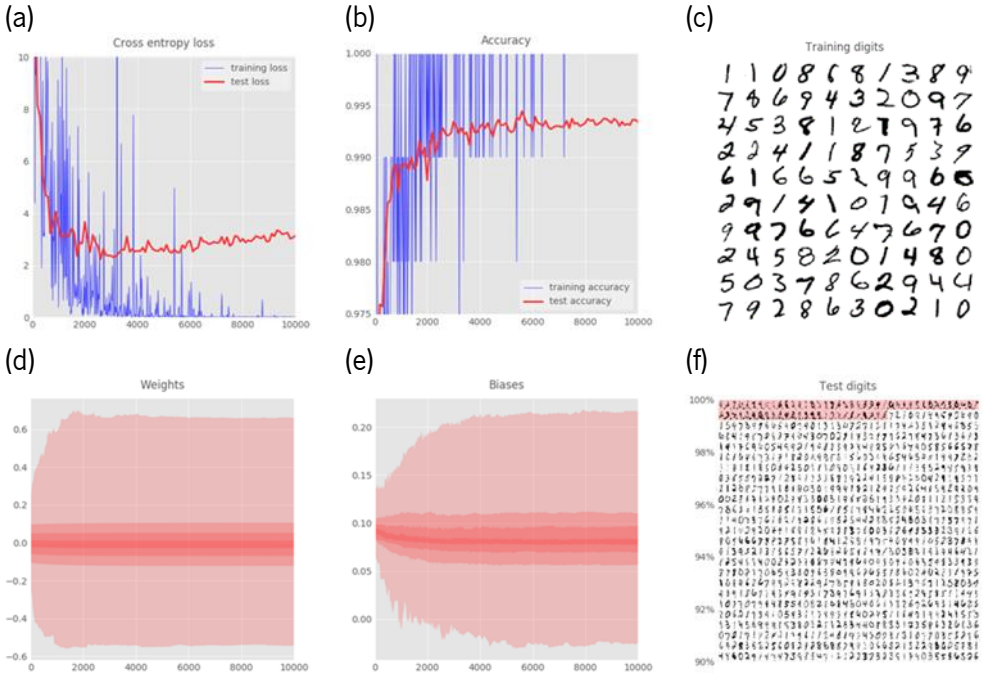


Figure 2.2 - Example of a solution to the MNIST classification problem.

2.1.2 Unsupervised Learning

The Unsupervised learning [20] algorithm is associated with the idea of learning without a teacher. It is also known as self-organisation and a method of modelling the probability density of inputs.

Cluster analysis is a sub-branch of unsupervised Learning that groups unlabelled, unclassified or uncategorised data. Instead of responding to feedback, cluster analysis identifies similarities in the provided data and responds based on the existence or absence of such similarities in each new sample of data. The core purpose of these types of algorithms is the study of data's intrinsic and hidden structures, to get meaningful insights, segment datasets in similar groups, or even to simplify them.

The other sub-branch of unsupervised learning is dimensionality reduction. Input data may have an abundant number of dimensions, as a result of a rich or overcomplex input data. Representing data with fewer dimensions allows a more accessible learning in subsequent tasks, a more straightforward visualisation and a more natural discovery of intrinsic data patterns.

2.1.3 Reinforcement Learning

The early history of reinforcement learning [21] has two extensive main threads that were independently pursued before linking in present-day reinforcement learning. The first thread concerns trial-and-error learning and emerges from the psychology of animal learning. The second thread concerns the optimal control problem and its solution using value functions and dynamic programming. Even though both

threads have been mostly independent, the exception is a third thread concerning temporal-difference methods. The three threads were linked together in the late 1980s to create the current field of reinforcement learning.

The "optimal control" term is used to describe the controller design problem to minimise a measure of a dynamical system's behaviour over time. In the 1950s, an approach to this problem was developed by Richard Bellman as an extension of the previous control theory of Hamilton and Jacobi. By using concepts of a dynamical system's state and a value function, to define a functional equation, the Bellman equations were created. The series of methods to solve the optimal control problems by solving the Bellman equations are known as Dynamic Programming. Bellman also introduced a discrete stochastic version of the optimal control problem known as the Markov Decision Processes (MDPs), and Ron Howard developed the policy iteration method for MDPs. These are all central fundamentals underlying the theory and algorithms of modern reinforcement learning. A thorough explanation is presented in sub-chapter, From Nothing to Q-Learning.

The first thread concerns trial and error learning and has its origin in the psychology of animal learning. Aristoteles was the first known ambassador of the reinforcement learning concept. The term *tabula rasa* means empty book, meaning that consciousness is devoid of any innate knowledge. It is the epistemological theory that an individual is born without any built-in mental content, and therefore, all knowledge comes either from experience or from perception. The first person to succinctly express the trial-and-error as a principle of learning was Edward Thorndike with the "Law of Effect" [22].

> "Of several responses made to the same situation, those which are accompanied or closely followed by satisfaction to the animal will, other things being equal, be more firmly connected with the situation, so that, when it recurs, they will be more likely to recur; those which are accompanied or closely followed by discomfort to the animal will, other things being equal, have their connections with that situation weakened, so that, when it recurs, they will be less likely to occur. The greater the satisfaction or discomfort, the greater the strengthening or weakening of the bond." - Edward Thorndike in [23], p. 244

Thorndike tested his theory by devising a classic experiment in which he used a puzzle box, to test the laws of learning empirically. He would place a cat in the puzzle box, which was encouraged to escape to reach a fish placed outside of the cage. The time it took for the cat to escape would be registered and compared to the next iterations. In the beginning, the cats would experiment with different strategies to escape the puzzle box and thus reaching the fish. Ultimately, they would stumble upon the lever which

would open the cage. After the escape, the cats were put again in the puzzle box, and the time it took to escape was noted. In consecutive trials, the cats would develop the knowledge that pressing the lever would have positive consequences, and would rapidly adopt this behaviour, becoming increasingly quick at pressing the lever.

The term "Reinforcement" in the context of animal learning came into use well after Thorndike's expression of the "Law of Effect", first appearing in this context in the 1927 english translation of Pavlov's monograph on conditioned reflexes. Pavlov described reinforcement as the firming of a pattern of behaviour due to an animal receiving a stimulus.

The idea of implementing trial-and-error learning in a computer appeared among the earliest thoughts about the possibility of artificial intelligence. In a 1948 report [24] named: "Machinery Intelligence", Alan Turing described a design for a "pleasure-pain system" that worked along the lines of the "Law of Effect":

> "When a configuration is reached, for which the action is undetermined, a random choice for the missing data is made and the appropriate entry is made in the description, tentatively, and is applied. When a pain stimulus occurs all tentative entries are cancelled, and when a pleasure stimulus occurs they are all made permanent." - Alan Turing in [24], p. 13

Andrew G. Barto and Richard S. Sutton are considered to be founding fathers of modern computational reinforcement learning. Both developed several breakthrough algorithms and strategies. The book [21] is considered the best introductory book on reinforcement learning. Andrew Barto considers that reinforcement learning is a complement of both search and memory. Search can be characterised as trial-and-error, generate-and-test and variation-and-selection. However, reinforcement learning must also be capable of recalling what worked best for each situation.

Trial and error learning did not have an intensive study from an AI point of view in the next decades. Opposing to error correction learning algorithms, such as performed in supervised learning, Trial and error, does not have such in-depth research. It was only in 1981 that Barto and Sutton in [25], developed an associative network where the outputs were not specified by examples. The system had to try things and had them evaluated in order to come up with the output that produced the highest reward, without that ever being presented to the system. In this particular system, the sole purpose of the network's action was to create the reward signal.

In 1983, a new algorithm was created where the network's action did not just create the reward signal but would also return the new state the agent was. The control problem presented was a pole balancing system, also known as the inverted pendulum problem, Figure 2.3, which led to the conception of an Actor-Critic (AC) algorithm [26]. The Actor-Critic architecture consists of two different networks. The actor decides which action to perform according to the actual state, and the critic helps the agent's process of decision as well as the network configuration as an external viewer. The AC algorithms are still used to date as an attempt to solve control problems in reinforcement learning.

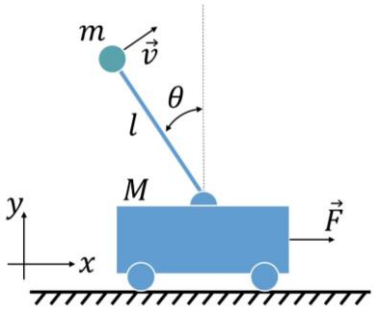


Figure 2.3 - Inverted pendulum control problem.

In parallel with the actor-critic algorithms, Richard Sutton created in his PhD [27] the temporal difference (TD) algorithms. The Temporal Difference error correction technique was different from supervised learning because it not only predicts the reward as an external signal but also predicts its future predictions. One of the most significant achievements was in an artificial checkers player. In [28], Arthur Samuel describes that the TD algorithm outperformed his evaluation function. His evaluation function relied only on the current state of the board, in contrast with Temporal Difference, which predicted future states until it reached a terminal state.

> "... we are attempting to make the score, calculated for the current board position, look like that calculated for the terminal board positions of the chain of moves which most probably occur during actual play." – Arthur Samuel in [28]

Temporal Difference Learning also led to an essential connection between Optimal Control and Dynamic Programming. Richard Bellman introduced Dynamic Programming in 1953, which was an algorithm very similar to the checker's solution. TD Learning made it possible to adapt this technique with Optimal Control Problem.

In the early 1990s, the first Temporal Difference Learning board game, named TD Gammon outperformed human experts. This project introduced several new techniques that surprised the scientific community

for its innovation and success. In addition to using Temporal Difference Learning, it used a multilayer artificial neural network (ANN) trained with backpropagation and Monte Carlo (MC) Learning.

At the time, reinforcement learning had the reputation of being a significantly slow algorithm. Monte Carlo simulation based optimisation was being studied in other fields, and its success was a revelation due to reinforcement learning reputation. Monte Carlo method consists of learning through multiple simulated games. In Backgammon the different configurations of the playing board, the states, are about 10^{20}. Strategies like Dynamic Programming would take an unfeasible amount of time to sweep all these states, resulting in years and years of continuous simulation. This problem is called the Curse of Dimensionality. With such a big state-space, it turned out that many states were irrelevant. Through multiple Monte Carlo games, the reinforcement learning computation could be focused only on the relevant parts of the state-space. Monte Carlo brought a great time-efficiency to reinforcement learning algorithms by looking down on irrelevant or semi-irrelevant states.

Neuroscience is also an essential contributor to reinforcement learning. Many ideas to reinforcement learning come from biology and the human's brains study. In neuroscience, some comparisons with how the human brain processes information were made to Temporal Difference error. Dopamine, as displayed in Figure 2.4, is one of several neural modulators sent via nerve fibres, like the mesolimbic pathway, whose essential purposes are reward and reinforcement. Many studies discuss the comparison between TD error and the way dopamine works on the brain.

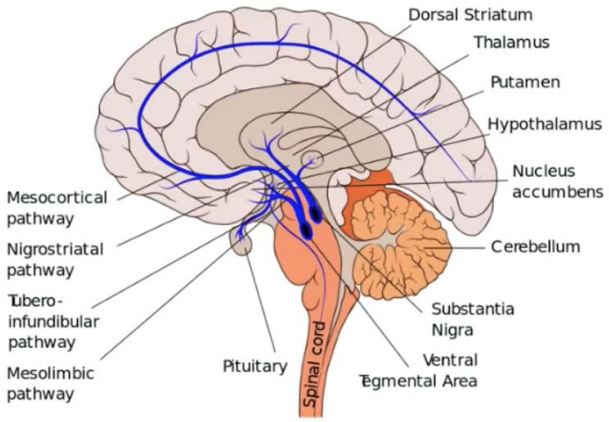


Figure 2.4 - Neuroscience representation of how dopamine travels through the human brain (from [29]**).**

Due to the work of Wolfram Schultz, it was discovered that some aspects of the signal are identical to the temporal difference error. Schultz research consisted of recording dopamine activity, among other chemical processes, in awake behaving monkeys, while they were performing specific tasks. In Figure 2.5 is shown an adapted display of a famous collection of pictures from Schultz's research [30], on

predictive reward signal of dopamine neurons. Bursts of activity of dopamine cells are initially triggered by a reward (a), i.e. monkey putting food in his mouth, as a consequence of effectively performing a task. However, with continuous trials, if the monkey develops a predict stimulus (b), the response of the dopamine rewards seizes at the time of reward and moves back to the time of the reward predictor, even when the reward does not occur (c). Moreover, If the animals develop what is known as a "predictor of a predictor" or an earlier predictor, the activity of dopamine cells will take place on the initial predictor.

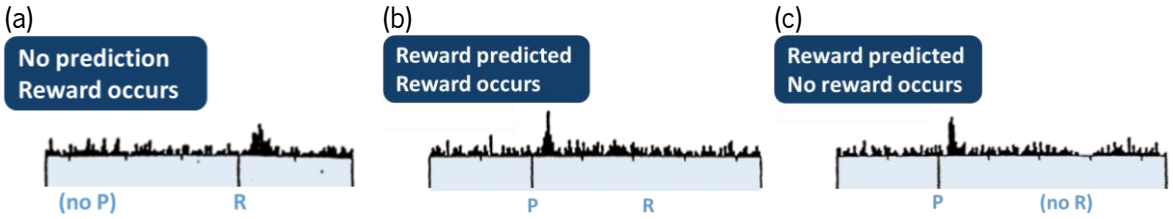


Figure 2.5 - Schultz experiment demonstrates that after learning a task, the monkeys started having dopamine bursts when predicting a positive situation, R denotes the time a reward is awarded, and P denoted the time a prediction is made (adapted from [30]**).**

This reaction is precisely the Temporal Difference error's purpose. Additionally, this realisation has revolutionised neuroscience view on the brain's reward system. These cells are not the reward themselves; they are the reinforcement signal that corresponds very closely to the Temporal Difference error.

Nowadays, reinforcement learning is widely used in numerous areas. One of the most significant achievements for reinforcement learning algorithms is AlphaGo Zero. From the AlphaGo family, AlphaGo Zero is the most recent algorithm, which has learned entirely from scratch without any human data. The essential idea of AlphaGo Zero is to learn entirely from *tabula rasa*. Thus, starting with a clean knowledge and learning just from self-playing, without any human knowledge, data, examples, features, or interventions. In Figure 2.6, AlphaGO Zero self-play (a) and neural network training (b) are demonstrated. It uses a Monte Carlo Tree Search when it is playing and a deep convolutional neural network for its training.

AlphaGo Zero uses an innovative form of reinforcement learning, in which it becomes its own teacher. The system starts with an artificial neural network that does not know anything about the game of Go. It plays against itself, combining the neural network with the Monte Carlo tree search. As it plays, the network is tuned and updated to predict, both the moves and the eventual winner of the game. At each game, the system's performance improves by a small amount, and consequently, the quality of the self-play games increases. This leads to a constant improvement of the system, creating ever more robust versions of AlphaGo Zero.

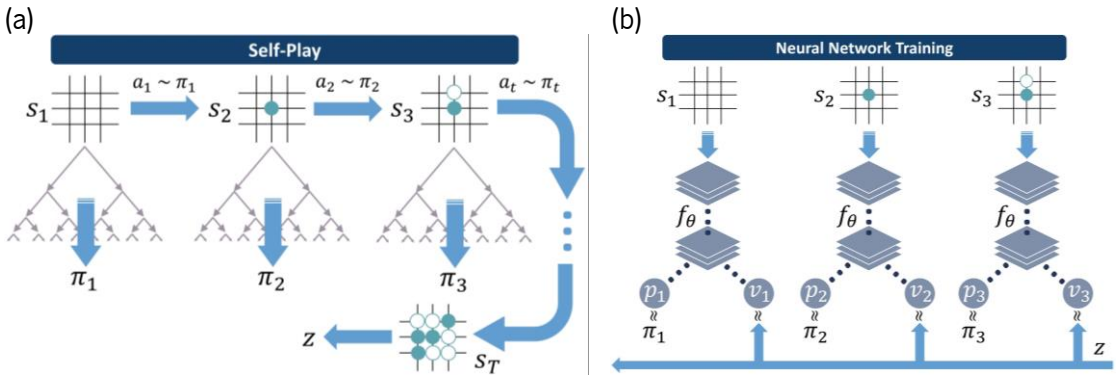


Figure 2.6 - AlphaGo Zero architecture diagrams, (a) represents the self-play tree searches, and (b) represents the afterwards training using the episode information (image adapted from [1], with permission from David Silver and Springer Nature).

This system outperforms the previous versions of AlphaGo because it learns from *tabula rasa*, no longer constrained by the limits of human knowledge, from the strongest player in the world: AlphaGo Zero itself. The low time-costing AlphaGo Zero learning process is also a significant technological advancement. The technique is defined by the following timeline milestones (Figure 2.7):

- 3 hours: AlphaGo Zero plays like a human beginner, abdicating of long-term strategies to focus on greedier policies capturing as many stones as possible.
- 19 hours: AlphaGo Zero has learnt the fundamentals of more advanced Go strategies such as life-and-death, influence and territory.
- 70 hours: Alpha GoZero plays at a super-human level. The game is disciplined and involves multiple challenges across the board
- 3 days: AlphaGo Zero surpasses the abilities of AlphaGo Lee, the version that beat world champion Lee Sedol in four out of five games.
- 21 days: AlphaGo Zero reaches the level of AlphaGo Master, the version that defeated 60 top professionals online and world champion Ke Jie in three out of three games in 2017
- 40 days: AlphaGo Zero surpasses all other versions of Alpha Go and, arguably, becomes the best Go player in the world. It does this entirely from self-play, with no human intervention and using no historical data

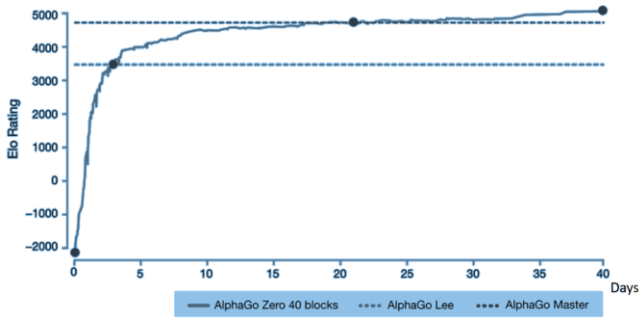


Figure 2.7 - AlphaGo Zero day-to-day learning evolution.

At the moment, a significant number of scientific teams are studying Artificial General Intelligence (AGI). The idea behind this concept is to create an algorithm that can learn multiple different tasks. Atari Games are well-known to the reinforcement learning community as they provide different environments which require different approaches from the traditional control problems. These games are also used for benchmarking reinforcement learning algorithms as well as AGI systems. In [31], a reinforcement learning algorithm is presented that solves six different Atari games (Figure 2.8) and surpasses human experts on three of them.

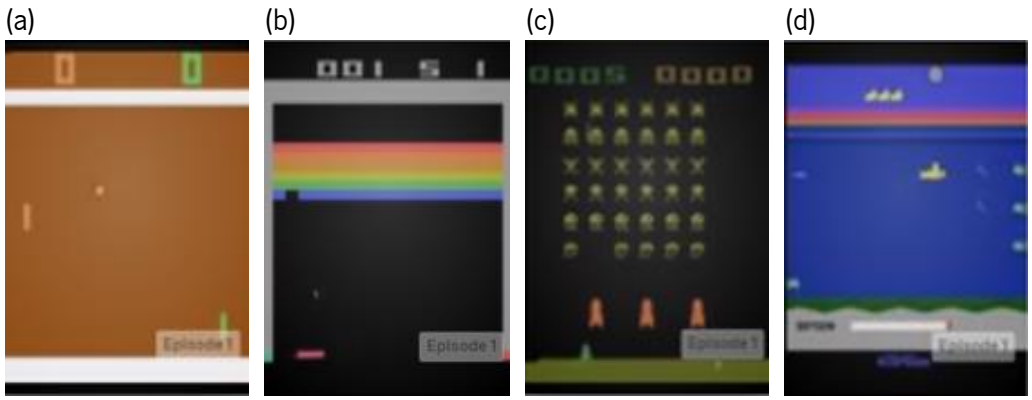


Figure 2.8 - Atari environments solved using deep reinforcement learning in [31]**, (a) Pong, (b) Breakout, (c) SpaceInvaders, (d) SeaQuest.**

Another significant milestone for reinforcement learning was the success of OpenAI Dota 2 player [32]. OpenAI Five is an RL algorithm that plays Dota 2, a multiplayer online battle arena game. This game requires long-term strategy planning. OpenAI initial version was able to play one versus one set of games with only one hero and was able to beat the world champion player reliably. Even though this was a significant achievement, it was only a stepping stone towards playing the real five versus five Dota 2 game. In 2018, it was unveiled an improved version of OpenAI Five, a five versus five players with a limited hero pool, that was able to defeat competent players.

In April 2019, OpenAI Five challenged the reigning world champions to a five versus five set of games, with a pool of seventeen heroes. This algorithm, like AlphaGo Zero, started from *tabula rasa* and learned only from millions of self-played games. It developed unorthodox strategies that surprised the world champion team. Also, the confidence value during the games was always noticeably high, reaffirming the algorithm confidence. This algorithm defeated two games to none the best team of humans in the world and the second game is known as a completely one-sided game. In Figure 2.9, a screenshot from one of the decisive moments in game one, right after this stand-off the RL algorithm predicted a 95% chance of winning. Even though it is a videogame, the decision complexity and planning of many real-time situations is a notable improvement. Their victory required teamwork and collaboration between algorithms.

Figure 2.9 - Decisive moment on Dota 2 OpenAI Five game two against world champions, in this confrontation the human team lost three out of five players.

2.2 Reinforcement Learning in Robotics

For a robot to operate efficiently in an environment, it must make some sense of it, be able to plan its actions and execute them while using feedback to ensure everything is proceeding according to plan. These components, contrary to human, are not trivial for a robot to perform. Tasks like recognising objects, dialoguing, predicting, to some extent, human's intentions, multiple object manipulation are very challenging for a machine to perform. In recent years, the rise of Deep Learning has prominence in multiple computer vision tasks [33]. However, the broad spectrum of skills required for a typical pedestrian street scene is still not entirely within the grasp of current technology.

The recent successes of reinforcement learning and deep learning led many researchers to develop new methods to control robots using reinforcement learning. The capability of automating the sensing, planning and control algorithms via a self-learning strategy by autonomous trial-and-error would solve numerous nowadays robotics problems.

After the surprising successes in several video games, board games and simulated control problems, deep reinforcement learning scientific researches started to focus on robotics. On the previously specified tasks, the environments on which the agent must learn are the same on which it must operate, and the simulation's efficiency allows the agent to learn in fewer episodes. Deep reinforcement learning systems are notoriously data inefficient, and often require millions of episodes before successfully learning how to solve a task such as playing an Atari game. If the same systems were to be applied to a real-world robot's learning, it would take an unfeasible amount of time as well as physical damages to the robot. An alternative is to use a simulated training environment and posteriorly deploy the trained policy into the real robot. However, a real-world completely thorough simulation of all the details are both problematic

and costly to achieve, and such policies do not generally perform well on the physical robots. The challenges of implementing reinforcement learning into physical robots can be divided into four categories:

- Sample Efficiency;
- Simulation to Real-World Transition;
- Reward Specification;
- Safety.

Two different approaches regarding the sample efficiency are commonly used, model-based and model-free RL. Model-free learning refers to the process of learning a policy based only on rewards obtained by interaction with the environment. Model-based learning refers to the process of an agent trying to discover an environment's model and use it to plan and improve the policy. This method often results in a lower performance compared to model-free learning, and occasionally fails due to errors in the learned model. Model-based RL shows significant success in problems where the dynamics can be represented by simpler models, such as a linear one. However, in more complex environments like non-linear models (artificial neural networks), model-free learning tends to have better success. In [34], to solve this model-based problem, it is proposed the system to learn several different models of the environment and to uniformly sample from the different learned models, regularising the learning process.

A recent paper [35], regarding a model-free learning algorithm, has demonstrated a significant sample efficiency. In the paper, a description of an off-policy actor-critic algorithm called Soft Actor-Critic (SAC) demonstrates that it was possible to train a robot to walk in about four hours of environment interaction, among other implementations.

The simulation to the real-world gap is another problem of reinforcement learning in robotics. The simulation of an environment a robot is likely to encounter realistically is a common challenge. Reinforcement learning is a valuable solution to problems where simulation can sufficiently capture the dynamics. In [36], a policy was trained to learn recovery manoeuvres for a quadrupedal robot using a simulation environment. The deployment to the real robot resulted in a 97% success rate. However, the state-space was significantly low dimensional, contrasting with robotic visual representations.

A solution to cope with robotic visual inputs such as cameras was also developed in [37]. By randomising the environment's visual inputs (Figure 2.10) and training the policy to be robust to these changes, this system implied the real world to be just like a simulation variation. The algorithm created multiple variations of the same environment, where the real-world would be tested as just another variation.

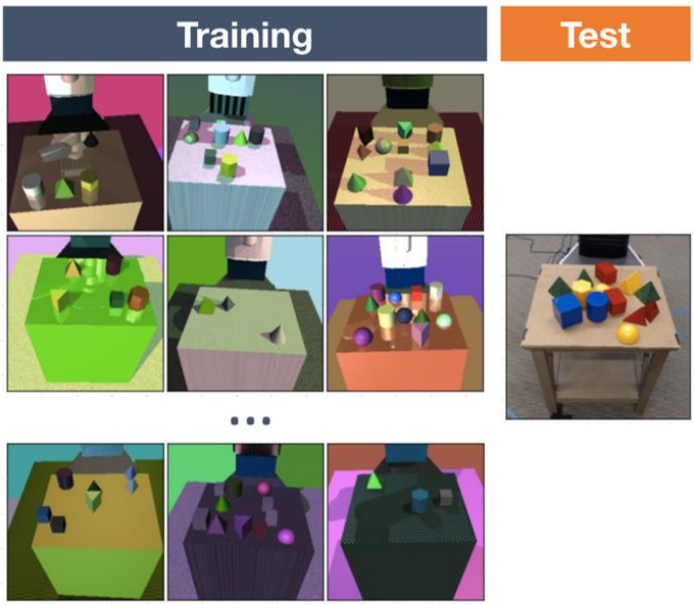


Figure 2.10 - Training and testing comparison of domain randomisation for object manipulation (figure from [37] **displayed with permission from Josh Torbin).**

For a machine to learn through environment interaction, it must receive a reward signal as a performance evaluation. The reward will diverse depending on the task, the environment and its outcome. In some tasks, like checkers, the goal is to win the game. However, for more complex tasks such as laundry folding or placing a book on a bookshelf, the reward function is not trivial. In a recent paper [38], instead of stipulating a reward function, an algorithm with several images was provided as a goal state combined with human binary labels, for bookshelf arranging, among others. This algorithm managed to restrain from a manually reward function partially.

Lastly, a crucial challenge to reinforcement learning implementation on real-world robots is safety. Three factors must be considered when discussing safety: the robot, the environment and the robot operator. Any upset in one of these factors is a key problem either financially, time-costing and even life-threatening situations. Thus, significant attention is given concerning safety issues in RL implementations on real-world robots.

Mobile robots obstacle avoidance is another problem in which reinforcement learning algorithms are broadly studied for real-world applications. Researches like [39], demonstrate reinforcement learning implementations in robots with low-level sensors such as individual distance sensors. As in researches like [40], a multiagent collision system is capable of finding collision-free, time efficient paths around other agents.

A grand milestone in reinforcement learning in robotics is set by the Brainstormers RoboCup Middle Size League (MSL) soccer team (Figure 2.11). By using a multi-layer neural network, it learned various soccer

sub-tasks [41], [42], such as different defence strategies, pass interception, position control, kicking, motor speed control, dribbling and penalty shots.

The resulting components contributed to winning the MSL world cup twice. As neural networks are function approximators, the value-function overestimation at a recurrently occurring state will increase the values predicted by the neural network for all other states, causing fast divergence. In [42], Riedmiller solved this problem by defining an absorbing state where the value predicted by their neural network was set to zero, which "clamps the neural network down" and therefore prevents divergence.

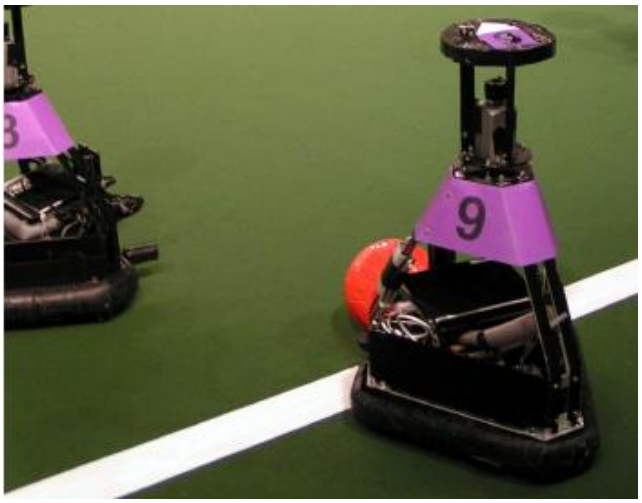


Figure 2.11 - Brainstormer Tribots robots that won the MSL in both RoboCup 2006 and 2007 (figure from [2] displayed with permission from Martin Riedmiller).

2.3 Simulators and Frameworks

In Osório's dissertation [43], a thorough description and comparison of different robotics simulation environments are presented. The discussed simulators are Microsoft Robotics Developer Studio, Virtual Robot Experimentation Platform (V-REP), Webots and Gazebo. A novel simulator dedicated for reinforcement learning algorithms, with a division dedicated to robotics, was introduced in 2016 named OpenAI Gym [44]. Gym is a toolkit for development and comparison of reinforcement learning algorithms. It makes no assumptions about the agent's structure and is compatible with any numerical computation library, such as TensorFlow or Theano. OpeanAI's Gym is an essential contribution as it is used as a benchmarking platform for reinforcement learning algorithms and the standardisation of environments for results comparison. At the time of writing this dissertation, Open AI Gym provides the following sections: Algorithms, Atari, Box2D, Classic Control, MuJoCo, Roboschool, Robotics and Toy Text.

The Classic control section consists of some basic examples that are commonly used for benchmarking and testing new algorithms. Some of its control problems are well known to allow a more straightforward

introduction to OpenAI Gym. CartPole (Inverted Pendulum) and Mountain Car are the most commonly used environments. The Robotics section, Figure 2.12, offers eight simulation environments where the agents are manipulators. A robotic arm (a), as well as a robotic anthropomorphic hand (b), must perform tasks such as Pick and Place, Pushing, Reaching and Sliding.

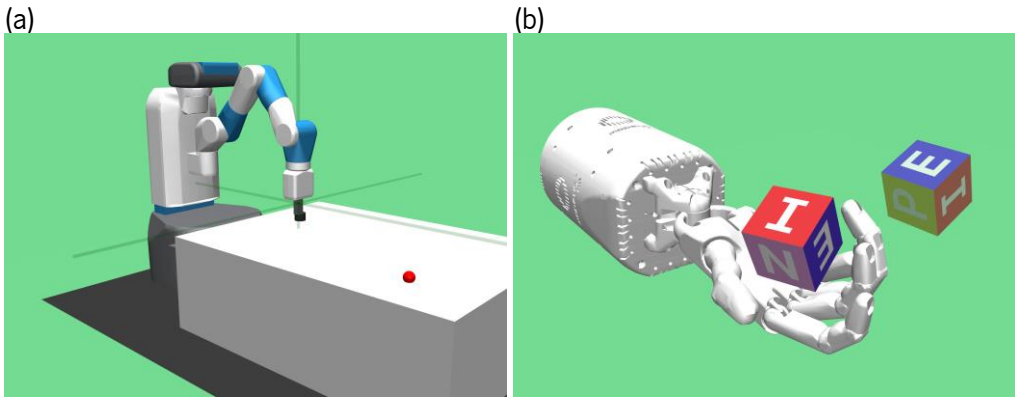


Figure 2.12 - Some examples of the robotics section environments, (a) FetchManipulator and (b) HandManipulate.

Both MuJoCo and Roboschool are significantly related to robotics. In Figure 2.13 are displayed some of the possible agents such as quadrupeds, humanoids (a) and (b), planar walkers (b), hoppers (b), and even ATLAS robot (c) from Boston Dynamics. Roboschool has the addition of using Bullet as its physics engine, allowing a smooth 3D experience. The tasks these agents must perform consist of walking, running and hopping as fast as possible while maintaining equilibrium and reaching target coordinate.

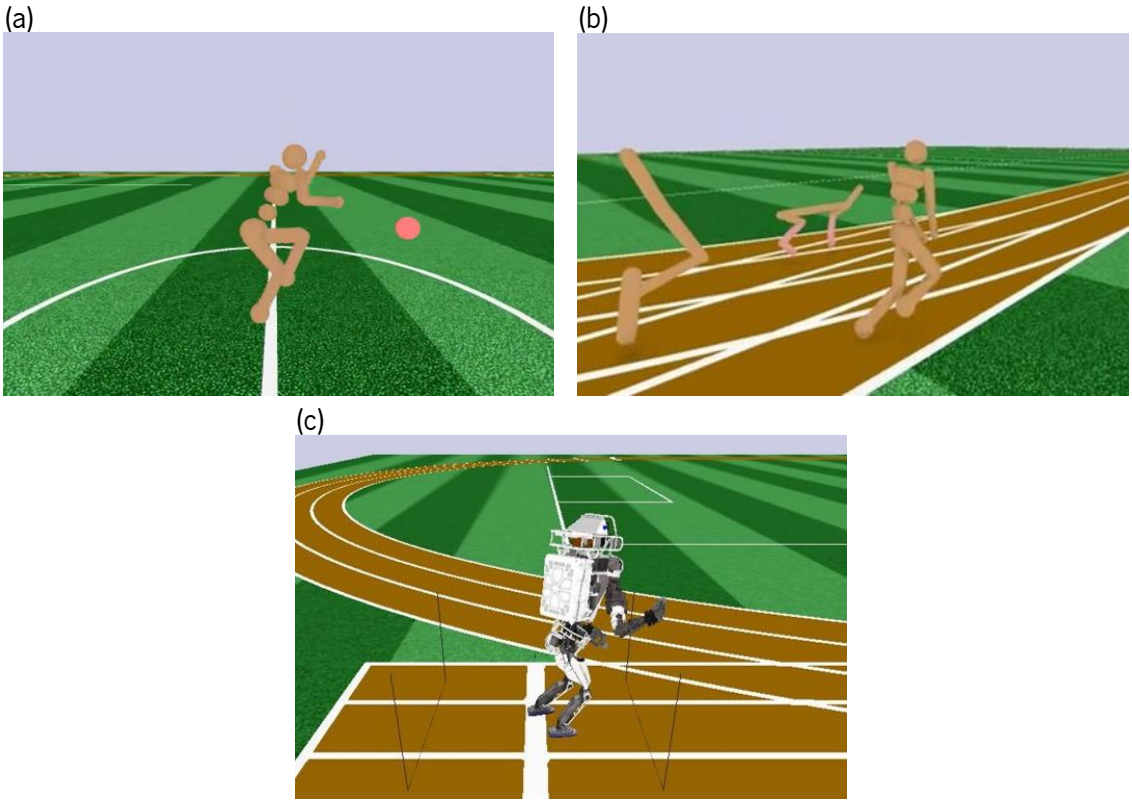


Figure 2.13 - Some examples of the roboschool section environments, (a) HumanoidFlagRun, (b) Humanoid, Hopper and HelfCheetah, and (c) Atlas from Boston Dynamics.

2.4 Conclusions

In this chapter, it is presented a review on machine learning and its three main branches, namely supervised learning, unsupervised learning and reinforcement learning. Reinforcement Learning is an experience-based type of learning, which relies solely on trial-and-error methods, which is a good fit for robots to learn how to perform new tasks, with any human bias.

The two historical threads of reinforcement learning are also presented. A more practical thread concerning the optimal control problem and its solution using value functions and dynamic programming, and a more theoretical thread regarding trial and error learning original in the psychology of animal learning. Both psychology and neuroscience are two subjects deeply connected to reinforcement learning.

Due to the trial-and-error experience method reinforcement learning prevails as the best ML method for robotics. The possibility to teach robots how to perform tasks is beyond reason the future of robotics. However, this raises new problems such as sample efficiency, simulation to real-world transition, reward specification and safety.

Simulation environments are an essential strategy to achieve results in a cost-effective and timely way. In chapter 4, a detailed description of the simulators used to design all the agents as well as how their components were developed is presented.

In the next chapter, both theoretical and mathematical foundations behind reinforcement learning are introduced. This includes a description of all the RL algorithms implemented throughout this dissertation.

3

Algorithm-Architecture: Theoretical Foundations

In this chapter, the theoretical foundations of reinforcement learning are presented. A wide variety of RL algorithms with their respective mathematics fundamentals and pseudocodes are demonstrated. The algorithms introduced, cover an extensive range of different methods, from model-based to model-free, and from on-policy to off-policy. Also, a novel algorithm developed by Google Deepmind, introducing continuous control is also explained in this chapter. This algorithm is a significant breakthrough in reinforcement learning in robotics since continuous control is an essential part of nowadays robotics.

3.1 From Nothing to Q-Learning

As referred in subchapter 2.1.3, a reinforcement learning algorithm is a machine learning strategy which can learn without a teacher, just by environment exploration and learning from each episode. The algorithms studied in this dissertation must be compatible with real-world robot implementations. Supervised and unsupervised learning techniques are unfeasible in robot implementations due to the curse of dimensionality, both time-consuming and costly dataset creation and human knowledge bias. The reinforcement learning strategies must be able to solve different tasks just by self-play and discovery of solutions for the problem. Therefore, a detailed analysis of all the mathematics behind these algorithms, as well as different RL algorithms, are presented. In this chapter, theoretical foundations that govern both the fundamentals and the performance of RL algorithms will be evidenced.

3.1.1 Introductory Terminology

As a start point, it is essential to define some new concepts regarding reinforcement learning, some terminology for an essential understanding. Figure 3.1 shows how the different components are linked

and influence each other. For each terminology, two different examples are considered, the first is a videogame, and the second is a robot performing a household task.

- Agent: simulated or physical component that must perform the tasks. What the RL algorithm will be implemented into, for example, the player of the videogame or the robot that is learning how to perform something;
- Environment: the world through which the RL agent moves. It takes the agent's current state and action as an input, and returns as output the agent's reward and its next state, as presented in Figure 3.1, for example, the videogame layout or the room where the robot must perform something;
- State (s): concrete and immediate situation in which the agent finds itself, information the agent knows from the environment to select which action must perform next. For example, in a videogame, the place where the agent is, the place where the opponent agents are, or the robot's position, the target position and which objects it must interact with;
- Action (a): A_t is the set of all possible moves the agent can make. Throughout an episode, an agent has several possible ways to change its state on the environment by performing differently, for example moving up, down, left, right or altering the motor speed, robot movement, turning on or off the robot's outputs;
- Policy (π): the strategy employed by the agent to determine the next action based on the current state. It maps states to actions, and if it is a greedy policy selects the action that promises the highest reward, for example, if an obstacle is presented in the videogame the agent must jump, or if a door is locked a robot must choose another path;
- Reward Signal (R): the feedback by which the success or failure of an agent's actions is measured. For example, in a videogame, if an agent collects a coin it receives a positive reward for the actions that led to the feat, or if a robot bumps into a wall it receives a negative reward to avoid repeating that action;
- Episode: an episode is a sequence of interactions from the initial state to some terminal state. In a videogame, an episode is every time a new game starts until it terminates, or for the robot, it may be an attempt to perform a specific task, until it either is successful or comes across another terminal situation.

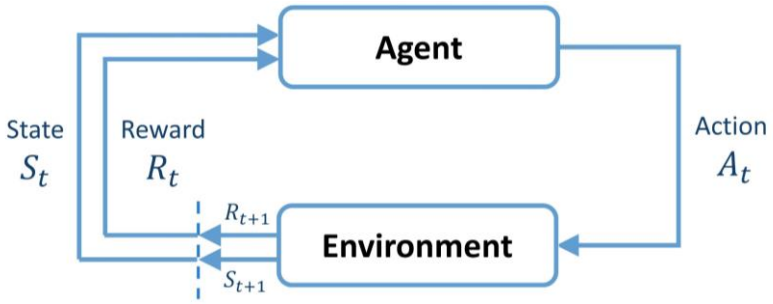


Figure 3.1 - Diagram of the different reinforcement learning components, the interaction between the action and the environment and the state-action-reward (SAR) triple.

3.1.2 Explore-Exploit Dilemma

An essential feature that distinguishes reinforcement learning from other kinds of learning is that it uses training information to evaluate the chosen actions, instead of receiving instructions with the correct answers. This evaluation process creates the need for active exploration to achieve an explicit search for beneficial behaviour. In this subchapter, the evaluation aspect study of reinforcement learning is presented in a simplified setting. The feedback evaluation problem explored is a simplified version of the k-armed bandit problem.

I K-Armed Bandit Problem

The k-armed bandit problem is named by analogy to a slot machine or one-armed bandit, but with k levers instead of one. Each action is similar to pulling a slot machine lever, and the reward is the outcome of the action, the total payoff.

Consider the following problem. A choice among k different actions, levers, must be repeatedly made. Afterwards, a numerical reward chosen from a stationary probability distribution is received, depending on which action from k was selected. The goal is to maximise the total reward over a certain period, for example, N action selections.

For each of the k actions, given that a specific action is selected, an expected or mean reward is given, named the value of that action. The selected action on time step t is denoted as A_t moreover, its corresponding reward R_t. The arbitrary action value is a, additionally, $q_*(a)$ is the expected reward, when a is selected:

$$q_*(a) = \mathbb{E}[R_t | A_t = a] \tag{3.1}$$

If the value of each action were known beforehand, the solution to the k-armed bandit problem would be trivial, always selecting the action with the most significant value. However, it is assumed that the action

values are unknown, but can be estimated. The estimated value of an action a at time step t is denoted as $Q_t(a)$. The goal is to have $Q_t(a)$ as close as possible to $q_*(a)$.

By maintaining an estimate of the action values, at any time step, there is always an action whose estimated value is the highest. These are called greedy actions and selecting these results on the exploitation of the current action value knowledge. If instead a non greedy action is selected, it is considered to be exploration since it enables a non greedy action value estimate improvement. To maximise the instant expected reward, the exploitation method is the best solution, since it chooses the known action capable of returning the higher reward. However exploration may produce a greater total reward in the long run. The following sub-chapters describe some solutions to the exploration-exploitation dilemma.

