Politecnico di Torino

Master of Science Degree
in MECHATRONIC ENGINEERING

Master Thesis

Deep Reinforcement Learning for Dynamical Systems

CANDIDATE
George Claudiu Andrei

SUPERVISORS
Prof. Didier Theillol
Prof. Alessandro Rizzo

AA. 2022/2023

UNIVERSITE' DE LORRAINE
POLITECNICO DI TORINO
ENSIL-ENSCI

Deep Reinforcement Learning for Dynamical Systems

Master Degree in Robotics and Automatic Control

George Claudiu ANDREI

Supervisors
Dr. Mayank Shekhar JHA (CRAN - CNRS UMR 7039)
Prof. Jean Christophe PONSART (CRAN - CNRS UMR 7039)
Prof. Didier THEILLIOL (CRAN - CNRS UMR 7039)

University Tutor
Prof. Alessandro RIZZO (Politecnico di Torino)

April 6, 2023

A Mamma e Papà.

Acknowledgements

What began as an assignment of a deceptively simple concept, compressing information to be used on Reinforcement Learning, sooner became an exploration quest on search for Deep Learning methods applied for dynamical systems that could enhance representation and approximation of states, opening doors to a vast scientific horizon of possibilities.

Firstly, I would thank to my external thesis supervisors Dr. Mayank Shekhar JHA, Prof. Jean Christophe PONSART and Prof. Didier THEILLIOL for give me the possibility of approaching this powerful and very interesting world in a motivated environment where the words 'passion' and 'team work' make our spirit, but also for them guidance and support: they allowed me to work independently on an inspiring and engaging project in an innovative field of research, steering me in the right direction whenever they thought I needed it identifying my limits and helping me to extend them. It was an honour working with all of them. Although not explicit on this thesis, there was a tremendous amount of hours, I spent learning about Statistics, Information theory, Machine Learning, Reinforcement Learning and finally Deep Reinforcement Learning which greatly improved my curiosity and motivation.

Secondly, a profound thank goes also to my father Florin, my mother Luminita and my brother Paul that are my strength: they have always supported me in every choice and interest by teaching me the meaning of sacrifice, commitment, constancy despite the distance. I thank from the bottom of my heart my colleagues from the Politecnico di Torino and ENSIL-ENSCI, and all my friends who believed in my during this path: Valentina, Rocco Jack, Ester, Lorenzo, Luca, Alex, Tommaso, Francesca, Eleonora, Margari.

A special thanks goes also to my colleagues Soha, Julien and Juan for their steady presence and support, but also our mutual inspiration and constructive confrontation but especially becoming as family in Polytech Nancy.

A special thanks goes also to every people I met during my stay in France and in Torino. Every person I met in my study path is essential and allowed me to have this fantastic experience including the semester in Limoges that changed my life.

A special thanks goes also to my internal supervisors Prof. Juan ESCARENO from ENSIL-ENSCI and Prof. Alessandro RIZZO from Politecnico di Torino for giving me motivation and right spirit for approaching research environment.

I would never have been able to meet many of these people and opportunities without the financial support I received from the Politecnico di Torino regarding my international mobility achievement and the one from CRAN for my work as an internship.

Abbreviations

ANN Artificial Neural Network
DDPG Deep Deterministic Policy Gradient
DNN Deep Neural Network
DRL Deep Reinforcement Learning
MDP Markov Decision Process
ML Machine Learning
RL Reinforcement Learning
MSE Mean-Squared Error
HJB Hamilton-Jacobi-Bellman

Abstract

Control systems influence our society enormously. Although they are invisible to most users, they are essential to the functioning of almost all devices, be they basic household appliances, aircraft or nuclear power plants. A common denominator among those different applications of control is the need to influence or modify the behavior of dynamical systems to achieve specified goals. In this context, one of the main objective of Artificial Intelligence is to solve complex control problems with high-dimensionality of observation spaces or system models which are unavailable. Recent research has shown that Deep Learning techniques can be combined with Reinforcement Learning methods to learn useful representations allowing to solve the problems mentioned above. In particular, due to the recent progress in Deep Neural Networks, Reinforcement Learning (RL) has become one of the most important and useful new technology in Control Engineering. It is a type of Machine Learning technique in which a computer agent learns to perform a specific task by replacing the classic controller of the traditional control, through these repeated trials and error interactions with a dynamic environment by selecting control inputs sequence based on measured outputs. This thesis intends to provide an in-depth introduction of Deep Learning using Neural Networks combined with Reinforcement Learning methods and then apply them to learn an intelligent controller able to meet specific system requirements without any kind of human supervision. In particular, the application of Deep Deterministic Policy Gradient (DDPG) model-free method to autonomous control such as DC motor speed control as well as inverted pendulum stabilization will be presented based on Reinforcement Learning MATLAB Toolbox. Moreover, a contribution in form of conference communication has been submitted to the upcoming 16th European Workshop on Advanced Control and Diagnosis (ACD 2022) as contribution in improving and speed-up the learning phase of the controller will be presented.

Keywords: Reinforcement Learning, Dynamical Systems, Deep Deterministic Policy Gradient, Deep Learning.

Contents

List of Figures

Chapter 1

Introduction

Optimal control theory is a mature mathematical discipline that finds optimal control policies for dynamical system by optimizing used-defined cost function which capture desired design objectives. One of the main principle for solving such problems is Dynamic Programming (DP). DP provides a sufficient condition for optimality by solving a partial differential equation, known as the Hamilton-Jacobi-Bellman equation. This method is used in traditional optimal control methods which are offline and require complete knowledge of the system dynamics. Artificial Intelligence (AI) has been used for enabling adaptive autonomy and it is a branch of computer science involved in the creation of computer programs capable of demonstrating intelligence. Traditionally, any piece of software which displays cognitive abilities such as perception, planning and learning is considered part of AI. The sub-field of AI which is specifically focused with developing computer algorithms that can solve problems by intelligently extracting knowledge from data is called Machine Learning (ML)[19]. In accordance with the amount, quality and the way of collecting system feedback data, ML is divided into three major categories:

1. Supervised learning;
2. Unsupervised learning;
3. Reinforcement learning;

In *Supervised learning*, labeled training data set is used as the feedback data for the learning algorithms. The goal is to build a system model that represents the learned relationship between the input, output, and system parameters so that the learning agent can generalize its responses to cases that are not in the training set.

Contrarily, in *Unsupervised learning*, the algorithm receives no feedback information and uses unlabeled data as input to find hidden patterns such as clusters or anomalies. It receives no feedback from the supervisor [19].

Finally, *Reinforcement learning (RL)* is the type of learning in which an agent uses actions, also called decisions, on a system in order to achieve the desired system performance over a long period of time. The time variable can be discrete or continue and actions are taken at every time step leading to a sequential decision-making problem. The actions are taken in a closed loop, which means that the outcome of earlier actions is monitored and taken into account when choosing new actions. The goal is to maximise long-term performance, which is determined by the total reward accumulated over the course of interaction between the agent and the system. Rewards are given in order to evaluate the one or more step decision-making performance. Sequential decision-making problems appear in a wide variety of fields, including automatic control, operations research, economics, and medicine. In particular, unlike traditional optimal control, RL finds the solution to the HJB equation online in real time. This has motivated control system researchers to enable adaptive autonomy in an optimal manner by developing RL-based controllers. [3][16]

How is RL different from Supervised (or Unsupervised) Learning?

- A sizable (labeled or unlabeled) static data set that was previously collected is not used for the training phase. Instead, the data that the agent needs is dynamically provided through feedback from the system with which the agent is interacting.
- Over a series of time steps, decisions are made interactively. For the purpose of producing an output prediction in a classification problem, the agent performs inference once on the input data. In RL, the controller constantly runs inference while navigating the real-world environment.

What problems are solved using RL?
Rather than the typical ML problems such as Classification, Regression, Clustering and so on, RL is most commonly used to solve a different class of real-world problems, such as control systems by finding an optimal way to achieve a given and specific task:

- Eg. a robot or a drone that has to learn the task of picking a divide from one box and putting it in a container.
- Manipulating a robot to navigate the environment and perform various tasks.
- Fault Tolerant Control.

1.1 Goals

The underlying motivation for this master's thesis was to explore *Deep Reinforcement Learning based on the control design for dynamical systems* using *Reinforcement Learning Matlab Toolbox*. In order to investigate such goal, some subtasks were given:

1. Conduct a literature study into the realm of Deep Reinforcement Learning and identify a suitable algorithm for control;
2. Learning the use of Reinforcement Learning Matlab Toolbox;
3. For reference, implement a controller with reasonable tuning for both linear and non linear systems;

1.2 Thesis Outline

Following this introductory chapter, the thesis branches between two main concepts, *Reinforcement Learning* and *Deep Learning* and finally their implementation on dynamical systems using Matlab. In particular, the thesis is structured as follows:

1. **Introduction**: explains the thesis's background and objectives.
2. **Reinforcement Learning**: presents a thorough description of the state-of-the-art in Reinforcement Learning in order to give the reader useful tools to enter this research field and a nomenclature comparison with traditional control.
3. **Deep Reinforcement Learning**: presents the use of *Neural Networks* to Reinforcement Learning as well as an outline of *Deep Learning* with a focus on*Deep Deterministic Policy Gradient (DDPG) algorithm* for continuous in time systems.
4. **DDPG Implementation using RL MATLAB Toolbox**: presents the design of a linear and then a non linear control system to perform reinforcement learning experiments in real applications described by simulated environment using Reinforcement Learning Matlab Toolbox
5. **Complementary reward function based learning enhancement for DLR**: proposed method to generate efficient sampled data during the learning phase speeding it up and favouring optimal solution convergence.
6. **Conclusions**: presents a summary of the results obtained from experiments with some considerations.
7. **Perspectives**: presents a conclusions from a possible future work point of view.

Chapter 2

Reinforcement Learning

The powerful algorithms provided by Reinforcement learning (RL) can be used to find the optimal controllers for both linear and non linear systems with deterministic or even stochastic dynamics that are unknown or highly uncertain. RL is a model-free framework for solving optimal control problems stated as Markov decision processes (MDPs). This chapter mainly covers AI approaches to RL from the control engineer point of view. Firstly, some fundamental Control theory concepts as well as RL ones related to traditional control will be discussed in order to focus more on RL techniques.

2.1 Control theory fundamentals

The area of mathematics and engineering known as automatic control investigates how to manipulate input variables to change the behaviour of the system through a command input. In particular, a control system is a system of devices that manages, commands, directs or regulates the behavior of other devices to achieve a desired performance. To put it another way, a control system may be defined as a system that directs other systems to reach a specific state. Many control system types exist, and they may be roughly divided into *linear control systems* and *non-linear control systems.*

In this thesis *feedback control system* only are considered. A feedback control system is a system whose output is controlled using its measurement as a feedback signal. This feedback signal is compared with a *reference signal* to generate an *control error signal* which is filtered then by a *controller* to produce the system's control input. The controlled system is called the *plant* and its output y is measured using sensors. The block diagram in Figure 2.1 illustrates a general feedback signal.