II Action-Value Methods

A study parcel of methods for estimating the values of actions and using the estimates to make action selections decisions is called action-value methods. One way to estimate the value of an action is to average the received rewards:

$$Q_t(a) = \frac{sum\ of\ rewards\ when\ a\ taken\ prior\ to\ t}{number\ of\ times\ a\ taken\ prior\ to\ t} = \frac{\sum_{i=1}^{t-1} R_i \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}} \quad \textbf{(3.2)}$$

Where $\mathbb{1}_{predicate}$ denotes the random variable that is one if the predicate is true, and zero if it is not. As the denominator tends to infinity, $Q_t(a)$ is expected to converge to $q_*(a)$. For this method, all $Q_0(a)$ are assumed to be zero. A simple action selection rule is to select one of the actions with the highest estimated value, performing a greedy action selection, that can be written as:

$$A_t = argmax_a\ Q_t(a) \quad \textbf{(3.3)}$$

Greedy action selection always exploits current knowledge for immediate reward maximisation. A simple alternative is to act greedier a significant percentage of the time, but every once in a while, with a small probability ε, select a random action among every action with equal probability, independently of the action-value estimates. These methods that use near-greedy action selection rules are named ε-greedy. This method grants that with time, every action is sampled an infinite number of times while still having a mostly greedy action selection.

A variant of the ε-greedy named decreasing ε-greedy consists of an exponential decrease of the greedy selection actions. By starting with a broad exploration value, it explores all the different possibilities at a

higher frequency. However, with time, and as the purpose of exploration decreases, an exponential decay function deviates the interest of this method to exploitation, resisting just a residual value for exploration.

III Incremental Implementation

Even though the action-value method previously presented is a great estimation strategy, it still has its disadvantages regarding computational efficiency. The incremental implementation offers a new strategy with constant memory and constant computation per time step. As a representation simplification, the following demonstration focus on a single action. As notation, R_t denotes the reward received after the t iteration and Q_n denotes the action value estimate after being selected $n - 1$ times, which can be written as:

$$Q_n = \frac{R_1 + R_2 + \cdots + R_{n-1}}{n-1} \tag{3.4}$$

In the previous method, it is needed to maintain a record of all the previously seen rewards. However, the memory and computational requirements would grow over time as more actions are executed. For each additional reward, additional memory storage and computation are necessary to calculate Q_n. The developing of an incremental formula to update averages with small and constant computation is required to process each new reward. With just given Q_{n-1} moreover, the nth reward R_n, the current average can be calculated by:

$$\begin{aligned} Q_n &= \frac{1}{n}\sum_{i=1}^{n} R_i \\ &= \frac{1}{n}\left(R_n + \sum_{i=1}^{n-1} R_i\right) \\ &= \frac{1}{n}\left(R_n + (n-1)\frac{1}{n-1}\sum_{i=1}^{n-1} R_i\right) \\ &= \frac{1}{n}(R_n + (n+1)Q_{n-1}) \\ &= \frac{1}{n}(R_n + nQ_{n-1} + Q_{n-1}) \\ &= \left(1 - \frac{1}{n}\right)Q_{n-1} + \frac{1}{n}R_n \end{aligned} \tag{3.5}$$

The memory required by this implementation is only for Q_n and n and only a small computation. The update rule **(3.5)** can also be written in the general update form:

$$NewEstimate \rightarrow OldEstimate + StepSize[Target - OldEstimate] \quad \textbf{(3.6)}$$

$$Q_n = Q_{n-1} + \frac{1}{n}[R_n - Q_{n-1}] \quad \textbf{(3.7)}$$

The following pseudocode demonstrates a solution to a complete bandit algorithm using incrementally calculated sample averages and ε-greedy.

Algorithm 3.1 – K-armed bandit using ε-greedy and incremental average samples

1: $Init, for\ a = 1\ to\ k$:
2: $Q(a) \leftarrow 0$
3: $N(a) \leftarrow 0$
4: $Loop\ (for\ each\ episode)$:
5: $A \leftarrow \begin{cases} argmax_a Q(a), & with\ probaability\ 1-\varepsilon \\ a\ random\ action, & with\ probaability\ \varepsilon \end{cases}$
6: $R \leftarrow execute(A)$
7: $N(A) \leftarrow N(A) + 1$
8: $Q(A) \leftarrow Q(A) + \frac{1}{N(A)}(R - Q(A))$

IV Optimistic Initial Values

The previous method is dependent on the initial action-value estimate, $Q_0(a)$, considered to be biased by its initial value. Initial action values can be used to encourage exploration. As an alternative of manually setting the initial action value estimate to zero, it is set to a value significantly above the one it is estimated to be around. This optimistic value for the estimate forces the action-value methods to explore. For every action initially selected, the reward received is always less than the initial estimate. Thus the algorithm must switch to other actions since the received rewards will always lower the estimates. All actions are tried a sizable number of times before the estimate values converge.

V Upper Confidence Bound

A particularity of ε-greedy is that the action selection chooses which non-greedy action is tried with no preference regarding non-greediness degree. While exploring, the action selection process is based on the potential for optimality. By taking into consideration, both how close the estimates are to being optimal and the uncertainties of the estimates. An effective way of performing this action selection method is:

$$A_t = argmax_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right] \quad \textbf{(3.8)}$$

As notation, $N_t(a)$ denotes the number of times action a has been selected before time step t. If $N_t(a)$, is zero then a is interpreted as a maximising action. The parameter c that must be higher than zero controls the exploration degree.

The concept of this upper confidence bound action selection is that the square root term works as an uncertainty measure or variance in the action value estimates. For every selection of action a the uncertainty is reduced: as $N_t(a)$ increments thus decreasing the uncertainty term. However, every time a is not chosen, t increases, but $N_t(a)$ does not, therefore the uncertainty estimate increases. The usage of this algorithm results on the following outcomes: a) the increases get smaller over time, b) all actions are eventually selected and actions with lesser value estimates, or recently chosen, are selected with a decreasing frequency over time.

VI Nonstationary Problem

Some reinforcement learning problems have reward probabilities that change over time, meaning they are nonstationary. In such cases, recent rewards should have a more significant weight in comparison to rewards from a long time ago. A solution to this problem is the use of a constant step size parameter. Equation **(3.7)** can be rewritten as:

$$Q_n = Q_{n-1} + \alpha[R_n - Q_{n-1}] \tag{3.9}$$

The step size $\alpha \in [0,1]$ is a constant value. Q_n is a weighted average of past rewards and the initial estimate Q_0:

$$\begin{aligned}
Q_n &= Q_{n-1} + \alpha[R_n - Q_{n-1}] \\
&= \alpha R_n + (1-\alpha)Q_{n-1} \\
&= \alpha R_n + (1-\alpha)[\alpha R_{n-1} + (1-\alpha)Q_{n-2}] \\
&= \alpha R_n + (1-\alpha)\alpha R_{n-1} + (1-\alpha)^2 Q_{n-2} \\
&= \alpha R_n + (1-\alpha)\alpha R_{n-1} + (1-\alpha)^2 \alpha R_{n-2} + \cdots \\
&\qquad + (1-\alpha)^{n-1}\alpha R_1 + (1-\alpha)^n Q_0 \\
&= (1-\alpha)^n Q_0 + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} R_i
\end{aligned} \tag{3.10}$$

The weight $\alpha(1-\alpha)^{n-i}$ given to the reward R_i, depends on how many rewards ago $n-i$, was observed. The quantity $1-\alpha$ is less than one, thus the weight given to R_i decreases as the number of rewards increases. The weight decays exponentially according to the exponent on $1-\alpha$.

It is usually suitable to vary the step size parameter from step to step. $\alpha_n(a)$ denotes the step size parameter used to calculate the received reward after the nth selection of action a. As previously demonstrated $\alpha_n(a) = \frac{1}{n}$ is the sample average method, that is guaranteed to converge because of the true action values by the law of large numbers. However, convergence is not certain for every choice of $\alpha_n(a)$. The conditions to guarantee convergence that results from a stochastic approximation theory are both:

$$\sum_{n-1}^{\infty} \alpha_n(a) = \infty \tag{3.11}$$

$$\sum_{n-1}^{\infty} \alpha_n^2(a) < \infty \tag{3.12}$$

Condition **(3.11)** is required to overcome any initial conditions or random fluctuations eventually, through large steps. Condition **(3.12)** guarantees that eventually, the steps become small enough to assure convergence.

3.1.3 Markov Decision Processes

In this subchapter, a formal problem of finite Markov Decision Processes is described. This formal problem, evaluates feedback, similar to the bandits, previously described, but with a new feature: the capability of choosing different actions in different situations. MDPs are a conventional formalisation of sequential decision making, where actions impact not only on immediate rewards but also on long-time rewards. Therefore, a crucial part of the MDPs is the trade-off between instant and delayed rewards. Opposing to the bandit problem, in this subchapter, the $q_*(a)$ of each action a in each state s, will be estimated, or the value $v_*(s)$ of each state, given optimal action is selected. These state-dependent quantities are crucial to assign credit for long-term outcomes to a specific action selection. Also, in this subchapter, a mathematical reinforcement learning introduction to elements, such as returns, value functions and the Bellman equations, is presented.

I Agent-Environment Interaction and Markov Property

The agent and the environment interact with each other on a sequence of discrete time steps, $t = 0,1,2, \dots$. At each time step t the agent receives from the environment a state, $S_t \in S$, and selects an action $A_t \in A$. On the following time step, as a consequence of the previous time step action, the agent receives a numerical reward, $R_{t+1} \in R \in \mathbb{R}$ and the current and updated state S_{t+1}.The agent with the MDP creates a sequence of states, actions and rewards similar to:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, S_3, A_4, \dots \tag{3.13}$$

When working with a finite MDP, the sets of states, actions and rewards (SAR) have a finite number of elements. The random variables R_t and S_t, have a discrete probability distribution dependent on the next state and action. Thus, for specific values of these variables, $s' \in S$ and $r \in R$, the probability of occurring at time step t, given previous state and action specific values. The following function, p defines MDP's dynamics:

$$p(s', r \mid s, a) = p\{S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a\} \tag{3.14}$$

In an MDP, the probabilities characterise the environment dynamics entirely. Thus, the probabilities of each possible value for S_t and R_t only depends on the directly preceding state and action S_{t-1} and A_{t-1}. This property is better viewed as a state restriction, not on the decision process. To surpass the property, the current state must now include information on all the noteworthy interactions between the agent and the environment. If the state possesses this past information, it is known to have the Markov property.

From the MDPs dynamics function p, it is possible to calculate all the other information about the environment, for example, the state-transition probabilities:

$$p(s' \mid s, a) = p\{S_t = s' \mid S_{t-1} = s, A_{t-1} = a\} = \sum_{r \in R} p(s', r \mid s, a) \tag{3.15}$$

The expected rewards for state-action pairs as a two-argument function can also be calculated as:

$$r(s, a) = \mathbb{E}[R_t \mid S_{t-1} = s, A_{t-1} = a] = \sum_{r \in R} r \sum_{s' \in S} p(s', r \mid s, a) \tag{3.16}$$

Also, the expected rewards for the (s, a, s') triplet is a three-argument function calculated as:

$$r(s, a, s') = \mathbb{E}[R_t \mid S_{t-1} = s, A_{t-1} = a, S_t = s] = \sum_{r \in R} r \frac{p(s', r \mid s, a)}{p(s' \mid s, a)} \tag{3.17}$$

In Figure 3.2, an example of a state transition diagram is presented. The larger circles, denoted as S represent the possible states, the smaller circles marked as a denote the possible actions, the large arrows denote the terminal states and respective reward, and the smaller arrows represent the state probabilities.

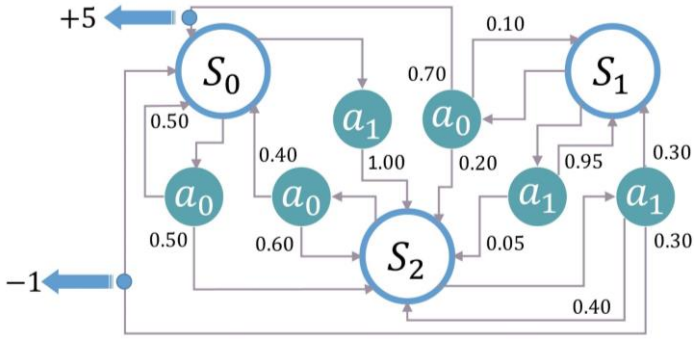


Figure 3.2 - State transition diagram, with three states, two actions, two terminal states and the state transition probabilities.

II Goals and Rewards

At each time step, a reward signal $R_r \in \mathbb{R}$ is sent from the environment to the agent. The agent's primary goal is to maximise the overall amount of reward it receives. Not only maximises the immediate reward but also the cumulative reward in the long run. Reward signals used as a goal formalisation are a distinctive feature of reinforcement learning. The design of goals is not trivial, and even though it usually demands a rigorous study, it is proved to be considerably flexible and widely applicable. Some hypothetical examples of how it can be used are presented for a more fundamental understanding:

1. For a robot to learn how to move, rewards on each time step can be provided reliant on the task at hand, for example, proportional to the robot's motion.
2. For a robot to escape a periodic maze, the reward can be a negative constant value for every time step needed to escape. This method encourages the agent to perform the task on minimal steps for a minor negative reward.
3. For a robot to collect garbage for recycling, another strategy might be to present null rewards except when successfully picking up garbage.
4. Negative rewards are also an essential part as they present strategies to avoid situations that the agent must avoid. For example: when a navigation robot hits a wall or breaks some part.

In the previous examples and many other reward implementations, it is crucial that the agent always learns how to maximise its rewards. The system that provides the rewards must be programmed in such a way to maximise the overall reward results on achieving the goal. The problem with reward specification is that many times the agent gets stuck in sub-optimal situations instead of focusing on the ultimate goal.

For example, in a game of chess, an agent should not be rewarded for destroying opponents pieces but for winning a game. These are different things and may compromise the agent's overall performance.

III Returns and Episodes

As previously discussed, the agent's goal is to maximise the cumulative reward it receives in the long run. The interest of an agent must reside on total future reward, everything from the current time step onwards. For the agent to learn, it needs to maximise the expected return, where G_t is the return. A simple return definition can be the sum of future rewards:

$$G_t = R_{t+1} + R_{t+2} + \cdots + R_T \tag{3.18}$$

Time step T represents a terminal state. Each episode must end in a specific state, known as a terminal state, which leads to the end of the episode and consequently the start of the next episode. Episodes are entirely independent of each other. The end of an episode does not affect in any way the start of the next episode. Thus the episodes can all be considered to finish in the same terminal state, but with different rewards for different outcomes. This kind of task is known as an episodic task. However, in many cases, such as control tasks, the agent-environment interaction cannot be broken logically into identifiable episodes but goes on continually. These are called continuous tasks. Some examples of continuous tasks are process-control and robotics tasks. It is infeasible to apply **(3.18)** to continuous tasks since time step T is unkown and could result in $T = \infty$ and consequently $G_t = \pm\infty$.

The use of a discount factor introduces the concept of discounting rewards. As a return in discounting rewards, the agent must select an action so that the future discounted rewards sum received is maximised:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{3.19}$$

γ denotes the discount factor, $0 \leq \gamma \leq 1$. The discount factor determines the value that future rewards are worth on the current time step. A reward obtained k time steps in the future is worth γ^{k-1} times what would be worth if it were to be received in the present time step. Returns at successive time steps are related in an essential way for reinforcement learning algorithms to be able to learn:

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma^2 R_{t+3} + \cdots) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{3.20}$$

IV Policies and Value Functions

Value functions are used in reinforcement learning algorithms to estimate either the value of the agent to be in a given state or the value to perform a specific action in a given state. The concept of value or how good, is defined in terms of expected returns. The rewards an agent may expect are a consequence of the actions it will perform. Value functions are defined by specific ways of acting, named policies.

A policy π is a map from states to probabilities of selecting each possible action. If an agent follows a policy π at time step t, then $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$. The value function $v_\pi(s)$, of a state s under a policy π, is the expected reward when starting in s and following π. For MDPs, v_π can be formalised by:

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right] \tag{3.21}$$

In the same way, the action-value $q_\pi(s, a)$ can also be defined by taking an action a in state s under a policy π:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a\right] \tag{3.22}$$

A central value function has a recursive property. For any policy π and state s, it is possible to iteratively calculate the s value from the value of its possible previous states:

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[G_t \mid S_t = s] \\
v_\pi(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\
v_\pi(s) &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_t = s']\right] \\
v_\pi(s) &= \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a)[r + \gamma v_\pi(s')]
\end{aligned} \tag{3.23}$$

Equation **(3.23)** is the Bellman equation for v_π. The Bellman equation states the association between the value of a specific state and the values of its successor states. Figure 3.3 shows a v_π visual representation diagram, where the white circles represent a state s and the blue circles represent the possible actions a. Based on the policy π, from state s, the agent can take an action a. The environment would respond with a reward r and the next state s', depending on the dynamics given by p.

The Bellman equation **(3.23)** returns an average over all the possibilities, weighted on their occurring probability, and states that the initial state-value is equivalent to the expected next state discounted value, plus the expected reward.

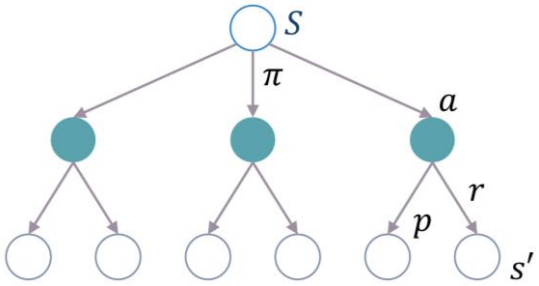


Figure 3.3 - v_π diagram, with states, policy, actions, rewards and probabilities.

V Optimal Policy and Optimal Value Function

To solve a reinforcement learning problem, an algorithm must discover a policy that achieves a considerable reward over the episode's long run. Value functions express a partial ordering over policies. A policy π_a is described as a superior policy compared to π_b if its expected return is greater or equal for all states, $v_{\pi_a}(s) > v_{\pi_b}(s)$, for all $s \in S$. The optimal policy denoted as π_*, represents a policy for which no other is superior. Optimal policies may not be unique and may share optimal state-value functions, denoted as v_*, that can be defined as:

$$v_*(s) = max_\pi v_\pi(s), for\ all\ s \in S \quad \textbf{(3.24)}$$

Optimal policies may share optimal action-value functions, denoted as q_*, defined as:

$$q_*(s,a) = max_\pi q_\pi(s,a), for\ all\ s \in S\ and\ all\ a \in A \quad \textbf{(3.25)}$$

Since equation **(3.25)** returns the expected reward for doing action a in state s following an optimal policy, it can be rewritten as:

$$q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \quad \textbf{(3.26)}$$

3.1.4 Dynamic Programming

Dynamic Programming denotes a collection of algorithms that are used to calculate optimal policies when given an environment complete model as an MDP. Dynamic Programming algorithms are known to be a limited solution to reinforcement learning because they need a perfect model and significant computational expense. However, it is an important starting point for other algorithms and as a theoretical foundation.

The central concept of Dynamic Programming is the application of value functions to structure and organise the search for superior policies. As previously demonstrated, once the optimal value functions have been found the optimal policy that satisfies the Bellman optimality equations are:

$$\begin{aligned} v_*(s) &= max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')] \end{aligned} \tag{3.27}$$

$$\begin{aligned} q_*(s, a) &= \mathbb{E}[R_{t+1} + \gamma max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r|s, a)[r + \gamma max_{a'} q_*(s', a')] \end{aligned} \tag{3.28}$$

DP strategies are attained by varying Bellman equations **(3.27)** and **(3.28)** into assignments.

I Iterative Policy Evaluation

Policy evaluation also referred to as the prediction problem, consists of calculating the state-value function v_π for an arbitrary policy π. It can be demonstrated from the value function definition and equation **(3.20)** to equation **(3.23)**. If the environment dynamics are entirely acknowledged, the algorithm possesses a system of $|S|$ linear equations and $|S|$ unknown variables. The algorithm consists of a sequence of approximate value functions, where the initial approximations must be chosen arbitrarily, except for terminal states, and each successive approximation is found using the Bellman equation for v_π in **(3.23)** as an update rule:

$$\begin{aligned} v_{k+1}(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \end{aligned} \tag{3.29}$$

The $\{v_k\}$ sequence is shown to converge to v_π when $k \to \infty$ under the same conditions that guarantee the existence of v_π. This algorithm is named iterative policy evaluation.

Iterative Policy Evaluation applies a similar operation to each state s. By replacing the previous s value with a novel value that in turn is calculated from the successor state s former value and the expected instant rewards, along with all the one-step transitions probable under the evaluated policy, achieves the expected update.

II Policy Improvement

In order to find superior policies, it is necessary to calculate the value function for a policy. If a value function v_π is determined for an arbitrary deterministic policy π, it is required to know whether a change

on the policy must be considered to deterministically chose an action $a \neq \pi(s)$. To know how prejudicial or beneficial it is to select a new policy, a strategy to select a different action must be developed:

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')] \end{aligned} \tag{3.30}$$

The crucial factor rests on if equation **(3.30)** is higher or lower than $v_\pi(s)$. If it is higher, meaning it is a better option to select the action a in state s and afterwards follow policy π, then it is better still to choose a everytime s is come across. It is expected for the algorithm to consider changes in all states and all possible actions and therefore selecting at each state the superior action according to $q_\pi(s,a)$, considering a new greedy policy π':

$$\begin{aligned} \pi'(s) &= arg\max_a q_\pi(s,a) \\ &= arg\max_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \\ &= arg\max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')] \end{aligned} \tag{3.31}$$

After a one-step lookahead, the greedy policy selects the action that will perform best in the short term. To make a new improved policy from a previous version allowing a greedy strategy concerning, the original policy value function is known as policy improvement. If the novel greedy policy π' is of similar value to the old policy π, then $v_\pi = v_{\pi'}$ and from equation **(3.31)** it respects for all $s \in S$:

$$\begin{aligned} v_{\pi'}(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_{\pi'}(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_{\pi'}(s')] \end{aligned} \tag{3.32}$$

However, this is similar to the Bellman optimality equation **(3.27)** and consequently v_π must be v_* and both π and π' have to be optimal policies. Thus, the Policy Improvement has to return an improved policy except when the original policy is already optimal.

III Policy Iteration

After the improvement of a policy π by using v_π to create a new and improved policy π', $v_\pi{}'$ can be calculated and improved once again to generate an even better policy π''. This process can be iteratively performed until an optimal policy and value function are achieved:

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \dots \xrightarrow{I} \pi_* \xrightarrow{E} v_* \tag{3.33}$$

Where $\xrightarrow{E}$ denotes a policy evaluation, and $\xrightarrow{I}$ a policy improvement. This method for finding an optimal policy is called policy iteration.

IV Value Iteration

Even though policy iteration solves the task at hand, it has a significant drawback since it consists of an iterative algorithm inside an iterative algorithm. It is computationally expensive and requires multiple sweeps through the state set. Policy evaluation steps end when v_π converges, but there is a point before the convergence that the resulting greedy policy no longer changes. Value Iteration can translate an update operation that combines the policy improvement and truncated policy evaluation steps:

$$\begin{aligned} v_{k+1}(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \end{aligned} \tag{3.34}$$

For random v_0, the sequence $\{v_k\}$ converges to v_* under similar conditions that assure the existence of v_*.

3.1.5 Monte Carlo Method

In the previous subchapter concerning Dynamic Programming, a complete environment model was necessary to perform both prediction and control problem solving. However, this can be considered to go against reinforcement learning philosophy, which is to learn from experience. Monte Carlo (MC) methods only require experience in the form of samples of sequences of states, actions and rewards from actual or simulated interactions with the environment. The ability to learn from experience is astonishing since it requires no prior knowledge of the environment dynamics.

Monte Carlo algorithms perform the value estimates as well as the policy changes at the end of each episode, working offline. The term "Monte Carlo" is generally used for estimation methods whose operation involves a significant arbitrary component.

I Monte Carlo Prediction

Monte Carlo methods purpose is to learn the state-value function for a given policy. To learn from experience is to average the returns observed after visits to a specific state. With time, as additional returns are observed, the average is expected to converge to the expected value. This concept underlies all Monte Carlo methods.

Thus, to estimate $v_\pi(s)$, the value of a state s under policy π, it is assumed the episode's collection obtained by following π and passing though s. An occurrence of a state s in a specific episode is commonly known as a visit to s. It is possible to have multiple visits to a specific state s in an explicit episode. This aspect introduces two different Monte Carlo methods, first visit MC and every visit MC, which have different theoretical properties. As the name indicates first visit MC estimates $v_\pi(s)$ as the returns average following first visits to s, and every visit MC averages the returns following all visits to s.

II Monte Carlo Control

A Monte Carlo estimation method is a solution to the control problem, a strategy to approximate optimal policies. In Generalised Policy Iteration, an approximate policy as well as an approximate value function are stored. The value is consistently updated to better approximate the current policy value function. The policy is also consistently improved concerning the current value function. In Figure 3.4, a visual diagram of both these updates is presented.

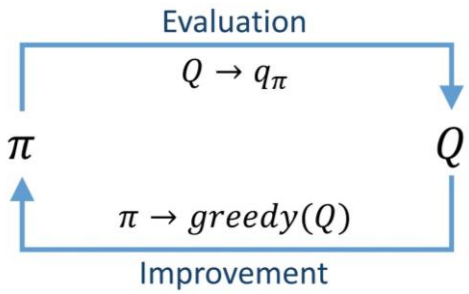


Figure 3.4 - Evaluation and improvement update of a Monte Carlo control problem.

In Monte Carlo version of classical policy iteration, complete steps of policy evaluation and policy improvement, are alternately performed. Starting from a random policy π_0 and, finishing with both the optimal policy and optimal action-value function:

$$\pi_0 \overset{E}{\rightarrow} q_{\pi_0} \overset{I}{\rightarrow} \pi_1 \overset{E}{\rightarrow} q_{\pi_1} \overset{I}{\rightarrow} \pi_2 \overset{E}{\rightarrow} \dots \overset{I}{\rightarrow} \pi_* \overset{E}{\rightarrow} q_* \quad \textbf{(3.35)}$$

Policy Evaluation is performed as previously discussed. However, in this case a considerable number of episodes are performed with the approximate action-value function asymptotically approaching the correct function. The experienced episodes make use of exploring starts (ES), as a source of initial randomness.

To perform Policy Improvement, it needs to have a greedy policy concerning the present value function. The action-value function allows a detachment from the need to have a model in order to perform a greedy policy. For an action-value function q, the greedy policy can be translated to:

$$\pi = arg\max_a q(s,a) \quad \textbf{(3.36)}$$

In Monte Carlo policy evaluation, it is natural to alter between evaluation and improvement on an episode-by-episode basis. Following the episode, the observed returns are worked on policy evaluation and consequently, the policy is improved for all states s experienced in the episode. A complete straightforward algorithm named Monte Carlo with Exploring Starts is given in pseudocode in the following algorithm.

Algorithm 3.2 – Monte Carlo with Exploring Starts, for estimating $\pi \approx \pi_*$

Init:
  $\pi(s) \in A(s), (randomly), for\ all\ s \in S$
  $Q(s,a) \in \mathbb{R}, (randomly), for\ all\ s \in S, a \in A(s)$
  $Returns(s,a) \leftarrow empty\ list, for\ all\ s \in S, a \in A(s)$
Loop $(for\ each\ episode)$:
  $Choose\ s_0 \in S, a_0 \in A(s_0)\ randomly\ that\ all\ pairs\ have\ probability > 0$
  $Generate\ an\ episode\ from\ s_0, a_0, following\ \pi: s_0, a_0, r_1, \dots, s_{T-1}, a_{T-1}, r_T$
  $G \leftarrow 0$
  $Loop\ (for\ each\ episode's\ time\ step), t = T-1, T-2, \dots, 0$:
    $G \leftarrow \gamma G + r_{t+1}$
    $Unless\ the\ pair\ s_t, a_t\ appears\ in\ s_0, a_0, s_1, a_1, \dots, s_{T-1}, a_{T-1}$
      $Append\ G\ to\ Returns(s_t, a_t)$
      $Q(s_t, a_t) \leftarrow average(Returns(s_t, a_t))$
      $\pi(s_t) \leftarrow argmax_a Q(s_t, a_t)$

III Monte Carlo without Exploring Starts

Even though exploring starts is a solution to consider when solving the control problem, it has its disadvantages. This algorithm's major drawback regards environments where the full state space and action space is unknown.

Two different approaches are presented: on-policy methods and off-policy methods. On-policy methods try to evaluate or improve the policy used to make decisions, contrary to off-policy methods which evaluate or improve a policy different from the one used to generate the data. MC with ES is an example of an on-policy method. In on-policy control methods, the policy π is known as soft. Meaning that $\pi(a|s) > 0$ for all $s \in S$ and all $a \in A$, but will progressively get closer to a deterministic optimal policy.

One way to implement this on-policy method is with ε-greedy policies (described in subchapter 3.1.2). This represents the choice of action with probability ε to be an action not with a maximal estimated action-value. All non-greedy actions have a low selection probability of $\varepsilon/|A(s)|$. However, without exploring starts, it is not possible to improve the policy only by performing greedier since it would avoid supplementary exploration of non greedy actions. Thus, since General Policy Iteration (GPI) does not

require an utter greedy policy, but only that it follows a greedy policy such as ε-greedy. The complete algorithm is demonstrated in the following pseudocode.

Algorithm 3.3 – Monte Carlo without Exploring Starts, for estimating $\pi \approx \pi_*$

Algorithm parameter: *small* $\varepsilon > 0$
Init:
  $\pi \leftarrow$ *a random* ε *soft policy*
  $Q(s,a) \in \mathbb{R}$ *(randomly), for all* $s \in S, a \in A(s)$
  $Returns(s,a) \leftarrow$ *empty list, for all* $s \in S, a \in A(s)$
Loop (for each episode):
  Generate an episode following $\pi: s_0, a_0, r_1, \dots, s_{T-1}, a_{T-1}, r_T$
  $G \leftarrow 0$
  Loop (for each episode's time step), $t = T-1, T-2, \dots, 0$:
    $G \leftarrow \gamma G + r_{t+1}$
    Unless the pair s_t, a_t *appears in* $s_0, a_0, s_1, a_1, \dots, s_{T-1}, a_{T-1}$
      Append G *to* $Returns(s_t, a_t)$
      $Q(s_t, a_t) \leftarrow average(Returns(s_t, a_t))$
      $a^* \leftarrow argmax_a Q(s_t, a)$
      For all $a \in A(s_t)$
        $$\pi(a|s_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|A(s_t)|, & if\ a = a^* \\ \varepsilon/|A(s_t)|, & if\ a \neq a^* \end{cases}$$

The Policy Improvement Theorem guarantees that a ε-greedy policy π' concerning q_π is an improvement over any ε-soft policy. The Policy Improvement Theorem conditions apply because for any $s \in S$:

$$\begin{aligned} q_\pi(s, \pi'(s)) &= \sum_a \pi'(a|s)\, q_\pi(s,a) \\ &= \frac{\varepsilon}{|A(s)|} \sum_a q_\pi(s,a) + (1-\varepsilon) max_a q_\pi(s,a) \\ &\geq \frac{\varepsilon}{|A(s)|} \sum_a q_\pi(s,a) + (1-\varepsilon) \sum_a \frac{\pi(a|s) - \frac{\varepsilon}{|A(s)|}}{1-\varepsilon} q_\pi(s,a) \\ &= \frac{\varepsilon}{|A(s)|} \sum_a q_\pi(s,a) - \frac{\varepsilon}{|A(s)|} \sum_a q_\pi(s,a) + \sum_a \pi(s|a)\, q_\pi(s,a) \\ &= v_\pi(s) \end{aligned} \quad \textbf{(3.37)}$$

3.1.6 Temporal Difference Learning

Temporal Difference Learning is considered by the reinforcement learning scientific community as a central and novel strategy of reinforcement learning. TD learning combines ideas from both Dynamic Programming and Monte Carlo methods. From MC methods, TD takes advantage of the learning from

raw experience strategy, without the need of an environment dynamics model. From DP methods, TD update estimates based on the learned estimates, without waiting for an outcome. This process is known as bootstrapping. These three methods can be combined in many ways and are a recurrent theme in reinforcement learning theory.

This subchapter will initially present the theoretical foundations on the policy evaluation or prediction problem, which is the problem of estimating the value function v_π for a policy π. Secondly, for the control problem or finding the optimal policy, the three methods use some variant forms from generalised policy iteration. The alternatives of GPI are mainly variations on trying to solve the prediction problem.

I Temporal Difference Prediction

Both TD and MC methods use the agent's experience in the environment to solve the prediction problem. By experiencing following a policy π, these methods update their estimate V of v_π for every nonterminal experienced state S_t. The main difference between both methods is that MC must wait until the return G_t after a visit is known, at the end of an episode, then use that return as a target for $V(S_t)$, as demonstrated in the following equation:

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)] \tag{3.38}$$

Whereas TD methods must only wait for the following time step $t + 1$ and can already make a suitable update using both the observed reward R_{t+1} and the estimate $V(S_{t+1})$:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{3.39}$$

The target for the MC update is G_t, while the target for the TD update is $R_{t+1} + \gamma V(S_{t+1})$. This target update is known as TD(0). In Figure 3.5, a display of the differences in how MC methods (a) and TD methods (b) solve the prediction problem.

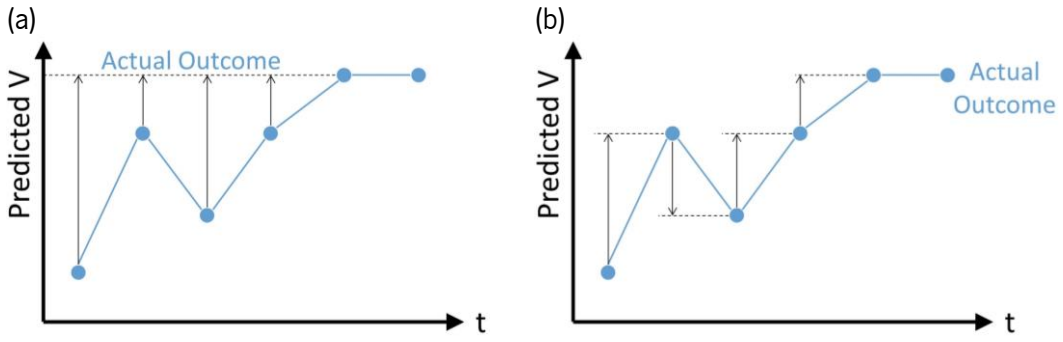


Figure 3.5 - Differences regarding the prediction problem for Monte Carlo methods (a) and Temporal Difference methods (b).

The following algorithm provides the pseudocode for a tabular TD(0) to estimate v_π.

Algorithm 3.4 – Temporal Difference (0) for estimating v_π

1: $Input: the\ policy\ \pi\ to\ be\ evaluated$
2: $Algorithm\ parameter: step\ size\ \alpha \in [0,1]$
3: $Init\ V(s), for\ all\ s \in S, randomly\ except\ for\ V(terminal) = 0:$
4: $Loop\ (for\ each\ episode):$
5: $\quad Initi\ s$
6: $\quad Loop\ (for\ each\ episode's\ time\ step):$
7: $\quad\quad a \leftarrow action\ given\ by\ \pi\ for\ s$
8: $\quad\quad Take\ action\ a, observe\ r, s'$
9: $\quad\quad V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$
10: $\quad\quad s \leftarrow s'$
11: $\quad until\ s\ is\ terminal$

The part in brackets in the TD(0) update, equation **(3.39)**, is known as the TD error, that is the difference between the estimated value of S_t and the improved estimate $R_{t+1} + \gamma V(S_{t+1})$, denoted as δ_t:

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \qquad \textbf{(3.40)}$$

At each time step, the TD error is the error in the estimate made at that time. Since the TD error depends on the next state and the next reward, it is not available until the following time step. This means δ_t is the error in $V(S_t)$ available at time step $t + 1$.

II Advantages of TD Prediction Methods

Temporal Difference methods have an advantage over DP methods, as neither it requires an environment model nor the reward and next-state probability distributions. TD methods diverge from DP as a model-free method whose updates are based on other estimates. From the process of guessing from a guess, the correct terminology used is the method bootstraps.

The advantage over MC methods is that in TD learning the methods are naturally implemented as an online fully incremental strategy. In Monte Carlo methods, the algorithm can only correctly update the value function when the episode finishes, as only then receives the full return function. In TD Learning this process can be performed every time step. This is a critical consideration as some applications have considerably long episodes, and suspending all the learning until the episode's end substantially slows down the process. A different range of applications, named continuing tasks must also be solved with TD methods as there are no episodes at all. This is in some cases the instance for robotics reinforcement learning. The TD methods are less susceptible to these problems as they learn every time step regardless of the following actions taken. In Figure 3.6, the differences between DP (a), MC (b) and TD (c) can be

visualised in the diagram. The white circles denote the states and the blue circles denote the actions. The blue cloud over the circles denotes the required information to adjust S_t.

When studying and comparing TD Learning to DP and MC methods, two questions tend to arise. Firstly for TD Learning method, if the convergence to the correct answer is guaranteed just by learning from estimates of estimates without waiting for an actual outcome.

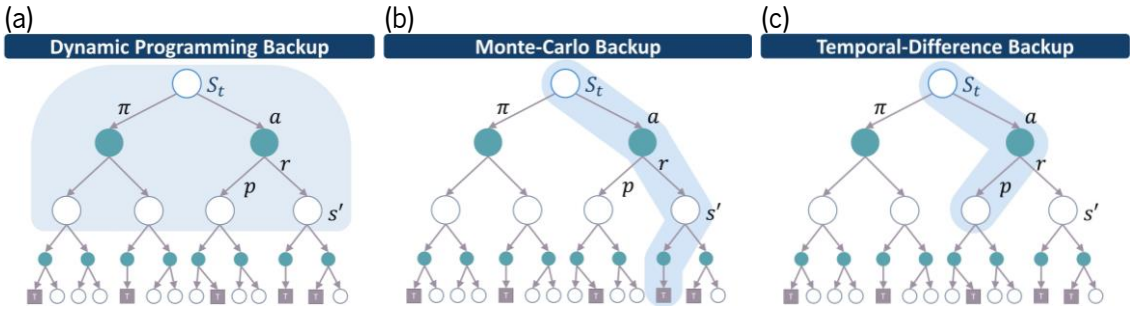


Figure 3.6 - Differences between Dynamic Programming (a), Monte Carlo (b), and Temporal Difference (c) learning.

For a fixed policy π, TD(0) has been proved to converge to v_π for an adequately small constant step-size parameter, and with a full probability, if the step-size parameter decreases as the previously mentioned stochastic approximation conditions **(3.11)** and **(3.12)**. Secondly, since both TD and MC methods converge asymptotically to the correct predictions, the issue lays on if there is a way to discover which learns faster by making more efficient use of the same limited data. At the time, this is considered to be an open-question since it has not been proven mathematically that one of the methods converges faster. However, in practice TD methods tend to converge faster than regular MC methods on stochastic tasks.

III Solutions to TD Control Problem

After a TD prediction methods analysis, the present and the following sections will demonstrate two different solutions to the control problem. The goal, as in previous subchapters is to follow the outline of GPI but using TD methods for the estimation or prediction part. As MC methods, the trade-off between exploration and exploitation must be considered. The two discussed approaches consist of on-policy and off-policy approaches. The main difference between the two algorithms is that in an on-policy solution, like SARSA, the learned policy value is carried out by the agent, including the exploration steps, whereas in an off-policy solution like Q-Learning, the action chosen is independent of the policy being followed.

In TD Learning methods, contrary to MC, the state-value function $V(S_t)$ is replaced by the action-value function $Q(S_t, A_t)$. For on-policy methods, $q_\pi(s, a)$ must be estimated for the current behaviour policy π, all states s and all actions a. As previously described, an episode consists of an alternating sequence of states and state-action pairs, like the one presented in Figure 3.7.

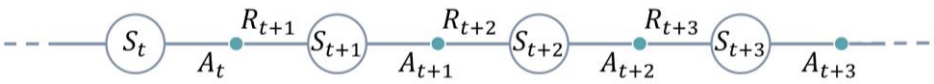


Figure 3.7 - Sequence of alternating states and state-action pairs.

IV SARSA: On-Policy TD Control

In TD Learning, the transitions considered are from one state-action pair to another and learned the values of state-action pairs. These cases, for state-action instead of action-value, are identical since both are Markov chains with a rewarding process. The deductions which assure the convergence of state-values under TD(0) may also apply to the corresponding algorithm for action-values:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \quad \textbf{(3.41)}$$

The update equation **(3.41)** is performed after every transition from a nonterminal state S_t. In case a terminal state is presented in S_{t+1} then $Q(S_{t+1}, A_{t+1})$ is zero. This rule employs every element from the events quintuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ that comprises the transitions from one state-action pair to the following one. The need to use quintuple's five components rises the name SARSA.

As it is typical in on-policy methods, to solve the control problem, $q_\pi$is continually estimated for the behaviour policy π and simultaneously change π towards greediness concerning q_π. SARSA control algorithm pseudocode is presented next.

Algorithm 3.5 – SARSA (on-policy Temporal Difference Control)

Algorithm parameter: *step size* $\alpha \in [0,1]$, *small* $\varepsilon > 0$
Init $Q(s,a)$, *for all* $s \in S, a \in A$, *randomly except for* $Q(terminal, .) = 0$
Loop (*for each episode*):
  Init s
  Choose a *from* s *using policy derived from* Q
  Loop (*for each episode's time step*):
    Take action a, *observe* r, s'
    Choose a' *from* s' *using policy derived from* Q
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s', a') - Q(s,a)]$
    $s \leftarrow s', a \leftarrow a'$
  until s *is terminal*

SARSA convergence properties depend on how the policy depends on Q, as ε-greedy or ε-soft policies can be used as exploration methods. SARSA converges to an optimal policy and action-value function if all the state-action pairs are experienced an infinite amount of times and the policy converges to the greedy policy.

V Q-Learning: Off-Policy TD Control

Q-Learning [45], is one of the early breakthroughs in reinforcement learning. It was a significant discovery as it introduced an off-policy Temporal Difference control algorithm, defined by:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \quad \textbf{(3.42)}$$

In Q-Learning, the learned action-value function, Q, directly approximates to the optimal action-value function q_*, independently of the policy being followed. This process simplifies the algorithm analysis, and it is proven to enable early convergence to the optimal action-value function. However, the policy still affects this algorithm as it determines which state-action pairs are both visited and updated. Moreover, the only requirement for natural convergence is that all action-value pairs are continuously updated. The Q-learning control algorithm pseudocode is shown next.

Algorithm 3.6 – Q-Learning (off-policy Temporal Difference Control)

Algorithm parameter: *step size* $\alpha \in [0,1]$, *small* $\varepsilon > 0$
Init $Q(s,a)$, *for all* $s \in S, a \in A$, *randomly except for* $Q(terminal, .) = 0$
Loop (*for each episode*):
  Init s
  Loop (*for each episode's time step*):
    Choose a *from* s *using policy derived from* Q
    Take action a, *observe* r, s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_a Q(s', a') - Q(s,a)]$
    $s \leftarrow s'$
  until s *is terminal*

Since it is an off-policy algorithm, it allows the agent to perform any action rather than the one best suited by the policy π. Instead, it updates the state-action function regarding the most valuable action, despite following a different policy. This algorithm allows the agent to learn while experiencing or performing an episode and despite the which, actions are taken during the episode.

3.2 Approximation Methods

Linear models provide convergence guarantees and are in practice very efficient in terms of data as well as computation wise. The efficiency critically depends on how the states are represented in terms of features. Feature selection is an essential task as it adds a prior domain knowledge to reinforcement learning systems. The features should correspond to the state-space aspects where a generalisation process may be appropriate. For examples when valuing geometric objects, some useful information would be to set different features for each shape, colour, size or function. When valuing states of a mobile

robot, some examples of a natural set of features are locations, sensor readings, internal processes and battery power. A linear form limitation is that it can not consider any interactions between features, such as the occurrence of feature i is only a positive case in the absence of feature j.

In this dissertation, the only two presented approximation method is a nonlinear function approximator named artificial neural networks and a radial basis function (RBF), since these are the only used methods in the reinforcement learning algorithms implemented.

3.2.1 Nonlinear Function Approximator: Artificial Neural Networks

Artificial Neural Networks (ANNs) are broadly used for nonlinear function approximation. An ANN is a network of interconnected units that has some neuron's properties reproducing the nervous systems main components. ANNs have a long history, with the latest advances in training deeply-layered ANNs, named as Deep Learning, being responsible for some of the most impressive abilities across all machine learning algorithms, including reinforcement learning.

In Figure 3.8, a visual diagram of a generic feedforward ANN, meaning a network with no loops within the network by which a unit's outputs can influence its input. The network presented in Figure 3.8 has an output layer, made of two output units, one input layer with three input units and two hidden layers, which are intermediate layers between the input and output layers. For each link represented by the arrows, is an associated weight value. The weight corresponds to a synaptic connection in a real neural network. If an ANN has at least a connection from an output unit to input or intermediate unit, it is considered to be a recurrent neural network. Even though feedforward and recurrent neural networks are both extremely used in reinforcement learning, in this subchapter as well as the practical implementation of this dissertation, only the feedforward neural networks are discussed and implemented.

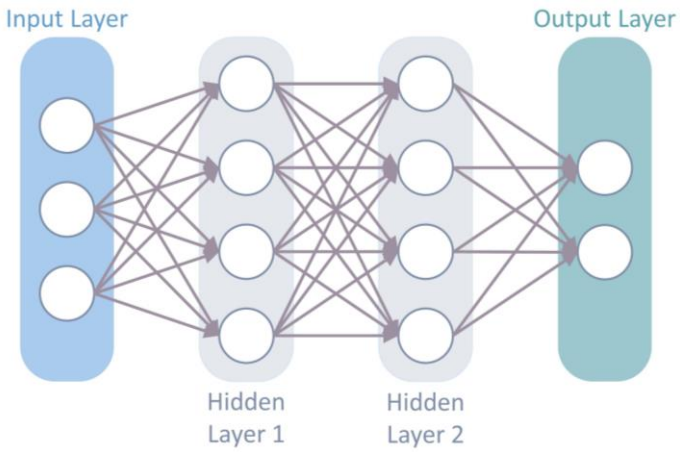


Figure 3.8 - Generic feedforward ANN with three input units, two output units, and two hidden layers with four units each.

The units represented as circles in Figure 3.8 and described in Figure 3.9 are usually semi-linear units, which means that they calculate a weighted sum of the input signal and then apply the result to a nonlinear function, named the activation function, to produce the unit's output. Different activation functions are commonly used, like sigmoid, logistic functions, RELU, inverse tangent and SoftMax.

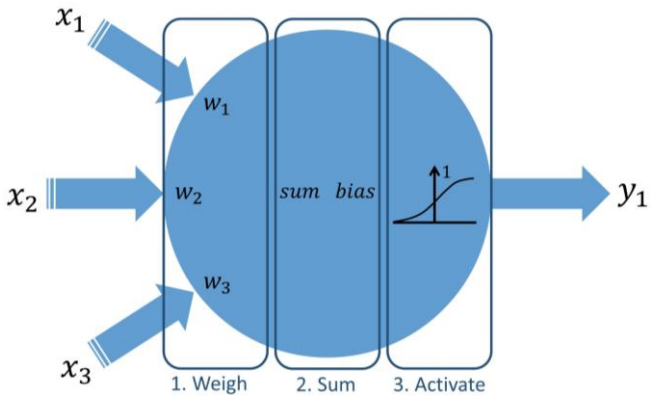


Figure 3.9 - Artificial neuron diagram with the three components: weigh, sum and activation.

The activation of every single output unit of a feedforward ANN is a nonlinear activation pattern function over the network's input units. The connection weights parameterises the functions. An ANN with no hidden layers can only represent a low number of the possible input-output functions. However, as demonstrated in [46], an ANN with a single layer containing a large number of sigmoid units can estimate any continuous function. For other nonlinear activation functions, this can also be proved. However, nonlinearity is essential since if all the units in a multi-layered feedforward ANN had linear activation functions, the whole network would be equivalent to a network with no hidden layers.

Despite this property, to approximate high-complexity functions needed for many artificial intelligence tasks, abstractions, which are hierarchical compositions of numerous layers of lower-level abstractions such as deep architectures, may be required. The successive layers of a deep ANN calculate an increasingly abstract network's input representation. Each unit can provide a specific feature contributing to a hierarchical representation of the overall input-output network function.

Training ANN hidden layers is a way to automatically create appropriate features for a specific problem so that hierarchical representations can be produced without exclusively relying on hand-crafted features. ANNs are known to learn by a stochastic gradient method. Each weight is regulated in a direction aimed at improving the network's overall performance. In supervised learning cases, the objective function is the expected error over a set of labelled training examples. In reinforcement learning, ANNs can use TD errors to learn value functions or aim to maximise expected reward as a policy-gradient algorithm. In this case, it is necessary to estimate the objective function's partial derivative concerning each weight, given the current values of all the network's weights. The gradient is the vector of these partial derivatives.

The most successful way to train ANNs with hidden layers, provided the units have differentiable activation functions, is the backpropagation algorithm. This learning algorithm consists of alternating between forward and backward passes through the network. Each forward pass calculates the activation of each unit, assuming the current activation of the network's input units. After each forward pass, a backward pass efficiently calculates the partial derivatives for each weight. Similar to other stochastic gradient learning algorithms, the partial derivative vector is a precise gradient estimate.

3.2.2 Radial Basis Function

A Radial basis function (RBF), is a natural generalisation process to continuous features. Rather than having discrete features, the RBF kernels allow for the input value to anything in the interval $[0,1]$. A common RBF feature x_i has a Gaussian response $x_i(s)$ dependent on the state s, the centre state c_i and the feature's width σ_i:

$$x_i(s) = \exp\left(-\frac{\|s - c_i\|^2}{2\sigma_i^2}\right) \tag{3.43}$$

The distance metric choice is made accordingly to the way the states and tasks data is set. Figure 3.10 shows a one-dimensional example of three different RBF features

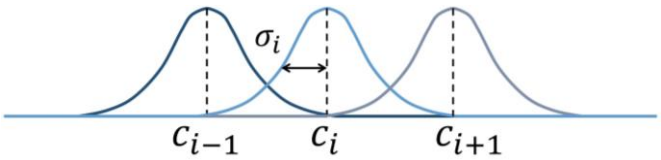


Figure 3.10 - One-dimensional radial basis functions.

The critical advantage of RBFs over binary features is that an approximate function is produced that varies smoothly and is differentiable. An RBF network is a linear function approximator that uses RBFs as its features. Some learning methods for RBF networks have dynamic centres and widths of the features, which turns them into nonlinear function approximators. Nonlinear methods can fit target functions with higher precision. However, the downsides are the high computational complexity and the need to perform some manual tuning before it becomes a robust solution.

3.3 Policy Gradient Methods

In this subchapter, a novel idea is presented. Until the end of the Q-Learning subchapter, all the methods have been action-value methods. These learned the values of action and consequently selected actions based on their estimated action values. On this basis, the policies would not even exist if it was not for

the action-value estimates. In Policy Gradient (PG), the methods learn a parameterised policy that selects an action without checking the value function. It would still be used to learn the policy parameter, but not for the action selection. The notation $\theta \in \mathbb{R}$ is used for the policy parameter vector.

$$\pi(a|s,\theta) = p\{A_t = a \mid S_t = s, \theta_t = \theta\} \tag{3.44}$$

It represents the probability to take an action a at time step t knowing that the environment is in state s with parameter θ. The only methods described in this subchapter, are the methods for learning the policy parameter based on a gradient of a scalar performance measure $J(\theta)$ concerning the policy parameter. These methods pursue a performance maximisation, so the updates resemble gradient ascent in J:

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t) \tag{3.45}$$

Where $\nabla J(\theta_t)$ denotes a stochastic estimate, whose expectation approximates the performance measure gradient concerning θ_t. The methods that follow this process logic are named policy gradient methods, independently of also learning an approximate value function. A new method, known as the actor-critic method, learns approximations to both policy and value functions. The actor refers to the learned policy, and the critic is a reference to the learned value function, usually a state-value function

3.3.1 Policy Approximation and its Advantages

In policy gradient methods, a policy parameterisation can be performed in any way, if $\pi(a|s,\theta)$ is differentiable concerning its parameters. As a way to ensure exploration, it is commonly required that the policy never becomes deterministic, for example, that $\pi(a|s,\theta) \in (0,1)$, for all a, s and θ. In this subchapter, one of the most common parameterisations for discrete action space is presented as well as the advantages it offers over action-value methods. Policy-based methods also offer useful strategies for dealing with continuous action spaces, as will be further described in chapter 3.4.

If the action-space is both discrete and small, a natural parameterisation method is to arrange parameterised numerical preferences $h(a,s,\theta) \in \mathbb{R}$ for each state-action pair. The actions with the highest likelihood in each state s have the highest probabilities of selection, for example, an exponential SoftMax distribution:

$$\pi(a|s,\theta) = \frac{e^{h(a,s,\theta)}}{\sum_l e^{h(a,l,\theta)}} \tag{3.46}$$

The denominator of **(3.46)** is only required so the sum of the action probabilities in each state is one. This policy parameterisation is known as SoftMax in action preferences. The action preferences can be

randomly parameterised, for example, as an artificial neural network, where θ is the vector of all the network's connection. Alternatively, as a simpler method, the preferences can be linear features:

$$h(a, s, \theta) = \theta^T x(s, a) \quad \textbf{(3.47)}$$

The first advantage of parameterised policies is that an approximate policy may approach a deterministic policy, but with ε-greedy action selection, there is always a ε probability of selecting an arbitrary action. As an alternative, soft-max distributions based on action values could be used, but this method does not permit the policy to approach a deterministic policy. The action-value estimates would instead converge to their actual corresponding values, which are different by a finite amount. Action preferences differ as they do not approach specific values. Instead, these try to create the optimal stochastic policy. When an optimal policy is deterministic, the optimal actions preferences will be driven infinitely higher than all suboptimal actions.

The second advantage of policy parameterisation according to SoftMax in action preferences is the selection enabling of actions with arbitrary probabilities. In cases with significant function approximation, the best approximate policy can be stochastic. Action-value methods have no natural way of finding stochastic optimal policies, whereas policy approximating methods can.

In Figure 3.11, an example of a situation where with the provided information, the optimal play is often to do two different things with specific probabilities. The agent starts on a grey square, not knowing which the state is whether there is square in one of the four directions it can move. The goal of the agent must be to find the treasure, denoted in blue, avoiding the skulls denoted as black. Both are terminal states.

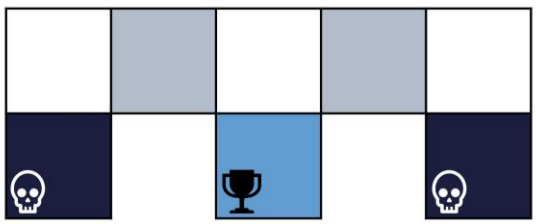


Figure 3.11 - Environment with two similar initial states, grey squares, two negative terminal states, black squares, and one positive terminal state, light blue.

As the grey squares provide the same state, if the agent follows a purely deterministic policy, with time its initial position will determine its success state. If it starts in one square, it will always be successful. However, if it starts on the other square, it will always fail as the policy will turn to what is described in Figure 3.12 (a). Though, if a stochastic policy enables actions with arbitrary probabilities, the agent may repeat a few steps, but will always be successful, as shown in Figure 3.12 (b).

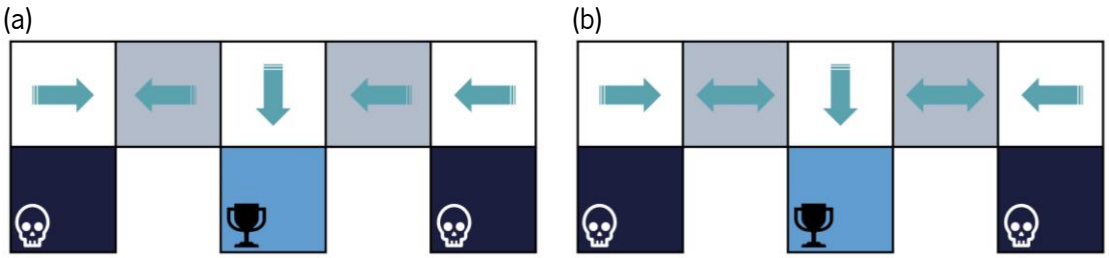


Figure 3.12 - Difference between deterministic policy (a) and stochastic policy (b).

Another advantage over action-value parameterisation relies on the policy being a more straightforward function to approximate. Different policy and action-value complexities vary from problem to problem, which means that in some cases it is easier to do policy approximation rather than action-value. Lastly, the choice of policy parameterisation is usually a straightforward way of injecting prior knowledge about the desired policy format into the reinforcement learning system.

3.3.2 Policy Gradient Theorem

The advantages of policy parameterisation also provide an essential theoretical advantage. Continuous policy parameterisation allows the action probabilities to change smoother as a learned parameter function, opposing to the ε-greedy selection where the action probabilities may have significant changes for an arbitrarily minor change in the estimated action-values. Consequently, it is the policy dependence continuity on the parameters which enables policy-gradient methods to resemble gradient ascent, as in **(3.45)**. The episodic and continuing cases define the performance measure, $J(\theta)$, differently and thus have to be treated separately to some extent.

For the episodic case, the performance value is defined as the episode's start state value. It can turn to a more straightforward notation to assume that every episode starts in a specific state s_0. Moreover, the performance can be defined as:

$$J(\theta) = v_{\pi_\theta}(s_0) \tag{3.48}$$

Where, v_{π_θ} is the exact value function for π_θ, the policy determined by θ.

With function approximation, it is not straightforward to change the policy parameter to ensure improvement. The problem is the performance dependence on both the action selection and the state's distribution in which those selections are made and the fact that both are affected by the policy parameter. Assuming a state, where the policy effect parameter on the actions and consequently on rewards is simply calculated from the parameterisation. However, the policy effect on state distribution is an environment function and is commonly unknown. Thus the policy gradient theorem mathematically demonstrates [47]

a strategy to estimate the performance gradient concerning the policy parameter, provided an analytic expression for the gradient of performance concerning the policy parameter. As so, the policy gradient theorem establishes that:

$$\nabla J(\theta) \propto \sum_{s} \mu(s) \sum_{a} q_{\pi}(s,a)\, \nabla \pi(a|s,\theta) \tag{3.49}$$

Where the gradients are partial derivative vectors concerning the θ components and π denotes the policy corresponding to parameter vector θ. The distribution μ is the on-policy distribution under π. On episodic cases, the proportionality constant is the average length of an episode. Moreover, in continuous cases, it is assigned a unitary value so that the relationship remains similar.

3.3.3 Actor-Critic Methods

An actor-critic method is a policy gradient learning algorithm that learns both a policy, which acts like an actor and a state-value function which acts as a critic, as represented in Figure 3.13. To learn a state-value function and use it as a critic means to use it for bootstrapping, by updating the value estimate for a state from the estimated values of subsequent states. The distinction between actor-critic methods to other methods such as Monte Carlo, is the bootstrapping factor for only through this bias and asymptotic dependence on the function approximation quality are introduced. The bias introduced through bootstrapping and state representation confidence is often beneficial since it reduces variance and accelerates the learning process. Temporal Difference methods can flexibly adjust the bootstrapping degree.

In this demonstration, only the one-step actor-critic methods are considered, similar to the algorithms previously demonstrated in 3.1.6IV and 3.1.6V, SARSA and Q-Learning, respectively. The one-step methods main advantages are being entirely online and incremental and avoiding the complexity of eligibility traces. These consist of a more straightforward eligibility trace case method. One step actor-critic methods, use one-step returns and a learned state-value function, defined as:

$$\theta_{t+1} = \theta_t + \alpha(G_{t:t+1} - \hat{v}(S_t, w)) \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)} \tag{3.50}$$

$$= \theta_t + \alpha(R_{t+1} + \gamma \hat{v}(S_{t+1}, w) - \hat{v}(S_t, w)) \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)} \tag{3.51}$$

$$= \theta_t + \alpha \delta_t \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)} \tag{3.52}$$

Where w denotes the state-value weights, and the natural state-value function learning method to pair with this is semi-gradient TD(0). Next, it is presented the complete algorithm pseudocode. It is now an entirely online incremental algorithm, with states, actions, and reward processes as they occur without the need to ever being revisited.

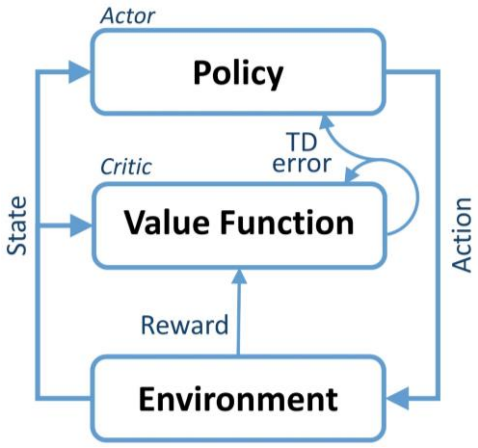


Figure 3.13 - Actor-Critic method diagram, where the actor represents the policy and the critic the value function.

Algorithm 3.7 – One-step Actor-Critic (episodic), for estimating $\pi_\theta \approx \pi_*$

Input: *a differentiable policy parameterisation* $\pi(a|s,\theta)$
Input: *a differentiable state-value function parameterisation* $\hat{v}(s,w)$
Parameters: *step sizes* $\alpha^\theta > 0, \alpha^w > 0$
Init policy parameter $\theta \in \mathbb{R}^{d'}$ *and state-value weights* $w \in \mathbb{R}^{d'}$
Loop (*for each episode*):
  Init s (*first state of episode*)
  $I \leftarrow 1$
  Loop (*for each episode's time step*):
    $A \sim \pi(.|s,\theta)$
    Take action a, observe s', r
    $\delta \leftarrow r + \gamma\hat{v}(s',w) - \hat{v}(s,w)$
    $w \leftarrow w + \alpha^w \delta \nabla\hat{v}(s,w)$
    $\theta \leftarrow \theta + \alpha^\theta I \delta \nabla \ln\pi(a|s,\theta)$
    $I \leftarrow \gamma I$
    $s \leftarrow s'$
  until s is terminal

3.4 Deep Deterministic Policy Gradient

In 2016, Google's Deepmind released a paper [48], linking the ideas underlying the success of Deep Q-Learning to the continuous action domain. They presented an actor-critic, model-free algorithm based on the deterministic policy gradient (DPG) that can operate over continuous action spaces. This algorithm, known as Deep Deterministic Policy Gradient (DDPG) robustly solved more than twenty simulated physics

tasks, including classic control problems, such as the cart-pole swing-up, dexterous manipulation, legged locomotion and car driving.

Continuous action spaces are challenging implementations in Q-Learning, since in continuous spaces, finding the greedy policy requires optimisation of A_t on every time step. This optimisation is considered to be significantly slow to be practical with large unconstrained function approximators and nontrivial action-spaces. Instead, an actor-critic approach based on the Deterministic Policy Gradient (DPG) [49] is used.

The DPG algorithm has a parameterised actor function $\mu(s|\theta^{\mu})$ that specifies present policy by deterministically mapping the states to a specific action. $Q(s,a)$, the critic, is learned using the Bellman equations in Q-Learning. The actor is updated, with a chain rule on the expected return from the start distribution J concerning actor parameters.

$$\begin{aligned}\nabla_{\theta^{\mu}} J &\approx \mathbb{E}_{s_t\sim\rho^{\beta}}\left[\nabla_{\theta_{\mu}} Q(s,a|\theta^{Q})|_{s=s_t,a=\mu(s|\theta^{\mu})}\right] \\ &= \mathbb{E}_{s_t\sim\rho^{\beta}}\left[\nabla_{a} Q(s,a|\theta^{Q})|_{s=s_t,a=\mu(s_t)} \nabla_{\theta_{\mu}} Q(s|\theta^{\mu})|_{s=s_t}\right]\end{aligned} \quad \textbf{(3.53)}$$

In [49], it is proved that this is the policy gradient, and as is commonly done in these implementations, the discounting in the state-visitation distribution ρ^{β} is ignored.

Similar to Q-Learning, the introduction of a nonlinear function approximator implies that convergence is no longer guaranteed. However, these approximators are an essential part of being able to generalise for large state spaces. DDPG emerged by providing modifications to DPG inspired by Deep Q-Networks, that allow the use of neural network function approximators to learn online in large state-spaces and action-spaces.

A challenge of using ANNs in reinforcement learning is the assumption made by most optimisation algorithms that the samples are identically and independently distributed. However, when the samples are generated from sequentially exploring an environment, this assumption is no longer valid. Moreover, to improve hardware efficiency, it is essential to learn in batches rather than online.

Replay buffers address this issue and are a finite sized cache. Transitions are sampled according to the exploration policy and the tuple $(s_t, a_t, r_t, s_{t+1}, a_{t+1})$ is stored in the replay buffer. At each time step t, both the actor and the critic are updated by sampling a small batch from the buffer. Since DDPG is an off-policy algorithm, it is possible to have a large replay buffer, which allows the algorithm to benefit from learning across a set of uncorrelated transitions.

DDPG creates a copy of the actor and critic networks, $Q'(s,a|\theta^{Q'})$ and $\mu'(s,a|\theta^{\mu'})$ respectively used for calculating the target values. The target network weights are updated by having them slowly track the learned networks:

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta' \tag{3.54}$$

Where $\tau \ll 1$, whose purpose is to define how much influence the learned network has over the target network weights. It results in the target values being constrained to slow changes, significantly improving the learning's stability. Moreover, with just a simple change, the unstable problem of learning the action-value function changes to a similar case in supervised learning, where a wide variety of robust solutions exist. Having both μ' and a Q' is required to have stable targets, to consistently train the critic without divergence. Since the target network delays the value estimation propagation, it causes the learning process to be slower. However, this problem is outweighed by learning stability.

Learning in continuous action spaces has new challenges, such as exploration. An advantage of off-policy algorithms is that the exploration problem can be taken care of independently from the learning algorithm. The exploration policy μ' is designed by adding noise sampled from a noise process N to the actor policy:

$$\mu'(s_t) = \mu\left(s_t|\theta_t^{\mu}\right) + N \tag{3.55}$$

N is commonly chosen as a way to better suit the environment. In the following algorithm, the pseudocode for the Deep Deterministic Policy Gradient is presented.

Algorithm 3.8 – Deep Deterministic Policy Gradient Algorithm

Initi Critic network $Q(s,a|\theta^Q)$ *with weights* θ^Q *randomly*
Initi Actor network $\mu(s,a|\theta^\mu)$ *with weights* θ^μ *randomly*
Init target networks Q'*and* μ'*with weights,* $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$
Initialise replay buffer φ
Loop (*for each episode*):
  Init random process N *for action exploration*
  Receive initial observation state s_0
  Loop (*for each episode's time step*):
    Select action a_t *according to* μ *and* N
    Take action a, *observe* s', r
    Store transition (s_t, a_t, r_t, s_{t+1}) *in* φ
    Sample random batch of j *transitions* (s_t, a_t, r_t, s_{t+1}) *from* φ
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta)|\theta^{Q'})$
    Update critic by minimising the loss $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient $\nabla_{\theta^\mu} J$
    Update the target networks Q'*and* μ': $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$
    $s \leftarrow s'$

3.5 Conclusions

In this chapter, the theoretical foundations behind the selected reinforcement learning algorithms are detailed. Reinforcement learning algorithms can learn purely from environment exploration, without any interference from an outside agent. The Markov Decision Processes and Bellman equations are the mathematics foundations behind RL algorithms. Different properties from both Dynamic Programming and Monte Carlo methods were merged to create a novel RL strategy named Temporal Difference Learning. The abstraction from the environment dynamics model and the capability of updating estimates based on previously learned estimates, without waiting for an outcome known as bootstrapping, is what makes Temporal Difference Learning as one of the top reinforcement learning methods. It also introduced two new algorithms, on-policy SARSA, and off-policy Q-Learning.

Approximation methods are an essential tool for feature selection. ANNs are a nonlinear function approximation accountable for introducing Deep Learning techniques. These are responsible for some of the most impressive abilities across all machine learning algorithms, including reinforcement learning.

Policy Gradient methods allow the agent to learn a parameterised policy that can select an action without checking the value function. By parameterising both the value function and the policy, a new set of algorithms known as the Actor-Critic are created and are commonly used as a state of the art strategy.

Lastly, Deep Deterministic Policy Gradient is an actor-critic, model-free algorithm based on the deterministic policy gradient that can operate over continuous action spaces. DDPG has been shown to robustly perform a wide variety of tasks unfeasible for the previously presented algorithms, due to being able to function over continuous action spaces. DDPG is an excellent solution to implement in robotics since it is robust enough to overcome the problems that the robotics subject brings.

One of the goals of this dissertation is to provide an introductory mathematical and theoretical foundation on reinforcement learning as well as some of the most common algorithms such as Monte Carlo methods, SARSA and Q-Learning and state of the art algorithms such as DDPG. However, for a more thorough explanation of all these algorithms and many more, the recommended literature lies on [21], as it is commonly known to be an essential book on reinforcement learning. On further chapters in this dissertation, three different algorithms of the ones previously described will be implemented, which are: Q-Learning, Policy Gradient and Deep Deterministic Policy Gradient.

4

Simulation Framework: Development

As referred to in subchapter 2.3, reinforcement learning algorithms designed for robotics must consider simulation environments. These simulation environments provide an operational imitation of a real-world process or system over time. Reinforcement learning requires a significantly high volume of interaction with the environment, commonly known as trial-and-error episodes, to learn an optimal policy. In order to teach an anthropomorphic robot to walk, one must perform around hundreds of thousands of episodes until the robot stops falling and having to be consistently picked up, successfully learning to walk. Therefore, simulators are an essential part of achieving results in a cost-effective and timely way.

All simulations performed were made on an Acer Aspire V15 Nitro laptop (VN7-591G-744P). The Central Processing Unit (CPU) used is an Intel Core i7-4720HQ, and the Graphics Processing Unit (GPU) is an NVIDIA GeForce GTX 960M (2GB GDDR5). Additionally, the computer has a two storage drives, 1 TB Hard Disk Drive (HDD) and a 128 GB Solid-State Drive (SSD) with an 8GB DDR3 of Random-Access Memory (RAM).

In this chapter, the study and comparison between two simulation environments is presented, along with, the development of all the simulated parts necessary for the reinforcement learning algorithms to be worked on: the environments, the agents, and the complete simulation layout. Additionally, a description of the communication between the different systems using Robot Operating System (ROS) and the embedded scripts developed.

4.1 Simulation Environments: V-REP and Gazebo

Initially, two different simulation environments were considered for this dissertation: V-REP (Virtual Robot Experimentation Platform) [50], Figure 4.1, and Gazebo [51], Figure 4.2. As demonstrated in [43], both

simulators present similar simulation capabilities, regarding worlds, sensors and actuators. In [52], it is presented a study regarding a thorough feature and performance comparison between the two simulators. Considering the physics engines, V-REP offers multiple engines such as Bullet 2.78, 2.83, ODE, Vortex and Newton whereas in Gazebo only ODE physics engine is available by default. Different physics engines may be used in Gazebo, yet they require Gazebo to be built from source. Both include a code and scene editor, and the objects of the scene allow full interaction.

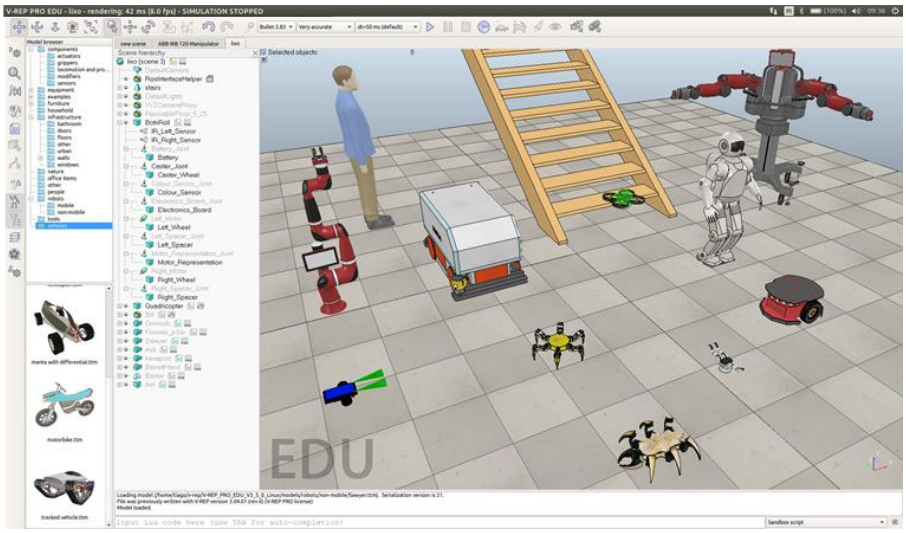


Figure 4.1 - V-REP simulation environment.

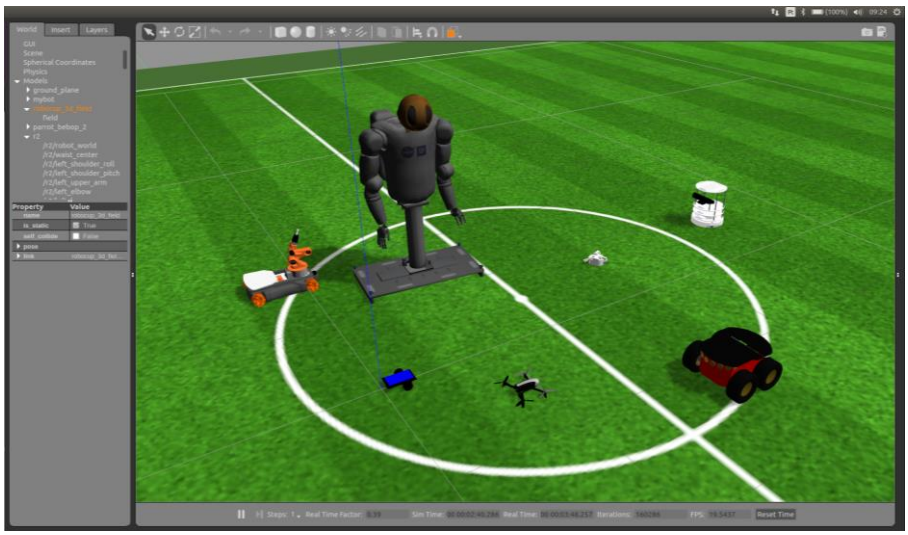


Figure 4.2 - Gazebo simulation environment.

The most significant differences between the two simulators are associated with the poor Gazebo usability and the restrictions regarding editing points on 3D meshes. Even though V-REP is the most resource-consuming simulator, its ability to automatically spawn new threads on multiple CPU cores and thus taking advantage of the full amount of CPU power, is an aspect Gazebo fails to exhibit. In Table 4.1, the essential distinctions which lead to the choice between V-REP and Gazebo are presented. Figure 4.3 shows a simplistic design from the agent in both simulation environments.

Table 4.1 - Comparison between V-REP and Gazebo.

Characteristics	*V-REP*	*Gazebo*
Default Physics Engine	ODE, Bullet 2.78, Bullet 2.83, Vortex and Newton	ODE
Mesh Manipulation	Yes, in real time	No
Robot Variety	High: bi-pedal, hexapod, wheeled, flying, snake robots	Low: wheeled, flying robots
Programming Functionality	Scripts attached to robots, plug-ins, ROS, and RemoteAPI	C++ plug-ins or via ROS programs
Model Description	Included Scripts	Hard to recognise how a third-party robot works
Documentation	Vast tutorials library and code examples, large community	Step-by-step tutorials and a large user community
User Interface	No freezing issues. All functionality is fairly intuitive	The interface froze a few times. Low UI usability
Simulator Performance: One robot + Small scene	Standard	Standard
Simulator Performance: Ten robots + Small scene	Slower than Real Time	Standard

(a) (b)

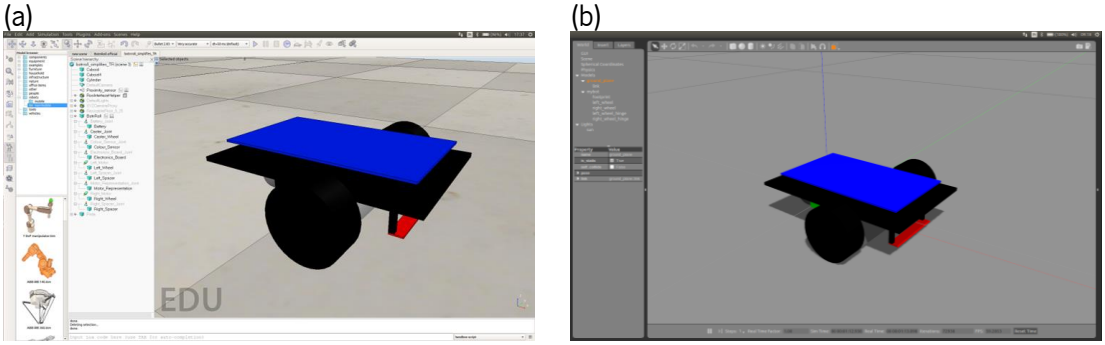


Figure 4.3 - Simplistic robot designed on both V-REP (a) and Gazebo (b) simulation environments.

In conclusion, the chosen simulator: V-REP, offers several useful features, like numerous physics engines, a comprehensive model library, the ability to interact with the world during the simulation and, most importantly, mesh manipulation and optimisation. As described in [52], V-REP is suitable for high-precision modelling of robotic applications where only a few robots are operating simultaneously.

4.2 The Agent: Bot'n Roll ONE A

Bot'n Roll ONE A [53] Arduino compatible is an Open Source didactic robot, intended for those who wish to start in the world of mobile robots, with little knowledge about electronics or computers. It has two microcontrollers, one ATmega328 running at 16MHz, programmable in C language with Arduino IDE, and one PIC18F45K22 running at 80MHz programmable in C with MPLABX Microchip supplied with firmware developed by botnroll.com [54]. The robot comes equipped with an LCD, two motors for differential drive, two infra-red obstacle sensors and all the electronics necessary for assembling a significant number of both inputs and outputs as seen in Figure 4.4. Its compatibility with Arduino shielding allows a diverse range of possible sensors to couple with the robot. Bot'n Roll ONE A is a robot commonly used in the laboratory for testing new algorithms, concepts and ideas.

In order to implement the reinforcement learning algorithms on Bot'n Roll ONE A, its simulated version must be developed on V-REP. The Coppelia Robotics [55], creators of V-REP, provide a very intuitive step-by-step introductory tutorial to create a new robot named BubbleRob [56]. The simulated version of Bot'n Roll ONE A was developed in conformity with this tutorial with some variants introduced for easier integration with the reinforcement learning methods developed.

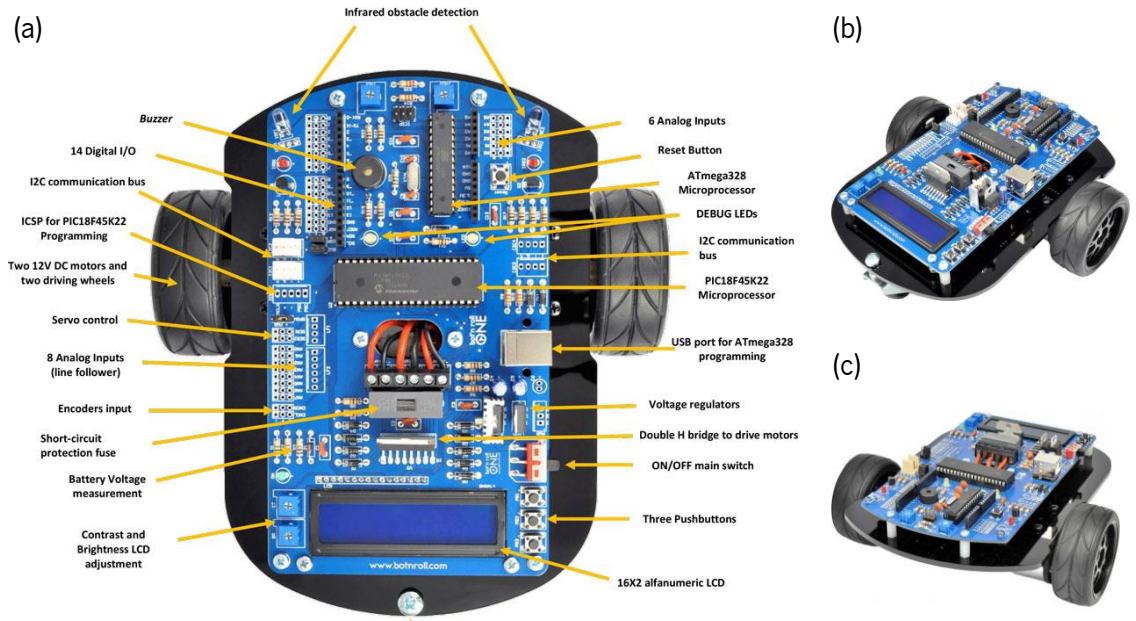


Figure 4.4 - Bot'n Roll ONE A robot (figure displayed with permission from botnroll.com).

Two different simulation variants of the robot were developed. As presented in Figure 4.5, this simulation variant illustrates a complete robot model developed in a 3D modelling framework and imported to V-REP, allowing a much more realistic version of the robot. This version, due to its detail complexity, can only be simulated with a high-density mesh. However, as previously seen, V-REP works more accurately with simplistic versions of robots. Moreover, after a few tests, this simulation concept was abandoned due to its reduced performance regarding resource-consuming.

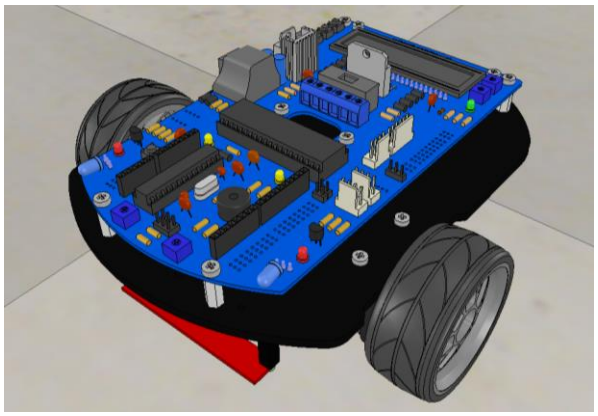


Figure 4.5 - Simulated Bot'n Roll ONE A with a high detail complexity resulting in a high-density mesh.

Thus, the second variant results in a simplistic simulation of Bot'n Roll ONE A, Figure 4.6. This model was drawn using only primitive shapes, which result in very simplistic meshes for low resource consumption. This method allowed the simulation environment to work with multiple robots at the same time while all the robots were running a smooth movement.

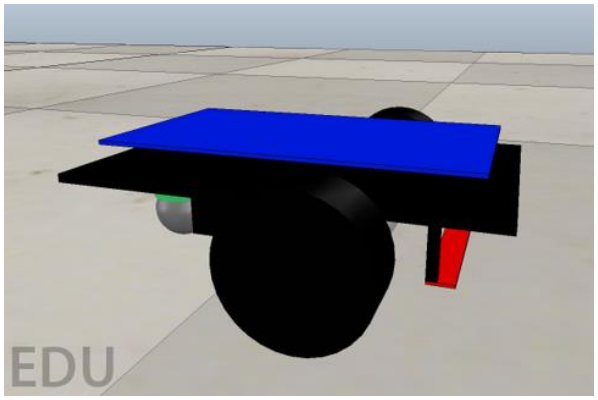
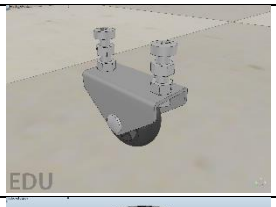
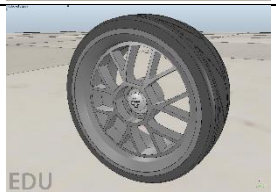
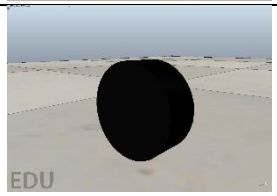
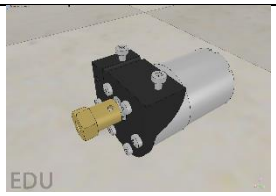
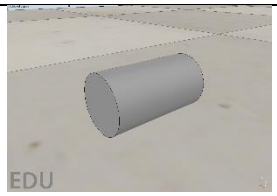
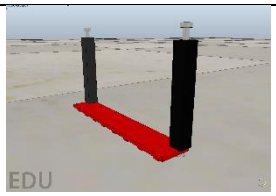
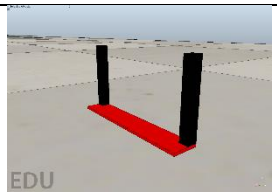
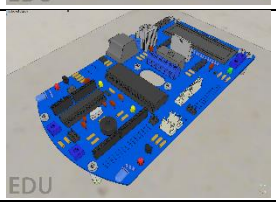
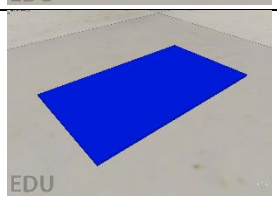
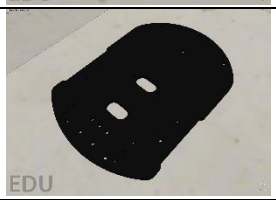
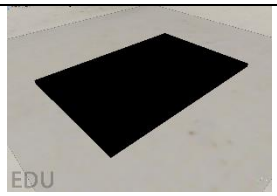
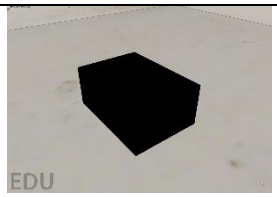


Figure 4.6 - Simplistic simulated Bot'n Roll ONE A for a low-resource consumption implementation.