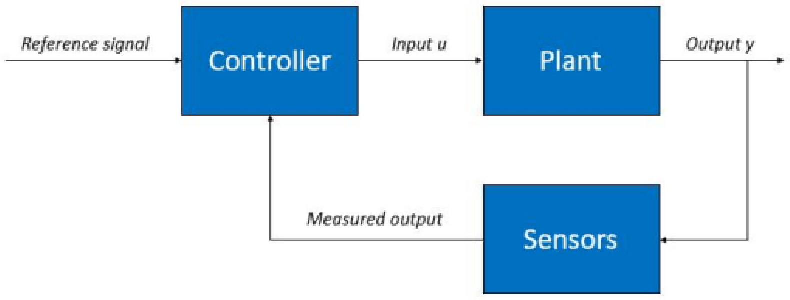

Figure 2.1: *Block diagram representation of a closed loop control system*

The main objectives of feedback control is to ensure that variables of interest in a process or a system, thought of as the *output signals, either* are able to track reference trajectories or maintained close to their set points.

The use of feedback is needed for three reasons mainly:

1. **To counteract disturbance signals affecting the output.**
2. **To improve system performance in the presence of model uncertainty.**
3. **To stabilize an unstable plant.**

2.1.1 Linear Control Systems

Linear systems are characterized by linear differential equations, that is, ordinary differential equations that linear in the dependent variables, linear in their derivatives with respect to the independent variable (time), and linear in the input function or control. Linear systems obey different fundamental principles:

- Additivity: if input x_1 leads to output y_1 and x_2 leads to y_2, then the sum of each $x_1 + x_2$ leads to the sum of each output $y_1 + y_2$.

- Homogeneity: if input x leads to output y then kx leads to ky where k is a constant.
- Superposition: if input x_1 leads to output y_1 and x_2 leads to y_2, then $k_1x_1 + k_2x_2$ leads to $k_1y_1 + k_2y_2$.

As an example of linear system, let's analyze the 1D mass-spring-damper linear system shown in Figure 2.2 to highlight several key principles in control theory.

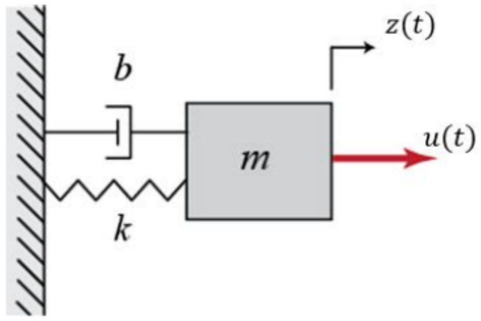

Figure 2.2: *Mass spring damper 1D model*

The mass-spring-damper system can be written as first differential equation of the form:

$$m\ddot{z} + b\dot{z} + k = u(t) \tag{2.1}$$

The first differential equation is obtained by introducing the variable $x_1 = z$ and consequently $x_2 = \dot{z}$. A standard dynamic system matrix representation is known as State Space Representation:

$$\dot{x} = \begin{bmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \end{bmatrix} = \begin{bmatrix} x_t(t) \\ -\frac{k}{m} - \frac{b}{m}x_t(t) + \frac{1}{m}u(t) \end{bmatrix}$$

A compact form of the system 2.1.1 is:

$$\dot{x} = Ax + Bu \tag{2.2}$$

where x is an n-dimensional state vector containing state variables that describe the system with n representing the system order. They are represented in this instance by the mass displacement x_1 and its velocity x_2. The input control action u is an m-dimensional control vector. A is the state matrix with appropriate dimensions representing the system dynamics by taking into account the evolution of the state while B represents the contribution of the input u in vector form with appropriate dimension. The system output y shown in eq. 2.3 is in general a linear combination of the state variables and in case of *closed loop systems* they will be measured to get compared with a reference signal.

$$y = Cx + Du \tag{2.3}$$

where C is the output matrix with appropriate dimension representing which state variables can be measured based on sensor availability while D is a matrix concerning the possibility of the input to affect the output. If the terms A,B,C and D remain constant over time, the system is called Linear time-invariant (LTI).

2.1.2 Non-linear Control Systems

A non-linear system in mathematics does not adhere to the superposition principle or have an output that is directly proportionate to its input. From an engineering perspective, nonlinear systems are essential to control systems. This is because all plants actually have nonlinear natures in practise. The saturation condition, which arises because no system in the real world can deliver constrained energy, serves as the finest illustration of nonlinearity.

In general, the state and output equations for nonlinear systems may be written as follows:

$$x(t) = f[x(t), u(t)]$$

$$y(t) = g[x(t), u(t)]$$

2.2 Reinforcement learning for optimal control applications

A control system's main objective is to determine the optimal control inputs in order to generate the desired system behavior. The controller in feedback control systems uses measured outputs as feedback to enhance performance and correct random disturbances and errors. Engineers design the controller in such a way to meet the system requirements using the feedback of the control system along with a model of the plant.

The goal of reinforcement learning is similar to the control system one but with a different approach and it can be summarized as follows: an agent attempts to learn a policy represented by the sequence of actions or control inputs which will allow to control a dynamic environment to maximize some objective function.
As an example, consider the task of parking a vehicle using an automated driving system. This task requires the vehicle computer (agent) to park the car in the proper orientation and location. The controller generates steering, braking, and acceleration control inputs (actions) by using data coming from cameras, accelerometers, gyroscopes, a GPS receiver, and lidar (observations). The control inputs are sent to the actuators that control the vehicle. The resulting observations depend on the actuators, sensors, vehicle dynamics, road surface, wind, and many other less-important factors.

The main purpose of this formulation is to draw the interest of control theory experts because it should seem very similar to what their own field of study seeks to achieve. The problem of controlling dynamic systems is approached differently by the two fields, with machine learning being primarily data-driven and control theory being predominantly model-driven.

The schema block shown in Figure 2.3 represents a general control system block representation which can be translated to a Reinforcement Learning representation:

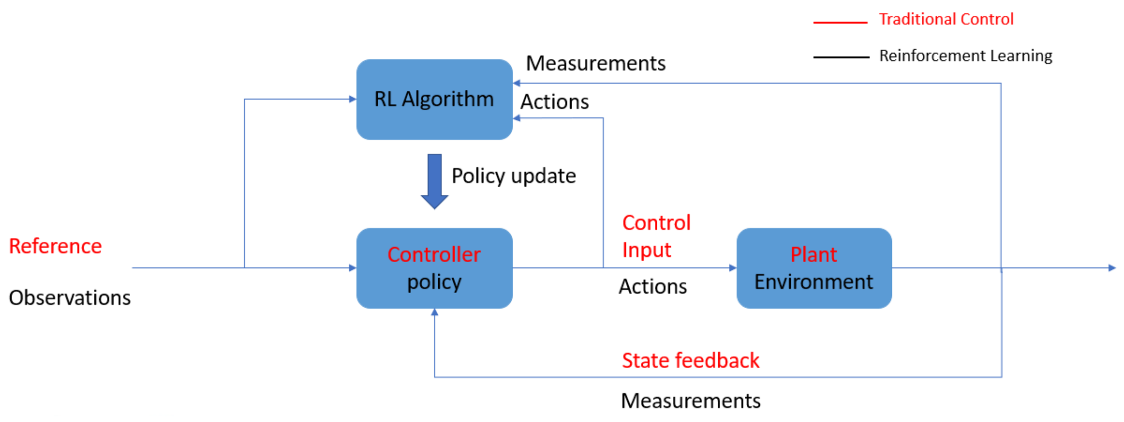

Figure 2.3: *Reinforcement Learning vs Traditional Control*

It should be noted that the plant, the reference signal, and the error computation are all considered to be part of the *environment* from an RL perspective. In general, the environment can also include additional elements such as measurement noise, disturbance signals, filters and converters.

In addition, the *observation* is any measurable value coming from the environment that is visible to the agent which depends on the available sensors for example the state system, measured output used then to in RL algorithms to find the optimal sequence of control input. The control input is defined by the *action* taken by the controller itself. The sequence of actions is called *policy* usually denoted as π the best ones are learned by the *agent* which is the *controller* based on a reinforcement learning method.

The *reward* is a function used for measuring the performance of the agent such as cost function in control theory. Moreover, the reward function allows the agent to understand if it is responding in a proper manner with respect to the desired system performance. As example, it is possible to implement reward functions which minimize the steady-state error while minimizing control effort.

Most of the material in this chapter is distilled from various books, papers and articles. Worth mentioning specifically are the books "Artificial Intelligence: A Modern Approach" by Russel and Norvig [19], "Reinforcement Learning: An Introduction" by Sutton and Barto [22], the UCL reinforcement learning course by Dr. David Silver, the papers "Human-level control through deep reinforcement learning" by Mnih et al. [13], "Deterministic policy gradient algorithms" by [20], and "Continuous control with deep reinforcement learning" by Dr. Lillicrap et al. [11].

Once the basic definition of what RL is and its comparison with control theory has been given, in the next section the main mathematical concepts that define the formulation of the RL problem with the possible approaches and resolution algorithms will be explored.

2.3 Reinforcement Learning fundamentals

Reinforcement learning is a computational approach to sequential decision making. It provides a framework that is exploitable with decision-making problems that are unsolvable with a single action requiring a sequence of actions, a broader horizon, to be solved. This section aims to present the fundamental ideas and notions behind this research field in order to help the reader to develop a baseline useful to approach the main algorithms.

2.3.1 Trial-error-learning

RL uses a trial-and-error learning cycles to maximize a decision-making agent's total reward observed from the environment. Compared to optimal control theory, this maximization problem can be viewed as minimizing the cost function since the reward function is designed based on the control problem and system requirements.

The *environment* is represented by a set of variables related to the problem and system. The combination of all the possible values this set of variables can take is referred to as *state space* usually denoted as S. A *state* s_t is a specific set of values the variables observed at any given time t. Agents may or may not have access to the *actual* environment's *state* s_t depending on sensors availability. The set of variables the agent can perceives at any given time t is called an *observation.* The combination of all possible values these variables can observe is the *observation space* denoted as O. State and observation are terms used interchangeably in the RL community. This is because usually agents are allowed to see the internal state of the environment, but this is not always the case. At every state s_t, the environment makes available a set of *actions* a_t the agent can choose from. Often the set of actions is the same for all states, but this is not required. The set of all possible actions in each states is known as *action space* usually denoted as A.