All the robot parts were designed accordingly to the robot weight and size. All parts were weighed and measured for a realistic implementation compared to the real-world robot. In Table 4.2, all the information regarding the values necessary for a simplistic design is provided as well as the seven parts developed. Both simplistic and realistic robot objects designs are present. The principal moments of inertia were calculated using V-REP automatic formulas. The wheels use standard rubber surface specifications provided by V-REP to simulate the movement friction. The simplistic caster wheel consists of a sphere

with no friction. When robots tend to be considerably light regarding weight, such as this one, V-REP advises the users to use a weight multiplication factor to improve the movement similarities to real-world during the simulation. Each object weight was multiplied by a 16 factor to improve the robot stability and movement control.

Table 4.2 - Specifications regarding the simplistic created simulated agent.

Object	*3D Model*	*Simplified 3D Model*	*Size [m]*	*Weight [kg]*
Caster Wheel			x: $3.20e-2$ y: $3.20e-2$ z: $3.20e-2$	Real: $4.80e-2$ Implemented: $7.68e-1$
Wheel			x: $6.50e-2$ y: $6.50e-2$ z: $2.60e-2$	Real: $3.30e-2$ Implemented: $5.28e-1$
Motor			x: $3.70e-2$ y: $3.70e-2$ z: $6.70e-2$	Real: $2.10e-1$ Implemented: $3.36e+0$
Line Sensor			x: $1.30e-2$ y: $3.85e-2$ z: $7.50e-2$	Real: $1.20e-2$ Implemented: $1.92e-1$
PCB			x: $1.80e-1$ y: $1.02e-1$ z: $2.00e-3$	Real: $1.58e-1$ Implemented: $2.53e+0$
Acrylic Base			x: $2.03e-1$ y: $1.34e-1$ z: $4.00e-3$	Real: $1.13e-1$ Implemented: $1.81e+0$
Battery			x: $5.00e-2$ y: $5.00e-2$ z: $3.00e-2$	Real: $1.93e-1$ Implemented: $3.10e+0$

All the robot's primitive shapes have a detailed set of properties for a specific robot's usage. Every physical part, known as an object have their properties configured accordingly to Figure 4.7. All the robot's objects are dynamic, collidable, measurable, detectable and renderable.

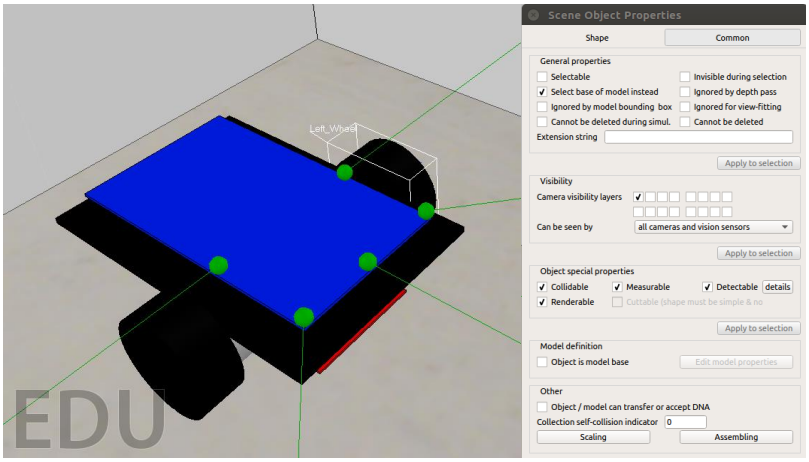


Figure 4.7 - Simulated agent object properties.

For the task at hand, the robot's only actuators are its two motors, which grant a differential drive as the robot movement platform. No more actuators will be considered, as the moving platform is a robot's physical restriction in a sense that, to add a different moving system, the robot had to be physically changed. The movement of the robot is the output for all reinforcement learning algorithms implemented, as it is crucial to have a well-configured motion system. In Figure 4.8, the implemented rotational actuators are represented.

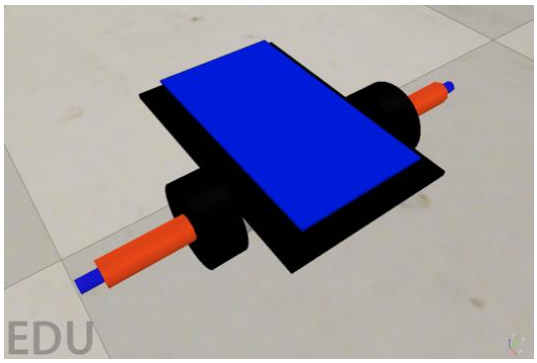


Figure 4.8 - Simulated Bot'n Roll ONE A, two rotational actuators representing the motors.

The physical robot was designed in such a way that numerous modular distance sensors may be attached to the robot. However, for the challenge further described on the next subchapter, the robot must only use the embedded infra-red distance sensors. As seen in Figure 4.9, two different approaches were considered, one respecting the embedded sensors physical restriction (a), and the other, using simulated Time-of-Flight (ToF) distance sensors with different topologies (b).

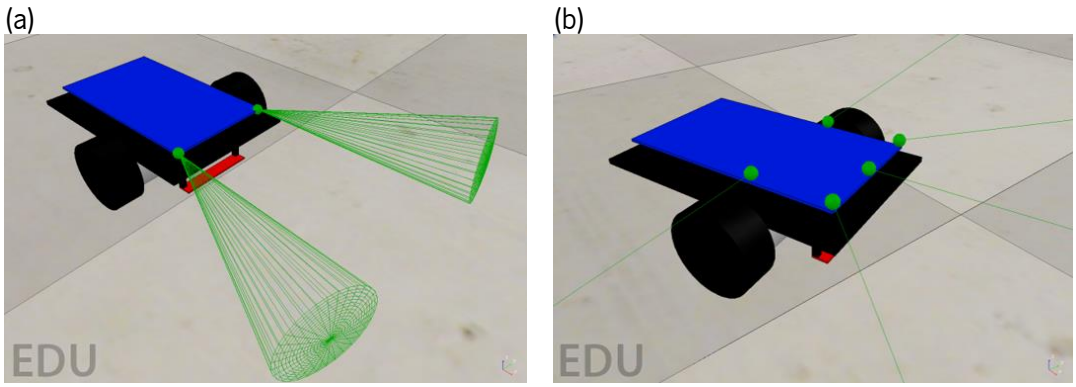


Figure 4.9 - Different obstacle sensor implementations using two infra-red wide obstacle sensors (a), and two ToF obstacle sensors (b).

The detection range of the embedded sensors was designed accordingly to the one presented on the robot. A simple parameter measurement of range, height, angle, radius, and offset was performed so the simulated agent sensors remark as precisely as possible the physical sensors. The ToF distance sensors are just a theoretical implementation of a laser beam with a maximum detection distance of one meter. The topologies used on each reinforcement algorithm are described in chapters 5 and 6.

4.3 The Environment: RoboParty Obstacle Maze

Similar to the agent, an environment must also be simulated. The specific task the simulated Bot'n Roll ONE A must surpass a maze similar to the one presented on the RoboParty Obstacle Maze challenge. RoboParty [57] is an international pedagogical event for new teams to learn how to build autonomous mobile robots in a fun and straightforward way, guided by qualified professionals. For three days and two nights, the teams are presented with an entirely disassembled Bot'n Roll ONE A, whereas they must build the mechanical components, solder the electronics and program the ATmega328. On the last day, teams are presented with three different challenges, one of them being the RoboParty Obstacle Maze challenge described in Figure 4.10.

Figure 4.10 - RoboParty obstacle maze challenge (figure displayed with permission from RoboParty).

The obstacle challenge consists of a maze with approximately $20\ cm$ of height walls with a single entrance point and a single exit point. The teams must only use the robots embedded infra-red obstacle sensors to avoid the maze walls in order to reach with no wall-bumping penalties the exit point. The robot must not have any modification to its original form to guarantee all teams compete under the same conditions.

Simulation wise, three iterative complex mazes were developed for different reinforcement learning strategies study. In order to succeed, the agent must be able to reach the endpoint, thus solving the mazes. The first maze (approximately $1.2\ m$ long) consists of a straight narrow line. The second maze (approximately $5.0\ m$ long) has two curves to both sides, which force the robot to learn how to perform turns on non-straight walls and avoiding obstacles from both sides. The third maze (approximately $8.3\ m$ long) has curves on both sides, slim and narrow wall distances, sharp edges and deep turns. The distance between walls is on average $0.5\ m$.

In Figure 4.11, the three designed mazes are presented and consist of simple wood walls represented in white. The endpoints are made of laser proximity sensors to detect whether the robot has successfully reached the end of the maze and are denoted by the magenta lines. The start points have no physical attributes and are represented by the robot's position in the mazes.

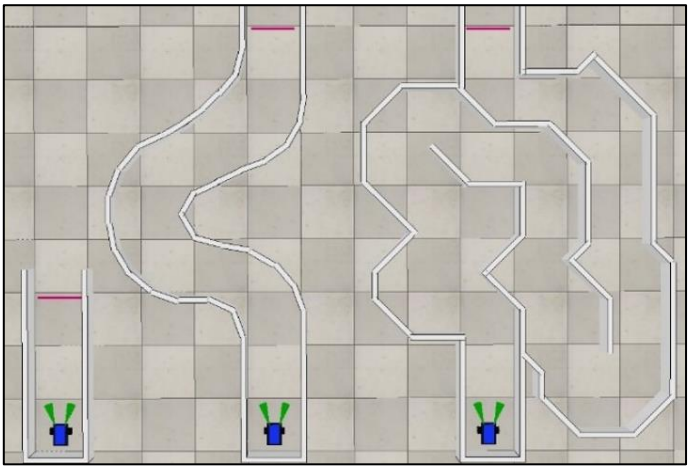


Figure 4.11 - Three iteratively complexity simulated mazes.

The maze walls object properties, Figure 4.12, are similar to the agent's primitive shapes in a way that is both collidable and measurable yet assume a static component, not able to be pushed around by the robot.

The Laser detection sensor, Figure 4.13, works as an episode termination state since the robot has managed to discover the maze's end successfully. It consists of a straight line at the robot's main board

height to detect. This sensor, via an attached embedded script, sends the information to the control script which terminates the particular episode.

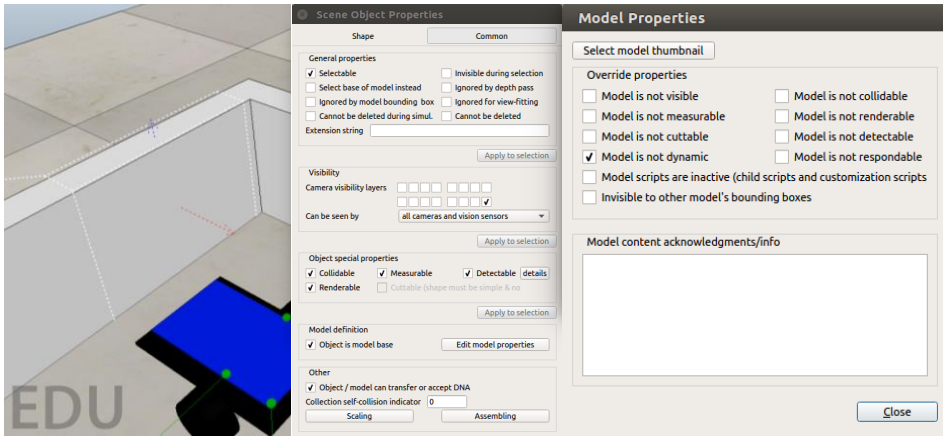


Figure 4.12 - Maze walls object properties.

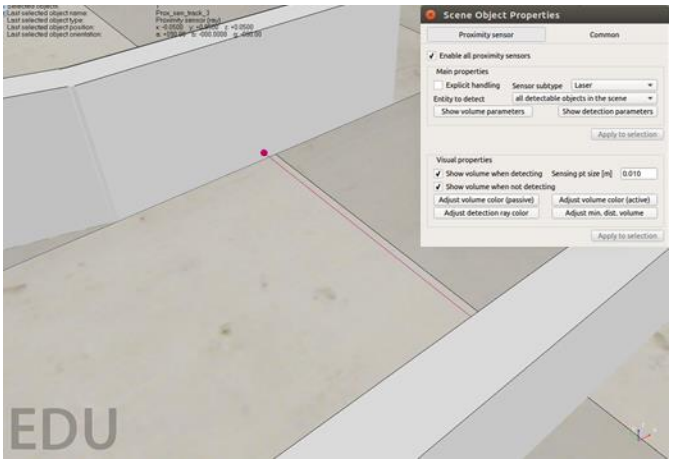


Figure 4.13 - Endpoint laser detection system properties.

4.4 ROS Framework

One of the ways provided by V-REP to control a robot or a simulation is via a ROS node. ROS is a distributed pseudo operating system that allows easy management and communication between multiple computers connected in a network. It is both a flexible and convenient communication platform for distributed processes. V-REP can create a ROS node, which other nodes can communicate with via ROS services, ROS publishers and ROS subscribers. It can virtually connect any number of processes with each other, and a large number of compatible libraries are available.

Even though V-REP offers a wide range of possible robot control systems, the choice of ROS rested on its practical applicability, the large number of compatible libraries and the majority of the laboratory robots having this pseudo operating system implemented.

All the reinforcement learning algorithms software were developed on a Python script external to V-REP. Throughout this dissertation, the Python control script will be named control script. The robot control system consisted of two ROS nodes, one for the simulator, named */ vrep_ros_interface* and another for the control script, named */ AI_bnr_control*. In Figure 4.14, the implemented ROS *rqt_graph* is presented describing how the two ROS nodes interact, via the seven ROS topics.

In order to established an organised and efficient communication between the two ROS nodes, seven different ROS topics were designed either for data transition or simulation control:

- */Collision_Data* (1 Boolean) – Information on whether or not there is a collision state on one of the only three robot objects capable of colliding with the walls: left wheel, main board and right wheel;
- */Goal_Data* (1 Boolean) – Information on whether or not the endpoint detection system has detected a robot coming through;
- */Motor_Data* (2 Float) – Intended speed for both motors from -1, going backwards to 1 going forward;
- */privateMsgAux* – Automatically created by V-REP, it does not perform anything relevant to this specific ROS framework;
- */Sensor_Data* (Dependent on the number of sensors and either if they are discrete (Bool) or continuous (Float)) – The distance read by the robot's obstacle sensors;
- */startSimulation* (1 Boolean) – Information sent to V-REP to indicate the start of a new simulation iteration, also known as an episode;
- */stopSimulation* (1 Boolean) – Information sent to V-REP to indicate the end of the current episode for reaching a terminal state.

Some of these ROS topics are directly connected to the implemented V-REP embedded scripts described on the next subchapter.

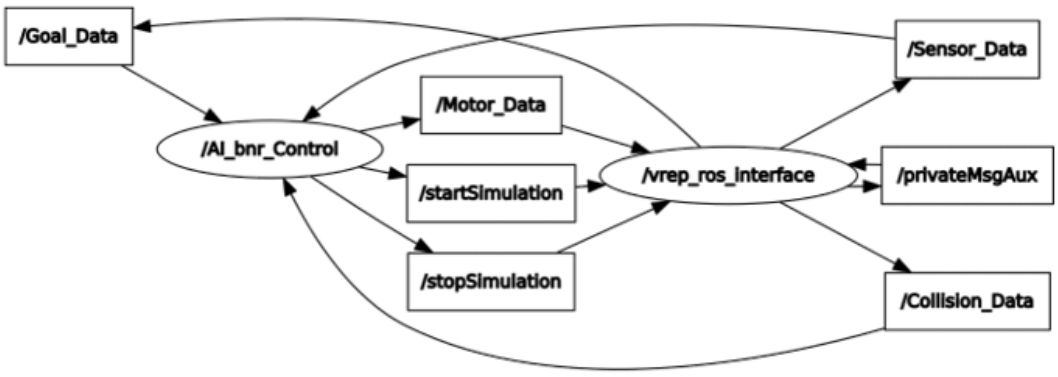


Figure 4.14 - ROS rqt_graph with the two ROS nodes and the seven ROS topics.

4.5 Embedded Script Configuration

V-REP provides embedded scripts to customise a simulation, either a scene, a model or an object via scripting. This configuration method, which consists of writing Lua scripts, is flexible and straightforward, with guaranteed compatibility with every other default V-REP installation. The Lua scripts are embedded scripts provided by V-REP to configure all the simulation parts. This method allows customising a particular simulation, a simulation scene, and the simulator itself. The embedded Lua scripts allow the user to perform a new range of operations not possible with just the ROS framework. Opposing to ROS, embedded scripts can customise the simulator, and are entirely contained in a scene or model, being highly portable and not having any communication lag.

Four embedded scripts were developed, one regarding the agent and its ROS communication interface, and the other three regarding the environment, more precisely the three endpoints sensor on each maze. The script regarding the agent can be divided into six fragments. The two initial scripts refer to initialisation fragments: ROS initialisation, Figure 4.15, and model initialisation Figure 4.16.

```
-- Check if RosInterface is correctly initialized:
moduleName=0
index=0
rosInterfacePresent=false
while moduleName do
    moduleName=sim.getModuleName(index)
    if (moduleName=='RosInterface') then
        rosInterfacePresent=true
        if (DEBUG) then
            print("Ros Interface Ready to be used!")
        end
    end
    index=index+1
end

-- Initialization of both publisher and subscriber:
if rosInterfacePresent then
    sensor_pub    = simROS.advertise('/Sensor_Data','geometry_msgs/PoseArray')
    collision_pub = simROS.advertise('/Collision_Data','std_msgs/Bool')
    motor_sub     = simROS.subscribe('/Motor_Data','geometry_msgs/Vector3','motor_data_subscriber_callback')
    -- test_pub      = simROS.advertise('/Test_Data','geometry_msgs/PoseArray')

end
```

Figure 4.15 - ROS initialisation.

```
-- Bot'n Roll ONE A Robot:
bnr = sim.getObjectAssociatedWithScript(sim.handle_self)

-- Infra_Red Obstacle Sensors:
IR_Left_Sensor  = sim.getObjectHandle("IR_Left_Sensor")
IR_FLeft_Sensor = sim.getObjectHandle("IR_FLeft_Sensor")
IR_Front_Sensor  = sim.getObjectHandle("IR_Front_Sensor")
IR_FRight_Sensor = sim.getObjectHandle("IR_FRight_Sensor")
IR_Right_Sensor = sim.getObjectHandle("IR_Right_Sensor")

-- Joint/Motor:
Left_Motor  = sim.getObjectHandle("Left_Motor")
Right_Motor = sim.getObjectHandle("Right_Motor")

-- Collision:
base_collisionObjectHandle   = sim.getCollisionHandle("base_collision")
leftw_collisionObjectHandle  = sim.getCollisionHandle("leftw_collision")
rightw_collisionObjectHandle = sim.getCollisionHandle("rightw_collision")
```

Figure 4.16 - Model initialisation.

The following three fragments refer to data transfer. These embedded script parts are the link between the simulated agent and the ROS framework, as they provide a sort of interface to send and receive data.

The script fragments are the sensor, motor and collision data. In Figure 4.17, the collision data (a) and the sensor data (b), receive the information gathered by the agent's sensors and proceed to send it via its own ROS topic. The motor data is sent the opposite way. It is calculated by the control script and sent to the agent via its ROS topic.

(a)

```
-- Collision:
base_collisionState=sim.handleCollision(base_collisionObjectHandle)
leftw_collisionState=sim.handleCollision(leftw_collisionObjectHandle)
rightw_collisionState=sim.handleCollision(rightw_collisionObjectHandle)
```

(b)

```
-- IR_Obstacle:
obs_left,   left_sensor_analog   = sim.readProximitySensor(IR_Left_Sensor)
obs_fleft,  fleft_sensor_analog  = sim.readProximitySensor(IR_FLeft_Sensor)
obs_front,  front_sensor_analog  = sim.readProximitySensor(IR_Front_Sensor)
obs_fright, fright_sensor_analog = sim.readProximitySensor(IR_FRight_Sensor)
obs_right,  right_sensor_analog  = sim.readProximitySensor(IR_Right_Sensor)
```

Figure 4.17 - Sensor data (a) and collision (b) embedded script fragments.

Additionally, the collision part of the script adds visual information in the format of red colour to the robot object, that collided with the walls, as shown in Figure 4.18.

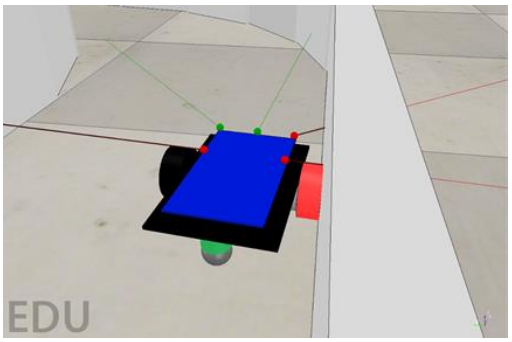


Figure 4.18 - Collision with a maze wall, the red object represents the agent's object that collided.

The remaining fragment regarding the agent refers to the synchronisation between the simulation and the control script. This sync process is necessary due to starts and stops to the simulation and the delay for initiating the ROS transmission from the V-REP node to the control script node. The other three embedded scripts are similar to the ones previously presented. They consist of a straightforward ROS topic initialisation and a sensor reading to inform the control script whenever the agent successfully solves the maze resulting in a terminal state and consequently the end of an episode.

4.6 Supplementary Simulation Environments

In addition to the two simulation environments compared for Bot'n Roll ONE A implementations, a different set of simulation environments was necessary. These set of simulation environments were an essential platform to implement and test all the reinforcement learning algorithms successfully on the

simulated agent. OpenAI's Gym is a toolkit for developing and comparing reinforcement learning algorithms. The Gym library is a collection of test problems, named environments, that can be used to work out reinforcement learning algorithms. It provides a set of easy-to-setup environments of many categories and is compatible with any numerical computation library.

These additional simulation environments were an essential tool to prepare the algorithms to be implemented on Bot'n Roll ONE A. They served as a base algorithm testing platform before transferring the implementations to the simulated agent. Additionally, these implementations granted a new view on how different hyperparameter configurations influence the learning process, and how these need to be adjusted to make the most out of them. These simulation environments share a few similarities regarding state and action space with the created simulation environment, which makes them a comfortable starting point for the experimentation of the selected RL algorithms. Three different Gym environments were experimented as an RL testing platform, namely MountainCar, CartPole and Pendulum.

4.6.1 MountainCar

The MountainCar environment, Figure 4.19, consists of a car on a one-dimensional track, located between two mountains. The goal is to drive up the mountain on the right reaching the flag. However, this car has an underpowered engine not strong enough to climb the mountain on a single try. Thus, the only way to reach the target flag is to drive back and forth building up momentum. MountainCar proved to be a simple Q-Learning and Policy Gradient implementation environment, described in chapters 5 and 6.

I Observation

There are two observation inputs for MountainCar as presented in Table 4.3, one for its position on the horizontal axis and the other for the instantaneous velocity the car reaches. These are visually represented in Figure 4.20 as the red shapes.

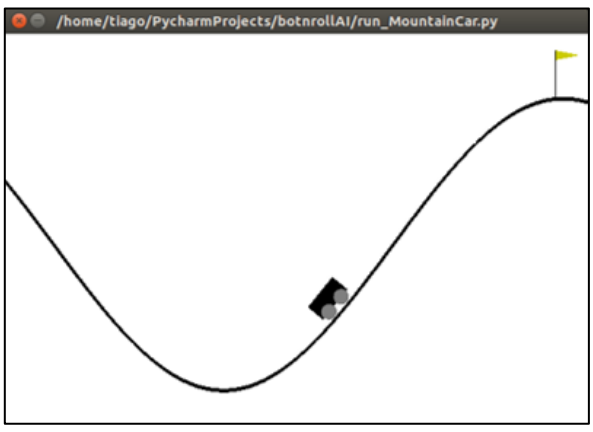


Figure 4.19 - OpenAI Gym MountainCar simulation environment.

Table 4.3 - MountainCar observation.

Number	*Notation*	*Observation*	*Minimum*	*Maximum*
0	o_0	Position	-1.2	0.6
1	o_1	Velocity	-0.07	0.07

II Actions

MountainCar provides three different action outputs, as demonstrated in Table 4.4. All the actions consist of the different car pushing movements, respectively pushing to the left, to the right and no pushing.

Table 4.4 - MountainCar actions.

Number	*Notation*	*Action (Discrete)*
0	a_0	Push Car to the Left
1	a_1	No Push
2	a_2	Push Car to the Right

The no pushing action is similar to letting the car run in neutral, lying in the hands of its previous velocity and the simulated movement laws. The three actions are visually represented in Figure 4.20 as the light blue shapes.

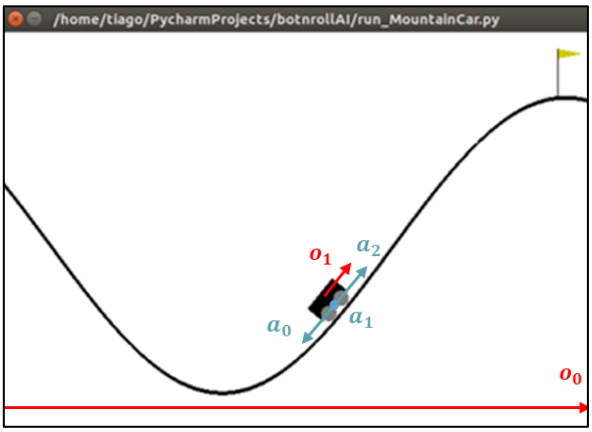


Figure 4.20 - MountainCar environment observations and actions.

III Initial State, Reward and Terminal States

The initial position is an arbitrary bounded position on the horizontal axis with no velocity: $-0.6 \geq o_0 \geq -0.4$ and $o_1 = 0$. For each time step t, a constant -1 reward is awarded to the agent until it reaches its success terminal state. This reward methodology privileges solutions that reach the success terminal

state as quickly as possible. The goal for MountainCar is to reach the flag on top of the right mountain. Computation wise success is achieved when the position observation is the same as the flag position. $o_0 \geq 0.5$. The only other terminal state results from a predefined iteration overflow, so the car does not get stuck in an episode, it cannot solve. The iteration overflow terminal state may be deactivated.

4.6.2 CartPole

The CartPole Environment, Figure 4.21, consists of a pole attached by an unactuated joint to a cart that is moving along a frictionless track. The goal is to keep the pendulum upright, preventing it from falling down by action over the cart's velocity. Cartpole also proved to be a virtuous Q-Learning and Policy Gradient implementation environment presented in chapter 5 and 6.

I Observation

The CartPole environment provides four observation parameters, shown in Table 4.5. These are the pole's angle and angular velocity and the cart's position and velocity. Each observation parameter has its own minimal and maximal value and are visually characterised in Figure 4.22 as the red shapes.

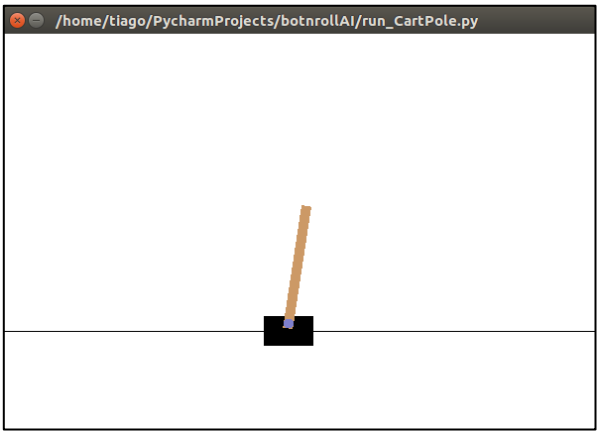


Figure 4.21 - OpenAI Gym CartPole simulation environment.

Table 4.5 - CartPole observation.

Number	*Notation*	*Observation*	*Minimum*	*Maximum*
0	o_0	Cart Position	-2.4	2.4
1	o_1	Cart Velocity	$-\infty$	∞
2	o_2	Pole Angle	$\approx -41.8°$	$\approx 41.8°$
3	o_3	Angular Velocity	$-\infty$	∞

II Actions

The CartPole environment offers two action outputs, presented in Table 4.6. The actions consist of increasing or decreasing the cart's speed on the horizontal axis. The actions work as a sum of the previous speed and the increment or decrement of speed chosen as the present action. The amount the velocity is reduced or increased depends on the pole's angle. This happens since the pole centre of gravity may differ the amount of energy necessary to move the cart. Both actions are characterised in Figure 4.22 as the light blue shapes.

Table 4.6 - CartPole actions.

Number	*Notation*	*Action (Discrete)*
0	a_0	Push Cart to the Left
1	a_1	Push Cart to the Right

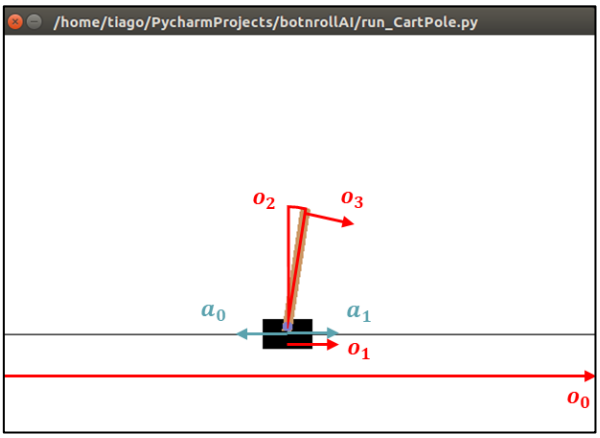


Figure 4.22 - CartPole environment observations and actions.

III Initial State, Reward and Terminal States

On the initial state, all the agent observations are assigned a uniform random value between ± 0.05. For every step taken, a constant reward of $+1$ is received from the environment. This reward methodology privileges longer episodes, which translates on the cart trying to balance the pole for as long as possible. There is no success terminal state since the cart goal is never to drop the pole. However, this environment has three different terminal states:

- Pole Angle is higher than $\pm 12°$;
- Cart Position more than ± 2.4;
- Episode Length is higher than T.

When the pole reaches the critical angle, $\pm 12°$ it is considered that the pole has fallen, and therefore terminating the episode. If the cart position reaches the edge of the simulation display, the episode is also terminated. The last termination parameter is similar to MountainCar, but in this case, to avoid the algorithm to getting stuck in a long episode, after successfully learning the task. T is the configurable value of maximum time steps in an episode.

4.6.3 Pendulum

None of the aforementioned environments can provide both continuous action-space and state-space. In robotics, it is not common to work with non-continuous environments since almost every information provided by the real world is continuous. Some real-world examples of continuous information regarding robotics are motor speed, the position of a particular object, the distance to a wall, the next position coordinates and strength required to move something.

Therefore, an environment with continuous action-space and state-space is necesary. The Pendulum environment, Figure 4.23, consists of a classic control problem. A frictionless pendulum starts in a random rotational position with an arbitrary angular velocity, and the goal is to swing the pendulum up, so it stays upright. The Pendulum environment allowed the study and test of Deep Deterministic Policy Gradient, further described in chapter 7.

I Observation

The Pendulum environment has three observation parameters, shown in Table 4.7. The first two are the cosine and sine of the angle θ, regarding the actual pendulum angle. The third observation is the angular velocity of the pendulum. All three observations have finite minimum and maximum values. The observation parameters for the pendulum environment are displayed in Figure 4.24.

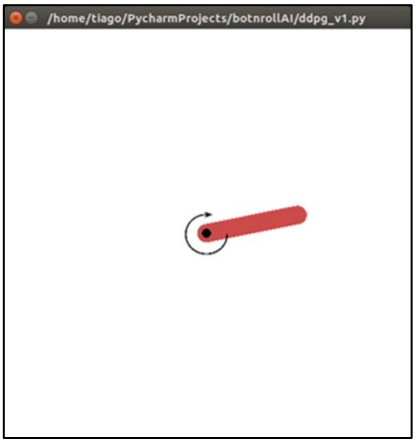


Figure 4.23 - OpenAI Gym Pendulum simulation environment, with continuous action-space.

Table 4.7 - Pendulum observation.

Number	*Notation*	*Observation*	*Minimum*	*Maximum*
0	o_0	$\cos\theta$	-1.0	1.0
1	o_1	$\sin\theta$	-1.0	1.0
2	o_2	Angular Velocity	-8.0	8.0

II Actions

Table 4.8 shows the single continuous action output in the Pendulum environment. It consists of the effort made on the joint to move the pendulum to a vertical position.

Table 4.8 - Pendulum actions.

Number	*Notation*	*Action (Continuous)*	*Minimum*	*Maximum*
0	a_0	Joint Effort	-2.0	2.0

This algorithm may sometimes present a wobbly stance and the need to invert the joint effort as balance its way to the top position. The action parameterisation is presented in Figure 4.24 as the light blue shapes.

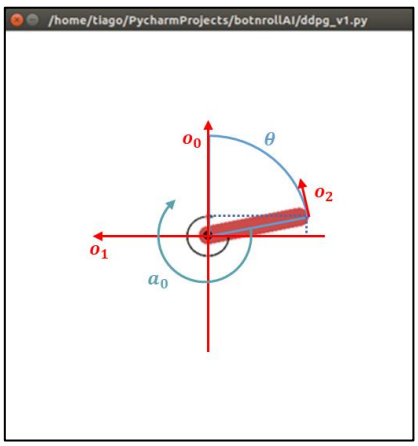


Figure 4.24 - Pendulum environment observations and actions.

III Initial State, Reward and Terminal States

The initial state is a random angle, bounded as $-\pi \leq \theta \leq \pi$ and an arbitrary velocity constrained as $-1 < o_2 < 1$. The reward function differs from the previous ones, since the complexity level increases. It depends on the present angle, the angular velocity and the chosen action. The selected action forces

the agent to solve the problem while selecting actions with lower values for smoother control. The reward equation is as follows:

$$R_t = -\left(\theta^2 + 0.1\frac{d\theta^2}{dt} + 0.001{a_t}^2\right) \quad \textbf{(4.1)}$$

θ is normalised between $-\pi$ and π. Therefore the further the pendulum is from the top position, the higher the angular velocity is, and the greater the action is chosen, the lower the reward is. The highest possible reward is zero. Thus the pendulum goal must reach the vertical position as soon as possible, while selecting low actions, and maintaining its control position. There are no terminal states. However, similar to the other two environments, the configurable maximum number of time steps is added for the environments never to get stuck in infinite episodes.

4.7 Conclusions

In this chapter, a thorough comparison between the two simulation environments, V-REP and Gazebo, was presented. The agent, a Bot'n Roll ONE A robot was designed on both simulators, to perform a practical comparison between the simulators. V-REP was chosen as the simulator to be used to implement the reinforcement learning algorithms due to its numerous physics engines, comprehensive model library, possibility to perform world interaction during the simulation and, most significantly, mesh manipulation and optimisation. The agent was designed in high detail requiring a significant mesh and in a simplistic version using only primitive shapes. The simplistic version was selected due to its effective computation performance.

A detailed description of how the simulated agent was developed is also presented in this chapter, as well as an environment description and development to fit the RoboParty Obstacle Maze challenge criteria. With the ROS framework used, describing every ROS node and ROS topic, it is possible to connect the simulation environment to the control script, linking all the information necessary for the reinforcement learning algorithms to work successfully. Lastly, in this chapter, the embedded scripts necessary to customise the simulator were described to allow a direct interface with the agent, which was not possible with only the ROS framework.

Three OpenAI Gym Environments, MountainCar, CartPole and Pendulum, were chosen to serve as a test platform for all the reinforcement learning algorithms described in chapter 3 and implemented on chapters 5, 6 and 7.

5

Q-Learning

Q-learning is a model-free, off-policy Temporal Difference control algorithm. It seeks to find the best action to take given a specific state. It only uses trial-and-error methods and estimations to predict the value function. Its off-policy characteristic allows the Q-learning functions to learn from actions that are outside its current policy. This TD control algorithm is one of the founding state-of-the-art strategies and is widely used in many low-level reinforcement learning solutions.

This chapter is divided into two parts. The first, consists of implementing Q-Learning on two different OpenAI Gym's environments, MountainCar and CartPole. This practical application will provide the initial feedback for the second part. A thorough explanation of every hyperparameter will be described and further compared to different Q-Learning tests.

The second part refers to the Q-Learning implementation on the created simulated agent, Bot'n Roll ONE A. Q-Learning will be the first reinforcement learning method tested on the created simulated robot. In this chapter, the algorithm will work with all the robot state-space constraints. Therefore, digital distance sensors will be considered as discrete values. Two different Q-Learning approaches are tested and compared as well as detailed hyperparameter study.

5.1 Implementation and Testing on Gym's Environments

The Q-Learning strategies implemented on MountainCar and CartPole environments are significantly similar. The only change between the two strategies is the difference in the observation and action-space. The remainder of the implementation is identical since Q-Learning must be easily adapted for a wide range of problems to be a suitable reinforcement learning method. For different problems, different hyperparameter configurations must be adapted for finer tuning. A hyperparameter is a parameter whose

value is set before the training process begins. Whereas, the values of the remaining parameters are calculated while training.

The algorithm can be divided into two parts, the model and the environment interaction. The model is the Q-Learning algorithm core. It starts by initialising the feature transformation class since the observation is composed of continuous variables in both cases. The feature transformation is used to convert a state into a feature representation. Then it is used a scaler to concatenate the results of multiple transformer objects. The RBF kernels, Figure 5.1, are used as an approximation method, with different deviations to cover different parts of the state-space. This approximation method applies a list of transformer objects in parallel to the input data, then concatenates the result. This is useful to combine several feature extraction mechanisms into a single transformer.

The creation of a Stochastic Gradient Descent (SGD) regressor per each action in an environment works similar to having an output neuron. Each SGD Regressor will use the data from the feature transformation composed by the RBF kernels to return a value. The regressor that returns the higher value is the greedy action for that state. The model class has three functions named predict, update and sample action. The predict function takes a state and after running through the SGD regressors, returns the greedy action. The update function receives the (s, a, G) tuple and updates the estimator. The sample action function takes the state and ε, to decide whether to perform exploration by choosing a random action, or exploitation, by sending the state to the predict function and returning the result.

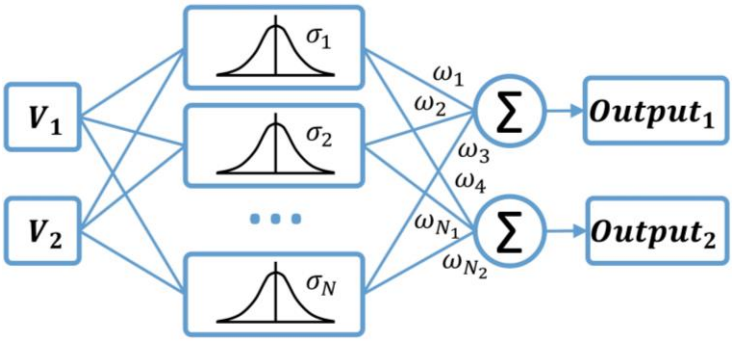


Figure 5.1 - Example of the neural network using RBF neurons used in this Q-Learning implementation.

The environment interaction is the software module responsible for connecting the model to the simulator. It includes all the functions necessary to send and receive data from the simulation environment. It starts its initialisation by writing on the environment registers, the agent configurations. Next, the algorithm defines the value of the hyperparameter. At the moment, all the steps necessary to start the Q-Learning algorithm have been carried out. Thus, the function to perform an episode is presented. It starts by resetting the environment, then from the initial state s_0 selects the action using the ε-greedy exploration

method and consequently receives the new state s_1. After receiving the new state, it calculates the return according to the Q-Learning function, for the present time step and updates the model. After the model update, it goes back to sending a new action to the environment repeating the same process all over. This algorithm is ran until the episode encounters a terminal state.

In both simulation environments, when a set of episodes ends, the graphs for that specific hyperparameter configuration are presented. The figures with four independent blue graphs represent the total reward achieved on each episode. The higher the reward achieved, the better the agent performed in that episode. In MountainCar, the objective is to reach the goal in the least number of iterations possible. Thus, the reward intended must be the highest negative value possible, since the reward are always negative for this environment. The optimal solution is for the rewards to tend to 0 over time. In CartPole, the objective is a bit different since the goal is to balance for as many iterations as possible. The reward intended must be as high as possible, meaning the agent can balance the pole for a great number of iterations. The optimal learning system has the total reward values tending to the defined maximum number of iterations. The graphs that present four different coloured lines are the running average rewards. These graphs demonstrate a comparison of the total rewards with smoother lines regarding the learning process. They represent essential information to determine how the learning process evolves throughout the episode. This is important since the total reward graphs sometimes produce results with high reward variation, and sometimes not providing a straightforward analysis. A high reward variation means that the rewards achieved change significantly from episode to episode, providing graphs with abrupt variations. In MountainCar it is possible to represent a three-dimensional graph, named cost-to-go, demonstrating the action-value for each set of observation pairs. This demonstrates the value determined by the agent for each state. This graph is negated, meaning the red values part of the graph represents the worst values and the blue part of the graph the best values. To end the algorithm, the user may choose to visualise the demonstration episodes using the learned policy π, to observe what the agent can perform.

5.1.1 MountainCar

For every simulation, it was possible to conceive three different graphs: a) the total reward for each episode that indicates how good each episode was; b) the reward running average that specifies how the learning evolution is proceeding over time; c) the cost-to-go function, that is the value of each action-value in a three-dimension graph. Since MountainCar only has two input variables, namely, the cart's speed and position, it is possible to arrange a three-dimensional graph concerning the value function for both

observations. For every simulation, a constant number of episodes is performed, in this case, a thousand episodes are performed, and the learning process throughout the episodes is compared. The approximate simulation time is also considered for this study.

I Discount Factor

The discount factor is the hyperparameter that determines the importance of future rewards. For MountainCar, four different discount factors are presented and compared. The values for the different discount factors are as follows:

$$\gamma_1 = 0.95, \gamma_2 = 0.97, \gamma_3 = 0.99, \gamma_4 = 0.999 \quad \textbf{(5.1)}$$

All the presented discount factors have the same set of other hyperparameters configurations, $\varepsilon_3 = 0.1 \times 0.97^n$. The graphs on Figure 5.2, compares the total reward on each episode for all the tested discount factors values **(5.1)** as well as in Figure 5.3 the average running reward of the last hundred episodes, to conclude the evolution of each discount factor.