Figure 2.4 shows graphically the interaction between the agent and environment which is repeated sequentially over time where at each time step t, the agent observes a *state* $s_t \in S$, then selects an *action* $a_t \in A$ based on its policy which represents the agent's way of behaving at a given time t. The policy might be *deterministic* or *stochastic*, and it is usually denoted as $\pi(a_t|s_t)$ (2.4):

$$\pi(s_t, a_t) = P[a_t|s_t] \tag{2.4}$$

where the policy π returns a probability $P[a_t|s_t]$ of taking an action in presence of stochastic environments or directly the action is taken in case of deterministic ones.
The agent receives then a new observation from the environment at the next time step s_{t+1} as consequence containing a *scalar reward* r_{t+1} defined in eq. 2.5 and the next state s_{t+1}. The scalar signal reward represents the *instantaneous benefit* after transition from state s_t to s_{t+1} and taking a particular action a_t.

$$r_{t+1} = \rho(x_t, u_t) \tag{2.5}$$

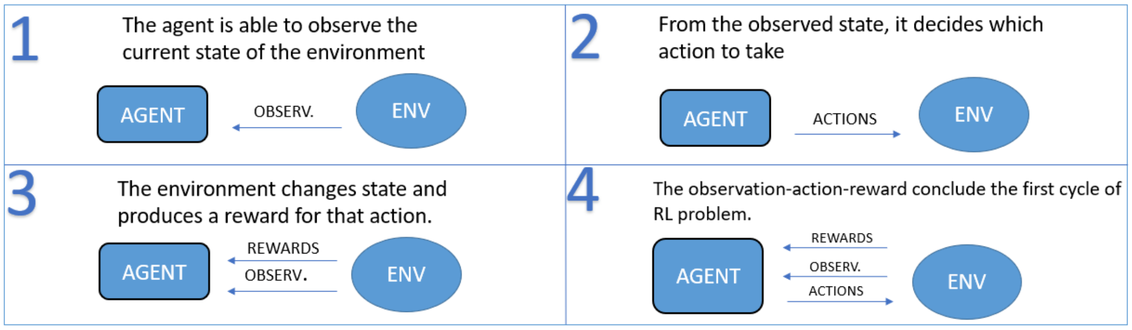
1
The agent is able to observe the current state of the environment
AGENT
OBSERV.
ENV
2
From the observed state, it decides which action to take
AGENT
ACTIONS
ENV
3
The environment changes state and produces a reward for that action.
AGENT
REWARDS
OBSERV.
ENV
4
The observation-action-reward conclude the first cycle of RL problem.
AGENT
REWARDS
OBSERV.
ACTIONS
ENV

Figure 2.4: *Trial-error-learning cycle*

The set of state, action, reward, and the new state observed by the agent at a specific time step t is called *experience* (2.6):

$$e_t =< s_t, a_t, r_{t+1}, s_{t+1} > \tag{2.6}$$

Every experience has an opportunity for learning and improving agent performance. The task which is being solved by the agent is may or may not have a natural ending. Tasks with natural ending, such as a games, are called *episodic tasks*. Conversely, tasks without natural ending are called *continuing tasks*, such as learning forward motion. The sequence of time steps from the beginning to the end of an episodic task is called an *episode*. Agents may take several time steps and episodes to learn the desired behavior. As it happens with human beings decisions, receiving conspicuous reward at a specific time step t does not exclude the possibility to receive a small reward immediately afterwards and sometimes it may be better to neglect immediate reward to gain greater ones later.

2.3.2 Return

The return value G_t is the accumulated reward mathematically defined as (2.7):

$$G_t(\tau) = r_{t+1} + \gamma r_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \quad \gamma \in [0,1] \tag{2.7}$$

where γ is the discount factor and it is mathematically required in order to bound the return value in case of an infinite-horizon sum of rewards. This parameter shows also the preference for immediate rewards rather than for future ones. The main goal of the agent is to *maximize the cumulative reward*. In this context, let's define τ as the sequence of states and actions in the environment defined in eq. 2.8:

$$\tau = (s_0, a_0, s_1, a_1, ...) \tag{2.8}$$

where s_0 is the measured output or state at time step $t = 0$, a_0 is the control input applied to the system or the action taken at time step $t = 0$, s_1 is the measured output or state at time step $t = 1$, a_1 is the control input applied to the system or the action taken by the agent at time step $t = 1$ and so on.

2.3.3 Markov Decision Process

The Markov Decision Process (MDP) provides a mathematical framework for modeling the environment, the policy and the agent.

It is defined by the tuple (S,A,r,P):

1. A set of states, S, which contains all possible states of the environment. In other terms, all the possible discrete or continuous measurable outputs.
2. A set of possible actions, $A(s)$, which contains all possible actions or control inputs to be applied to the system in each state.
3. A reward function $r_{t+1} = \rho(x_t, u_t)$, which is the immediate reward perceived after transition from state s_t to s_{t+1} with action a_t.
4. A transition model, $P(s_{t+1}|s_t, a_t)$, which denotes the probability of reaching state s_{t+1} when performing action a_t in state s_t.

The transition states of the environment are assumed to satisfy the *Markov property*, which means that the probability of reaching state s_{t+1} only depends on s_t, and not on the history of any earlier states. If the complete states of the environment are available to the agent through the state, the environment is *fully observable* otherwise is *partially observable.*

Moreover, the action and state spaces can be both discrete or continuous depending on the system. In case of discrete action space the number of possible actions are finite. On the other hand, a continuous action space is characterized by infinite number of possible actions. The type of action space will determine the possible approach to be applied to solve the RL problem.

At this point, it is required to identify other two main sub-elements that still need to be addressed in a RL problem that will be discussed in the following sub-sections.

2.3.4 Value function

The value function $V_\pi(s_t)$ is the expected E_π return value of state s_t when following policy π as defined in eq. 2.9. Practically, it represents what is the expected reward that the agent can presume to collect in the future starting from the current state s_t. The reward signal r_{t+1} represents only a local value of the reward in the sense that it takes into account only the state while the value function provides a broader view of future rewards: it is a *sort of prediction of rewards*.

$$V_\pi(s_t) = E_\pi[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ...|s_t] = E_\pi[r_{t+1} + \gamma G_{t+1}|s_t] \tag{2.9}$$

In other words, the value function estimates how good a state is, represented by the expected value E_π under policy π. Notice that it does not takes into account actions but only the goodness to be in that particular state $V_\pi(s_t)$ with respect to the specific task. By considering how good it is to be in a state by taking into account also the action taken by the agent, it is necessary to refers to the *action-value function*, also known as *Q-function* or *Q-value function* defined in eq. 2.10 and denoted as $Q_\pi(s_t, a_t)$, which gives the expected return for performing action a_t in state s_t at a generic time step t, under the policy π:

$$Q_\pi(s_t, a_t) = E_\pi[G_t|s_t, a_t] = E_\pi[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ...|s_t, a_t] = E_\pi[r_{t+1} + \gamma G_{t+1}|s_t, a_t] \tag{2.10}$$

In both of these value functions, $G_t(s_t)$ is the return value as defined in eq. 2.7. The optimal value function denoted as $V^*(s_t)$ is the one which yields maximum value compared to all other value function as defined in eq. 2.12:

$$V^*(s_t) = \underset{\pi}{max} V_\pi(s_t) \quad \forall s \in S \tag{2.11}$$

$$V^*(s_t) = \underset{a}{max} \sum_{s_{t+1} \in S} P(s_{t+1}, r_{t+1}|s_t, a_t) + [r_{t+1} + \gamma V^*(s_{t+1})] \quad \forall s \in S \tag{2.12}$$

In other words, it gives the maximum expected return when starting in state s_t, and acting according to the optimal policy π^* afterwards. On the other hand, the optimal action-value function denoted as $Q^*(s_t, a_t)$ and defined in eq. 2.13 is the one which yields the maximum value compared to all other action-value functions:

$$Q^*(s_t, a_t) = \underset{\pi}{max} Q_\pi(s_t, a_t) \quad \forall s \in S, a \in A \tag{2.13}$$

$$Q^*(s_t) = \underset{a}{max} \sum_{s_{t+1} \in S} P(s_{t+1}, r|s_t, a_t) + [r_{t+1} + \gamma \underset{a_{t+1}}{max} Q_\pi(s_{t+1}, a_{t+1})] \quad \forall s \in S, a \in A \tag{2.14}$$

and it gives the expected return when starting from state s_t, performing action a_t, and then acting according to the optimal policy π^* afterwards. There is an important connection between the optimal action-value function and the optimal action: the optimal policy in s_t will select the action which maximizes the expected return when starting in state s_t. Therefore, if $Q^*(s_t, a_t)$ is known, the optimal action $a^*(s_t)$ can be obtained directly as eq.2.15 shows.

$$a^*(s_t) = \underset{a}{\text{argmax}} Q^*(s_t, a_t) \tag{2.15}$$

2.3.5 Bellman Equation

The goal of the agent is to approximate an optimal policy π^* which maximizes the value of each state yielding V^* and Q^* as the optimal state value function and action value function.
Dynamic Programming (DP) is an optimization method which simplifies a complicated problem such as RL by recursively breaking it into simpler sub-problems which can then be solved. The idea is to exploit the fact that said value functions satisfy *Bellman Optimality Equation* (Barto and Mahadevan, 2003)[2], which yield:

$$V^*(s_t) = \underset{\pi}{max} V_\pi(s_t) \quad \forall s \in S \tag{2.16}$$

$$V^*(s_t) = \underset{a}{max} \sum_{s_{t+1} \in S} P(s_{t+1}, r|s_t, a_t) + [r_{t+1} + \gamma V^*(s_{t+1})] \quad \forall s \in S \tag{2.17}$$

$$Q^*(s_t, a_t) = \underset{\pi}{max} Q_\pi(s_t, a_t) \quad \forall s \in S, a \in A \tag{2.18}$$

$$Q^*(s_t) = \underset{a}{max} \sum_{s_{t+1} \in S} P(s_{t+1}, r_{t+1}|s_t, a_t) + [r_{t+1} + \gamma \underset{a_{t+1}}{max} Q_\pi(s_{t+1}, a_{t+1})] \quad \forall s \in S, a \in A \tag{2.19}$$

where $P(s_{t+1})$ is the *transition state probability* representing the probability of observing the reward function r_{t+1} and state s_{t+1} from the previous one s_t after performing action a_t.

2.3.6 Expectation

Since the objective is to maximize long-term future rewards, the *expectation* stated in eq. 2.20 is employed as the weighted sum of all conceivable outcomes multiplied by their probabilities. In other words, the expectation, also known as the *mean*, is computed by the summation of the product of every state value and its probability:

$$E[f(x)] = \sum_x P(x) f(x) \tag{2.20}$$

where $P(x)$ represents the probability of the occurrence of a general random variable x, and $f(x)$ is a general function denoting the value of the state x.

2.3.7 Exploration vs. Exploitation

An important problem in RL is the trade-off between *exploration* and *exploitation*. To achieve high rewards, the agent has to choose actions by applying a control input it has tried before and knows to be a good one. To discover these actions, the agent must choose actions that have not been tested using observed consequences from the environment. *Exploration* refers to the RL agent exploring the environment to collect more information, and *exploitation* means simply following the current best policy from the experiences collected in the past and available to the agent to gain as much reward as possible. In other words, short-term rewards have to be sacrificed in order to be able to find a good policy in the long run by exploring states that are not seen yet by the agent. The key challenge that arises in designing RL systems is in balancing the trade-off between exploration and exploitation. In a stochastic Environment, actions will have to be sampled sufficiently well to obtain an expected reward estimate. An Agent that pursues exploration or exploitation exclusively is bound to be less than expedient. It becomes worse than pure chance (i.e. a randomized agent).

There are some solutions such as naive explorations and the most used ones are called *greedy*, *ϵ-greedy* and *decayingϵ-greedy*decaying explorations. Those techniques are quite simple to comprehend and put into practise, but they suffer from major setback which is they have sub-optimal regret. As a matter of fact the regret of both greedy and -greedy grows linearly by the time. This is simple to comprehend intuitively since the greedy would focus on an activity that, while it may have produced positive outcomes in the past, was not in fact the best course of action. So, greedy will continue taking advantage of this situation while dismissing possible alternatives. In other words, it exploits too much.

The ϵ-greedy on the other hand, explores too much because even when one action seem to be the optimal one, the methods keeps allocating a fixed percentage of the time for exploration, thus missing opportunities and increasing total regret. Its formulation is fairly straight-forward:

$$\text{let } \epsilon \in (0,1)\text{: } \begin{cases} a^*(s_t) = \underset{a}{\operatorname{argmax}} Q^*(s_t, a_t) & \text{with probability } (1-\epsilon) \\ \text{random action} & \text{with probability } \epsilon. \end{cases} \tag{2.21}$$

The action $a^*(s_t)$ expressed in eq. 2.21 is taken by the agent according to the best current policy π.
Notice that at the beginning of the training, the probability of doing exploration will be huge with a high ϵ value, so most of the time the agent will explore. As training goes on and consequently Q-functions gets better and better in its estimations, it is needed to progressively reduce the ϵ value since the agent will need less and less exploration and more exploitation:

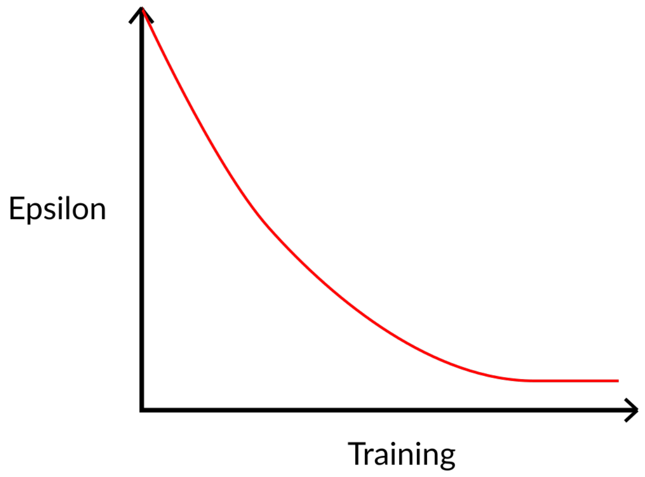

Figure 2.5: *Epsilon response during training*

On the other hand, as time passes, the decaying Epsilon Greedy approaches attempt to reduce the proportion set up for exploration. The graph in Figure 2.6 demonstrates how this might result in the best regret.

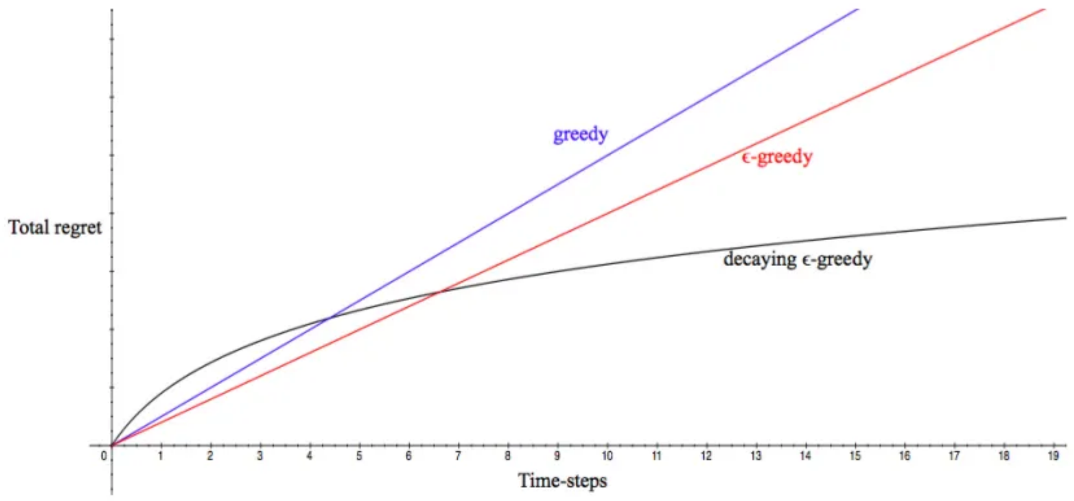

Figure 2.6: *Regret comparison with respect to exploration methods*

The difficulty is in being able to carry out the proper withering process.

2.3.8 Approaches to RL

Every agent consists of an RL algorithm that exploits to maximize the reward it receives from the environment. Each kind of algorithm has its own peculiarity. It is very important to understand the differences between in order to adequately understand what type of algorithm satisfies better the needs of the specific problem. Nowadays, RL algorithms are numerous and drawing the complete picture behind them could be a complicated purpose. The distinctions presented in this section aim to describe the most crucial distinctions that are useful in the context of the thesis without claiming to be exhaustive.

2.3.8.1 Learning Components

The first worthy distinction among RL algorithms can be made by analyzing how the algorithms exploit the different components of the agent: it is possible to explain the main strategies in RL using *policy*, *model* and *value function* quantities defined in sections 2.3.1 and 2.3.4 respectively.

One of the most crucial aspects of an RL algorithm is the question of whether the agent has access to the model of the environment: this element enables the agent to predict state transitions and rewards. A method is *model-free* when it does not exploit the model of the environment to solve the RL problem. All the actions control input applied by the agent to the system results from direct observation of the current situation in which the agent is. It takes the observation and then select the best control input to feed into the system. This last representation is in contrast with the *model-based* methods where the agent has the full knowledge of the environment including the system plant. Both groups of methods have strong and weak sides. Ordinarily, model-based methods show their potential in a deterministic environment by making the agent able to extract all the knowledge from the environment and learn an optimal policy to follow. However, the opportunity of having the knowledge about the environment and in particular about the plant is not always possible due to many factors such as the knowledge of the system parameters or systems which are too complex to model. At this time, model-free methods are more popular and have been more extensively developed and tested than model-based methods [6]. This thesis focuses mainly on the model-free approaches. On the other hand, model-free methods tend to be more straightforward to train and tune because it is usually hard to design and build models of complex systems.

The use of policy or value function as the central part of the method represents another essential distinction among RL algorithms. The approximation of the agent's policy is the base of *policy-based* methods. In this case, the policy representation is usually a probability distribution over available actions. This method aims to optimize the agent's behaviour directly and may require multiple observations from the environment which makes this method not so sample efficient. On the opposite side, methods could be *value-based*. In this case, the agent is still involved in finding the optimal behaviour and sequence of control input to follow in an indirectly way by exploiting the value functions 2.3.4. Particularly, it is not interested anymore about the probability distribution of actions but on determining the value of all actions available by choosing the best one. The main difference from the policy-based method is that this one can benefit from other sources such as old policy data or replay buffer.

2.3.8.2 Learning approaches

Many control issues in robotics and automated driving need for sophisticated, nonlinear control structures. These issues can be solved using methods like gain scheduling, robust control, and nonlinear model predictive control (MPC), but doing so frequently necessitates a control engineer with a high level of domain knowledge as well as a system able to be measured *online*. Gains and parameters, for instance, are challenging to tune. The complexity of nonlinear MPC's computational requirements can make the resulting controllers difficult to implement. Differently in RL, the learning setting of the agent's behaviour could be *online* or *offline*. In the first case, the learning process is done in parallel while the agent continues to collect new information from the system, while the second one progresses toward learning using limited data. In the context of this thesis, online learning will be considered: the learning phase is not bound to already gathered data, but the whole process goes on using both old data coming from replay buffers and brand new data obtained in the most recent episode.

Another significant difference in RL algorithms is the distinct use of policy to learn. *On-policy* algorithms profoundly depend on the training data sampled according to the current policy because they are designed to use only data gathered with the last learned policy. On the other hand, an *off-policy* method can use a different source of valuable data for the learning process instead of direct experience. For instance, this feature allows the agent to use large experience buffers of past episodes. In this context, these buffers are usually randomly sampled in order to make the data closer to being independent and identically distributed.

In the following sub-section the common used model-based approach *Dynamic Programming (DP)* will be explained. DP is very important to analyze since it is relatively straightforward and it contains all the main ideas and frame bases used by most RL algorithms.

2.3.9 Model-based approach: Dynamic Programming

Dynamic Programming (DP) is one of the approaches used to solve RL problems. Formally, it is a general method used to explain complex problems by breaking them into more manageable sub-problems. After solving all sub-problems, it is possible to sum them up in order to obtain the final solution to the whole original problem. Particularly, this technique provides a practical framework to solve MDP based problems by observing what is the best result achievable from it assuming *full knowledge* of the system. For this reason, it applies primarily to model-based problems.

DP methods are based on *bootstrapping* means that these strategies use one or more values estimated at the update stage for the same type of estimated value making the results more sensitive to the original values. The way of bootstrapping and updating step define the DP methods that mainly are *policy iteration* and *value iteration*.

2.3.9.1 Policy Iteration

The Policy Iteration aims to find the optimal policy π^* by directly manipulating a starting randomized one to properly evaluate the current policy by iteratively applying the *Policy Evaluation* 2.23. Subsequently, *Policy Improvement* 2.24 represents the second step towards Policy Iteration algorithm where a new better policy than previous one is sought by choosing the action in a specific state s_t with a more rewarding one. It is possible to check if a new policy $\pi^{'}$ is better than the previous one π by using the action-value function $Q_\pi(s_t, a_t)$. As explained in 2.10, this function returns the value of taking action a_t in the current state s_t and, after that, following the existing policy π.

In this context, it is possible to conclude that if:

$$Q_{\pi'}(s_t, a_t) > Q_\pi(s_t, a_t) \tag{2.22}$$

where $\pi^{'}$ is the new policy and π is the previous one, it follows that the selected actions coming from the new policy $\pi^{'}$ is better with respect to action chosen by the current policy π in terms of performances as well as reward resulting in a better policy. Based on that, it is reasonable to act greedily to find a better policy starting from the current one iteratively selecting the action which produces the higher $Q_\pi(s_t, a_t)$ for each state. This iterative application of policy improvement stops after an improvement step that no longer modifies the initial policy. In this way, the optimal policy π^* is obtained.