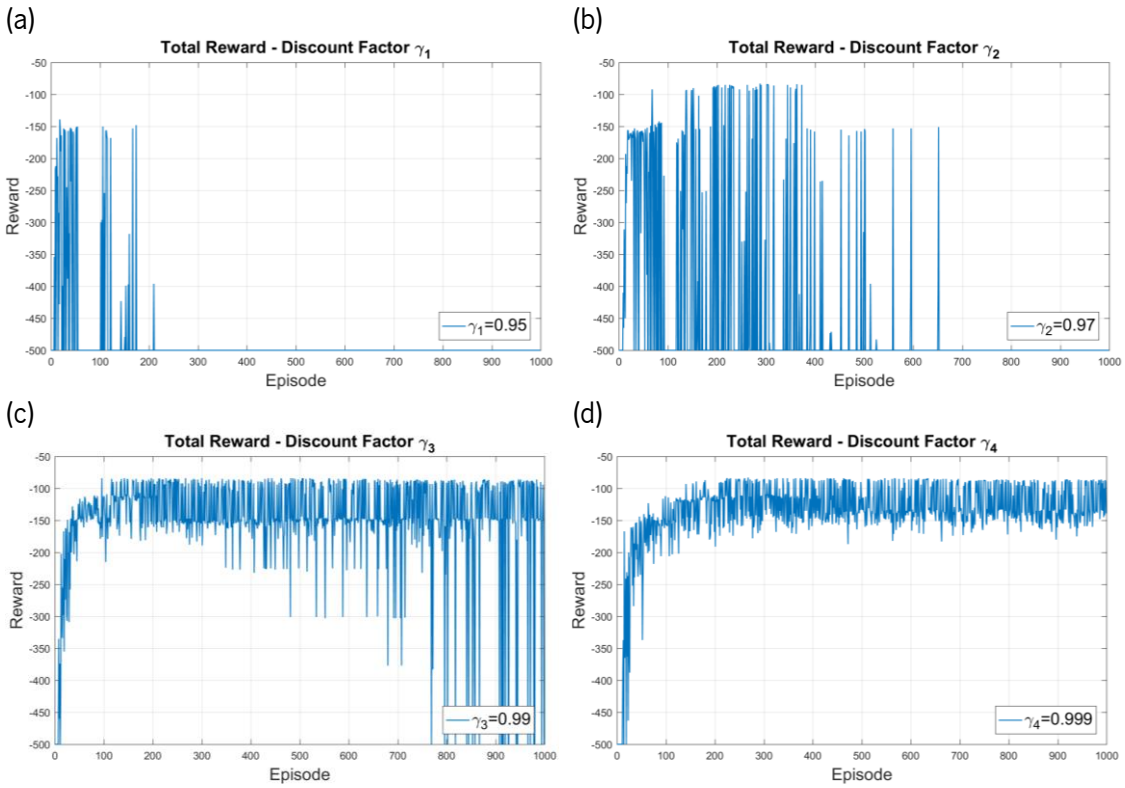


Figure 5.2 - Q-Learning MountainCar total reward comparison between different discount factors.

Both graphs show that for MountainCar when the discount factor tends to 1, the learning process is consistently more successful. The first discount factor γ_1 (a), is not capable of learning how to solve the problem. Initially, on the first 200 it solves some episodes but is not able to continue the learning process. γ_2 (b) the algorithm is still not able to solve the problem. However, it manages to solve more episodes

than γ_1. γ_3 (c) the evolution of the discount factor continues, and the algorithm already finishes the 1000 episodes having solved the problem but with a suboptimal policy. γ_4 (d) solves the problem very successfully with an optimal policy and a robust learning process. It manages to maintain an average reward over -150 throughout the 1000 episodes. In Figure 5.4, it is presented the cost-to-go function for the four selected discount factors. These graphs compare the value the algorithm has defined for the state-space observation. This type of graphs gives a visual idea of how the policy interacts with the environment and how good or bad the algorithm thinks it is to be in a specific observation.

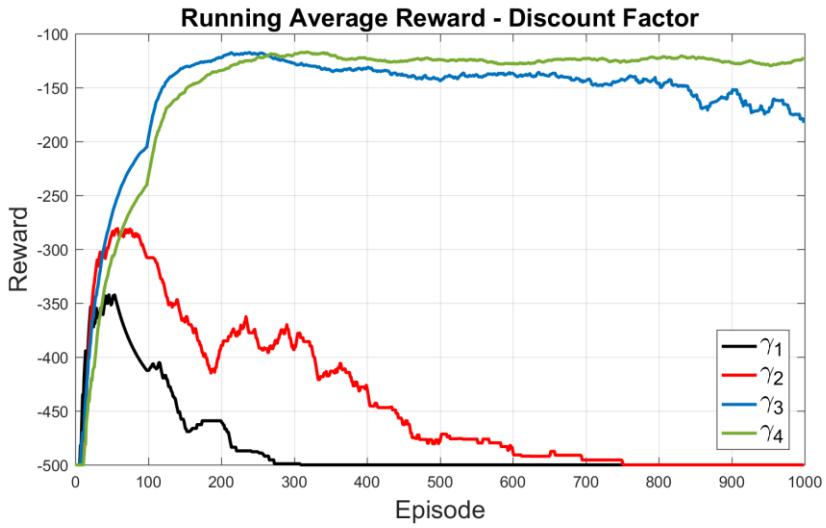


Figure 5.3 - Q-Learning MountainCar running average reward comparison between different discount factors.

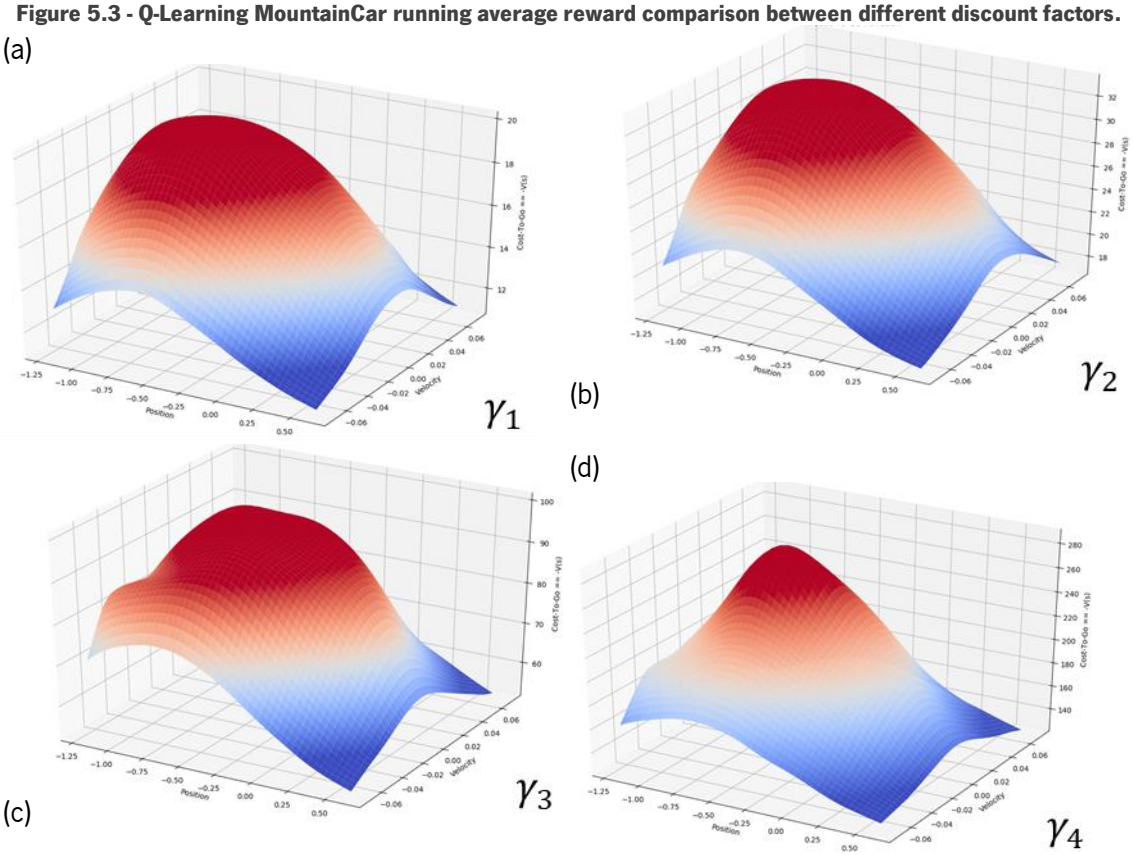


Figure 5.4 - Q-Learning MountainCar cost-to-go functions comparison between different discount factors.

In conclusion, Figure 5.2 and Figure 5.3 provide evidence that for MountainCar, it is a better solution to trust almost entirely on future rewards rather than the immediate reward received. It also shows that even for the discount factor γ_3 the learning softly deteriorates towards the end of the thousand episodes, thus the need to have a discount factor as close to the unitary value as possible. Figure 5.4 compares the value considered by the algorithm for the full state-space for the selected discount factors.

II Exploration Method

The exploration method selected for this implementation is ε-greedy. Four different equations were tested in order to determine which would better fit this specific learning method. The discount factor used is $\gamma = 0.999$ since it was determined to be the best of the studied discount factors. The equations are as follows:

$$\varepsilon_1 = 0.1, \varepsilon_2 = \frac{0.1}{0.1(n+1)}, \varepsilon_3 = 0.1 \times 0.97^n, \varepsilon_4 = 0.1 \times 0.99^n + 0.05 \quad \textbf{(5.2)}$$

Where n denotes the episode's number. In Figure 5.5, a graph comparing the four selected exploration functions is presented. It is possible to see that with the exception of the first one, they all differ in their initial values, the final value and the slope they have throughout the thousand episodes that results on different decreasing explorations.

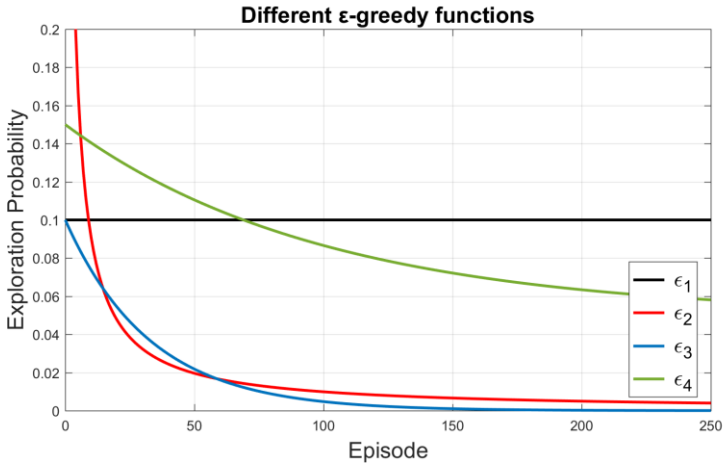


Figure 5.5 - Different ε-greedy functions.

Figure 5.6 and Figure 5.7 demonstrates the impact these ε-greedy explorations have on the total and average rewards over the thousand episodes. With the exception of ε_1 (a), the other three graphics demonstrate a very similar learning process. For ε_2 (b) ε_3 (c) and ε_4 (d) the learning does not present any significant changes as demonstrated in Figure 5.7. The three running average values continuously overlap. The difference between these three and ε_1 (a), is due to the non decay exploration throughout the 1000 episodes it will perform a more exploratory number of actions. In the running average reward

graph, it is possible to see the ε_1 line consistently below the other three. In Figure 5.8, the cost-to-go functions regarding the first four ε-greedy methods are presented. Similarly to the average rewards, this image demonstrates that the state-value function for the four exploration methods is considerably similar for all observations. Resulting in four very similar graphs.

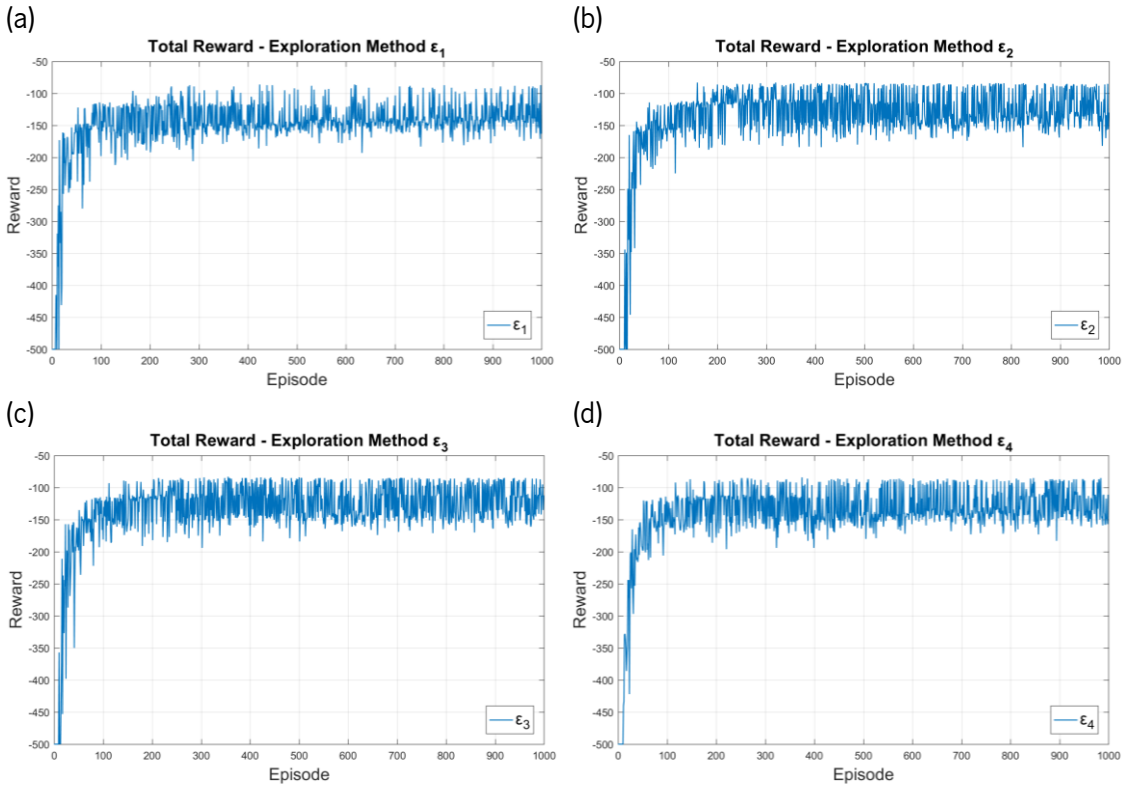


Figure 5.6 - Q-Learning MountainCar total reward comparison between different ε-greedy functions.

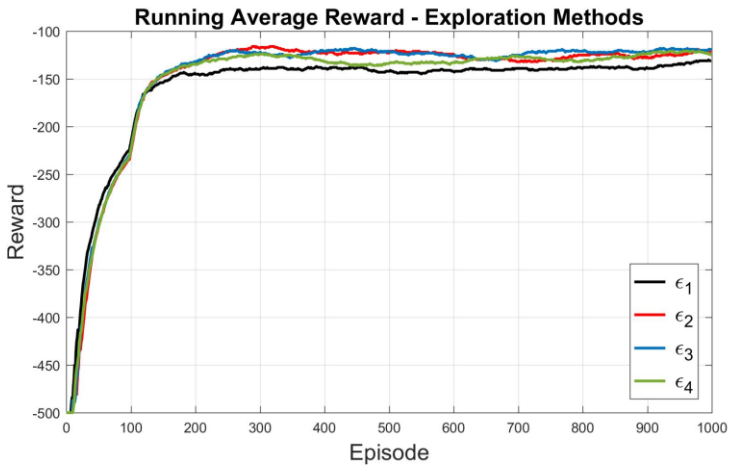


Figure 5.7 - Q-Learning MountainCar running average reward comparison between different ε-greedy functions.

In conclusion, all the ε-greedy functions have similar performances, as observed in Figure 5.6, Figure 5.7 and Figure 5.8. This determines that even though the functions are different, they do not differ significantly enough. Thus it is not possible to determine if one is considerably better than the other since the average rewards provided change even with all the same hyperparameters due to arbitrary variable initialisation.

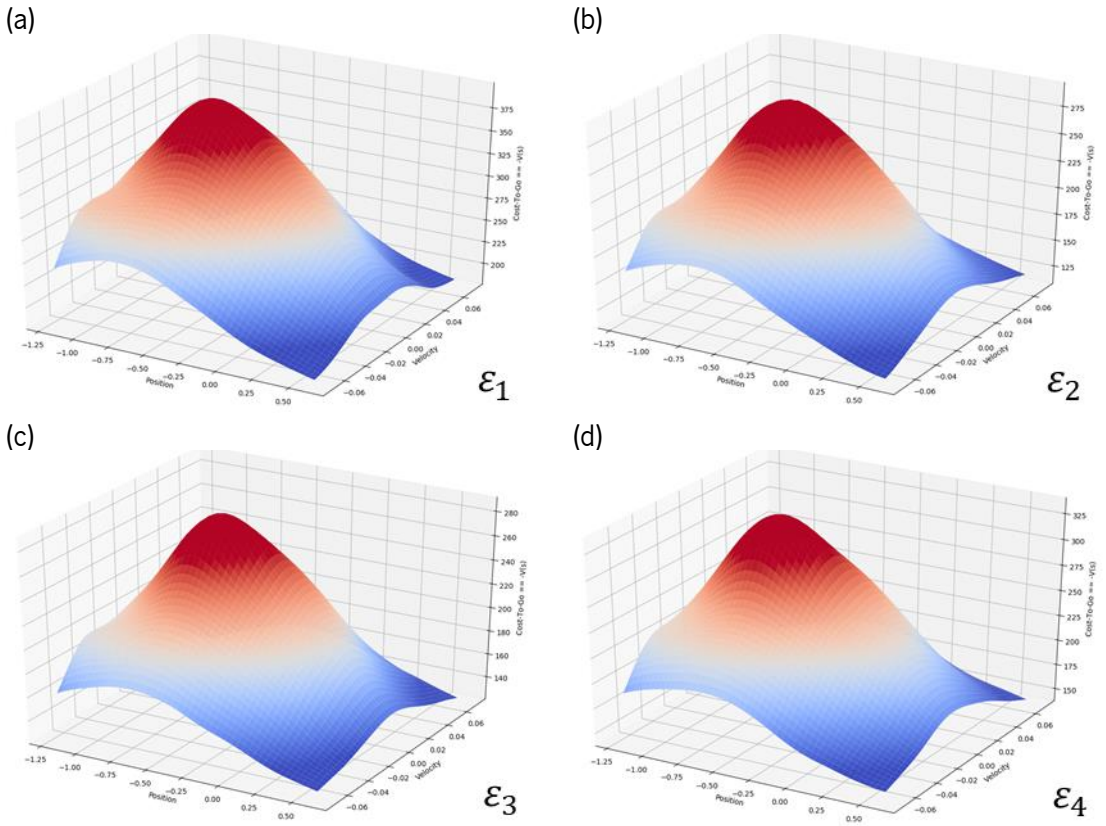


Figure 5.8 - Q-Learning MountainCar cost-to-go functions comparison between different ε-greedy functions.

III Long-Term Simulation

An important test to be considered is the long-term functionality. This test is made using ten times more the number of episodes used on the previous tests. The agent must perform ten thousand episodes to determine its learning stability. In Figure 5.9, both the total rewards and the average reward for each episode are provided. Even though the total rewards appear to have some reward variations in the learning, the average rewards show that the agent successfully learned the task. These variations are due to the agent's initial observation, that sometimes makes it impossible for the agent to reach the goal using only once the force provided by the left mountain. On average if the agent can reach the goal with just one force than it returns a total reward of -90. If not, the total reward can vary between -120 and -140. The agent can learn an almost optimal solution to MountainCar in approximately 200 episodes.

The cost-to-go graph of the long-term experiment shown in Figure 5.10 demonstrates learning outcome when the algorithm has thousands of episodes to learn from. The value function can considerably differ from the ones present in the previous sections. Allowing the algorithm to become much more efficient. In (d) the graph demonstrates that for a position next to the goal, the velocity given is not so important since the momentum gained is hardly shifted to a non-successful solution. The worst observation considered by the algorithm is when the position is at the beginning of the left mountain and the velocity

is null. A strong forward velocity at the top left mountain is also the best observation in this position since the agent starts its forward push to gain momentum.

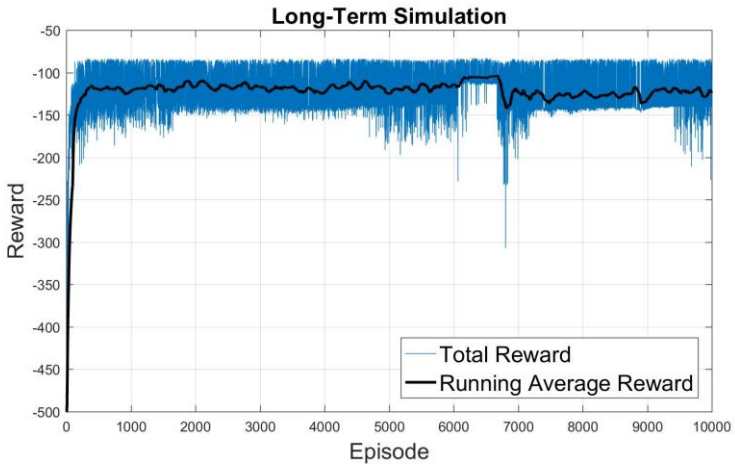


Figure 5.9 - Q-Learning MountainCar total and running average reward long-term simulation.

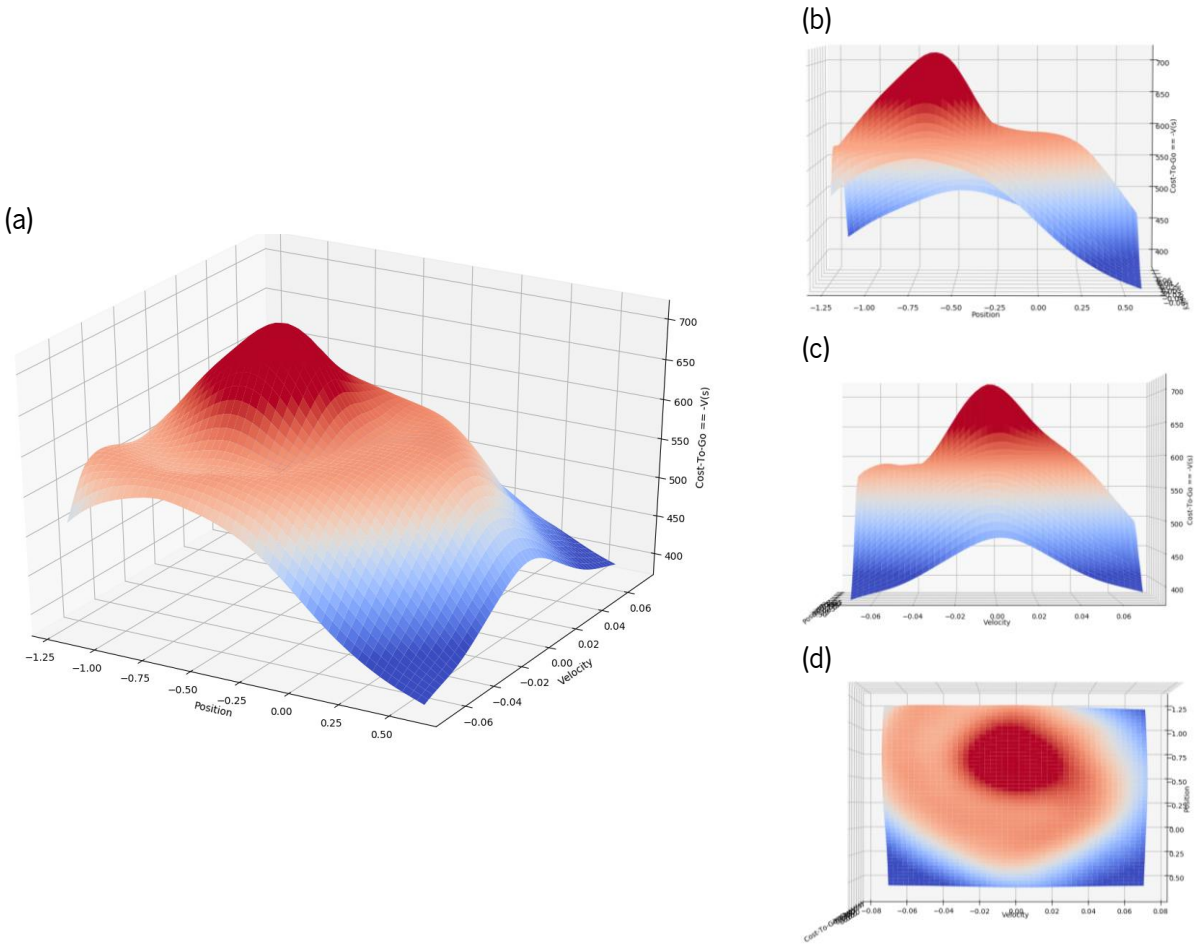


Figure 5.10 - Q-Learning MountainCar cost-to-go function long-term simulation.

One of the most important features of this graph is the C shaped white shade around the peak of the cos-to-go function. This shade is precisely the value intended for the backwards agent movement on positions lower than 0. Gaining the necessary momentum to overcome the right mountain slope.

The long-term test took nearly 62 minutes to conclude, while previous tests took an average of 7.5 minutes. The time is not proportional due to the performance difference. Episodes with higher number of time steps take longer to conclude. In MountainCar an episode with the maximum number of iterations 500 takes approximately 1.5 seconds.

5.1.2 CartPole

On Cartpole environment, since the state-space consists of four different observations, two graphs are presented for each hyperparameter configuration. The first is the full reward for each episode and the second is the average reward of the last hundred episodes, which shows a smoother and easier learning progress perception. Apart from the hyperparameters discussed in the previous subchapter, a new reward configuration is considered for a comparison to the standard reward function returned by the environment. For every simulation, a predefined number of a thousand episodes are performed, with exception to the long-term test. The approximate simulation times for each configuration are also presented.

I Discount Factor

In Figure 5.11 and Figure 5.12 are plotted the different total reward and average rewards for different discount factor s, respectively. Compared discount factors are as follows:

$$\gamma_1 = 0.95, \gamma_2 = 0.97, \gamma_3 = 0.99, \gamma_4 = 0.999 \quad \textbf{(5.3)}$$

The exploration method used is a standard decreasing ε-greedy, $\varepsilon = 0.1 \times 0.99^n$. From the information in the graphs, a discount factor of 0.95 (a) is not able to learn the specific task to equilibrate the pendulum. However, with the increase in the discount factor from 0.95, passing by 0.97 (b) and ending at 0.99 (c), it is demonstrated that the system increases its learning capability. It is going from finishing the 0.95 discount factor with an average of 500 time steps holding the pendulum, to being able to hold the pendulum on average for 2000 time steps at about 500 episodes.

When the discount factor is increased to a value over 0.99, it starts to be an inconsistent learner, as represented by the green line in Figure 5.12. γ_4 (d) after successfully reaching some episodes with a reward of 2000, that is the maximum value, unlearns the task and needs to start its learning process from the beginning. The problem is that, since the exploration method is continuously decreasing after 600 episodes, it is not easy to successfully return to learn an optimal policy.

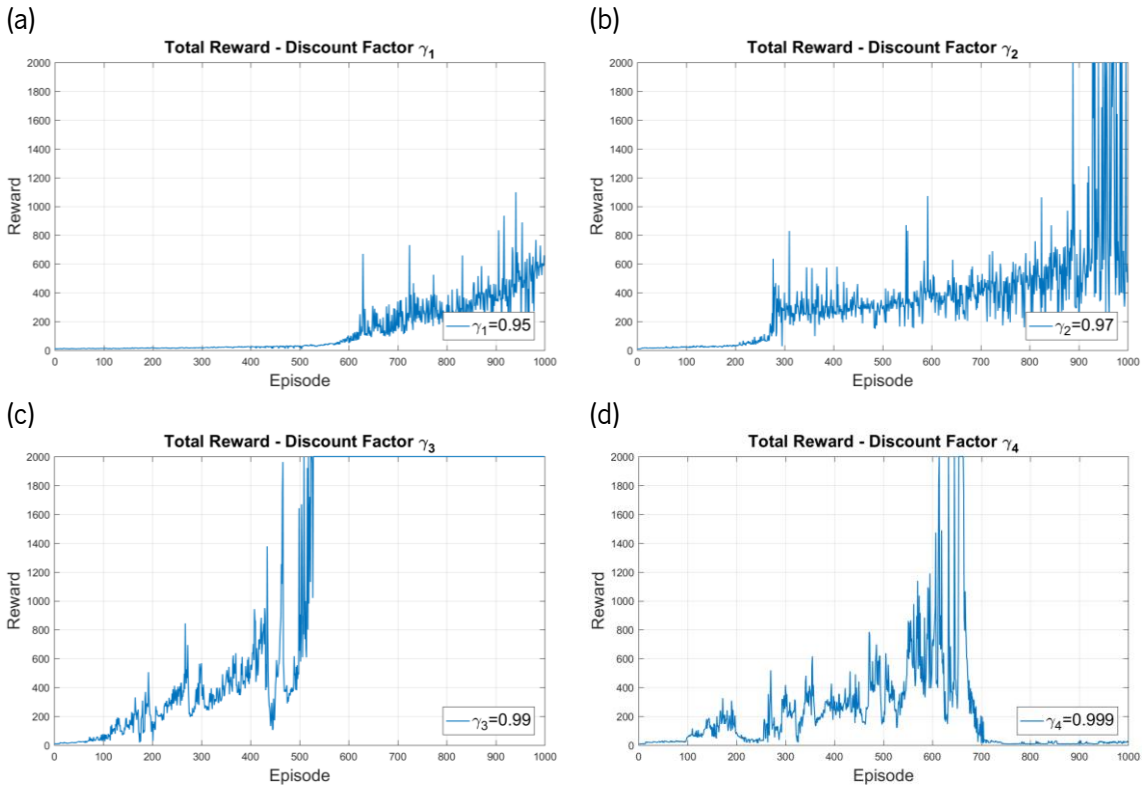


Figure 5.11 - Q-Learning CartPole total reward comparison between different discount factors.

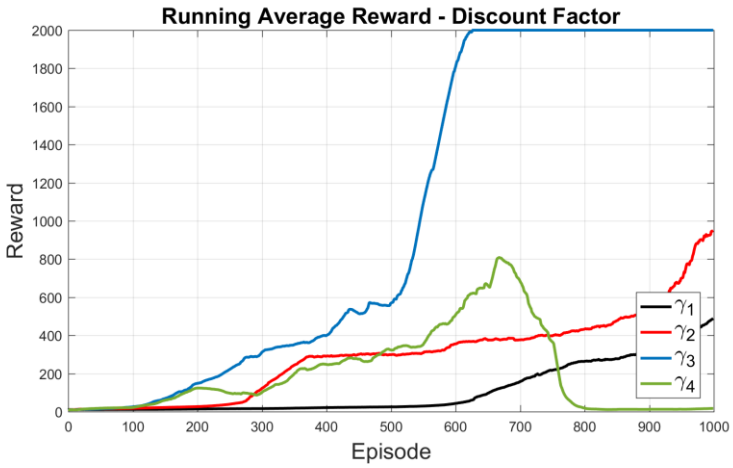


Figure 5.12 - Q-Learning CartPole running average reward comparison between different discount factors.

II Exploration Methods

As shown in the exploration methods on the MountainCar environment, the differences between the selected exploration methods were not relevant enough to make a significant conclusion. As so, in the CartPole Environments, a reduced comparison is presented, and ε-greedy methods are as follows:

$$\varepsilon_1 = 0.1, \varepsilon_2 = 0.1 \times 0.97^n, \varepsilon_3 = 0.1 \times 0.99^n, \varepsilon_4 = 0.1 \times 0.99^n + 0.05 \quad \textbf{(5.4)}$$

The used discount factor is $\gamma = 0.99$. In Figure 5.13, the total rewards for each exploration method demonstrate that methods with a constant ε (a), have a learning with higher reward variation. This

happens, contrary to the MountainCar results because the inverted pendulum is a highly unstable environment and just one exploration action may move the agent to a state where it is tough to come back to an equilibrium position, if not impossible. ε_2 (b) takes more time to reach an optimal policy than ε_3 (c). Whereas, ε_3 (c) and ε_4 (d) present very similar results. The difference between ε_2 (b), ε_3 (c) and ε_4 (d) lays on many episodes it takes the agent to discover an optimal policy. In Figure 5.14, the average rewards of all the tested exploration methods show that for CartPole, it is necessary a decreasing ε-greedy, preferably with a medium amount of exploration in the first hundred episodes.

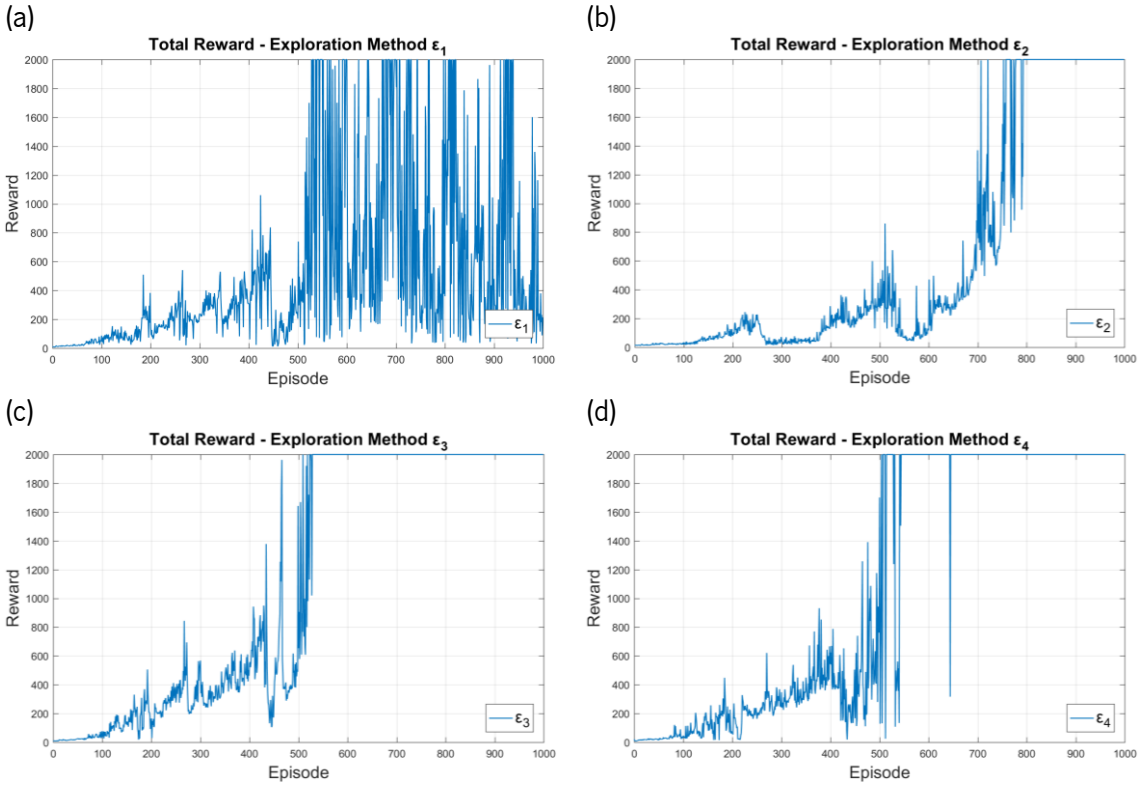


Figure 5.13 - Q-Learning CartPole total reward comparison between different ε-greedy functions.

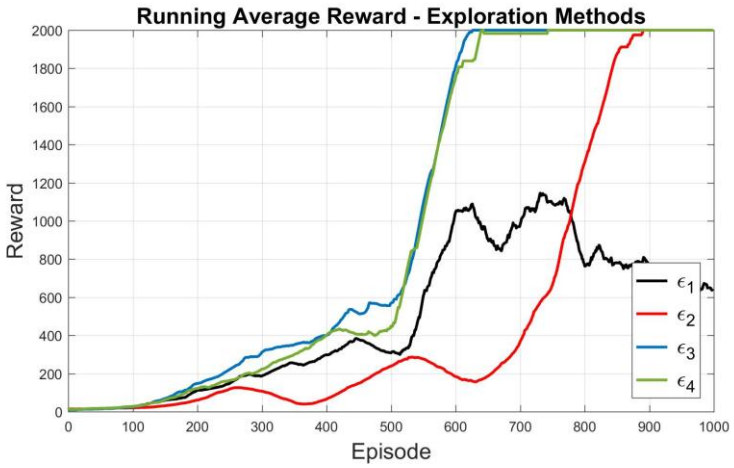


Figure 5.14 - Q-Learning CartPole running average reward comparison between different ε-greedy functions.

III Different Reward Configuration

The Gym Environments provide a standard reward function. For CartPole the environment returns a $+1$ reward for each time step the agent can equilibrate the pendulum. However, these can be adjusted to fit the algorithm's learning process better. An example of an adaption is to reward negatively when the pole falls. In this test, a -10 reward is applied whenever the pole drops.

This process creates a more significant value difference between the actions that led to the pole falling and the actions that were balancing the pole. This process forces the Q-Learning algorithm to learn faster. The defined hyperparameters are $\varepsilon = 0.1 \times 0.99^n$, and $\gamma = 0.99$ as demonstrated in Figure 5.15 and Figure 5.16, the solution with the adapted reward function learns an optimal policy quicker.

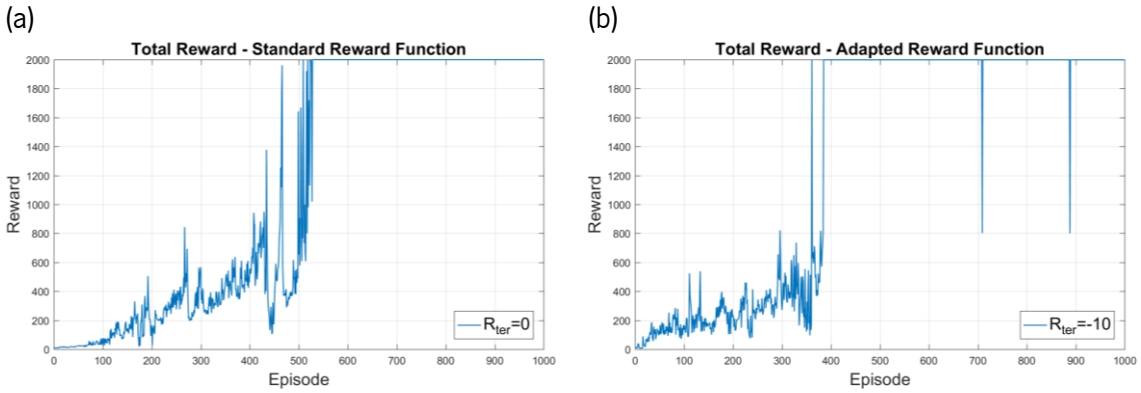


Figure 5.15 - Q-Learning CartPole total reward comparison between standard and adapted reward functions.

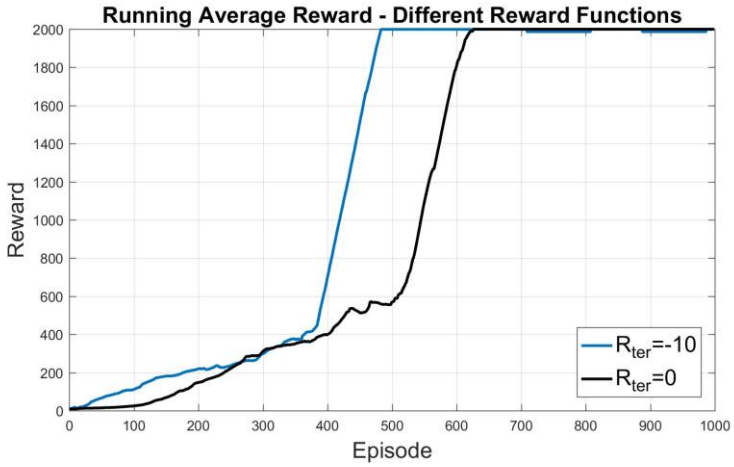


Figure 5.16 - Q-Learning CartPole running average reward comparison between standard and adapted reward functions.

IV Long-Term Simulation

The time for an average simulation with all the previous parameters is approximately between 23 minutes and 38 minutes. Contrary to MountainCar, where the episodes tend to be shorter as the agent is learning, in Cartpole the whole purpose is for the cart to balance the pole as long as possible. Thus the simulation time for Cartpole increases as the learning process starts to get better. For a long time simulation of

$10\,000$ episodes, the algorithm took about 12 hours and 50 minutes to conclude. The configured hyperparameters are as follows, $\gamma = 0.99$, $\varepsilon = 0.1 \times 0.99^n$ and the adapted reward function. This long-term simulation Figure 5.17, demonstrates that even though there were moments where the algorithm chose non optimal actions, it always managed to return to an optimal policy reaching the 2000 rewards consistently.

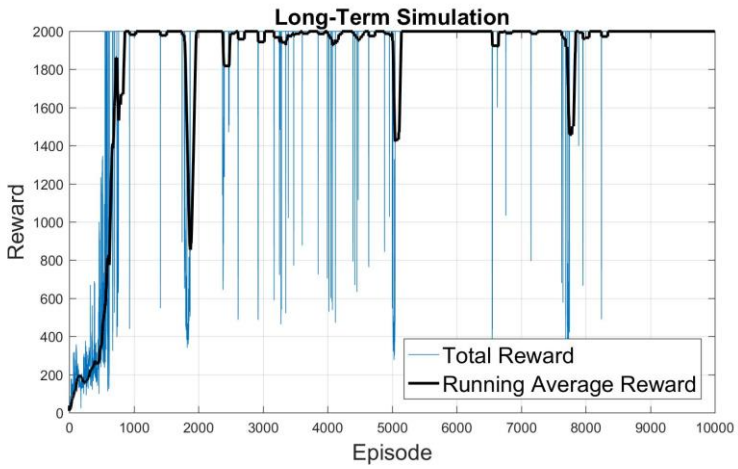


Figure 5.17 - Q-Learning CartPole total and running average reward long-term simulation.

The 2000 reward limit was defined as the maximum possible reward, and the episode terminates when reached. This limit had to be defined since the pole would discover policies that managed to balance the pole for hundreds of thousands of episodes.

5.2 Bot'n Roll ONE A

In this chapter, two different Q-Learning methods are described that were implemented and tested on a simulated Bot'n Roll ONE A. The goal is to solve three mazes with increasing complexity. The agent must go through the first maze for a specific number of episodes. Then, its learning is transferred to an agent that will try to solve the second maze. This learning process will be performed for the three mazes.

However, this implementation has distinctions from the two different Q-Learning implementations made on both MountainCar and CartPole. In the Gym environments, the state-space was continuous, where for the simulated agent, the state-space is discrete, due to the physical restrictions of the robot. The objective of this chapter is to implement Q-Learning in the simulated robot while using only sensors embedded in the physical robot. Some changes are necessary for the Q-Learning algorithm to be adapted to this problem, such as a different feature approximation method.

The two distinct Q-Learning methods were designed for an extensive task comparison and analysis. The different behaviour of both implementations serves the purpose of reasoning the best method. The

difference between the two lays on when $Q(s_t, a_t)$ is calculated, depending on the state transitions and actions executed. The implementations will be named Q-Learning method A and Q-Learning method B for a more straightforward explanation and a more natural understanding.

5.2.1 Reinforcement Learning Components

The reinforcement learning components introduced by the Q-Learning implementation differ from the components implemented using other algorithms for this specific environment. As so, in the following sections, a description of the different components regarding the two Q-Learning implementations is presented.