The Policy Evaluation and Policy Improvement steps could be resuming as follows:

1. **Policy Evaluation** = evaluate V-function with a given policy π ($v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_\pi$) using synchronous backups by iterating Bellman Equation explained in Section 2.3.5. Practically, it starts from some arbitrary V-function $V_\pi(s_t)$ for all the states and looking at a step forward and iterating the Bellman equation, the V-function of the initial states can be updated using the rule according to eq. 2.23.

$$V(s_t) \leftarrow E_\pi[r_{k+1} + \gamma V_\pi(s_{t+1})] \tag{2.23}$$

2. **Policy Improvement** = improve the policy by acting greedily with respect to $V_\pi(s_t)$ of the current evaluated policy π according to eq. 2.24. In other words, the best action for each state is extracting in order to improve π and ensure that it measures up to the best possible action. In this way, the current policy π is improved and by repeating all those loops until the policy will not change anymore it is possible to find the optimal policy.

$$\pi(s_t) \leftarrow \underset{a}{argmax} E_\pi[r_{t+1} + \gamma V_\pi(s_{t+1})] \tag{2.24}$$

The corresponding pseudo-algorithm concerning Policy Iteration is given below.

```
Algorithm 1: Policy Iteration Algorithm
  Data: $\theta$: a small number
  Result: $V$: a value function s.t. $V \approx v_*$, $\pi$: a deterministic policy
          s.t. $\pi \approx \pi_*$
  Function PolicyIteration is
      /* Initialization                                              */
      Initialize $V(s)$ arbitrarily;
      Randomly initialize policy $\pi(s)$;
      /* Policy Evaluation                                           */
      $\Delta \leftarrow 0$;
      while $\Delta < \theta$ do
          for each $s \in S$ do
              $v \leftarrow V(s)$;
              $V(s) \leftarrow \sum_{s',r'} p(s', r|s, \pi(s))[r + \gamma V(s')]$;
              $\Delta \leftarrow \max(\Delta, |v - V(s)|)$;
          end
      end
      /* Policy Improvement                                          */
      policy-stable $\leftarrow$ true;
      for each $s \in S$ do
          old-action $\leftarrow \pi(s)$;
          $\pi(s) \leftarrow arg \max_a \sum_{s',r'} p(s', r|s, a)[r + \gamma V(s')]$;
          if old-action != $\pi(s)$ then
              policy-stable $\leftarrow false$;
          end
      end
      if $policy - stable$ then
          return $V \approx v_*$ and $\pi \approx \pi_*$;
      else
          go to Policy Evaluation;
      end
  end
```

where $V(s) = V(s_t)$ 2.9, V^* is the optimal V-function defined in 2.12, $V_\pi = V^\pi$ is the V-function under the current policy π, π^* is the optimal policy to be found, S is the state space, $s^{'} = s_{t+1}$. Notice that θ is the parameter which defines the accuracy: the more the value is closer to 0, the more the evaluation would be precise.

2.3.9.2 Value Iteration

The second approach used by DP to solve MDPs is *Value Iteration (VI)*. Vi algorithm is and iterative technique that alternate policy evaluation and policy improvement steps until it converges to the optimal policy π^*. On the other hand, VI uses a modified version of policy evaluation to determine the V-function $V(s_t)$ and then it learn the optimal policy from it.

The corresponding pseudo-algorithm concerning Value Iteration is given below.

Algorithm 2: Value Iteration Algorithm

```
Data: $\theta$: a small number
Result: $\pi$: a deterministic policy s.t. $\pi \approx \pi_*$
Function ValueIteration is
    /* Initialization                                         */
    Initialize $V(s)$ arbitrarily, except $V(terminal)$;
    $V(terminal) \leftarrow 0$;
    /* Loop until convergence                                 */
    $\Delta \leftarrow 0$;
    while $\Delta < \theta$ do
        for each $s \in S$ do
            $v \leftarrow V(s)$;
            $V(s) \leftarrow \max_a \sum_{s',r'} p(s', r|s, a)[r + \gamma V(s')]$;
            $\Delta \leftarrow \max(\Delta, |v - V(s)|)$;
        end
    end
    /* Return optimal policy                                  */
    return $\pi$ s.t. $\pi(s) = arg\max_a \sum_{s',r'} p(s', r|s, a)[r + \gamma V(s')]$;
end
```

2.3.9.3 Policy Iteration vs Value Iteration

Policy Iteration and Value Iteration are both dynamic programming algorithms that find an optimal policy π^* in a Reinforcement Learning environment. They both make use of one-step look-ahead and modified Bellman updates as previously discussed.

The main differences between the two approaches are highlighted:

- In Policy Iteration, the starting point is a fixed policy. Conversely, in Value Iteration starts by selecting the value function. Then, in both algorithms, the improvement step is iteratively applied until convergence is reached.

- The policy is updated via the algorithm for Policy Iteration. Instead of iterating over the value function, the Value Iteration algorithm does thus. Nonetheless, each iteration of both algorithms indirectly updates the state value function and policy.

- The Policy Iteration function passes through two phases for each iteration. The policy is assessed in the first phase, and it is improved in the second. By taking the maximum over the utility function for all feasible actions, the value iteration function covers these two stages.

- The value iteration algorithm is straightforward. It combines two phases of the policy iteration into a single update operation. However, the value iteration function runs through all possible actions at once to find the maximum action value. Subsequently, the value iteration algorithm is computationally heavier.

- Both algorithms are guaranteed to converge to an optimal policy in the end. Yet, the Policy Iteration algorithm converges within fewer iterations. As a result, the policy iteration is reported to conclude faster than the value iteration algorithm.

It is possible to conclude that:

Policy Iteration	**Value Iteration**
Starts with a random policy	Starts with random value function
Algorithm is more complex	Algorithm is simpler
Guaranteed to converge	Guaranteed to converge
Cheaper to compute	More expensive to compute
Requires few iterations to converge	Requires more iterations to converge
Faster	Slower

Table 2.1: *Policy Iteration vs Value Iteration resume*

For a more in-depth discussion of model-based RL and their implementation, see *" Deep Reinforcement Learning "* by Dr. Li. [10], chapter 6. However, DP algorithms are limited by the high calculation cost caused by the analysis of the entire state space as well as the need for states and/or Q-functions storage in lookup tables. RL seeks to solve these issues by the limitation of the exploration to the more probable states only, based on the learning experience, and the later employment of *function approximators* such as *neural networks* [2].

2.3.10 Model-free approaches

The main hypothesis of DP methods is to have a comprehensive knowledge of the environment which is not always accurate from a practical point of view especially for very complex systems to be modelled or highly uncertain environments. In these cases, the agent or the controller, has to infer information using its experience by exploiting *model-free methods*, based on the assumption that there is no prior knowledge about state transition and rewards.

This section intends to provide a brief description of main model-free approach used for Control Systems: *Temporal Difference (TD).*

2.3.11 Temporal Difference Learning

Temporal Difference (TD) learning is a very central topic in Reinforcement Learning. TD learning methods are model-free where the agent can learn directly from experiences without a model of the environment. In this context, the agent learns by *bootstrapping* from the estimation of the present V-function or Q-function, which means the estimates are updated by using other learned function estimates without waiting for the actual outcome. TD learning often refers to the *prediction problem* with an update rule for the V-function as defined in eq. 2.25:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{2.25}$$

where α is known as the learning rate $\in$ (0,1) while $\gamma \in$ (0,1) is the discount factor. Notice that, the agent takes into account more recent reward for $\alpha = 1$ and learns nothing when $\alpha = 0$.

The term $r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$ defined in eq. 2.25 is known as the *TD error* and arises in various forms in many areas of RL.

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{2.26}$$

The terms that add up in the TD error are denoted as *TD Target*:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) \tag{2.27}$$

The TD learning technique for prediction is used in two different learning methods for solving RL problems, which are discussed below:

- SARSA (State-Action-Reward-next State-next Action)
- Q-Learning

2.3.11.1 SARSA learning

SARSA is an *on-policy* and *online* TD learning-based technique and it gets its namefrom the definition of experience that underlies the algorithm defined as a quintuple that represents a transition from one state-action pair to the next one: $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$. It works by taking the principle of TD prediction in order to learn $Q(s_t, a_t)$ instead of $V(s_t)$. It is an on-policy method since it estimates $Q_\pi(s_t, a_t)$ *for the current policy* $\pi(s_t)$, and in simultaneously update the current policy $\pi(s_t)$ greedily with respect to the estimated Q-function $Q_\pi(s_t, a_t)$. The Q-function is updated by bootstrapping as formulated in eq. 2.25) following the rule 2.28 after every transition to a non-terminal state s_t.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{2.28}$$

Notice that if a state s_{t+1} is found to be terminal, $Q(s_{t+1}, a_{t+1})$ is set to zero. It can be shown that SARSA converges to $Q^*(s_t, a_t)$ and $\pi^*(s_t)$ when all state-action pairs are visited an infinite number of times and at the same time the policy converges to be purely greedy. In other words, an agent interacts with the given environment and performs policy update based on selected actions.

SARSA pseudo-algorithm is given below.

Algorithm 1 SARSA (on-policy TD control) for estimating $\pi \approx \pi_*$

```
Algorithm parameters: step size $\alpha \in (0,1)$, small $epsilon > 0$;
Initialize $Q(s_t, a_t)$, for all $s_t \in \mathcal{S}, a_t \in \mathcal{A}$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$;
For each episode:
    Initialize $s_t$;
        For each step of episode:
            Choose $a_t$ from $s_t$ using policy derived from $Q(s_t, a_t$ (e.g., -greedy);
            Take action $a_t$, observe $r_{t+1}$, $s_{t+1}$;
            $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$;
            $s_t \leftarrow s_{t+1}$;
            $a_t \leftarrow a_{t+1}$;
        end;
end;
```

2.3.11.2 Q-learning

One of the early breakthroughs in Reinforcement Learning is the *off-policy* control algorithm *Q-learning*. It takes the Q-function and updates it very analogously to SARSA where the only difference is how the bootstrapping is done. In the Q-learning algorithm, the future action is taken using greedy policy i.e choose an action which maximize the Q-function of the next state as shown in eq. 2.29.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \underset{a}{max} Q(s_{t+1}, a_{t+1})] \tag{2.29}$$

where $\alpha \in$ (0,1) is the learning rate and $\gamma \in$ (0,1) is the discount factor.

Q-learning pseudo-algorithm is given below.

Algorithm 2 Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

```
Algorithm parameters: step size $\alpha \in (0,1)$, small $epsilon > 0$;
Initialize $Q(s_t, a_t)$, for all $s_t \in \mathcal{S}, a_t \in \mathcal{A}$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$;
For each episode:
    Initialize $s_t$;
        For each step of episode:
            Choose $a_t$ from $s_t$ using policy derived from $Q(s_t, a_t$ (e.g., -greedy);
            Take action $a_t$, observe $r_{t+1}$, $s_{t+1}$;
            $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_t, a) - Q(s_t, a_t)]$;
            $s_t \leftarrow s_{t+1}$;
        end;
end;
```

where a refers to the action which maximize the Q-function at its specific time step t.

2.3.11.3 SARSA vs Q-learning

The most important difference between the two is how the Q-function is updated after each action a_t. Off-policy methods such as Q-learning update its Q-function values using the one of the next state s_{t+1} with the *greedy* action a_{t+1} which maximize the Q-function at time step $t + 1$. In other words, it estimates the total discounted future reward for each Q-functions assuming a greedy policy were followed despite the fact that it's not following a greedy policy.

On the other hand, On-policy methods such as SARSA learning, updates its Q-functionv alues using the Q-value of the next state s_{t+1} with the current policy's action a_{t+1}. it estimates the total discounted future reward for each Q-functions assuming the current policy continues to be followed.