I State-Space

The observation is the sensory information resulting from the robot's environment observation. As previously described, the robot must only use its two infra-red distance sensors to detect the surrounding obstacles to define the current state. The infra-red distance sensors embedded on the Bot'n Roll ONE A are discrete, resulting in two binary sensor information. The number of state-spaces in this case study is four. In Figure 5.18 (a) are represented the robot's four possible states. The detection volume is conically shaped with a $20\ cm$ range and a $20°$ angle parameters.

II Action-Space

An action a is generated in the control script and sent to the simulation environment ROS node, which in turn is applied to the agent. It represents a choice of a pre-determined set of possible actions. The complete pre-determined set of actions consists of three forward, three backward, three right turns, and three left turns all with different speeds. Additionally, two self-rotation movements, one for each side. A total of 14 different possible actions are available for the agent, visually described in Figure 5.18 (b), where colour and size represent motion intensity.

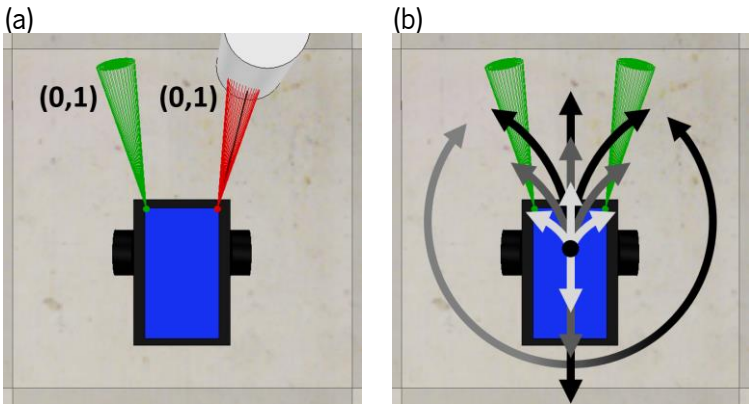


Figure 5.18 - Simulated agent state-space (a), and action-space (b).

III Policy

Q-Learning uses a state-action function Q, which is updated when an action a_t is taken at time step t, thus when all $Q(s_t, a_t)$ are maximised, the optimal policy is to select the actions with a maximum state-action function, $max_a Q(s_t, a)$.

IV Approximation Method

Since both the action-space and the state-space are discrete, it is over processing to create an approximation method, when the values are already discrete. The number of possible actions and possible states is considerably reduced and allow the implementation of solutions that do not overcomplicate. Thus, in this implementation, the tabular method was selected as a simple solution. The tabular method consists on the creation of a bidimensional array with the number of possible actions by the number of possible states. Each position is consequently an action-value condition. This method allows a straightforward use of the action-value function and its elements at all times.

V Reward Equations

The reward function is denoted as R, that represents the reward obtained for each action a, at time step t. The following function characterises the reward system:

$$R_i = \begin{cases} -1, per\ timestep \\ -10^3 \delta_u(i-1), per\ timeout \\ -10^3\left(40^{i-1}\right), per\ collision \\ 10^3\left(50^{i-1}\right), per\ success \end{cases} \quad \textbf{(5.5)}$$

Where i is a discrete variable from one to three that represents the maze level. The low negative time step reward forces the agent to try to reach success as fast as possible to maximise the reward. For the timeout, so the agent does not get stuck in an episode, δ_u stands for a unitary delta impulse function that can be represented as:

$$\delta_u(t-T) = \begin{cases} 1, t-T=0 \\ 0, t-T \neq 0 \end{cases} \quad \textbf{(5.6)}$$

It results in a negative reward only on the first level. This reward prevents the robot from getting stuck in a state or set of states, performing the same actions and never leaving the same spot or never being able to reach a terminal state. It is only used in the first level since it is essential to the initial learning and not so much in the later levels. The success and collision functions were calculated for an optimised robot's performance. The exponential increase characteristic is a crucial feature to solve the iterative complexity problem. Higher rewards for higher levels allow a coherent knowledge to solve all the different mazes..

5.2.2 Method A: New Action Every Time Step

On Q-Learning method A, the control script reads the sensor information, calculates the respective $Q(s_t, a_t)$ and sends the updated action to the simulator's ROS node. This occurs at a fixed Δt, which means the robot will go through a constant number of iterations per second. For every Δt, an action is calculated and selected via the exploration strategy, in this case, ε-greedy, and sent to the agent. This method allows the agent to experiment with different actions, even though there is no state change.

In this method, a total of a hundred episodes were performed for the first and second level mazes. The 100 episodes performed for each maze turned out to be insufficient for the agent to learn an optimal policy successfully. In Figure 5.19, the results in the form of rewards for this method are presented. The graphs only show the first and second levels together, since this method could not successfully solve the second maze. The orange line represents the total reward for each episode, and the blue line represents the average running reward. As seen, the robot barely solves the first maze, assuming a much more basic strategy than the one displayed in method B. The average reward line demonstrates a sub-optimal policy compared to the one in method B.

In Figure 5.19 are presented the total reward for each episode and the running average rewards for the last fifty episodes. These represent the reward evolution, which allows an understanding of the robot's level of successfulness. In Figure 5.20 are plotted the most noticeable changes made on Q on each episode. Showing how the highest action-value is adjusted for each episode, granting an understanding of how the intelligence system is evolving. Lastly, in Figure 5.21, the number of collisions denoted by the red line, the number of timeouts denoted by the blue line and the number of successes denoted by the green line are displayed. This allows a visual reference of how many collisions, timeouts and successes the agent faces along the learning process. On Table 5.1 the hyperparameter transition table is presented for the different levels. These values were adjusted for optimised learning.

(a) (b)

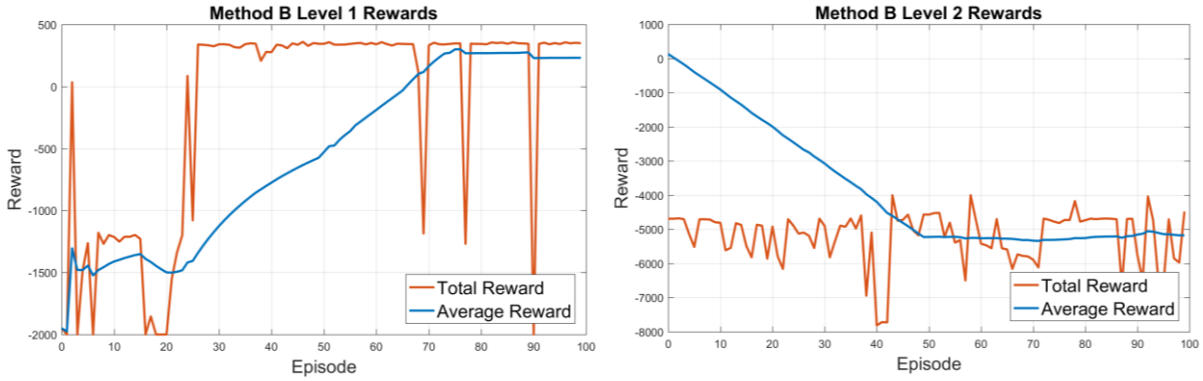


Figure 5.19 - Q-Learning Bot'n Roll ONE A, method A total and average rewards for the first two levels.

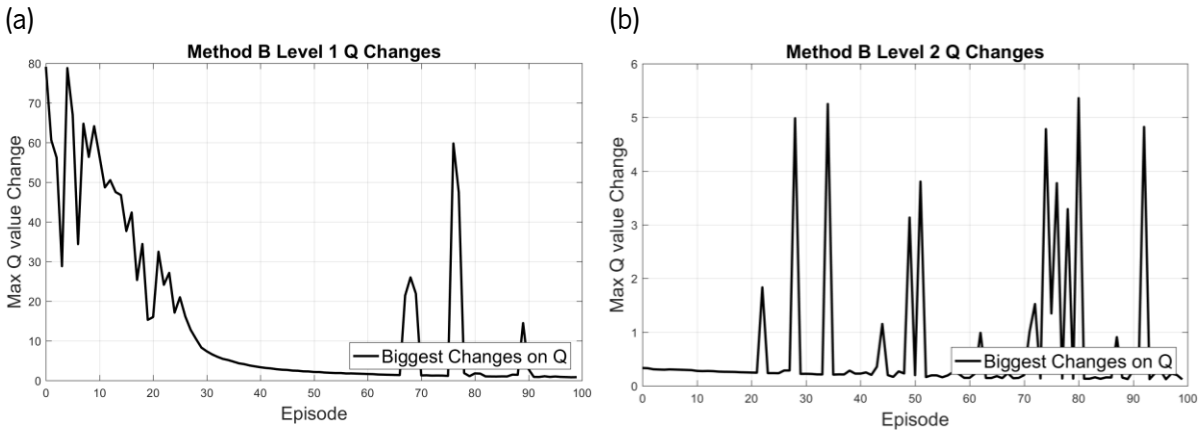


Figure 5.20 - Q- Learning Bot'n Roll ONE A, method A biggest change on the action-state function Q.

Figure 5.21 - Q-Learning Bot'n Roll ONE A, method A successes, collisions and timeouts.

Table 5.1 - Hyperparameter configuration for Q-Learning method A.

Hyperparameters	*Level 1*	*Level 2*	*Level 3*
α *(learning rate)*	0.1	0.01	0.0001
r_c *(collision reward)*	-1000	-4000	-1600000
r_s *(success reward)*	+1000	+5000	+2500000
r_t *(timeout reward)*	-1000	0	0
Timeout counter	1000	4000	6000
ε_{ini} *(initial ε-greedy)*	0.05	0.01	0.01

5.2.3 Method B: New Action Whenever the Observation Changes

The second implementation, Q-Learning method B, has the same Δt characteristic, meaning it will read the sensor value and calculate $Q(s_t, a_t)$ the same number of times per second. However, it differs from the previous method since it only chooses a new action when a state change occurs. The agent will keep performing the received action a and calculating the action-value function Q until a state transition occurs. Only when the sensor data differs, a new action is calculated, meaning an action will only go through the ε-greedy method whenever the state differs. In its essence, this can be interpreted as using options as introduced in [58]. The main difference between the two methods happens when the agent remains in the same state for a significant amount of time. Method A can alter the action the agent is performing every iteration if so the $Q(s_t, a_t)$ and the exploration strategy select. Also, method B must remain with the chosen action until a state change occurs.

The graphs for this method are presented in Figure 5.23 and Figure 5.24. The labels and indices are equal to the ones presented on method A. The total and average reward in Figure 5.22, the most significant changes in Q in Figure 5.23, and the number of different occurrences in Figure 5.24. For this method, the same number of a 100 episodes per level were performed for all three mazes. This value revealed to be enough for the agent to learn the specified task successfully. As presented, the simulated robot successfully learns the three mazes. However, when transferring to the following level, the robot needs to adapt to the different environment by adjusting its Q values. In Figure 5.22, the blue line demonstrates that the agent is successfully learning a strategy in order to solve the maze. In Table 5.2, the hyperparameter transition table between levels is presented. These values had to be adjusted for optimised learning.

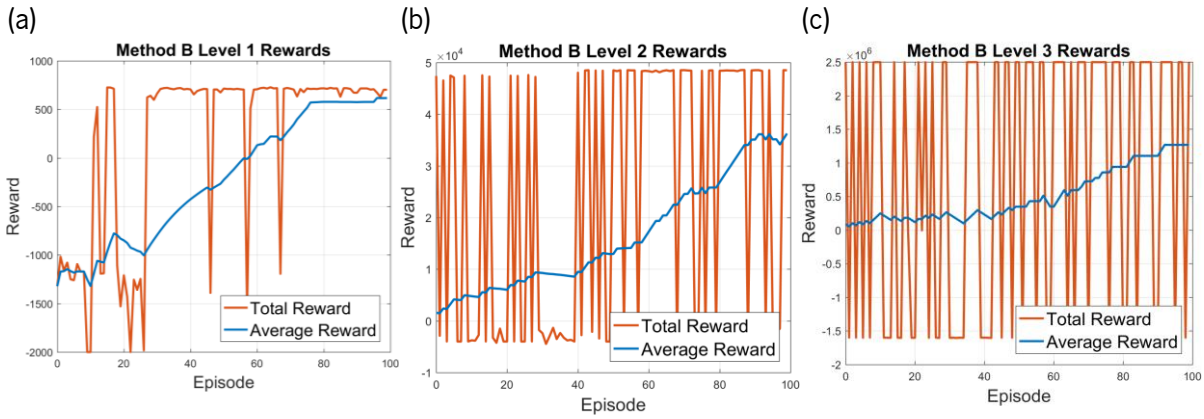


Figure 5.22 - Q-Learning Bot'n Roll ONE A, method B total, and running average rewards.

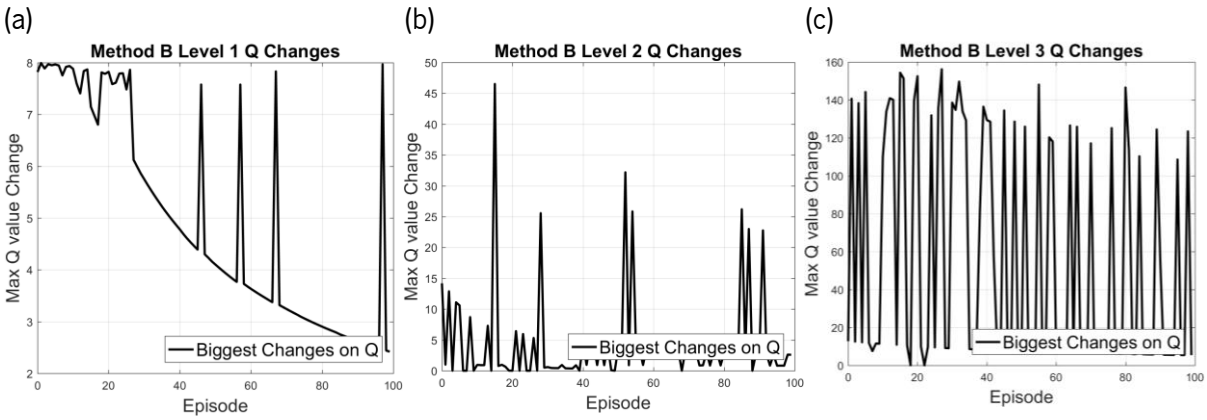


Figure 5.23 - Q-Learning Bot'n Roll ONE A, method B biggest change on the action-state function Q.

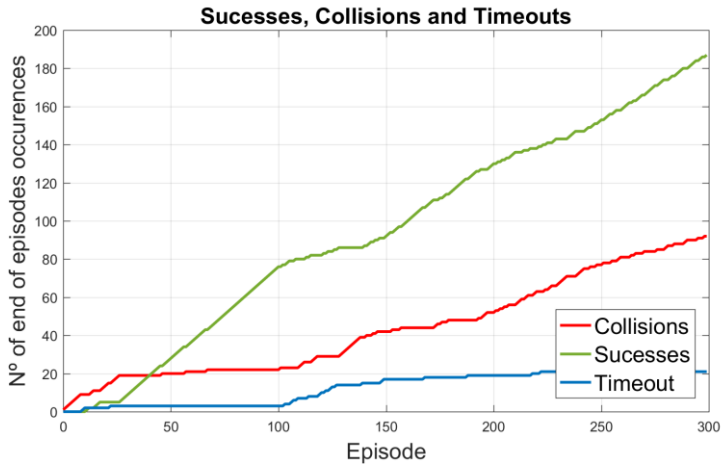


Figure 5.24 - Q-Learning Bot'n Roll ONE A, method B successes, collisions and timeouts.

Table 5.2 - Hyperparameter configuration for Q-Learning method B.

Hyperparameters	*Level 1*	*Level 2*	*Level 3*
α *(learning rate)*	0.008	0.001	0.0001
r_c *(collision reward)*	-1000	-4000	-1600000
r_s *(success reward)*	+1000	+5000	+2500000
r_t *(timeout reward)*	-1000	0	0
Timeout counter	1000	4000	6000
ε_{ini} *(initial ε-greedy)*	0.05	0.01	0.01

The value evolution of each action is another essential parameter. Tracking this information allows an action-space as well as action transition, proper understanding. Figure 5.25 presents the different actions

by colour, and the different action-value evolutions are plotted throughout 100 episodes on the same state. This extract is the method's B "Right Obstacle" state from the second level maze. For a "Right Obstacle" state, the agent's natural response should be to turn left, however in this case, not always the agent had that perception.

Due to exploration resulting from the ε-greedy exploration method, the agent finds a better solution. In this case, "Front Medium" represented as the black line, has a higher value than the "Left Strong" action, represented as the blue line, around episode 18. This solution caused the action-value to change its maximum action for this state. With time, the agent realises that the "Front Medium" action, which previously was considered to be a great answer, was probably a lucky action, and starts to decrease its trust on "Front Medium", eventually falling to a sub-optimal action. However, due to another exploration, that action managed to surpass again the now optimal action which then slowly again eventually fades, and the agent manages to continuously increase its trust on the "Left Strong" action. This trust continues to rise until the end of the training.

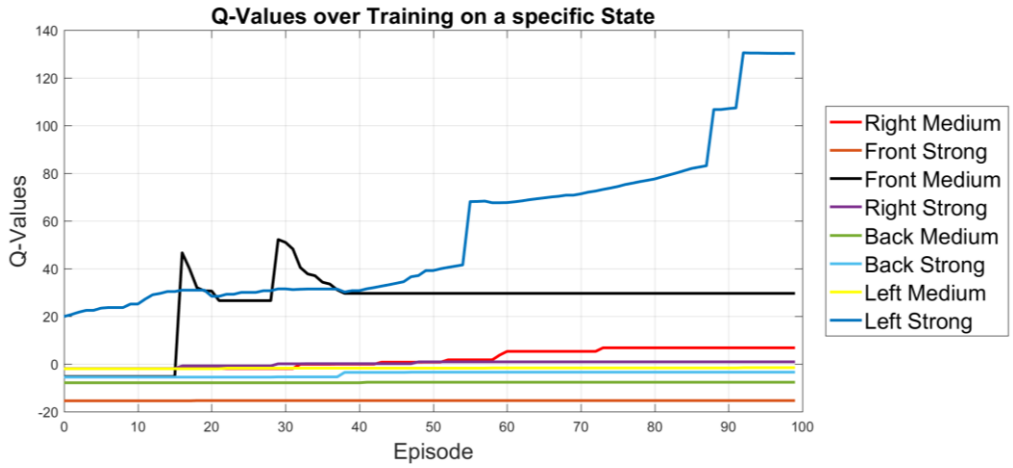


Figure 5.25 - Action evolution for method B, "Right Obstacle" state from the second level maze.

Another essential parameter studied is navigation evolution. In Figure 5.26, a visual representation of the navigation evolution is presented for this method. The colours red, orange, yellow and green correspond to the episode's number for each maze as described in the figure's label. These navigation details show the robot's navigation evolution throughout the reinforcement learning process. As demonstrated, the robot manages to adjust its parameters for each maze to increase its success rate as well as the amount of maze completed. For the first maze, the robot is able to learn two different successful policies to reach the endpoint. For the second and third maze, the robot manages to successfully improve its performance by making minor adjustments to Q. Overall, the robot acomplishes a strategy to either go further in the maze or find new policies to solve the navigation and obstacle avoidance problem.

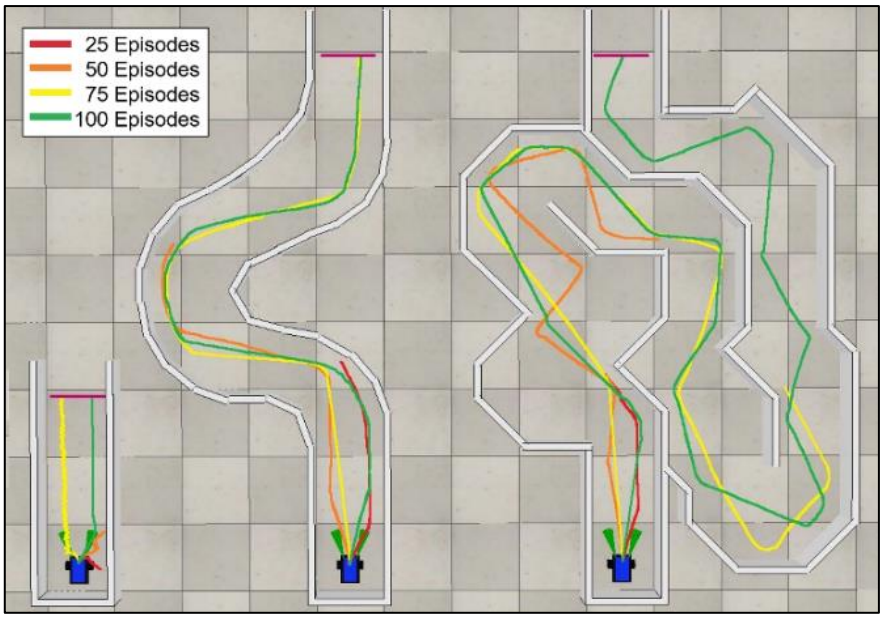


Figure 5.26 - Robot's navigation evolution for the three mazes every 25 episodes.

5.2.4 Results Discussion

Method A revealed to be a slow and safe but ineffective learning method for the case study even though the significantly high number of different hyperparameters tested. As described in Figure 5.19, Figure 5.20 and Figure 5.21, it managed to solve the first maze with some long times. However, it never managed to solve the second level maze. Not even with an extensive parameter search, it was possible to increase the method's learning capability. The rewards graph Figure 5.19, demonstrates that the agent tends to stagnate its learning. The policy derived by the agent on the first level can solve the maze, however, when tested on the second level, it tried to apply the same policy, yet, not effectively. This is mainly due to the considerable larger turns on the second maze.

In conclusion, this method was able to learn a sub-optimal policy that allows the agent to reach the goal on the first maze yet slower than method B. The policy mainly consists of continuously turning when it sees no obstacle, so it can toggle between moving against the object and moving away from the object. Since it chose two actions that when vectorially summed have a positive value for each motor, the agent manages to go through the first level maze very slowly and yet in a safer way. This turned out to be an impossible policy for the second level. This can be observed in Figure 5.21 since the robot has a very high number of collisions and an almost null number of timeouts or successes in the second maze.

Method B revealed to be a quick learning method. As seen Figure 5.22, Figure 5.23 and Figure 5.24, it managed to solve the three mazes on fewer episodes than method A. The most significant changes on the Q graphs in Figure 5.23 reveal that the agent had a smooth learning on level one, on level two a more

unstable learning and on level three a considerably unstable learning. However, to solve the three mazes, the reward equations had to grow exponentially which demonstrates good learning behaviour. The rewards graphs, Figure 5.22, reveals that the agent with the episode's course manages to consistently increase the average reward, which results in successful learning throughout all episodes. The occurrences graph in Figure 5.24 shows a slight slope decrease on successes for every level change, that results in slope increase on the other two lines. This method was able to learn to avoid walls on both sides and to go forward as fast as possible when there are no objects in sight.

In Figure 5.25, the agent makes use of the explore and exploit strategy efficiently. Initially, it starts by exploiting the "Left Strong" action, and then exploring the "Front Medium". This exploration was a positive try, resulting in a new optimal action. By exploiting this new optimal action, the agent continuously concludes that this action is not as virtuous as it predicted, returning the "Left Strong" action to be the optimal action. This process happened once again, and even then, the agent was capable of overcoming the sub-optimal action.

The robot's navigation evolution, as demonstrated in Figure 5.26, displays the different policies throughout the training in the three levels. For the first maze, the robot starts by experimenting different random policies. The navigation on episode 75, represented in yellow in the first level, shows a non-optimal policy the robot figures out. This policy ends up solving the problem by turning around until it sees a wall and then toggling between left and right turns until it reaches the endpoint, similar to method A. With more episodes, it figured a new policy that increases reward reaching the endpoint quicker. For the next two mazes, the learning process consists of making minor policy adjustments so that it can reach a further point in the maze with a higher speed.

In conclusion, both methods can solve the first maze, method A with a slower and safer strategy and method B with an agile approach. Each agent managed to develop a different solution to the same problem mainly residing on the differences between methods and their exploration strategies. Method A uses Optimistic Initial Values much more significantly than ε-greedy and method B works the other way around. A different reward function structure could be implemented to adapt to the method A strategy. Method B revealed to be an up-and-coming solution to this specific problem, solving all three mazes on a reduced number of iterations. Different solutions originated different parameters, methods and strategies, which demonstrates the consistency and potential of Q-Learning algorithms to solve straightforward obstacle avoidance problems.

5.3 Conclusions

In this chapter, the Q-Learning implementation in different environments was carried out. With these implementations, it is possible to conclude several aspects regarding the different agents and respective learning processes:

1. MountainCar was presented with different hyperparameters that can solve this environment. Different discount factors and exploration methods were tested and demonstrate that a Q-Learning implementation is a very versatile solution.
2. CartPole was presented as well with different hyperparameters. However, Q-Learning for this environment was not a very robust solution. Due to the inverted pendulum control problem instability, the hyperparameter configuration had to be more extensive. Long-term simulations also sometimes failed to learn. It was possible to present different parameter configurations that solved the problem very successfully, but these had to be hardly explored.
3. Two Bot'n Roll ONE A Q-Learning methods to solve an obstacle avoidance problem, more precisely the RoboParty Obstacle Maze challenge, were presented with three different mazes. Method A developed a safer and slower solution that was only able to solve the first maze. Method B adapted an agile policy to solve the three mazes. Moreover, it can continuously adapt and test the optimal actions throughout all the learning process.

The thorough work developed and described above allowed the publication of a scientific paper [59], named: Q-Learning for Autonomous Mobile Robot Obstacle Avoidance. This paper introduces, tests and compares both methods above described in this chapter.

6

Policy Gradient

Policy Gradient (PG) methods are a novel reinforcement learning algorithms technique to tackle model-free problems. PG methods target at modelling and optimising the policy directly. In this chapter, a Monte Carlo variant of PG is implemented. The agent throughout an episode collects a trajectory using its current policy and uses that to update the policy model once the episode finishes. The policy model is an artificial neural network named policy network. Since one full episode must be finished to create the tuple array necessary for the network optimisation, this is an off-policy method.

In this chapter, two Policy Gradient implementations are considered. The first regards the Monte Carlo Policy Gradient algorithm for OpenAI Gym environments MountainCar and CartPole. These serve as an initial platform to configure the methods. Thus, a detailed hyperparameter study is analysed and compared for different configurations. The second part of the chapter refers to a Policy Gradient implementation on the simulated Bot'n Roll ONE A. However, as an update from the previous chapter, the state-space restrictions of the physical distance sensors is abandoned. The sensors are now continuous distance sensors, for a realistic high-level robot obstacle avoidance situation.

6.1 Implementation and Testing on Gym's Environments

Similar to the Q-Learning implementation, the Policy Gradient method implemented in this chapter is identical for both OpenAI Gym environments. The only exception, similarly to the last chapter, is the action-space and state-space. The Policy Gradient strategy implemented is a Monte Carlo method. The policy model is based on Monte Carlo Learning as it waits until the end of the episode to update the model. The implemented model consists of a feedforward artificial neural network. The implemented ANN

is named policy network and is responsible for controlling how the agent acts. By tuning the policy network, updating the hyperparameters, different results are achieved. The model starts by initialising all the hyperparameters responsible for calibrating the neural network as well as some other model configurations. The policy network has one hidden layer and uses a $tanh$ activation function, with random normal distribution and a constant bias initialisation to connect the inputs to the hidden layer. The activation function from the hidden layer to the output is the SoftMax function since probabilities for each action must be provided. The selected optimiser for the ANN is the AdamOptimizer.

The functions provided by the model, are the action choice according to the probabilities returned by the policy-network, some debug print functions, the transition storage and the learning function. The transition storage is used to create the tuple necessary for the learning process. The learning function is the centrepiece of this Monte Carlo method since it is where the tuple is received and the update via backpropagation of the artificial neural networks occurs. The environment interaction is in all aspects similar to the one presented in the previous chapter since the environment is the same. Next, the hyperparameter configuration study is presented for the different configurable variables.

6.1.1 MountainCar

For the Policy Gradient implementation in MountainCar, two different graphs are presented, the total reward for each episode and the average reward for the previous hundred episodes. Even though the state-space still only has two possible observations, the cost-to-go graph is not presented since now it is a hidden process. The maximum number of time steps is $10\ 000$, after that the episode is terminated and a new one starts, to avoid situations where the agent gets stuck in a set of states. In this subchapter, a comparison between different learning rates, different reward decays and a different number of artificial neurons is demonstrated. Moreover, a long-term simulation is also displayed for long-term analysis.

I Learning Rate

The learning rate α, also known as step size determines how much newly acquired information prevails over previous information. A learning rate of 0 does not allow the agent to learn anything new, performing only exploitation of previous knowledge. A learning rate of 1 causes the agent only to consider the most recent information, thus ignoring previous knowledge to explore new possibilities. Typical α values tend to be very close to 0 since the algorithm performs better with smoother value transitions over time instead of unstable value changes. The following tests were performed with the same configurations for all the different hyperparameters, $\gamma = 0.99$, and $n = 10$. The test learning rates are the following:

$$\alpha_1 = 0.02, \alpha_2 = 0.01, \alpha_3 = 0.005, \alpha_4 = 0.002 \quad \textbf{(6.1)}$$

In Figure 6.1, a plot of four different learning rates compares the efficiency of each parameter, moreover in Figure 6.2, the average episode reward of each one of the four different learning rates is shown.

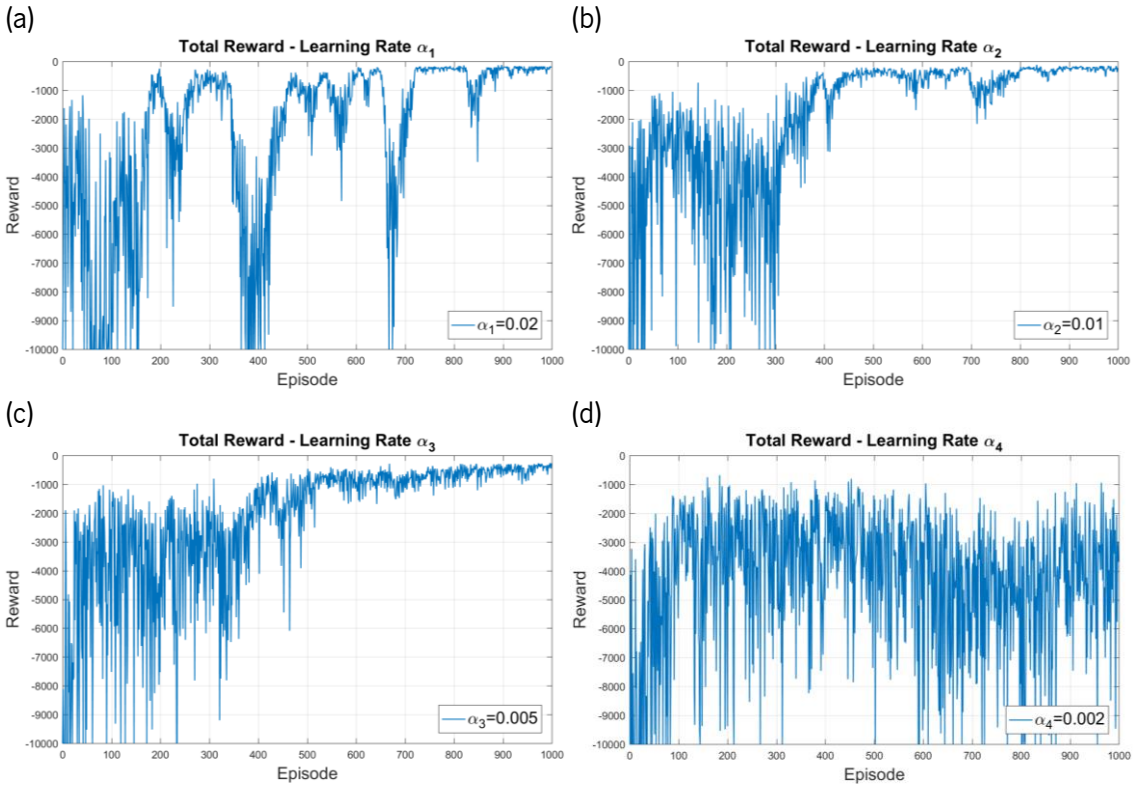


Figure 6.1 - Policy Gradient MountainCar total reward comparison between different learning rates.

The highest learning rate α_1 (a) shows unstable learning where the algorithm, after learning an optimal policy, adjusts and unlearns the task. However, it always recovers and relearns, yet with high reward variation. As the learning rate decreases α_2 (b) and α_3 (c), the algorithm improves the unlearning feature and after learning an optimal policy, it tries to improve smoothly. However, decreasing even more the learning rate creates new problems like the instability and reward variation demonstrated in α_4 (d).

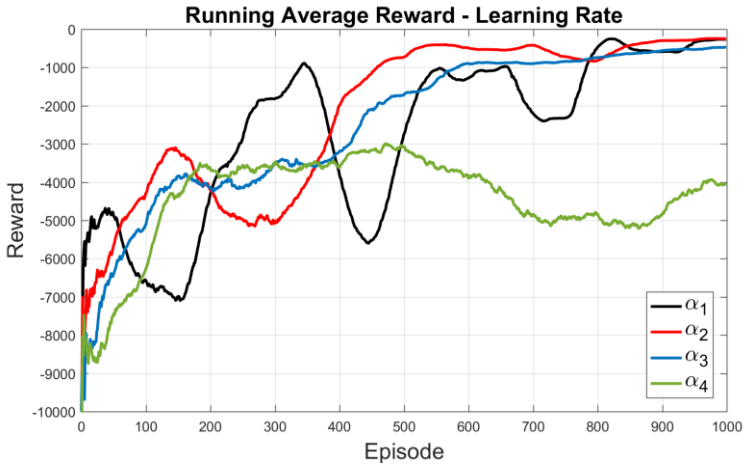


Figure 6.2 - Policy Gradient MountainCar running average reward comparison between different learning rates.

II Discount Factor

The discount factor γ determines the importance of future rewards. A comparison of four different discount factors in this environment with a learning rate $\alpha = 0.01$ and $n = 10$ is presented. The compared discount factors are:

$$\gamma_1 = 0.97, \gamma_2 = 0.99, \gamma_3 = 0.995, \gamma_4 = 0.999 \quad \textbf{(6.2)}$$

In Figure 6.3 for γ_1 or lower values (a), the algorithm is not capable of learning the task at hand. The low value compared to the others tested means that γ_1 (a) gives higher importance to reaching as soon as possible to the goal rather than the future values that may take the agent to the goal.

The difference between γ_1 and the closest tested value γ_2 (b) is only of 0.02. However, this difference makes a significant change in the learning process. From γ_2 on, (c) and (d), the agent successfully learns how to solve the MountainCar problem, with just some minor differences regarding reward variation, plotted in Figure 6.4.

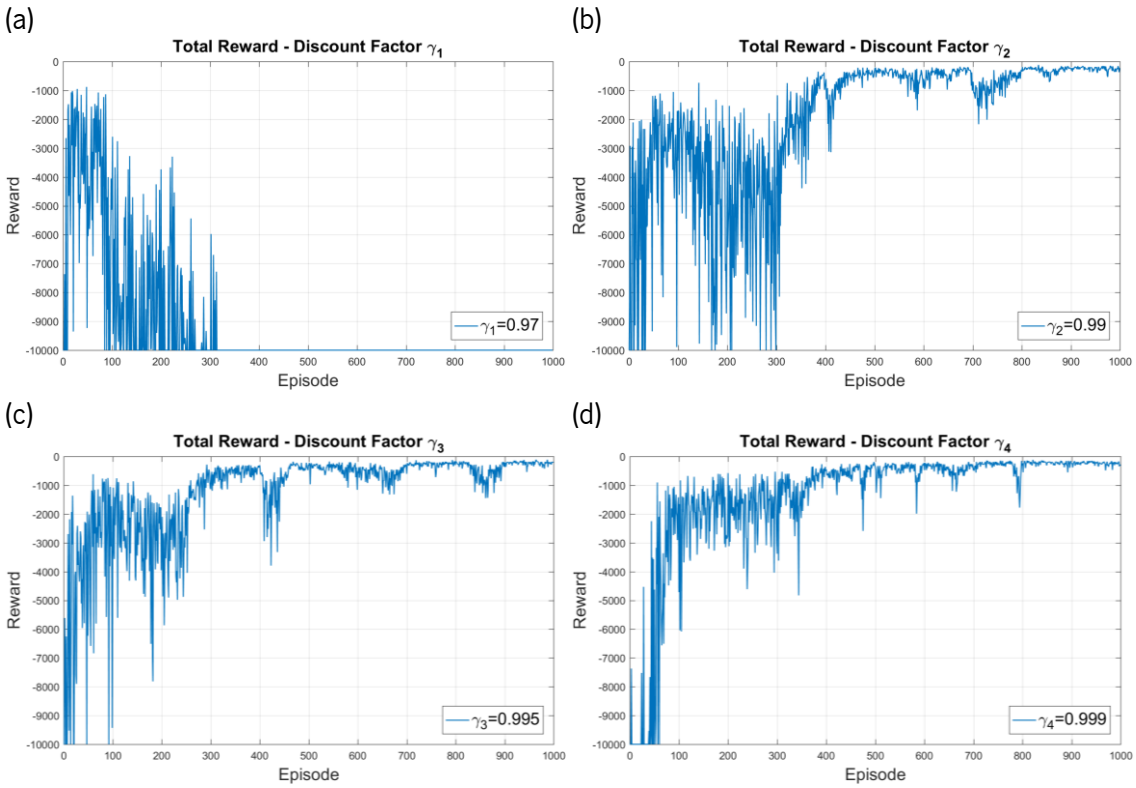


Figure 6.3 - Policy Gradient MountainCar total reward comparison between different discount factors.

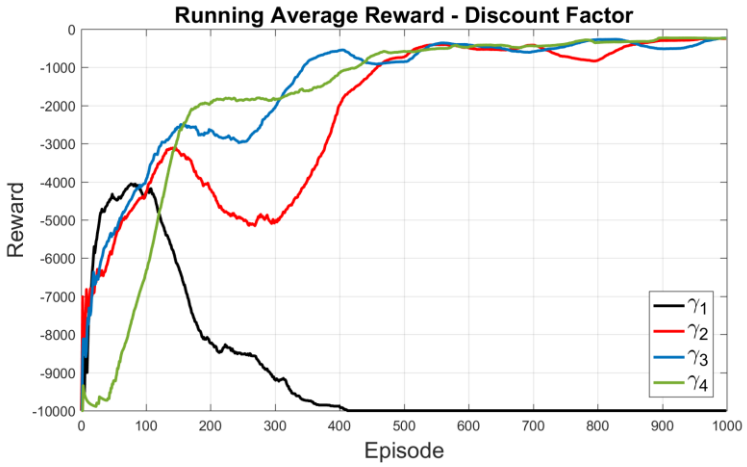


Figure 6.4 - Policy Gradient MountainCar running average reward comparison between different discount factors.

III Number of Artificial Neurons

A different hyperparameter introduced by the Policy Gradient algorithm is the different number of neurons in the neural networks. The policy network has a single hidden layer as part of its architecture, and differences in the number of units in the hidden layer are compared. The increase of neurons of an ANN network increases computational-costing, thus requiring higher processing units or more time to do the forward and backpropagation. The other hyperparameter values are $\alpha = 0.01$ and $\gamma = 0.999$. The different number of neurons tested are as follows:

$$n_1 = 5, n_2 = 10, n_3 = 20, n_4 = 40 \quad \textbf{(6.3)}$$

Similar to the other hyperparameters, Figure 6.5 demonstrates the plots of the total reward of each episode for a different number of hidden layer neurons, and Figure 6.6 the running average of the rewards throughout the episodes. A higher number of neurons such as n_4 (d) increases significantly the model complexity. When working with small action-spaces and state-spaces, these solutions tend to overcomplicate the problem, thus not being able to solve it. For implementations with a higher number of neurons, like n_4 (d) and even n_3 (c), it is demonstrated that the learning is not as smooth as a lower number of neurons like n_2 (b) and n_1 (a). For n_1, n_3 and n_4 around episode 600, they all try to adjust the optimal policy, even though this results in a learning deterioration. However, the three solutions are able to work around this and with a different number of episodes returning to a policy that solves the task. The two largest number of neurons, n_3 and n_4, increased the initial learning considerably, as displayed in the average reward graphs in Figure 6.6. The difference between the four different configurations and their efficiency may result as a part of the chosen learning rate and the discount factors.

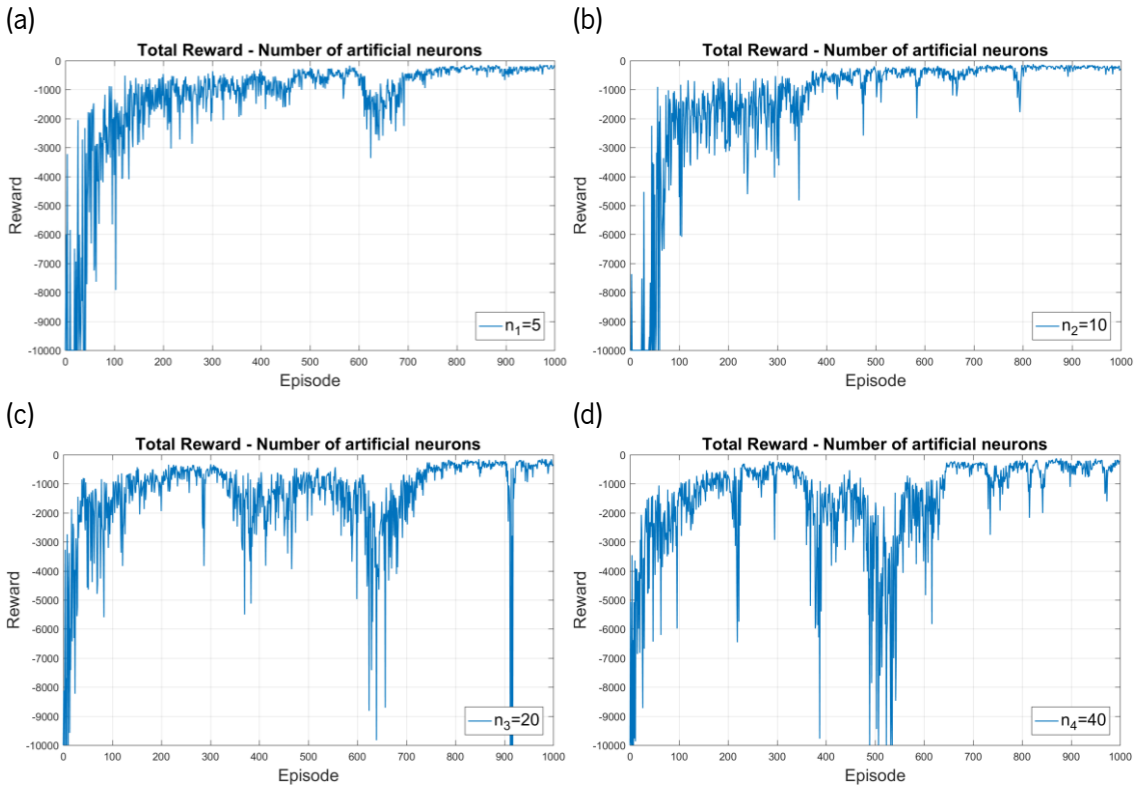


Figure 6.5 - Policy Gradient MountainCar total reward comparison between different number of neurons.