The following table highlights the main differences in the algorithm between Q-Learning and SARSA:

Q-learning	**SARSA**
1. Move one step selecting a_t from action space A	1. Move one step selecting a_t from action space $\pi(s_t)$
2. Observe r_{t+1}, s_{t+1}	2. Observe r_{t+1}, s_{t+1}, a_{t+1}
3. Update the Q-function $Q(s_t, a_t)$	3. Update the Q-function $Q(s_t, a_t)$
4. Update the policy $\pi(s_t) \leftarrow \underset{a}{argmax}(s_t, a_t)$	4. Update the policy $\pi(s_t) \leftarrow Q(s_{t+1}, a_t)$

Table 2.2: *Q-learning vs SARSA resume*

2.4 Conclusion: Reinforcement Learning drawbacks

The algorithms discussed in this chapter work with systems characterized by well-defined and time-discrete states and actions. In this context, a V-function-based or a Q-function-based table is created with a number of rows equal to the size of the state space and a number of columns equal to the action space. This way of proceeding can bring to some non-negligible problems as the V-function has an entry for each state while the Q-function has an entry for each state-action pair. This setting can be problematic in case of continuous action spaces over time or with a large number of action or state spaces from a computation point of view and memory availability as it becomes difficult to manage the storage of a large number of states and actions.
Furthermore, there may be obstacles regarding the slowness in learning the value of each state individually. This problem is called the *curse of dimensionality.* A solution to this problem can be the use *Artificial Neural Networks (ANNs)* as *function approximators.* The intersection of neural networks theory (known as *Deep Learning*) as function approximators with Reinforcement Learning is called *Deep Reinforcement Learning.*

In the next chapter some basic concepts from Deep Learning will be first explained in order to go more in detail through Deep Reinforcement Learning methods.

Chapter 3

Deep Reinforcement Learning

Reinforcement Learning has evolved a long way with the enhancements from *Deep Learning*. Recent research efforts into combining Deep Learning with Reinforcement Learning have led to the development of some very powerful Deep Reinforcement Learning (DLR) systems, algorithms, and agents which have already achieved some extraordinary accomplishment [5]. The basic idea of DLR is to no longer directly compute the value of V-function or Q-function as RL problems, but to estimate their values in order to solve the curse of dimensionality problem mentioned in section 2.4:

$$\begin{aligned} V(s_t) &\approx V_\pi(s_t) \\ Q(s_t, a_t) &\approx Q_\pi(s_t, a_t) \end{aligned} \tag{3.1}$$

The use of *Artificial Neural Networks* as estimators reduces the training time for high dimensional systems and especially it requires less space in memory. This point represents the bridge between traditional RL and recent discoveries in the theory of Deep Learning. One of the first steps towards DRL and general AI broadly applicable to a different set of various environments was done by *Deep Mind* with their pioneering paper [14] and the consequent [13].

The focus of this chapter will be model-free algorithms due to the nature of the work. The following sections aims to explain the state-of-the-art and the leading theory behind Deep RL framework together among an overview about the use of Neural Networks as function approximators followed by the presentation of two deep $actor - critic$ algorithms used for solving control problems in this thesis: *Deep Deterministic Policy Gradient* method.

3.1 Deep Learning fundamentals

Deep Learning (DL) involves using multi-layered non-linear function approximation, typically Neural Networks. DL is not a separate branch of ML, so it's not a different task than those described above. DL is a collection of techniques and methods for using neural networks to solve ML tasks. Deep Reinforcement Learning is simply the use of DL to solve RL problems.

This section focuses the attention to the *Deep Learning fundamentals* providing an outline of the basic concepts used by Deep Learning in combination with Reinforcement Learning.

3.1.1 Artificial Neural Networks

Artificial neural networks (ANNs) were created as an attempt to mimic how the animal and human brain functions. They turned out to be a oversimplification, but remain very useful as function approximators. ANNs are called *networks*, because they contain multiple neurons (i.e nodes) which are connected together as well as human brain. Each neuron consists of numerous inputs called dendrites coming from proceeding neurons. When a linear combination of the inputs exceeds a specific potential, it fires through its single output called axon. Mathematically, the elaboration phase of each neuron j consists in taking the inputs and elaborate them by taking the *weighted sum* and then adding a *bias* b:

$$in_j = \sum_{i=0}^{n} wx_{i,j} + b \tag{3.2}$$

where the index i ranges over all nodes in the previous layer connected to it. Then, an activation function g is applied, producing the output (see Figure 3.1):

$$a_j = g(in_j) = g(\sum_{i=0}^{n} wx_{i,j} + b) \tag{3.3}$$

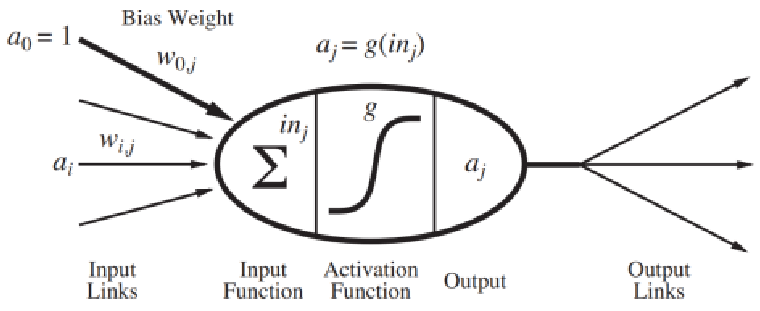

Figure 3.1: *Example of a simple model for a single neuron [19].*

These neurons are arranged in different *layers*, which can be divided into three broad categories: *input layers*, *hidden layers*, and *output layers*. Every neural network has one or more input layers where the input data is fed to the network as well as one or more output layers, that produce the resulting specific output based on the network's task. Input and output layers are connected through hidden layers which contain the network dynamics with respect to its task. Their number depends on the estimated complexity of the activation function. Figure 3.2 illustrates one of the most used configuration in which nodes in a feed-forward network are fully connected to all the nodes in the layer above them.

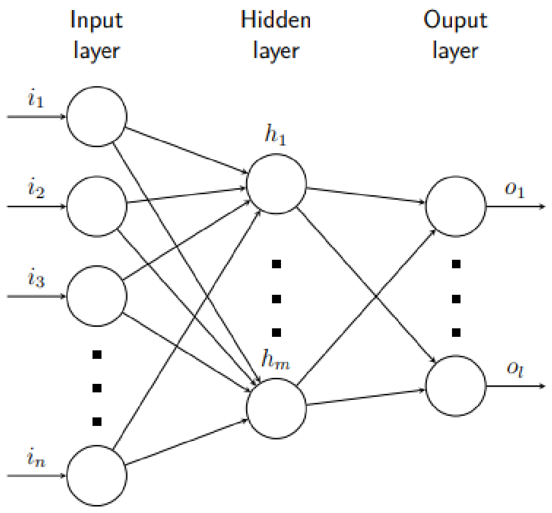

Figure 3.2: *Simple example showing a deep neural network with four layers [19].*

Each node implements a linear classifier, but the network as a whole can approximate nonlinear functions as well.
An early proof of the universal approximation theorem showed that two hidden layers may describe any function and later demonstrated that a single hidden layer is sufficient to represent any continuous function. This is done by varying the different weights of the nodes in the network collectively called *network parameters* and usually denoted as θ. Next, the problem is to find a weight combination that will produce a function nearest to the target function that the network is estimating based on the learning process[19].

3.1.2 Learning process

The learning process aims to seek the neural network set of parameters θ resulting in the best possible function approximation for a specific objective.

The learning process consists in the following steps:

1. **Forward path**: the vector containing the input-features X is forwarded through the neural network in order to predict or estimate the output value of interest $\hat{Y} = f(X, \theta)$.
2. **Loss**: the resulting estimated value ($\hat{Y}$) is compared with the *actual value* y by computing the *loss function* $L(\theta)$ based on the difference between the output of the neural network for a specific input data y_θ and the actual value y. There are several loss functions but one of the most used one is the *Mean-Squared Error (MSE)* defined in eq. 3.4.

$$L(y, \hat{y}) = (y_\theta - y)^2 \tag{3.4}$$

3. **Back-propagation**: the learning starts by computing the global gradient of the loss function, which is carried out together with its *back-propagation* through all the network. The back-propagation computes the gradient of the loss function with respect to its parameter $\nabla\theta_L$ for

each neuron in the hidden layers by back-propagating using the chain rule showed in eq. 3.5, which computes derivatives of composed function by multiplying the local derivatives.

$$y = g(x),\; z = f(g(x)),\; \frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x} \tag{3.5}$$

4. **Update**: the final step of the learning consists updating the weights of all neurons. The most common rule used to carry out the update phase is the *gradient descent* defined with the main purpose of *minimizing* the loss function by refreshing the internal parameters of the network in the negative direction of the gradient loss. Eq. 3.6 defines the update rule based on *gradient descent*. Notice that this process will bring the function approximation closer to the minimum at each iteration.

$$\theta_{t+1} \leftarrow \theta_t - \alpha \nabla \theta_L \tag{3.6}$$

   where α is the learning rate $\in [0, 1]$ which represents one of the hyperparameters of the neural networks.

5. **Regularization**: The final aim of the learning process is to obtain an approximate function able to generalize over data. This means that a neural network should show performances on unseen data with respect to the one obtained from training data otherwise the so-called effect *overfitting* is produced. On the other hand, if the neural network is not able to show good performances on both unseen and training data, it causes the opposite phenomenon to overfitting called *underfitting*.
   Regularization represents an approach to overcome and prevent the problem of overfitting and underfitting which consists on extending the loss function (see 3.4) with a *regularization term*:

$$L^{'}(\theta) = L(y, \hat{y}) + \lambda \Omega(\theta) \tag{3.7}$$

   where λ is the regularization factor.

   Equations 3.8 and 3.9 show two examples of regularization terms.
   The first one is L^1*-regularization* which exploits the squared sum of the weights θ in order too keep the weights small.
   The second approach is known as L^2*-regularization*: in this case, large weights are less penalized, but this method leads to a sparser solution.

$$L^1(\theta) = L(y, \hat{y}) + \lambda \frac{1}{2} ||\theta||^2 \tag{3.8}$$

$$L^2(\theta) = L(y, \hat{y}) + \lambda \frac{1}{2} ||\theta|| \tag{3.9}$$

3.1.2.1 Activation functions

The activation function is the responsible of bringing non linearity to the neural network in order to improve the approximation of complex and non linear function.
The most used activation functions are listed below and plotted in Figure:

$$\begin{aligned} \text{Sigmoid} \rightarrow g(x) &= \frac{1}{1 + e^{-x}} \\ \text{Hyperbolic Tangent} \rightarrow g(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}} \\ \text{Rectified Linear Unit (ReLu)} \rightarrow g(x) &= max(0, x) \\ \text{Leaky Rectified Linear Unit (Leaky ReLu)} \rightarrow g(x) &= max(\alpha, x) \end{aligned} \tag{3.10}$$

where $\alpha \in (0,1)$ is an hyperparameter network generally set to 0.01.