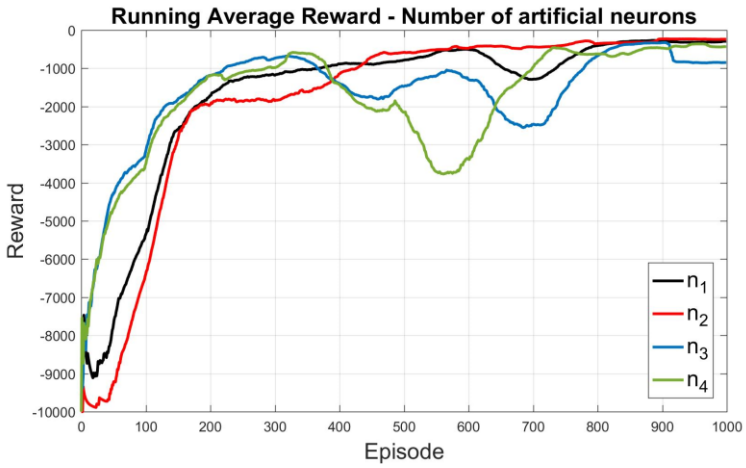


Figure 6.6 - Policy Gradient MountainCar running average reward comparison between different number of neurons.

IV Long-Term Simulation

The long-term simulation was set with the following hyperparameters:

$$\alpha = 0.01, \gamma = 0.999, n = 10 \quad \textbf{(6.4)}$$

In Figure 6.7, the total and running average are plotted. The algorithm successfully learns to perform actions to solve the problem. It is shown that this method explores different policies throughout the learning process and successfully adapts when these results in suboptimal policies. This long-term simulation took around 33 minutes to run, whereas the short-term simulations took approximately 5

minutes. The consistency of the average regarding the last 7000 episodes demonstrates the robustness provided by this learning method.

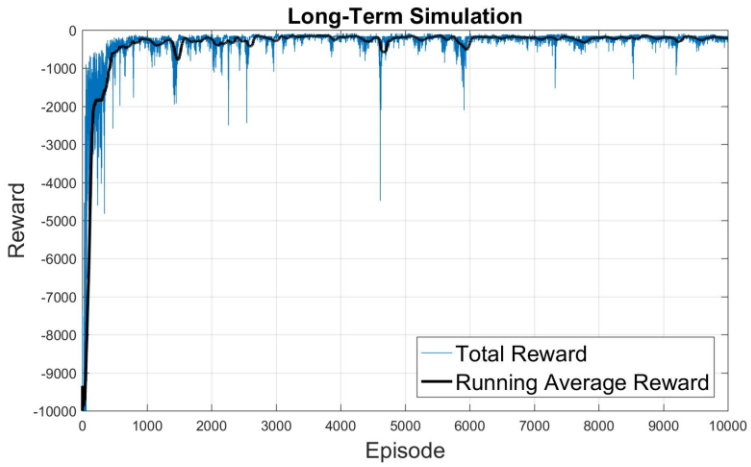


Figure 6.7 - Policy Gradient MountainCar total and running average reward long-term simulation.

6.1.2 CartPole

For the Cartpole environment, the same two graphs are presented, the total reward for each episode and the average reward for the previous hundred episodes. The main differences regarding the MountainCar environment are the high instability of the inverted pendulum problem and the time necessary to perform the simulations. Cartpole is an unstable environment, as every action has a great significance to the success or not of the episode. As discussed, along this subchapter, it is essential to have a good hyperparameter configuration. Unlike MountainCar, the parameter configurations that can make successful learning are significantly lower. In order to study the hyperparameters, a thousand episodes are performed, the rewards are compared, and the completion times are defined. A maximum amount of time steps is set at 2000 similar to the Q-Learning implementation. Since the agent after successfully learning the task would have results much higher than the stipulated maximum to avoid wasting unnecessary time.

I Learning Rate

The developed tests were made with the same configurations for all the different hyperparameters, $\gamma = 0.99$ and $n = 10$. The learning rates are the following:

$$\alpha_1 = 0.02, \alpha_2 = 0.01, \alpha_3 = 0.005, \alpha_4 = 0.002 \quad \textbf{(6.5)}$$

In Figure 6.8, the four plots respective to the tested learning rates are presented. α_1 (a) learns the task successfully after 1000 episodes but through its learning process it tries significantly different policies which result in bad rewards, even though this learning rate can recover its learning, it takes a considerable

number of episodes compared to α_2 (b). When testing learning rates lower than α_2 it is possible to determine that these need more episodes to reach the maximum reward, slowing the learning process. α_3 (c) and α_4 (d) solutions also demonstrate a higher instability due to the nature of the control problem, that consequently results in graphs with higher reward variation.

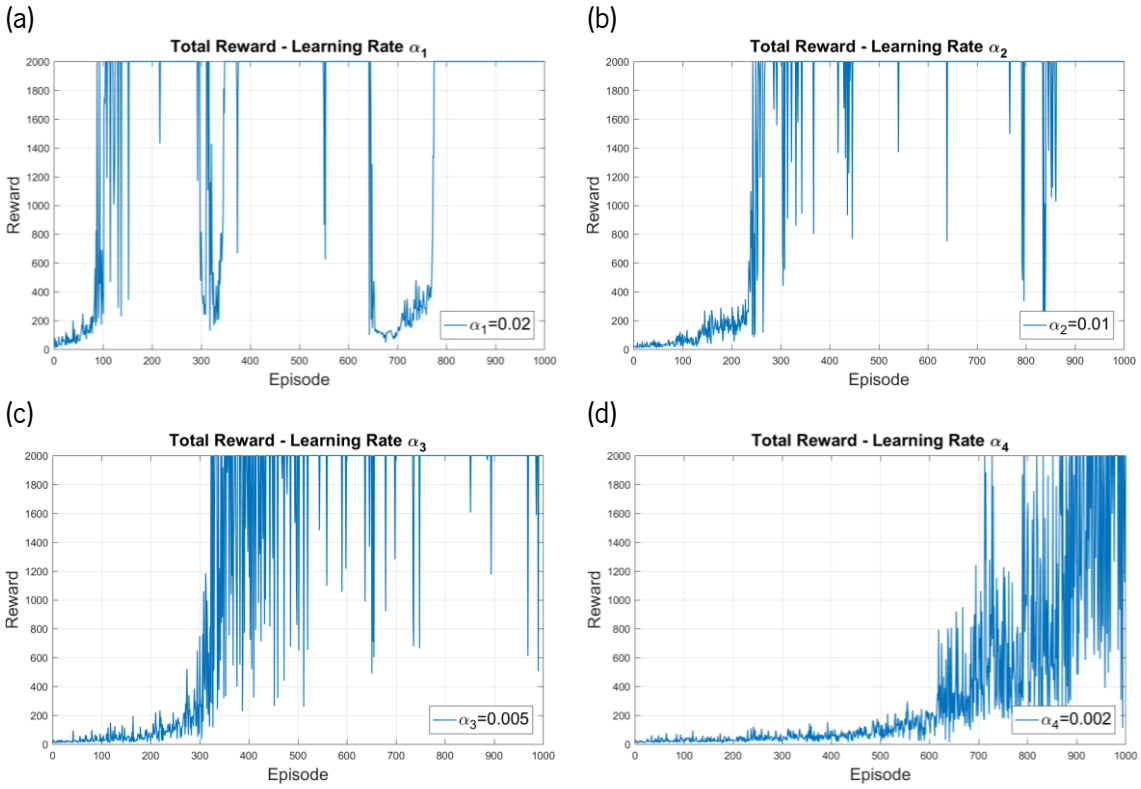


Figure 6.8 - Policy Gradient CartPole total reward comparison between different learning rates.

In Figure 6.9, it is possible to see the learning evolution without the high variation shown in the total reward graphs.

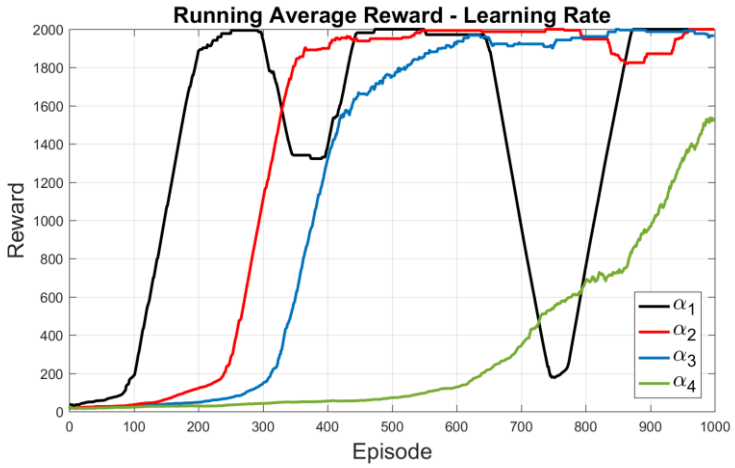


Figure 6.9 - Policy Gradient CartPole running average reward comparison between different learning rates.

II Discount Factors

Four different discount factors are introduced with a constant number of artificial neurons $n = 10$ and a learning rate of $\alpha = 0.01$. The tested discount factors are:

$$\gamma_1 = 0.97, \gamma_2 = 0.99, \gamma_3 = 0.995, \gamma_4 = 0.999 \quad \textbf{(6.6)}$$

Figure 6.10 demonstrates the different rewards in each episode for each discount factor. The three first discount factors γ_1 (a), γ_2 (b) and γ_3 (c) have learned an optimal policy throughout 1000 episodes. Of the three, γ_2 is the discount factor that has the most stable initial learning, from the first episode to the first time it is able to perform a 2000 reward for a couple of straight times.

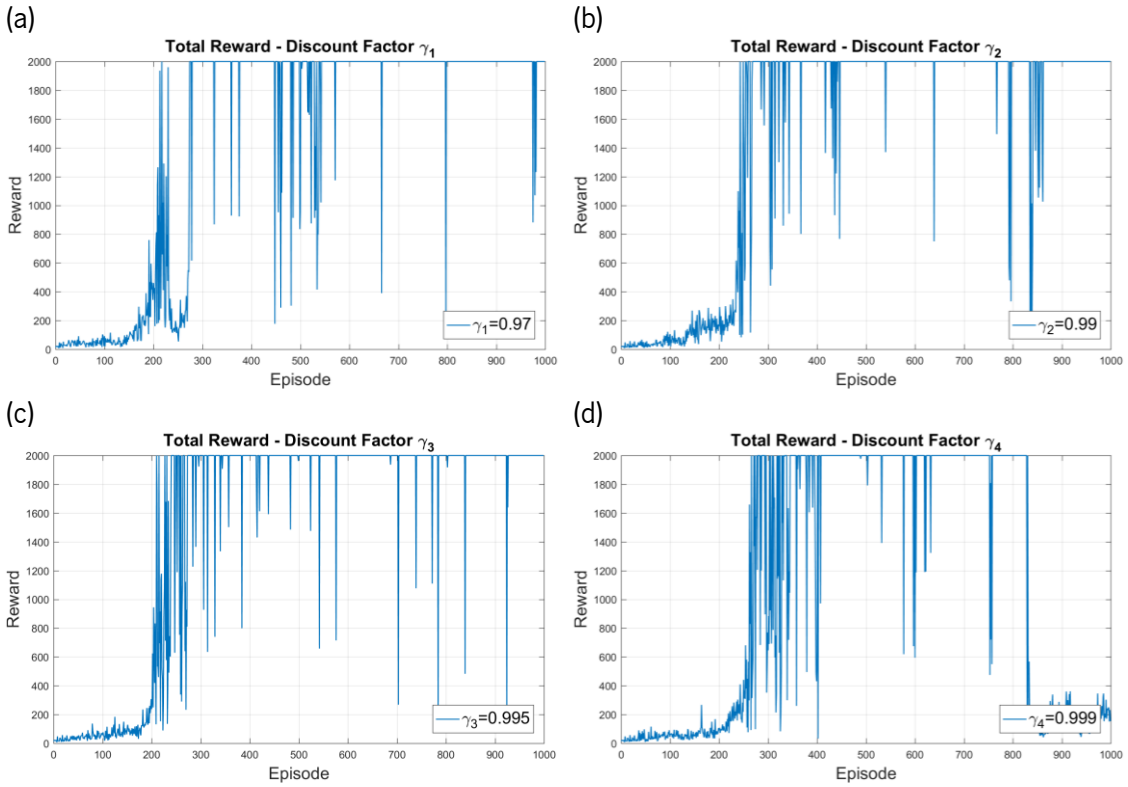


Figure 6.10 - Policy Gradient CartPole total reward comparison between different discount factors.

At approximately episode 300, this can be easier visualised in Figure 6.11. γ_3 introduces a solution with higher reward variation and with the discount factor increase, the learning process tends to be more unstable. γ_4 (d) introduces a highly unstable solution that even loses its optimal policy and is unable to have a successful learning process at the end of the 1000 episodes.

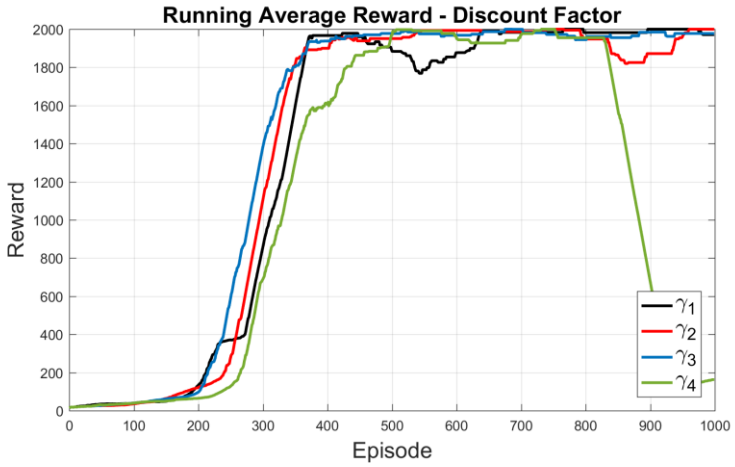


Figure 6.11 - Policy Gradient CartPole running average reward comparison between different discount factors.

III Number of Artificial Neurons

As stated in 6.1.1III, the Monte Carlo strategy used has an artificial neural network. It has a hidden layer and the number of artificial neurons in that layer is a hyperparameter. The compared number of hidden layer neurons are the following:

$$n_1 = 5, n_2 = 10, n_3 = 20, n_4 = 40 \quad \textbf{(6.7)}$$

In Figure 6.12 and Figure 6.13, the respective total episode reward and the running average are plotted. In this implementation, the increase in the number of neurons can be linked to the number of episodes necessary to reach a 2000 policy. From n_1, to n_4 the time necessary to reach the maximum reward for a couple of times decreases. Meaning that n_4 (d) is the solution that takes the smaller number of episodes. However when considering stability, n_4 has a considerable amount of episodes where the reward achieved shows that the experiments made translate into not so successful attempts. n_3 (c) is the most stable solution even though it takes approximately more 150 episodes to achieve its first straight maximum reward possible. n_2 (b) is a slower and more unstable solution than n_3. n_1 (a) is clearly an oversimplified solution, as seen in the number of iterations necessary to reach the 2000 reward consistently. n_1 is outperformed by the solutions with higher number of artificial neurons since it is significantly surpassed, both in stability and time necessary until it reaches an optimal policy.

All the solutions can provide an optimal policy after 1000 episodes. As also previously stated, these results are known to be connected to the selected learning rate and discount factor, different values on these hyperparameters would result in different learning processes.

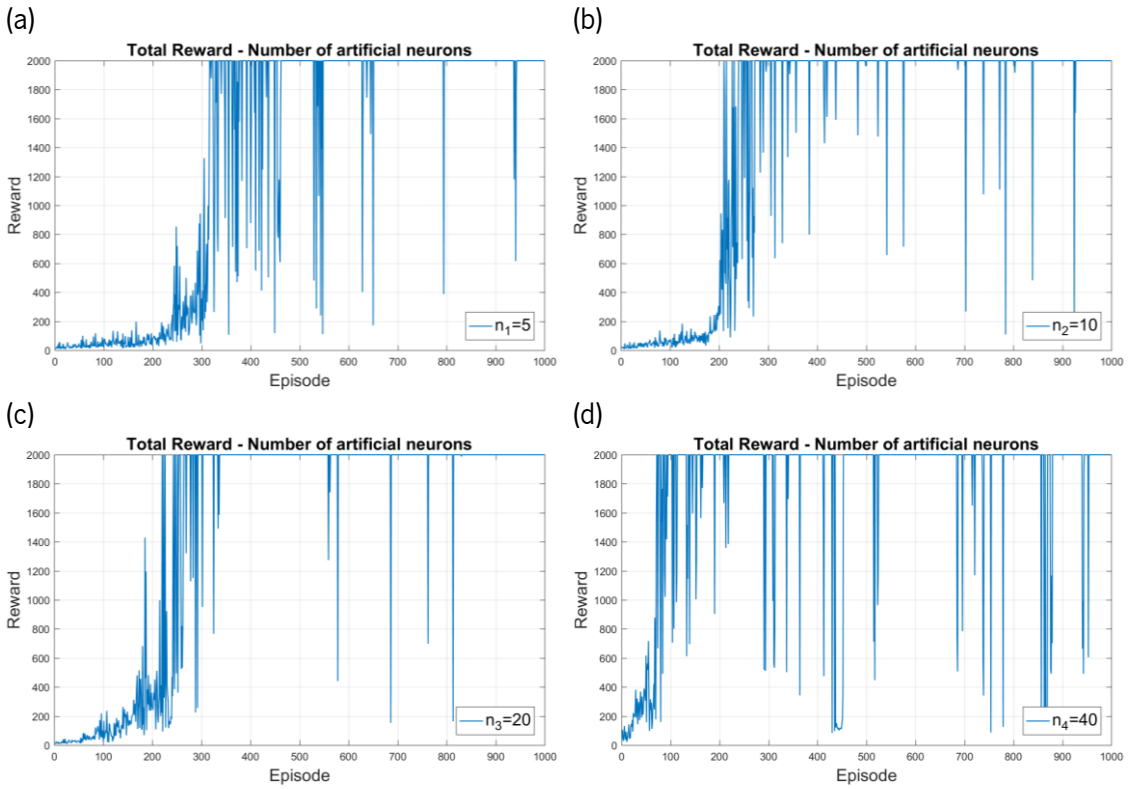


Figure 6.12 - Policy Gradient CartPole total reward comparison between different number of neurons.

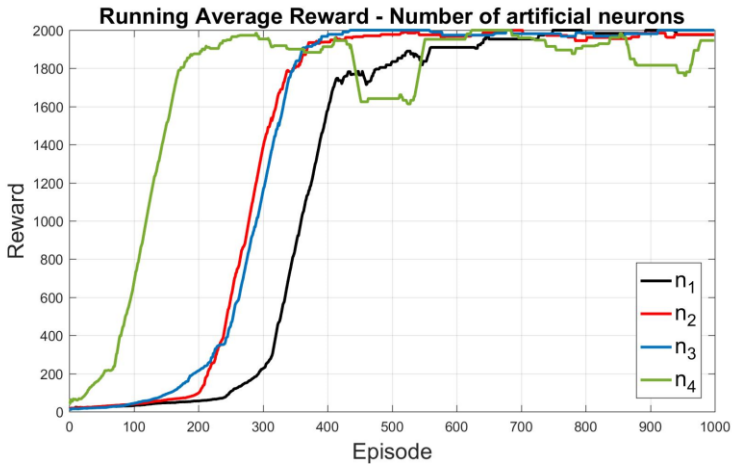


Figure 6.13 - Policy Gradient CartPole running average reward comparison between different number of neurons.

IV Long-Term Simulation

The long-term simulation was performed using the following hyperparameters:

$$\alpha = 0.01, \gamma = 0.99, n = 20 \quad \textbf{(6.8)}$$

The long-term simulation for CartPole using the Monte Carlo method is not as straightforward as the previous long-term simulations. Due to the instability of the inverted pendulum control problem, the long-term simulations are not always able to reach the $10\,000$ episode mark while having learned an optimal policy. In Figure 6.14 is an example of a non-successful long-term simulation. Even though it seems that

it has learned an optimal policy from episode 2800 to episode 8000, it fails the learning process. The non-linear function approximator used, ANNs can sometimes create unstable learning systems. This downfall, linked with the fact that just a minor change on the policy can make the inverted pendulum stop balancing, demonstrates that not always the learning process can be successful even using the same hyperparameters. The time to perform the long-term simulation was approximately 7 hours, whereas the short-term took around 30 minutes.

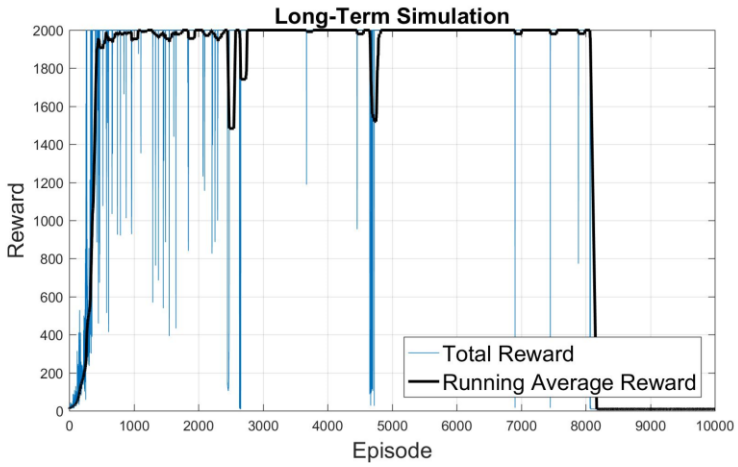


Figure 6.14 - Policy Gradient CartPole total and running average reward long-term simulation.

6.2 Bot'n Roll ONE A

In this chapter, a Policy Gradient Monte Carlo method is applied and tested on a simulated Bot'n Roll ONE A. The goal similar to the previous chapter, is to solve three mazes with increasing complexity. The agent must surpass each level by undergoing a fixed number of episodes on each maze. After completing a maze, the agent will be positioned on the following level, and so on until it solves the third maze.

The implementation on the Bot'n Roll ONE A and the respective environment is equal to the one presented on previous subchapters. The method consists of a neural network as the policy model and uses Monte Carlo methods to update the policy model at the end of each episode. However, the simulated version introduces a change in the obstacle sensors. Instead of a dual discrete solution, in this implementation, a solution with five continuous sensors is presented.

6.2.1 Reinforcement Learning Components

A precise reinforcement learning components definition is an essential introductory task to understand the algorithm's performance. In the following sections, a description of the different RL components regarding the implemented PG method is presented.

I State-Space

The information returned by the agent's sensors defines the state-space. Contrary to the state-space defined in 5.2.1I, in this chapter, the distance sensors physical restrictions are abandoned in favour of a more complex and realistic high-level obstacle avoidance framework. The robot sensors now consist of five Time-of-Flight (ToF) beams pointed to five different directions, such as a) left, b) $45°$ between left and front, c) front, d) $45°$ between the front and right, and e) right. These beams are displayed in Figure 6.15. Therefore, the number of state spaces is five, and each one is a continuous value between 0 and 1, representing no obstacle, and an obstacle right next to the robot, respectively. The detection volume is a laser with a $1\ m$ range.

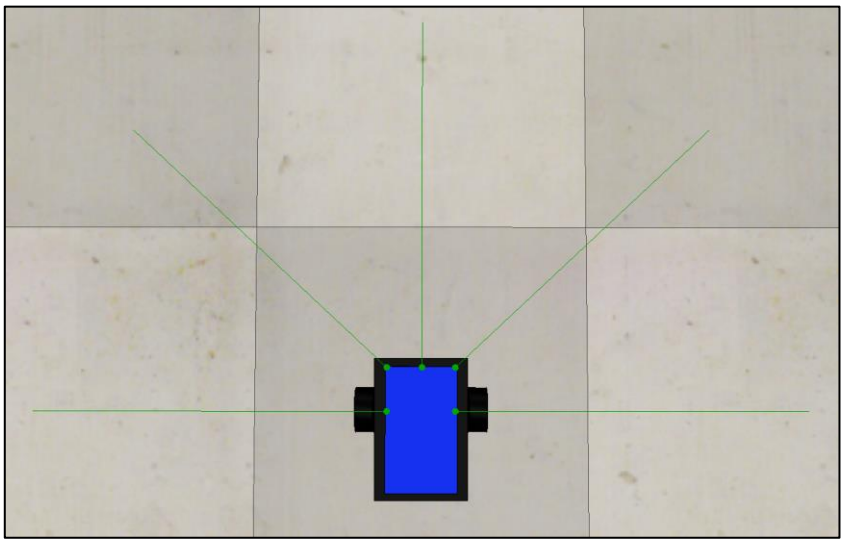


Figure 6.15 - Simulated agent adapted action-space with five simulated ToF distance sensors.

II Action-Space

The action-space is equal to the one presented in the previous chapter. Since this Monte Carlo policy gradient method still cannot handle continuous actions, a predefined set of discrete actions described in Figure 5.18 were created to fit the algorithm restrictions. The algorithm can choose from a range of fourteen possible actions. The action a after being calculated in the control script is sent to the simulator via the ROS node.

III Policy

This algorithm has a stochastic policy $\pi(a|s,\theta)$ that outputs the probability of an action being selected. Its model is an artificial neural network whose outputs are activated via a softmax function that introduces the action's probabilities. The number of output neurons is the number of actions, where the value of

each output neuron is the probability associated with each action. The neural network has one hidden layer, with an adjustable number of neurons.

6.2.2 Reward Functions Comparison

After implementing the policy gradient algorithm and testing it with thousands of different hyperparameters, the robot could only solve the first maze for a couple of different configurations but never managed to surpass the second maze. The main reason for this is that in policy gradient with Monte Carlo methods, the target policy is stochastic. An agent such as Bot'n Roll ONE A in an environment as the RoboParty Maze Challenge needs a deterministic target policy to solve the three levels. However, this implementation allowed a comprehensive analysis of how different reward functions affect the behaviour the agent manifests in a specific environment. Thus, in the following sections, five different reward functions are analysed using the Policy Gradient Monte Carlo method. All the graphs show both the total and running average rewards for the first 200 episodes. This value is considerably low, but it is enough to demonstrate how the reward functions influence the learning process. All the simulations are performed on the second maze.

I Constant Time Step Negative Reward

In this simulation, the policy model receives a constant negative reward. This reward system is similar to the one presented on MountainCar environment. It consists of a negative reward of -1 for every time step except when a collision or a success state occurs. The agent's goal is to exploit the environment reward system and try to follow the steps that achieve the best reward. The constant negative reward is commonly used to teach agents to learn a task in a minimum number of iterations, for every time step the robot takes to reach a terminal state, it will lower its total reward. However, this strategy comes with a drawback. If the robot never reaches the success terminal state to receive a positive reward, it assumes that the best possible reward is to reach a terminal state with the fewer time steps possible, as it does in MountainCar. The difference between MountainCar and this environment is that in MountainCar, there is no terminal state, that needs to be avoided. Thus the robot tries to reach a terminal state as quickly as possible. Since the simulated Bot'n Roll ONE A never manages to reach the success state, it assumes that the best reward possible is to collide with a wall with fewer time steps possible. In Figure 6.16, a display of the total rewards and average reward for this system shows precisely that the robot tries to maximise the solution of colliding as quickly as possible.

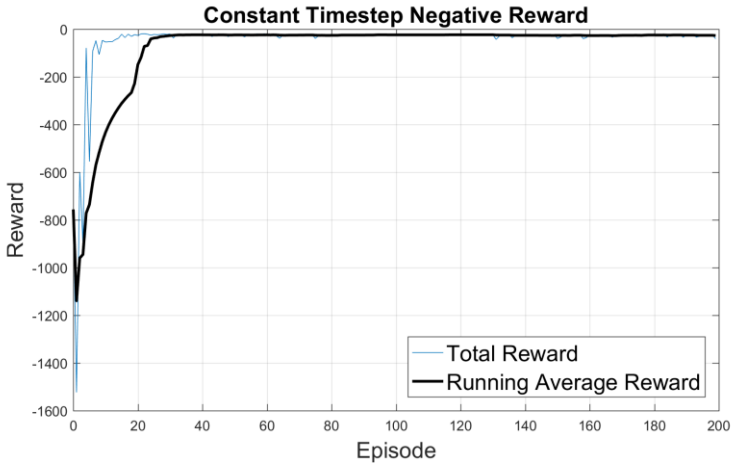


Figure 6.16 - Policy Gradient Bot'n Roll ONE A, total and running average rewards for constant time step negative rewards function.

II Constant Time Step Positive Reward

In this example, it experiments a contrary implementation from the approach to the last section. This simulation experiments a constant positive reward for every time step. In the CartPole environment, a similar solution is also presented, when the goal is for the agent to stay in an episode as long as possible. This simulation consists of a constant $+1$ reward for every time step except terminal states. This type of reward function is usually used for unstable systems, where the main goal is for the agent to perform a set of actions to avoid a single negative terminal episode. The purpose of this implementation was to have an agent whose primary goal is to avoid the maze walls. However, the robot can develop a policy that respects this primary goal but still does not solve the maze. It performs actions that when the movement vectors are summed turns out as null movement. This means that the robot continuously performs sets of actions that never gets out of its place, forward followed by backwards or left turns followed by right turns. Similarly to the last section, the difference between the CartPole and the Bot'n Roll ONE A is that in the created simulated environment two different goals are intended, obstacle avoidance and finishing the maze, whereas the Cartpole only goal is to avoid dropping the pendulum. Moreover, the constant positive solution only solves one of the problems: obstacle avoidance. In Figure 6.17, is displayed how the total episode rewards turned out throughout all the episodes. It is possible to see that the robot tries to increase the reward as high as possible, however at the 1000 time step mark the timeout system terminates the simulation.

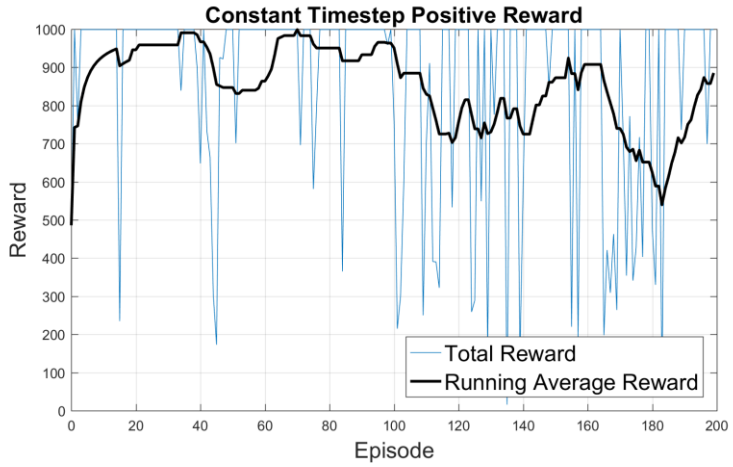


Figure 6.17 - Policy Gradient Bot'n Roll ONE A, total and running average rewards for constant time step positive rewards function.

III Distance to the End of the Maze Negative Reward

The next experiment introduces a new variable to the reward functions. The robot's distance to the endpoint of the maze, Figure 6.18. The issue with the last reward system is that the robot only goal was to avoid the maze walls. The distance to the endpoint of the maze is an attempt to create a reward system that has as its primary purpose to reach the end of the maze. The distance to the end of the maze is calculated and normalised: a) 0, that means the robot has reached the endpoint; b) 1 the robot is in the furthest possible position inside the maze. This distance would be negated to force the robot to reach the endpoint as quickly as possible.

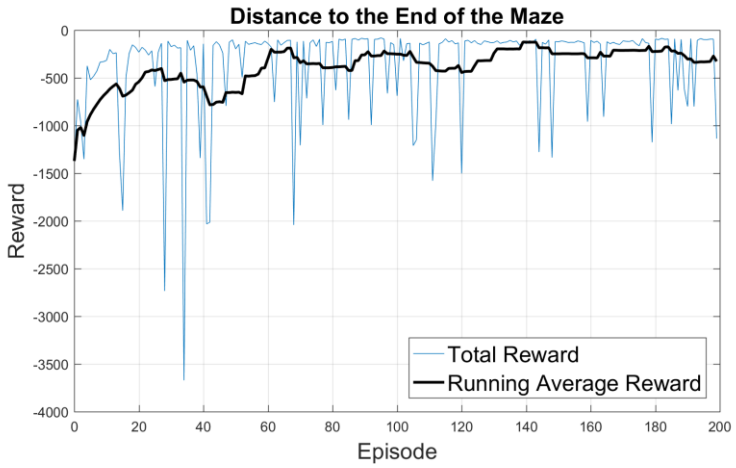


Figure 6.18 - Policy Gradient Bot'n Roll ONE A, total and running average rewards for distance to the end of the maze negative reward function.

However, this solution was still not capable of solving the environment. Mainly because of the maze length, the values from 0 to 1 were not sufficient for the robot to understand that going forward would result in better total rewards. Thus, the robot develops a similar policy to the constant negative reward. After some trials, the robot does not understand that moving forward awards a minor negative reward

and assumes as a primary goal to finish the episode as quickly as possible. In Figure 6.18, it is possible to see that the robot is forced to end the episode as quickly as possible, but in a significant number of episodes, it tries other solutions involving moving forward.

IV Distance to the End of the Maze Compared to the Previous Time Step

This section is a variation from the previous reward system, where the outcome is the distance travelled to the end of the maze since the previous time step, Figure 6.19. In this system, the algorithm measures the distance travelled from the previous iteration regarding the endpoint. If the robot moves in the endpoint's direction, a positive reward proportional to the distance covered is sent to the algorithm. On the other hand, if the robot moves away from the endpoint a negative reward is received. The central point from this reward function is to reinforce the moving in the endpoint direction and have a negative reward for when a collision occurs. For this case, a new policy was created by the agent, which consists of moving in the direction of the endpoint with no regard for collisions. Initially, the robot would successfully move to the endpoint and would avoiding both first and second turn, though as the episodes ran, it stopped avoiding both turns and started to only move forward with no capability of avoiding the maze walls.

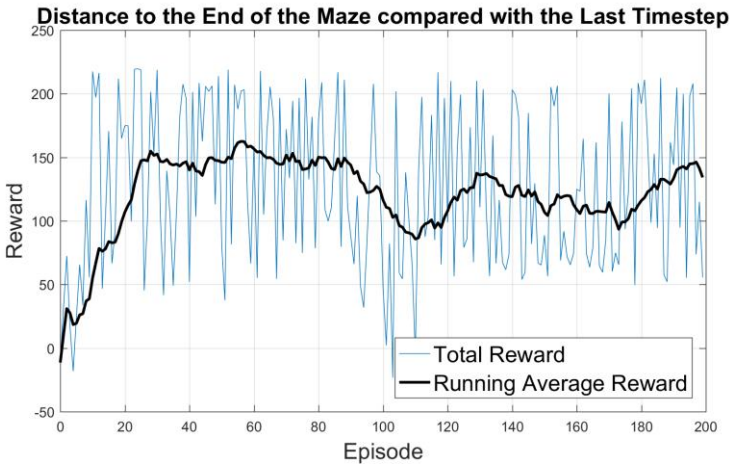


Figure 6.19 - Policy Gradient Bot'n Roll ONE A, total and running average rewards for distance travelled to the end of the maze compared to the last time step positive reward function.

In Figure 6.19, the rewards throughout the learning process show that the agent does not attempt to reduce the reward function and its only concern is on moving forward.

V Lowest Sensor Measurement Compared to the Previous Time Step

The last new variable tested as part of the reward functions regards the values of the robot sensors. The idea behind this is to use the information the robot receives from the observation and try to include as part of a reward. This reward system consists of collecting the sensor data with the lowest value from the

observation. Meaning that the sensor which detects the nearest wall compares the distance to the previous time step. If the distance to the nearest wall decreases, meaning a wall is closer, the reward is negatively proportional to the decreased distance. Instead, if the distance to the nearest wall increase, meaning the robot is moving away from the nearest wall, the reward is positively proportional to the distance moved. In Figure 6.20 the graphs demonstrate the results for the introduced system as a new way to reinforce the robot to avoid the maze walls.

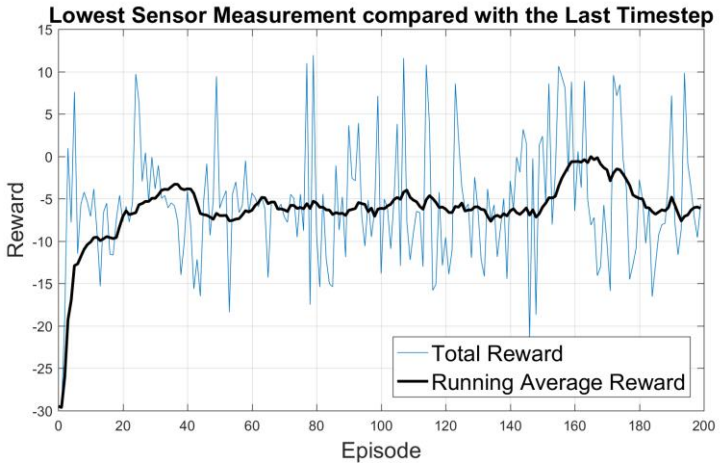


Figure 6.20 - Policy Gradient Bot'n Roll ONE A, total and running average rewards for the lowest sensor measurement compared to the last time step positive reward function

As displayed, the reward system average increases over time, as it tends to 0, which means that the robot maintains the distance on average to both walls. However, this solution created a unique and brilliant solution, even though it is not what was planned. The robot developed a policy of turning on its own to an angle of approximately $90°$, so the only sensor detecting a wall would be the one on the front. Then, the robot would start to move back to increase its reward. Since the only sensor detecting walls was the front sensor, moving backwards would always return a positive reward. The agent would consequently collide with the wall on its back, thus ending the episode. This reward system shows the potential of these types of reward systems to solve problems in unique and complex ways.

VI Integration of Multiple Reward Systems

Lastly, concepts from all the previous reward functions were implemented together with different topologies. The integration of different reward systems consists of assigning different percentages to the selected reward systems. However, this integration did not create any significantly different policy from the ones above presented. Many different topologies were tested, mainly joining functions that would primarily force the agent to avoid the maze walls and to move forward. However, none of the different topologies was able to develop a different policy from the ones selected as part of the integration. The

policies would always be one of the previous five policies presented in the five previous sections, not showing great improvement.

6.2.3 Results Discussion

Five different reward functions were implemented so the learned policies could be compared and analysed. The different policies translate into five different behaviours with different goals performed by the agent. The goals go from wall avoidance to improving the distance from the walls, to moving in the endpoint direction and reaching a terminal state as quickly as possible. Different topologies involving the five reward functions were tested to try to solve the second maze, but independently of the different conjugations no policy different from the above was successfully created. Even though none of the above could successfully solve the second maze, this study revealed to be an essential part of understanding how different reward functions influence the learning process.

6.3 Conclusions

In this chapter, a Policy Gradient algorithm using the Monte Carlo method was tested in three different environments. The first two, MountainCar and CartPole, are successfully solved with different solutions compared to the ones in Q-Learning, presented in chapter 5. A hyperparameter detailed study of how different configurations influence the learning process is also demonstrated.

The third algorithm was implemented on the simulated Bot'n Roll ONE A. The state-space was adapted to five continuous ToF sensors, so it could better resemble higher complexity obstacle avoidance systems. Even though thousands of different hyperparameters configurations were tested, the agent could only solve the first environment. This allowed thorough testing of how the reward functions influences the learning process. Five unique reward functions were designed and analysed. These introduced five distinct policies, each one with different primary goals. Different topologies joining the five different reward functions were also tested, but these could not create different policies than the five previously discovered.

The reason for the lack of success of this Policy Gradient algorithm does not reside on the hyperparameter configuration. After testing all the different reward functions, it is possible to analyse that a very specific reward function could indeed solve the second maze, but the percentage configuration would have to be extremely precise. However, the biggest reason for this algorithm to be unsuccessful is that its target policy is not deterministic. The obstacle avoidance problem is a complicated problem to be solved with a stochastic target policy as provided by Policy Gradient methods. A stochastic policy models a distribution

over actions, and selects an action according to this distribution, whereas a deterministic policy always returns the same action with the highest expected Q-value.

In the next chapter, a policy gradient method with a deterministic target policy named Deep Deterministic Policy Gradient will be analysed and detailed.

7

Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient is a Policy Gradient algorithm that uses a stochastic behaviour policy for an improved exploration but estimates a deterministic target policy, which is considerably easier to learn. Since DDPG is off-policy and makes use of a deterministic target policy, this permits the use of the Deterministic Policy Gradient theorem. DDPG is an actor-critic algorithm that is primarily composed of two artificial neural networks, one for the actor and one for the critic. The networks calculate action predictions for the current state and generate a temporal difference error signal at each time step. The input of the actor-network is the current state, and the output is an array of float values representing an action chosen from a continuous action-space. The critic's output is the estimated Q-value of the current state and the action given by the actor-network. The deterministic policy gradient theorem provides the weight update rules for the actor-network. The critic network is updated regarding the gradients obtained from the TD error.

In this chapter, a Deep Deterministic Policy Gradient implementation on the OpenAI's Gym environment Pendulum is presented. Different hyperparameter configurations and their impact on the learning process were analysed, tested and compared.

7.1 Implementation and Testing

In the implementation made in this chapter, the Deep Deterministic Policy Gradient algorithm is branded by its actor-critic strategy. It uses two distinct ANNs, the actor-network and the critic-network. The actor-network obtains as inputs the environment observation, the values from the three different states which are part of the environment's state-space. It has a hidden layer of 30 artificial neurons and its output is the action-space dimension. For this example, it is just the pendulum angular velocity. The critic-network

inputs are the current state, the action previously made, and the reward returned by the environment. This network also has 30 in its hidden layer with just one output, which is the estimated Q-value of the current state and the action the agent performed. In Figure 7.1, a diagram of the implemented DDPG algorithm is presented. The Pendulum Environment is an excellent choice for implementing Deep Deterministic Policy Gradient as it provides continuous action-space.