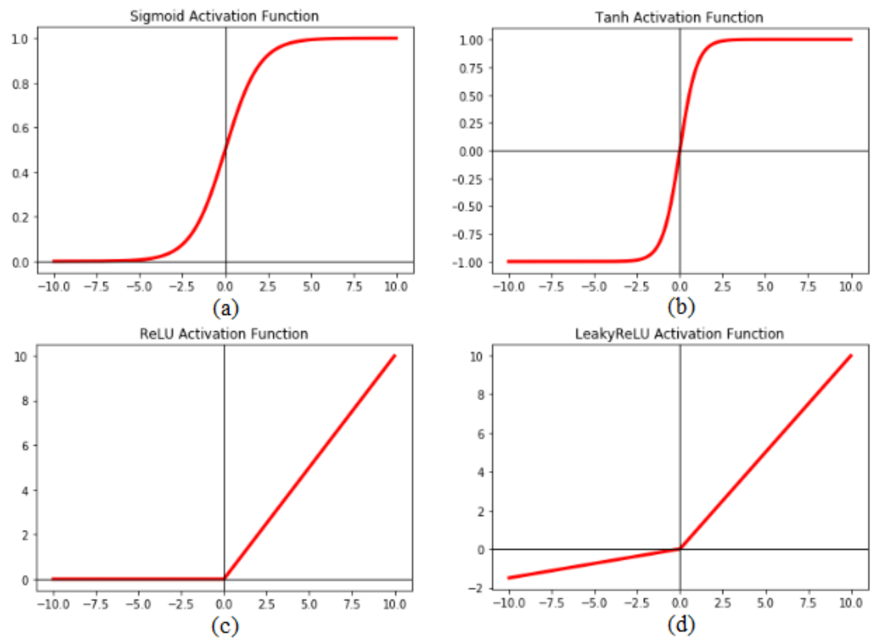

Figure 3.3: *Activation functions plot: (a) Sigmoid; (b) Tanh; (c) ReLU; (d) LeakyReLU*

Normally, neural networks use *ReLu* in the hidden layers because it is easy to compute and does not saturate and the *Tanh* in the output layer in order to normalize the output between 0 and 1.

These components serve as the foundation for the neural networks' learning phase, which is utilized to estimate the interested RL function. Algorithms which use networks for estimating and providing directly the control input are known as *Policy Gradient*. On the other hand, *actor-critic* algorithms use networks for estimating the Q-function in order to provide in a second step the best control input.

In the following section both of them will be discussed in detail in order to derive the so called *Deep Deterministic Policy Gradient algorithm*, the one used in this thesis.

3.2 Policy Gradient algorithms

Differently from Q-Learning, policy gradient algorithms has the main objective to generate a trajectory τ which maximizes the expected reward by learning the optimal policy π^* and providing directly the best action starting from any given time step t until the terminal time T.

From a mathematical point of view, policy gradient methods consist of maximizing $J(\theta)$ value by finding a proper policy by updating θ parameters of the neural network which parameterized the policy function:

$$\begin{gathered} J(\theta) = E[\sum_{t=0}^{T-1} r_{t+1}] \\ \theta^* = \underset{\theta}{argmax} J(\theta) \end{gathered} \tag{3.11}$$

Since this is a *maximization problem*, the policy is optimized by taking the gradient ascent with the partial derivative of the objective with respect to the policy parameter θ in order to refresh the parameters of the policy:

$$\theta_{t+1} \leftarrow \theta_t + \alpha \nabla_\theta J(\theta) \tag{3.12}$$

where α is the learning rate which defines the strength of the steps in the direction of the gradient.

Notice that the *gradient ascent* is the reverse of the *gradient descent* defined in eq. 3.6 and updates parameters θ_t in the positive direction of the gradient of the policy's performance measured by the $\nabla_\theta J(\theta)$ term.

3.2.1 Deriving the policy gradient

By substituting the expectation term with its definition the following result is obtained:

$$J(\theta) = E[\sum_{t=0}^{T-1} r_{t+1} | \pi_\theta] = \sum_{t=i}^{T-1} P(s_t, a_t|\tau) r_{t+1} \tag{3.13}$$

where i is an arbitrary starting point in a trajectory, $P(s_t, a_t|\tau)$ is the probability of the occurrence of s_t, a_t given the trajectory t_{tau}.
By differentiating both side with respect to policy parameter θ and using $\frac{\partial \log f(x)}{\partial x} = \frac{f'(x)}{f(x)}$:

$$\frac{\partial J}{\partial \theta} = \nabla_\theta J(\theta) \sim \sum_{t=i}^{T-1} \nabla_\theta \log P(s_t, a_t|\tau) r_{t+1} \tag{3.14}$$

Notice that the expected reward was replaced since random samples of episodes will be taken.
By using return definition as defined in eq. (2.7), the following result is obtained:

$$\nabla_\theta J(\theta) = \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t \tag{3.15}$$

Notice that the first term measures how likely the trajectory is under the current policy π and by multiplying it with the return it increases in case of high positive rewards. On the contrary, the likelihood of a policy is decreasing if it results in a high negative reward. In this context, the essence of policy gradient is increasing the probabilities for *good* actions and decreasing those of *bad* actions with will not be learned.

The main steps of Gradient Policy algorithms can be resumed as follows:

1. Perform a trajectory roll-out using the current policy π;
2. Store log probabilities and reward values at each step t;
3. Compute discounted cumulative future reward at each time step t;
4. Compute policy gradient and update policy parameter θ;
5. Repeat 1-4 until the optimal policy π^* is reached.

The main advantage of policy gradient algorithms consists in the stability of their convergence since they update the current policy directly at each time step t instead of renewing value functions from which to derive the policy. Furthermore, policy gradient algorithms can face infinite and continuous action space because the agent estimates the action directly instead of computing the Q-value for each possible action. The third feature is their ability to learn stochastic policies which can be useful in uncertain contexts or in case of partially observable environments. Despite the presence of the advantages just mentioned, policy gradient algorithms have a substantial disadvantage: they tend to converge to a local maximum instead of the global optimum since they are based on log of probabilities resulting in noisy gradients and this could cause unstable learning or convergence to a sub-optimal policy.

In this context, a mix approach known as *Actor-Critic architecture* was introduced in order to overcome this issue.

3.2.2 Actor-Critic architecture

Pure policy gradient approaches tend to learn slowly and they are difficult to implement for online applications due to estimates with high variance. However, the TD methods discussed in section 2.3.11 can be used to solve these issues.

Actor-critic methods combine *policy gradient* and TD techniques in order to learn both a policy and a Q-function simultaneosly using the last one as bootstrapping. The policy is learned by the *actor* neural network, and controls how the agent behaves and act. The Q-function is learned by the *critic* neural network, and measures how good or bad an action chosen by the actor is. In particular, the *actor* relates to the policy, while the *critic* deals with the estimation of a value function (e.g Q-value function).

In this context of Deep Reinforcement Learning, they can be represented using neural networks as function approximators: the actor exploits gradients derived from the policy gradient method and adjust the policy parameters, while the critic estimates the approximate value function for the current policy π. The learning phase is realized by the interaction between actor and critic neural networks with the environment at each time step t as follows:

1. The actor is a network that is trying to take the best action a_t which maximizes its output based on the current state s_t.

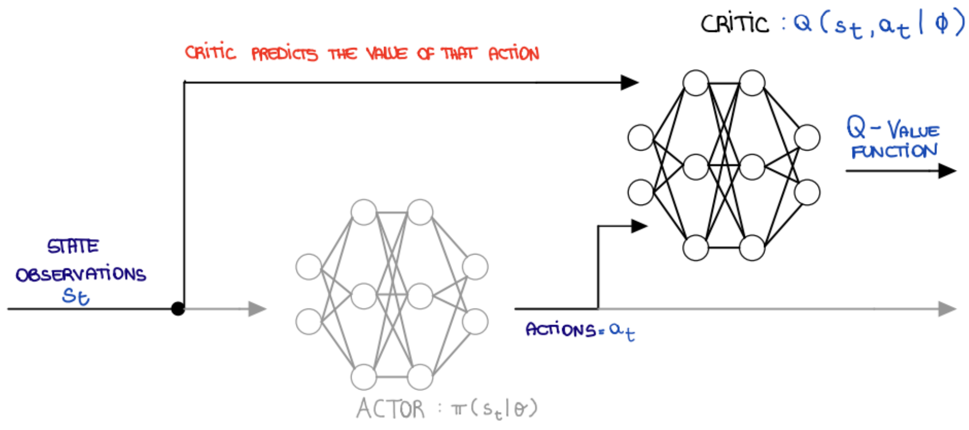
CRITIC : $Q(s_t, a_t | \phi)$
CRITIC PREDICTS THE VALUE OF THAT ACTION
Q - VALUE FUNCTION
STATE OBSERVATIONS s_t
ACTIONS = a_t
ACTOR : $\pi(s_t | \theta)$

Figure 3.4: *Actor-Critic learning phase: Step 1*

2. The critic is a second network that is trying to estimate the Q-function by taking as input the action took by the actor a_t at time step t and the observation from the environment s_t, s_{t+1} and r_{t+1}.

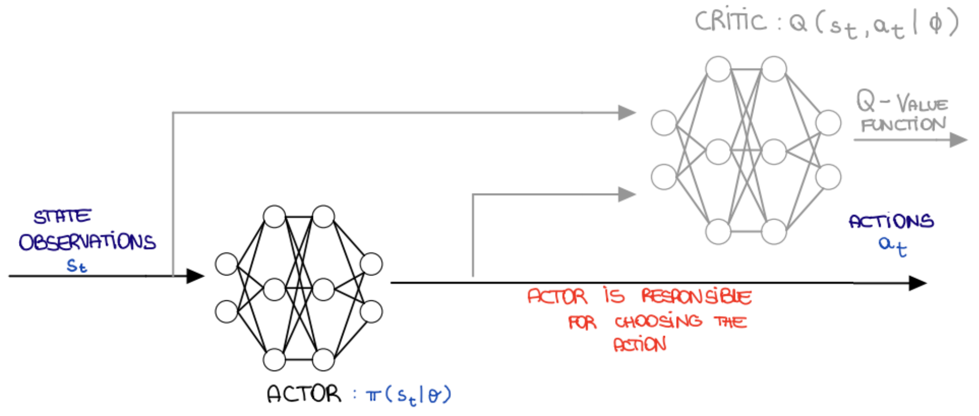
CRITIC : $Q(s_t, a_t | \phi)$
Q - VALUE FUNCTION
STATE OBSERVATIONS s_t
ACTIONS a_t
ACTOR IS RESPONSIBLE FOR CHOOSING THE ACTION
ACTOR : $\pi(s_t | \theta)$

Figure 3.5: *Actor-Critic learning phase: Step 2*

3. The critic then uses the reward r_{t+1} from the environment to determine the accuracy of its value prediction by computing the error defined as the difference between the new estimated value of the previous state (*target value*) and the old value of the previous state from the critic network *actual value*: the *TD error* defined in eq. 2.26 and used by TD methods. The new estimated value is based on the received reward and the discounted value of the current state. The error gives the critic a sense of whether things went better or worse than it expected.