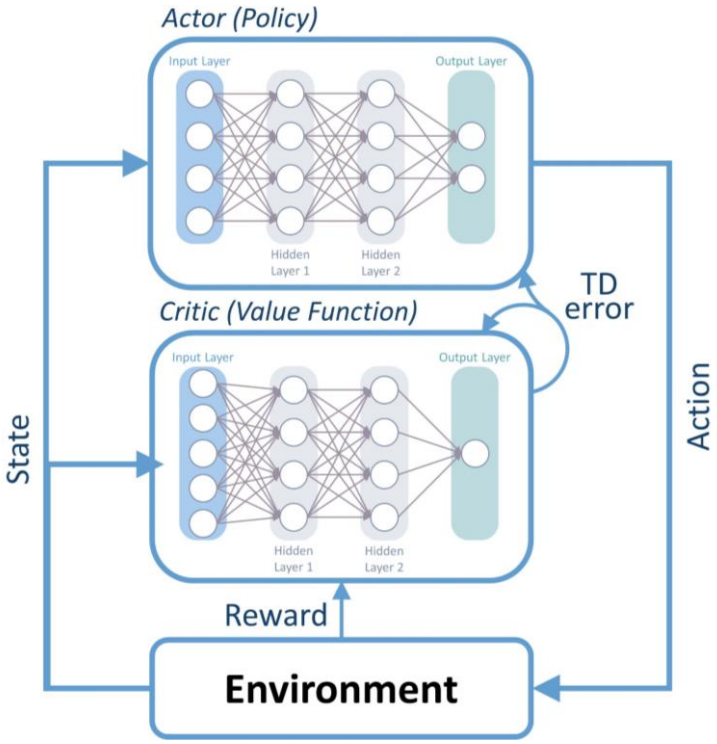


Figure 7.1 - Diagram of the implemented Actor-Critic method with the specific actor-network and critic-network.

However, the application of ANNs on Deterministic Policy Gradient method behaves poorly, not being able to converge. In general, training and evaluating a policy and value function with thousands of temporally-correlated simulated courses leads to a significant amount of variance in the approximation to the Q-function. The TD error signal consolidates the variance received by wrong predictions over time. Thus, to solve the lack of convergence problem, a replay buffer is used to store the agent experiences during training and arbitrarily sample previous experiences. This strategy is called experience replay, and it was used on this implementation with batches of 32 episodes.

7.1.1 Pendulum

The inverted pendulum swing-up problem is a classic literature control problem. In this environment, the algorithm receives from the simulator the pendulum's angle, as the cosine and sine, and its instantaneous angular velocity. In order to fulfil the goal of swinging it to the upright position, a continuous angular speed must be performed by the agent. For every simulation, two different graphs are presented, the total reward

for each episode and the running reward average. The total reward graph indicates how good each episode is. Similarly to MountainCar, the range of possible rewards are always negative, so the goal for the agent is to have rewards throughout the episodes that tend to 0. The running reward average demonstrates how the learning evolution is proceeding over time. A constant number of 1000 episodes, were performed for every hyperparameter configuration. A detailed hyperparameter configuration is presented and analysed throughout this chapter.

I Actor-Network Learning Rate

The Learning Rate for the Actor-Network denoted as α_a, is the hyperparameter responsible for the step size the artificial neural network performs. A comparison of four different α_a is presented in this section. The studied learning rates are the following:

$$\alpha_{a_1} = 0.05, \alpha_{a_2} = 0.01, \alpha_{a_3} = 0.005, \alpha_{a_4} = 0.002 \quad \textbf{(7.1)}$$

The values used for the rest of the hyperparameters are: $\alpha_c = 0.01, \gamma = 0.95,\ \sigma = 3, \omega = 0.9999$. In Figure 7.2, the plots considering the total rewards for each episode for all four actor-network learning rate configurations are presented. The actor learning rate has a significant influence on the learning reward variation, as can be noted by the difference between the four graphs. α_{a_1} (a) shows a learning process with high reward variation since the pendulum has many steps with low rewards.

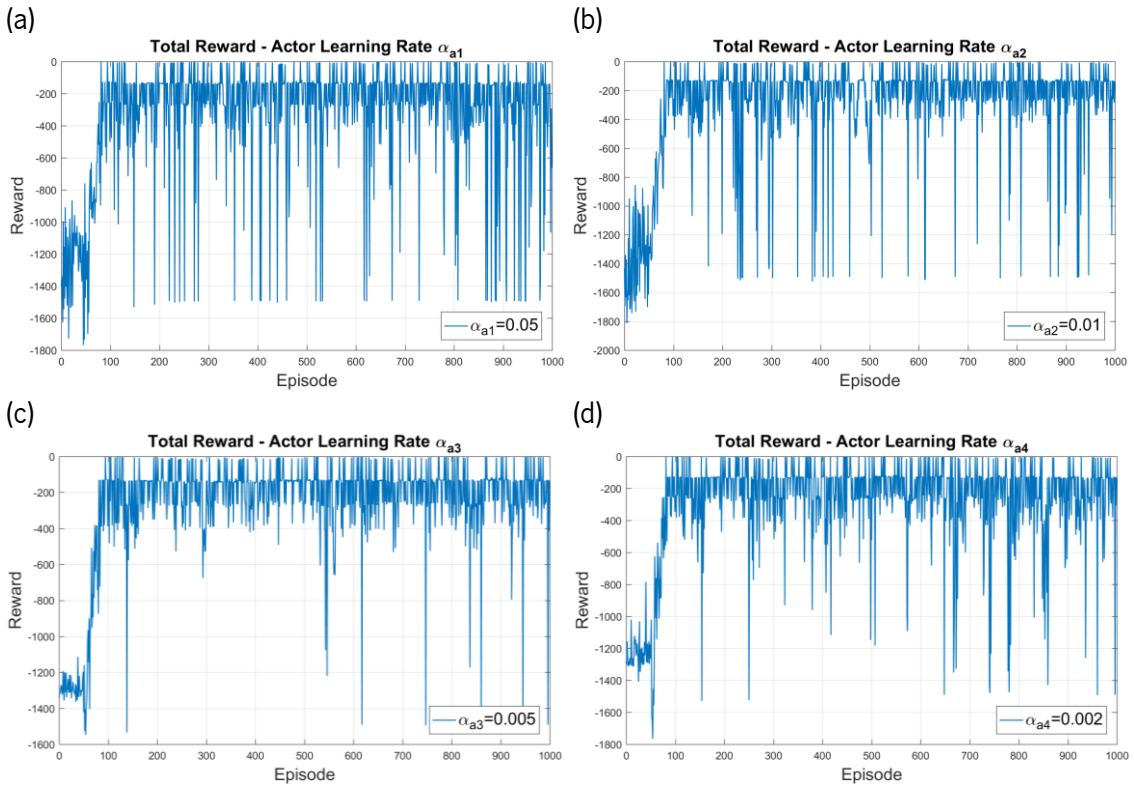


Figure 7.2 - DDPG Pendulum total reward comparison between different actor learning rates.

As the learning rate decreases, α_{a_2} (b) and furthermore α_{a_3} (c) it is noted that the reward variation starts to decrease. α_{a_3} presents a solution with very little reward variation regarding episodes with low rewards. However, with higher decreases such as α_{a_4} (d) the learning process starts to deteriorate in the last hundreds of episodes. Figure 7.2 and Figure 7.3 shows how the reward variation of all learning rate configurations influences medium-term learning.

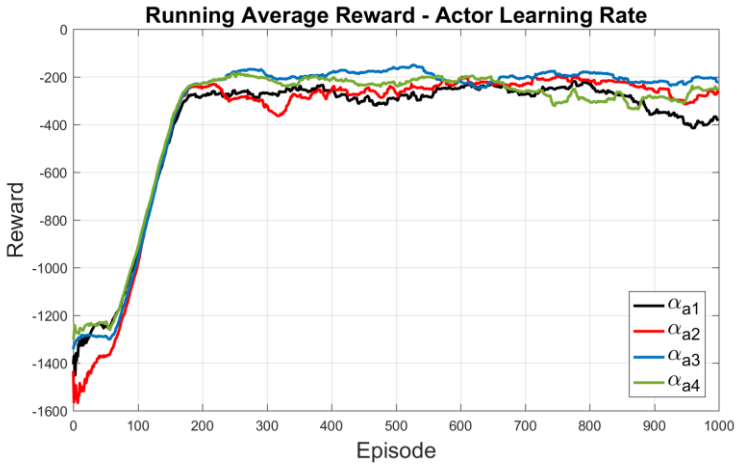


Figure 7.3 - DDPG Pendulum running average reward comparison between different actor learning rates.

II Critic-Network Learning Rate

Similarly to the previous section, the learning rate for the critic-network hyperparameter denoted as α_c, controls the ANNs step size. Four different critic-network learning rate configurations are presented as follows:

$$\alpha_{c_1} = 0.02, \alpha_{c_2} = 0.01, \alpha_{c_3} = 0.005, \alpha_{c_4} = 0.002 \quad \textbf{(7.2)}$$

The values used for the rest of the hyperparameters are: $\alpha_a = 0.005, \gamma = 0.95,\ \sigma = 3, \omega = 0.9999$. The four graphs shown in Figure 7.4 demonstrate the four learning rates for critic-network adjustments. Regarding this hyperparameter, there are differences in the initial learning and the reward variation registered throughout 1000 episodes. α_{c_1} (a) and α_{c_4} (d) show an initially slow exponential learning, whereas α_{c_2} (b) and α_{c_3} (c) have a high reward on the first couple of episodes and maintain that reward over 100 episodes.

Independently of this initial learning, the four learning rates have similar learning behaviours from episode 100 to 200. Regarding the learning reward variation, α_{c_2} (b) shows a more natural learning process with a higher number of solved episodes. Both the initial learning and the reward variation can be easily understood in Figure 7.5 as the reward running average.

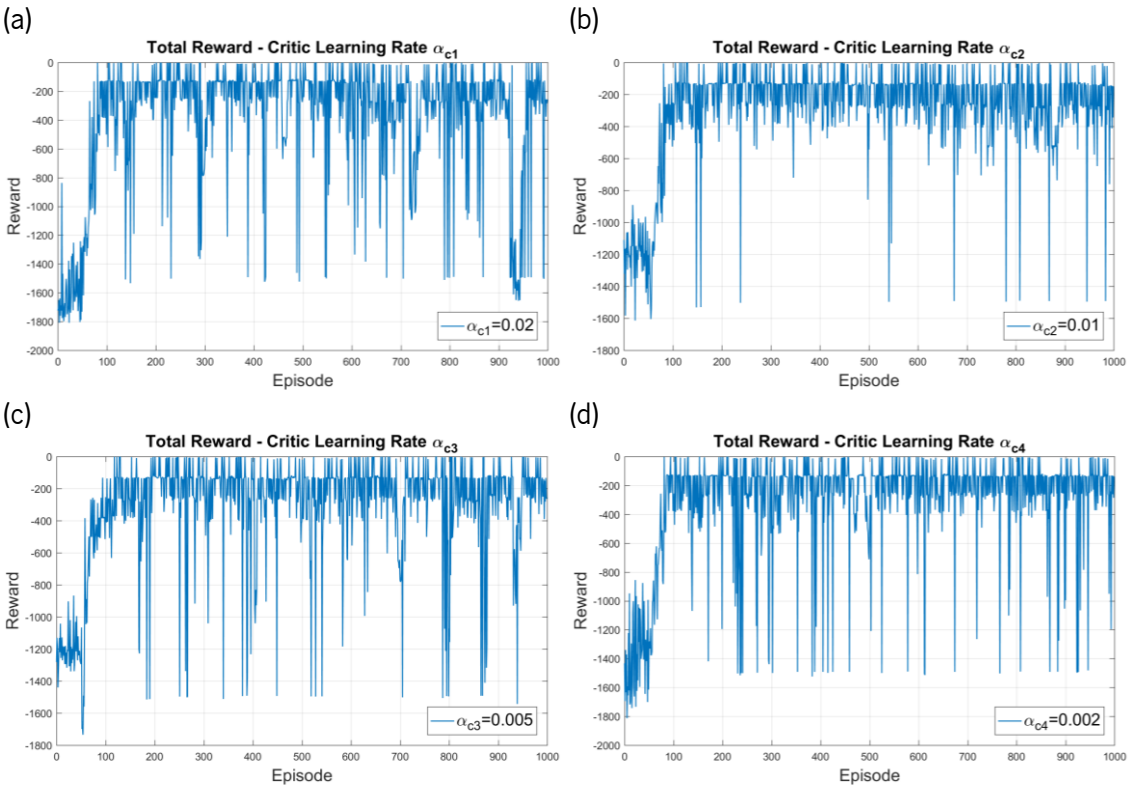


Figure 7.4 - DDPG Pendulum total reward comparison between different critic learning rates.

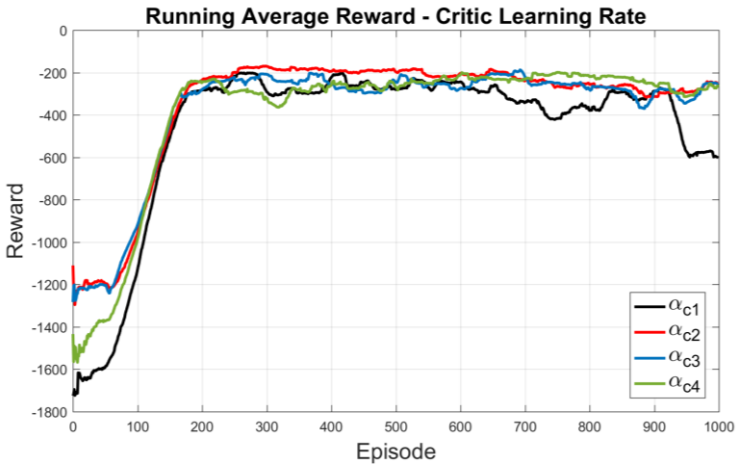


Figure 7.5 - DDPG Pendulum running average reward comparison between different critic learning rates.

III Discount Factor

The Discount Factor γ controls the importance of future rewards versus the present ones. In this section a comparison and analyses of four different γ values are presented:

$$\gamma_1 = 0.9, \gamma_2 = 0.95, \gamma_3 = 0.99, \gamma_4 = 0.999 \quad \textbf{(7.3)}$$

The values used for the other hyperparameters are: $\alpha_a = 0.005, \alpha_c = 0.01, \sigma = 3,\ \omega = 0.9999$. Figure 7.6 shows an interesting different learning configuration. γ_4 (d) is the discount factor that demonstrates the highest reward variation. In contrast from γ_1 (a) to γ_3 (c) it is noticeable that the reward variation on the learning process is transferred from the second half of the learning to the first half of the

learning. In γ_1 the biggest part of the reward variation only starts appearing on episode 600. In γ_2 (b) the reward variation starts appearing a bit sooner, approximately on episode 400. For γ_3 the learning process changes and only has a high reward variation on the first 500 episodes. This demonstrates that many times, the hyperparameters introduce certain details on the learning process essential for the success of the agent. In Figure 7.7 the differences on the reward variations appearances can be observed, where for γ_1 the average drops on the last half of the episodes run, γ_2 drops approximately at episode 400 and γ_3 average rises on the second part of the thousand episodes.

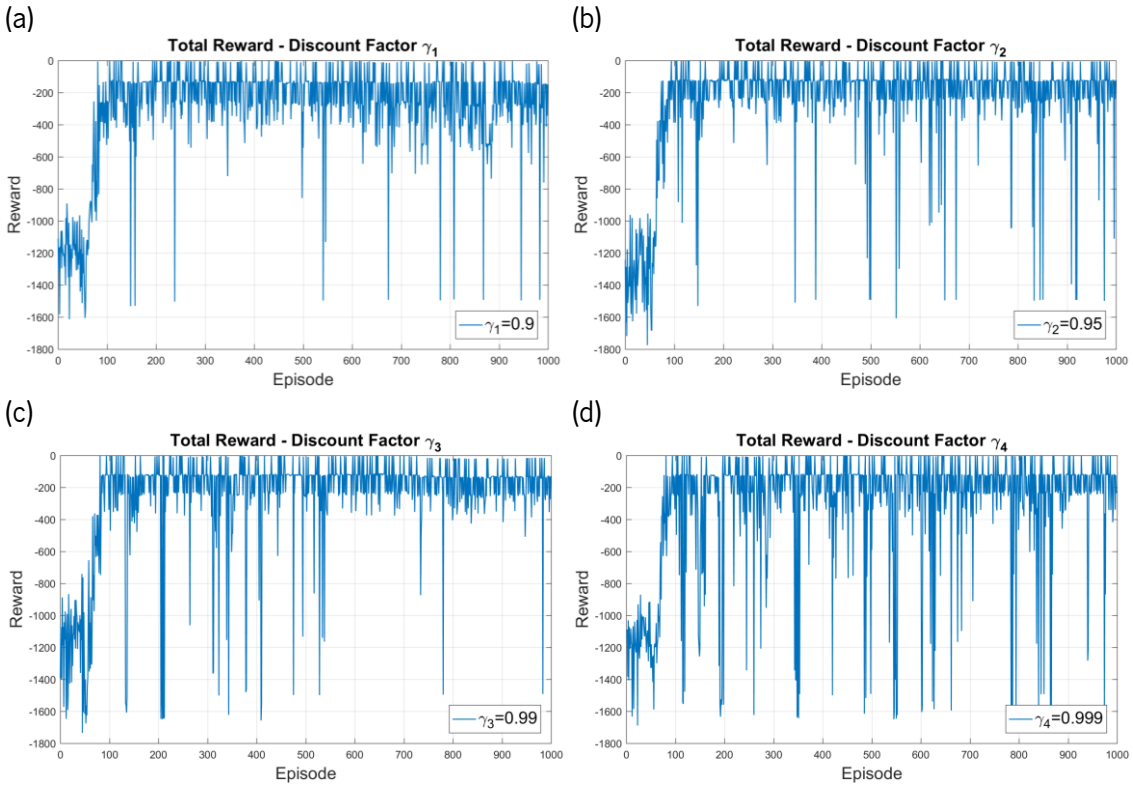


Figure 7.6 - DDPG Pendulum total reward comparison between different discount factors.

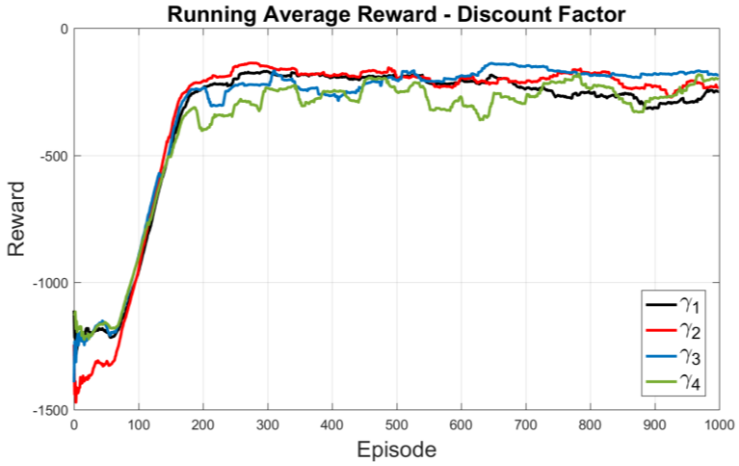


Figure 7.7 - DDPG Pendulum running average reward comparison between different discount factors.

IV Exploration Noise

The exploration method for continuous action-spaces is different from the discrete action-spaces. Instead of performing a different action for exploration, the algorithm introduces a noise function to the selected continuous action. In this implementation, a Gaussian distribution is implemented to provide the exploration as noise for the action. Similarly to the exploration methods introduced on discrete action-spaces, the exploration methods decrease over the number of episodes, using decay rates. A Gaussian distribution is defined by its mean, that is the algorithm's selected action, and the deviation. Four different simulations were performed using two different deviations σ, and two different decay rates ω:

$$\sigma_1 = 3,\ \sigma_2 = 3,\ \sigma_3 = 5,\ \sigma_4 = 5 \quad \textbf{(7.4)}$$

$$\omega_1 = 0.9995,\ \omega_2 = 0.9999,\ \omega_3 = 0.9995,\ \omega_4 = 0.9999 \quad \textbf{(7.5)}$$

The values used for the other hyperparameters are: $\alpha_a = 0.005, \alpha_c = 0.01, \gamma = 0.99$. By analysing both Figure 7.8 and Figure 7.9 with the two different deviations and decay rates, it is possible to stand out the differences between the two hyperparameters. For the two different deviations with the same decay rates, it is possible to point out that σ_1 (a) and (b) allows a learning with less reward variation since the exploration is also minor, while σ_3 (c) and (d) has higher reward variation.

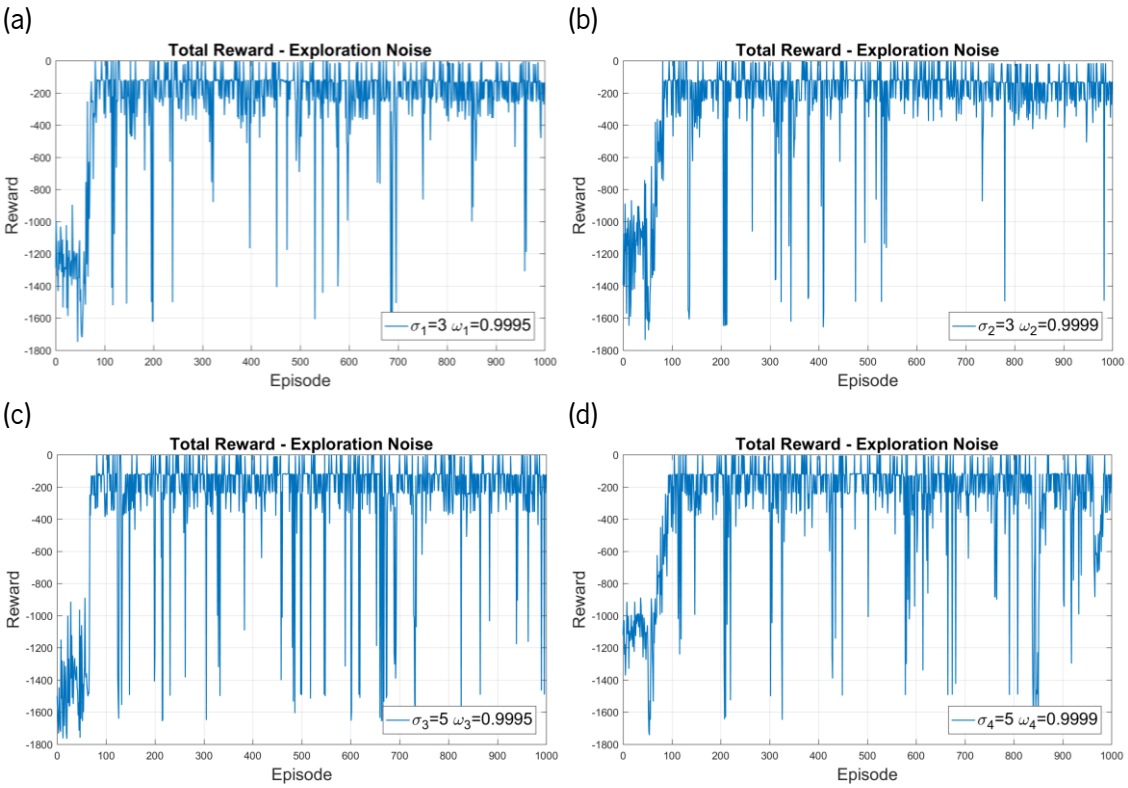


Figure 7.8 - DDPG Pendulum total reward comparison between different exploration noises.

This was expected to occur and it is due to the higher exploration that ultimately results in a longer exploration. The higher decay rate ω_2 (b) and (d) shows that the end of both simulations have less variance on the rewards than the lower ω_1 (a) and (c). Even though the exploration takes more episodes, it is made in an improved way that outperforms lower deviations.

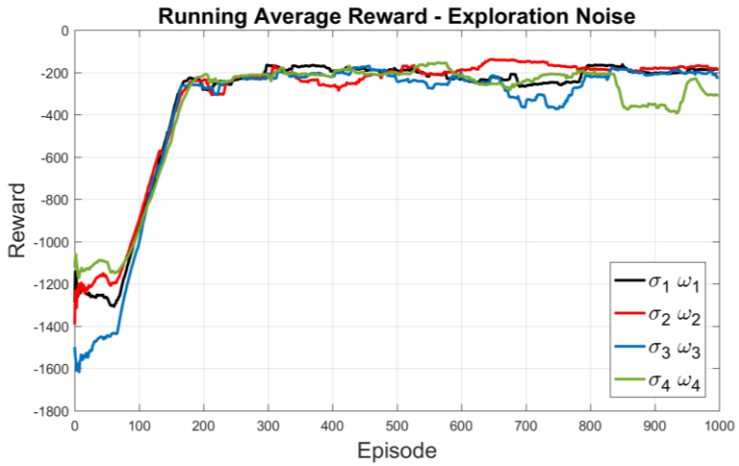


Figure 7.9 - DDPG Pendulum running average reward comparison between different exploration noises.

V Long-Term Simulation

The long-term simulation demonstrates how an optimal set of hyperparameters for short term simulations performs throughout $10\ 000$ episodes. The long-term simulation serves to prove the long-term efficiency of the selected algorithm as well as the configured hyperparameters. The hyperparameters were selected considering the study previously demonstrated and are as follows:

$$\alpha_a = 0.01, \alpha_c = 0.002, \gamma = 0.005, \sigma = 3, \omega = 0.9999 \quad \textbf{(7.6)}$$

Figure 7.10 shows the reward of each episode and the average running reward. This specific simulation demonstrates that the algorithm learned the task at hand and after the last episodes still successfully solves the environment with an average reward higher than -400.

However, in short-term simulations, the final average value rounds the -200. The average running rewards show that the algorithm maintains a high average until approximately episode 3500. After the first 3500 episodes, the rewards drop significantly until episode 7500, during a period where the developed policy does not perform as good as it formerly did. From episode 7500 onwards the agent solves the environment in approximately 300 iterations. For an improved result, the hyperparameters reconfiguration could create a different policy that better meets the long-term learning process.

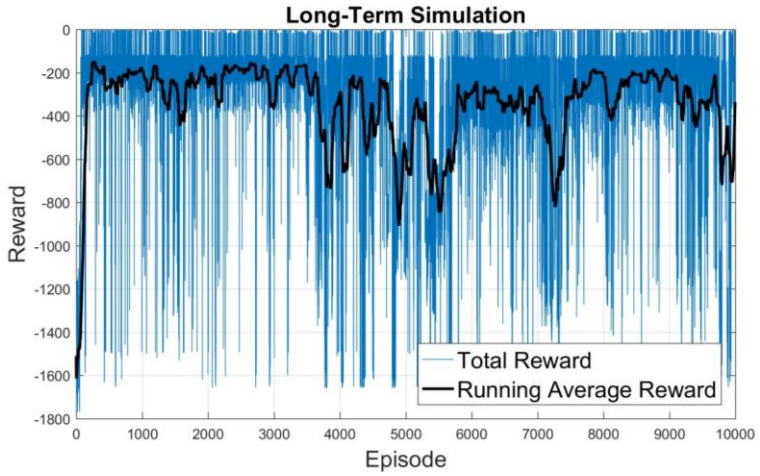


Figure 7.10 - DDPG Pendulum total and running average reward long-term simulation.

7.2 Conclusions

In this chapter, a Deep Deterministic Policy Gradient implementation in OpenAI's Gym environment Pendulum was accomplished. This algorithm made possible the conclusion of some essential topics regarding the advantages of using DDPG algorithms:

1. A stochastic behaviour policy associated with a deterministic target policy provides a virtuous exploration while contributing for a considerably smoother learning process.
2. The Actor-Critic algorithm aims to take advantage of both value-based and policy-based methods. The agent controls the agent's behaviour by learning the optimal policy, while the critic evaluates the action by calculating the value function. These two models collaborate so both can get better in their role. The overall architecture can provide efficient learning.
3. The artificial neural networks presented in both actor and critic methods allow a broad approximation to large state-spaces, thus working as a robust approximation method for complex reinforcement learning environments. The non-linear approximation methods are a big step to implementing RL in robotics.
4. The continuous action-space shows better resemblances to real-world problems. In non-simulation environments, such as the physical world, the majority of actions is not discrete, so continuous action-spaces algorithms are essential for the future of reinforcement learning applied to real-world problems.

The connection between all these points is the direction to implement reinforcement learning to real-world robotics. The DDPG studies show enormous potential to be one of the top leading algorithms of this scientific subject.

8

Conclusions

The work presented in this dissertation concerns a description of the theoretical foundations supporting reinforcement learning and the implementation of different reinforcement learning algorithms in OpenAI Gym and created environments. A detailed explanation of the mathematics behind RL algorithms is defined as one of the primary purposes of this dissertation to allow further projects of LAR, such as CHARMIE, to use this work as a foundation.

Initially, some introductory concepts were presented, such as basic terminology and exploration methods. Two different reinforcement learning methods, Dynamic Programming and Monte Carlo were described and analysed to introduce some of the fundamental problems faced. Temporal Difference emerges as a model-free method using advantages from both DP and MC. Temporal Difference Learning is used nowadays as the foundation method to a significant part of the state of the art algorithms, introducing Q-Learning and SARSA. Policy Gradient methods are described as the next step to solve problems with higher complexity, using policy parameterisation and introduce algorithms such as Actor-Critic and Deterministic Policy Gradient. The two combined create one of the newest reinforcement learning breakthroughs, Deep Deterministic Policy Gradient. DDPG is an off-policy algorithm that uses an Actor-Critic method, a stochastic behaviour policy and an estimation based on a deterministic target policy. It can also work with continuous action spaces, emerging as an algorithm with a significant potential regarding RL in robotics.

The second part of this dissertation focus on practical implementations of three different algorithms, Q-Learning, Monte Carlo Policy Gradient and Deep Deterministic Policy Gradient, on different environments. One of the goals was to create a specific simulation environment using a Bot'n Roll ONE A as an agent and the RoboParty Maze Challenge as the problem to solve. A detailed comparison between Gazebo and

V-REP simulators was carried out and concluded that V-REP was the most advantageous choice, mainly due to the possibility to perform world interaction during the simulation and, mesh manipulation and optimisation. The process of designing and script configuration was very straightforward and intuitive. The ROS framework was selected to work as the communication system between V-REP and the control script.

Q-Learning was the first RL algorithm implemented. It was initially tested on OpenAI Gym environments, specifically MountainCAR and CartPole. The fact that it is an off-policy algorithm and has a deterministic policy presented a successful implementation on both environments, being able to solve them on a low number of episodes and low hyperparameter search. The results of different hyperparameter configurations are also presented. The Q-Learning implementation on Bot'n Roll ONE A used discrete action-space and discrete state-space. It introduced two different methods regarding different action-choosing situations, method A only able to solve the first maze, whereas method B solved the three mazes. Method B revealed to be an efficient solution to the maze challenge and to solve never seen situations. The work regarding the Q-Learning implementation of both methods allowed the publication of a scientific paper [59] on an international conference.

The Policy Gradient algorithm implemented used a Monte Carlo method rather than the Temporal Difference. The noted difference regards the lack of estimations during the episode that instead happened at the end of each episode. The tests were performed in the same Gym environments. In MountainCar, the efficiency results were similar to the Q-Learning implementation, but a bit slower to converge. In CartPole the policy gradient method demonstrated the need for fewer episodes to converge to an optimal policy but revealed to be an unstable learning method, as sometimes the policy would unlearn how to solve the control problem. The Bot'n Roll ONE A implementation introduced a continuous state-space, with a change on the obstacle sensors, from infra-red to ToF.

The Policy Gradient method provides the tools for a detailed study and analyses how the reward functions alter the agent behaviour. Five different individual reward functions were developed regarding different parameters, such as the distance to the endpoint and sensor readings. All five solutions presented produced five different policies, and in some cases, even produced surprising policies that did not solve the environment but produced compelling information on how reward functions affect the learning evolution. After all the hyperparameter and reward configurations search, no possible combination was able to solve the second environment. The most significant motive for the unsuccess is the target policy not being deterministic. The navigation problem consists of two different tasks, obstacle avoidance and

target following, and both are very complicated problems to solve with a stochastic policy, as in Policy Gradient methods.

The last implemented algorithm is Deep Deterministic Policy Gradient. DDPG is an off-policy method that uses a deterministic target policy, with an actor-critic composed of two artificial neural networks. This algorithm was implemented in the Pendulum environment to explore its functionality to work with continuous action-space and state-space. The implementation as displayed by all the graphs, was very successful, since the algorithm solved the problem with a wide range of hyperparameter configurations. The continuous action-space, as well as the deterministic target policy, are essential characteristics for algorithms to use in robotics. Thus, DDPG is an excellent algorithm for reinforcement learning problems in robotics.

The code developed throughout this dissertation is publicly available on GitHub.

8.1 Further Work

The work made throughout this dissertation is expected to establish a starting point for future developments in reinforcement learning algorithms for robotics, in particular for navigation systems. By using the introduced theoretical foundations and the designed implementations, both theoretical and practical components may serve as a foundation for future work.

Regarding reinforcement learning navigation systems for robotics, the introduction of some novel developments and features are necessary to improve both the overall efficiency as well as the capability to transfer the learning into other navigation platforms. The upgrades that have been naturally selected and are planned to be studied in the future are as follows:

- The implementation of DDPG on the simulated Bot'n Roll ONE A
- Transfer the learning system to a physical Bot'n Roll ONE A
- The adaptation of DDPG and standardisation to different omnidirectional platforms, i.e. CHARMIE's navigation platform
- The design and implementation of other robotics functionalities that may upgrade CHARMIE's functionalities, i.e. manipulators and body movement.
- The study of new algorithms that are associated with recent breakthroughs, i.e. Advantage Actor-Critic (A2C), Asynchronous Advantage Actor-Critic (A3C) and Proximal Policy Optimization (PPO

)

References

[1] D. Silver *et al.*, "Mastering the game of Go without human knowledge.," *Nature*, 2017.

[2] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics : A Survey," *Int. J. Rob. Res.*, 2013.

[3] D. Hassabis, M. Suleyman, and S. Legg, "DeepMind," 2010. [Online]. Available: https://deepmind.com/. [Accessed: 20-May-2019].

[4] N. Heess *et al.*, "Emergence of Locomotion Behaviours in Rich Environments," *arXiv [cs.AI]*, p. arXiv:1707.02286v2, 2017.

[5] I. Garcia *et al.*, "Autonomous 4DOF Robotic Manipulator Prototype for Industrial Environment and Human Cooperation," *2019 IEEE Int. Conf. Auton. Robot Syst. Compet.*, 2019.

[6] T. Ribeiro, I. Garcia, D. Pereira, J. Ribeiro, G. Lopes, and A. F. Ribeiro, "Development of a prototype robot for transportation within industrial environments," *2017 IEEE Int. Conf. Auton. Robot Syst. Compet. ICARSC 2017*, pp. 192–197, 2017.

[7] F. Ribeiro *et al.*, "MinhoTeam 2016: Team Description Paper," vol. 2011, 2016.

[8] A. G. T. Lopes *et al.*, "MinhoTeam@Home," 2017. [Online]. Available: http://home.dei.uminho.pt/. [Accessed: 20-May-2019].

[9] F. RoboCup, "RoboCup@Home," 2007. [Online]. Available: http://www.robocupathome.org/. [Accessed: 20-May-2019].

[10] L. Iocchi, D. Holz, J. Ruiz-Del-Solar, K. Sugiura, and T. Van Der Zant, "RoboCup@Home: Analysis and results of evolving competitions for domestic and service robots," *Artif. Intell.*, vol. 229, pp. 258–281, 2015.

[11] D. Holz and L. Iocchi, "Benchmarking Intelligent Service Robots through Scientific Competitions : The RoboCup @ Home Approach," in *AAAI Spring Symposium - Designing Intelligent Robots: Reintegrating AI II*, 2013, pp. 27–32.

[12] J. Savage, E. Vazquez, E. Silva, J. Hernandez, and A. Nakayama, "DSPL Pumas @ Home 2019 Team Description Paper," 2019.

[13] Y. Tanaka *et al.*, "Hibikino-Musashi @ Home 2019 Team Description Paper," 2019.

[14] R. Memmesheimer *et al.*, "RoboCup 2019 - homer@UniKoblenz (Germany)," vol. 2014, no. 2013, pp. 1–9, 2019.

[15] J. G. M.F.B. van der Burgh, J.J.M. Lunenburg, L.L.A.M. van Beek, A. T. H. L.G.L. Janssen, S. Aleksandrov, K. Dang, H.W.A.M. van Rooy, and A. A. and M. J. G. van de M. D. van Dinther, "Tech United Eindhoven @Home 2019 Team Description Paper," pp. 1–10, 2019.

[16] A. G. T. Lopes *et al.*, "CHARMIE, Minho Team @ Home 2017 Team Description Paper," 2017.

[17] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach 3rd*, 3rd ed. Prentice Hall Press Upper Saddle River, NJ, USA ©2009.

[18] Y. LeCun, C. Cortes, and C. J. C. Burges, "THE MNIST DATABASE of handwritten digits," 1998. [Online]. Available: http://yann.lecun.com/exdb/mnist/. [Accessed: 20-May-2019].

[19] D. Cireşan, U. Meier, and J. Schmidhuber, "Multi-column Deep Neural Networks for Image Classification," *Int. Conf. Pattern Recognit.*, no. February, pp. 3642–3649, 2012.

[20] G. Hinton and T. J. Sejnowski, *Unsupervised Learning: Foundations of Neural Computation*. A Bradford Book, The MIT Press, 1999.

[21] S. Sutton, Richard and G. Barto, Andrew, *Reinforcement Learning, Second Edition An Introduction*. 2018.

[22] E. Thorndike, "Some experiments on animal intelligence," *Science*. 1898.

[23] E. L. Thorndike and Jelliffe, "Animal Intelligence. Experimental Studies," *J. Nerv. Ment. Dis.*, 1912.

[24] A. M. Turing, "Intelligent machinery: A report," *Key Pap.*, 1948.

[25] A. G. Barto, R. S. Sutton, and P. S. Brouwer, "Associative search network: A reinforcement learning associative memory," *Biol. Cybern.*, 1981.

[26] A. G. Barto, R. S. Sutton, and C. W. Anderson, "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems," *IEEE Trans. Syst. Man Cybern.*, 1983.

[27] R. S. Sutton, "Temporal Credit Assignment in Reinforcement Learning," 1984.

[28] A. L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers," *IBM J. Res. Dev.*, 1959.

[29] J. L. Patrick, "Dopaminergic pathways," 2015. [Online]. Available: https://pt.wikipedia.org/wiki/Ficheiro:Dopaminergic_pathways.svg.

[30] W. Schultz, "Predictive reward signal of dopamine neurons.," *J. Neurophysiol.*, 1998.

[31] V. Mnih *et al.*, "Playing Atari with Deep Reinforcement Learning," pp. 1–9, 2013.

[32] T. OpenAI, "OpenAI Five," 2015. [Online]. Available: https://openai.com/five/. [Accessed: 20-May-2019].

[33] T. A. B. Pinto, "Object detection with artificial vision and neural networks for service robots," Master's Thesis, Minho University.

[34] T. Kurutach, I. Clavera, Y. Duan, A. Tamar, and P. Abbeel, "Model-Ensemble Trust-Region Policy Optimization," *Int. Conf. Learn. Represent.*, pp. 1–15, 2018.

[35] T. Haarnoja *et al.*, "Soft Actor-Critic Algorithms and Applications," 2018.

[36] J. Lee, J. Hwangbo, and M. Hutter, "Robust Recovery Controller for a Quadrupedal Robot using Deep Reinforcement Learning," 2019.

[37] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," in *IEEE International Conference on Intelligent Robots and Systems*, 2017.

[38] A. Singh, L. Yang, K. Hartikainen, C. Finn, and S. Levine, "End-to-End Robotic Reinforcement Learning without Reward Engineering," 2019.

[39] Bing-Qiang Huang, Guang-Yi Cao, and Min Guo, "Reinforcement Learning Neural Network to the Problem of Autonomous Mobile Robot Obstacle Avoidance," *2005 Int. Conf. Mach. Learn. Cybern.*, no. August, pp. 85–89, 2005.

[40] Y. F. Chen, M. Liu, M. Everett, and J. P. How, "Decentralized non-communicating multiagent collision avoidance with deep reinforcement learning," in *Proceedings - IEEE International Conference on Robotics and Automation*, 2017.

[41] R. Hafner and M. Riedmiller, "Reinforcement Learning on a Omnidirectional Mobile Robot," *Proc. 2003 IEEE/RSJ Int. Conf. Intell. Robot. Syst. (IROS 2003)*, no. October, 2003.

[42] M. Riedmiller, T. Gabel, R. Hafner, and S. Lange, "Reinforcement learning for robot soccer," *Auton. Robots*, 2009.

[43] P. H. M. O. G. Silva, "Simulador 3D Multi-Plataforma para Competições Robóticas," Master's Thesis, Minho University, 2018.

[44] G. Brockman *et al.*, "OpenAI Gym," 2016.

[45] C. J. C. H. Watkins and P. Dayan, "Technical Note: Q-Learning," *Mach. Learn.*, 1992.

[46] G. Cybenko, "Approximations by superpositions of sigmoidal functions," *Approx. Theory its Appl.*, 1989.

[47] R. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy Gradient Methods for Reinforcement Learning with Function Approximation," *Adv. Neural Inf. Process. Syst. 12*, 1999.

[48] T. P. Lillicrap *et al.*, "CONTINUOUS CONTROL WITH DEEP REINFORCEMENT LEARNING." 2016.

[49] D. Silver, G. Lever, D. Technologies, G. U. Y. Lever, and U. C. L. Ac, "Deterministic Policy Gradient (DPG)," *Proc. 31st Int. Conf. Mach. Learn.*, vol. 32, no. 1, pp. 387–395, 2014.

[50] E. Rohmer, S. P. N. Singh, and M. Freese, "v-replros2013," pp. 0–5.

[51] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," 2005.

[52] L. Pitonakova, M. Giuliani, A. Pipe, and A. Winfield, "Feature and performance comparison of the V-REP, Gazebo and ARGoS robot simulators," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 10965 LNAI, pp. 357–368, 2018.

[53] A. F. Ribeiro, G. Lopes, N. Pereira, J. Cruz, and M. F. M. Costa, "Bot'n roll robotic kit as a learning tool for youngsters," *9th Int. Conf. Hands Sci.*, no. 266647, pp. 192–192, 2012.

[54] SAR, "botnroll.com," 2007. [Online]. Available: www.botnroll.com. [Accessed: 15-Jun-2019].

[55] R. Coppelia, "V-REP (Virtual Robot Experimentation Platform)," 2013. [Online]. Available: http://www.coppeliarobotics.com/. [Accessed: 15-Jun-2019].

[56] R. Coppelia, "BubbleRob Tutorial." [Online]. Available: http://www.coppeliarobotics.com/helpFiles/en/bubbleRobTutorial.htm.

[57] A. F. Ribeiro, G. Lopes, N. Pereira, and J. Cruz, "Learning robotics for youngsters-the roboparty experience," *Adv. Intell. Syst. Comput.*, vol. 417, pp. 729–741, 2016.

[58] R. S. Sutton and S. Singh, "Summary for Policymakers," *Clim. Chang. 2013 - Phys. Sci. Basis*, vol. 1, no. 1999, pp. 1–30, 2015.

[59] T. Ribeiro, F. Gonçalves, I. Garcia, G. Lopes, and A. F. Ribeiro, "Q-Learning for Autonomous Mobile Robot Obstacle Avoidance," *2019 IEEE Int. Conf. Auton. Robot Syst. Compet.*, 2019.