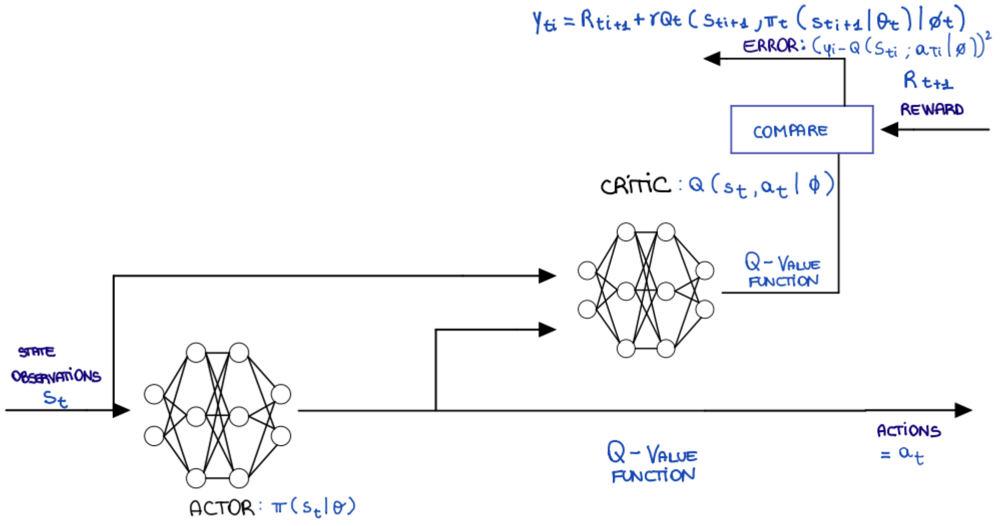
$y_{ti} = R_{ti+1} + \gamma Q_t(s_{ti+1}, \pi_t(s_{ti+1} | \theta_t) | \phi_t)$
ERROR: $(y_i - Q(s_{ti}; a_{ti} | \phi))^2$
R_{t+1}
REWARD
COMPARE
CRITIC : $Q(s_t, a_t | \phi)$
Q - VALUE FUNCTION
STATE OBSERVATIONS s_t
Q - VALUE FUNCTION
ACTIONS $= a_t$
ACTOR: $\pi(s_t | \theta)$

Figure 3.6: *Actor-Critic learning phase: step 3*

4. The critic uses this error to update itself in the same way that a value function would so that it has a better prediction the next time will be in this state.

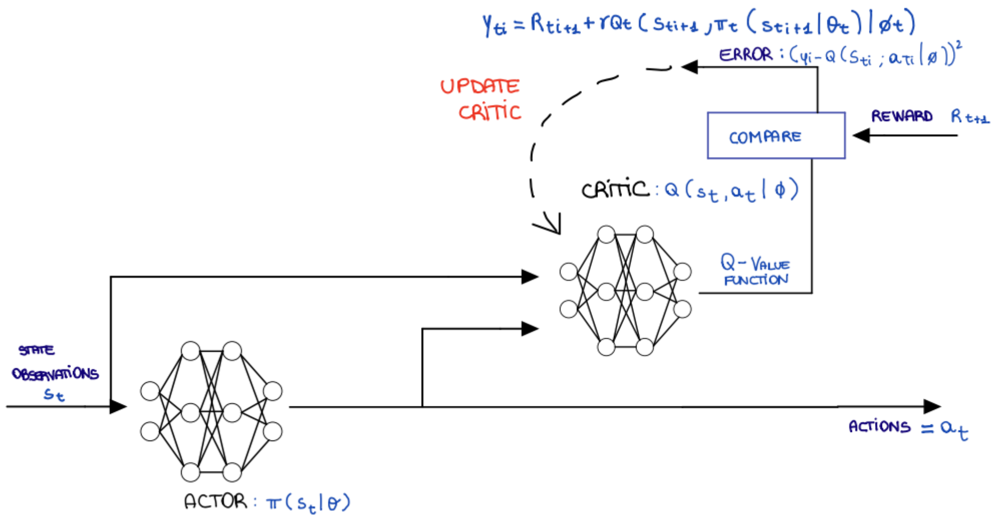
$y_{ti} = R_{ti+1} + \gamma Q_t(S_{ti+1}, \pi_t(s_{ti+1}|\theta_t)|\phi_t)$
ERROR : $(y_i - Q(S_{ti}; a_{Ti}|\phi))^2$
UPDATE CRITIC
COMPARE
REWARD R_{t+1}
CRITIC : $Q(s_t, a_t|\phi)$
Q-VALUE FUNCTION
STATE OBSERVATIONS S_t
ACTIONS $= a_t$
ACTOR : $\pi(s_t|\theta)$

Figure 3.7: *Actor-Critic learning phase: step 4*

5. The actor also updates itself using the critic output in order to adjust its probability of taking that specific action again in the future.

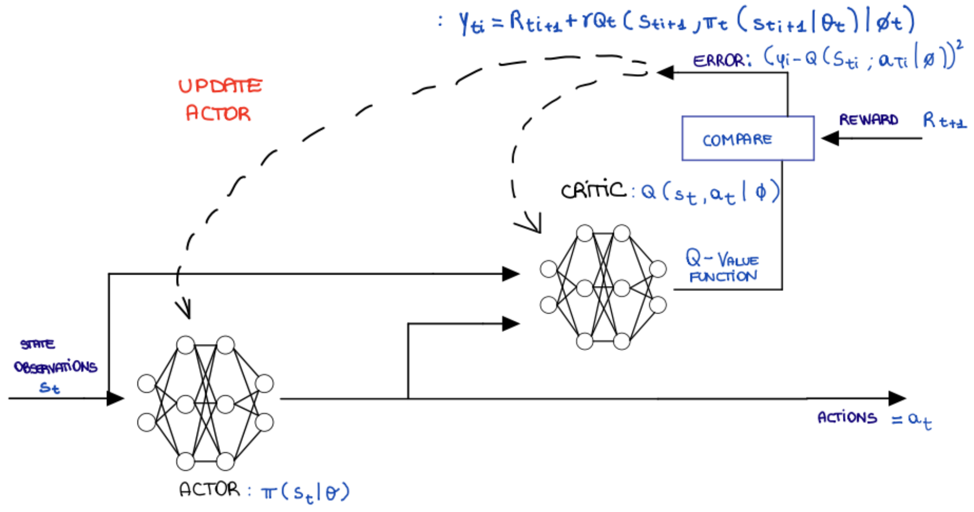
: $y_{ti} = R_{ti+1} + \gamma Q_t(S_{ti+1}, \pi_t(s_{ti+1}|\theta_t)|\phi_t)$
ERROR: $(y_i - Q(S_{ti}; a_{Ti}|\phi))^2$
UPDATE ACTOR
COMPARE
REWARD R_{t+1}
CRITIC : $Q(s_t, a_t|\phi)$
Q-VALUE FUNCTION
STATE OBSERVATIONS S_t
ACTIONS $= a_t$
ACTOR : $\pi(s_t|\theta)$

Figure 3.8: *Actor-Critic learning phase: step 5*

This idea of combining policy gradient algorithm and TD technique is nowadays considered standard for solving RL problems thanks to its performance and stability. Most modern algorithms rely on actor-critic architecture and expand this basic idea into more sophisticated and complex technique such as *Deep Deterministic Policy Gradient* algorithm which is one of the most used technique for control systems which will be explained in the next section.

3.3 Deep Q-learning

Q-Learning is an algorithm widely used in Reinforcement Learning. Initially it was considered an unstable algorithm when used with neural networks and therefore its use was limited to tasks and problems involving state spaces with limited dimensionality. However, it has been demonstrated in [25] that Q-Learning algorithms and techniques can be used with neural networks. This algorithm has shown the achievement of human-level performance on seven video games on the Atari 2600 console using only raw pixel images as input. The introduction of neural networks extended the algorithm name from Q-learning to Deep Q-Learning or Deep Q-Network (DQN).

DQN is a function approximation method of RL which works only with discrete action and state spaces. Particularly, it represents an evolution of the Q-Learning method where the state-action table is replaced by a single neural network, the Critic network as defined in actor-critic architecture. In this model-free, online and off-policy algorithm therefore learning does not consist in updating the table but consists in adjusting the weights of the neurons that make up the network through backpropagation. The learning of the value function in DQN is therefore based on the modification of the weights according to the loss function L_t:

$$L_t = (E[r_{t+1} + \gamma \underset{a}{max} Q(s_{t+1}, a_t)] - Q(s_t, a_t))^2 \tag{3.16}$$

where $(E[r_{t+1} + \gamma \underset{a}{max} Q(s_{t+1}, a_t)]$ represents the optimal expected return while $Q(s_t, a_t)$ is the neural network outcome at time step t.

The errors calculated by the loss function will be propagated backwards in the network by means of backpropagation, following the gradient descent logic. Indeed the gradient indicates the direction of greatest growth of a function and moving in the opposite direction the error is reduced.
The policy behavior is $\epsilon - greedy$ approach to ensure sufficient exploration.

The key aspect that makes DQN working well is the use of experience replay. With such technique, the agent experience $e_t = (s_t, a_t, r_{t+1}, s_{t+1}$ collected at each time step t is saved in a data-sed of the form $D = (e_t, e_{t+1}, ..., e_T)$ known as*replay memory.*

The training is carried out through a mini-batch technique, i.e. by taking a sub-set of experience samples randomly extracted from this replay memory. The use of this technique allows the use of past experiences of be used in more than one network update. In addition, the subset chosen randomly from the replay memory allows you to break the strong correlation between successive experiences, thus reducing the variance between updates.

3.4 Deep Deterministic Policy Gradient

Deep deterministic policy gradient (DDPG) is a *model-free*, *off-policy*actor-critic algorithm with continuous action spaces, presented by Dr. Lillicrap et al. [11] in 2015. It is the main algorithm chosen for study in this thesis, mainly due to being adapted specifically for environments with continuous action spaces, which most physical control tasks have, and because of its high performance. DDPG is an extension of two other algorithms, Deep Q-Networks (DQN) [13] and Deterministic Policy Gradient (DPG) [20]. Specifically, it utilizes the experience replay and target network techniques which will be discussed in-depth below, and uses actor-critic with a deterministic policy. In particular, DDPG concurrently learns a Q-value function $Q(s_t, a_t)$ and a policy π: it uses off-policy data and the *Bellman Equation* defined in eq. 3.17 in order to learn the Q-function thanks to which it will subsequently learn the policy ϕ:

$$Q_\pi(s_t, a_t) = E_{s_{t+1}\sim E}[r_{t+1} + \gamma E_{a_{t+1}\sim\pi}[Q_\pi(s_{t+1}, a_{t+1})] \tag{3.17}$$

where r_{t+1} is the reward observed from the environment after the action a_t at time step t, $s_{t+1} \sim E$ means that the transition is sampled from the environment defined as E, and $a_{t+1} \sim \pi$ means that the action is sampled from policy π. If the policy is deterministic, it is usually denoted as μ, and the inner expectation of the Bellman equation can be avoided:

$$Q_\mu(s_t, a_t) = E_{s_{t+1}\sim E}[r_{t+1} + \gamma Q_\phi(s_{t+1}, \mu(s_{t+1})] \tag{3.18}$$

Notice that Q_μ can be learned *off-policy*, by using transitions generated by a different policy β since expectation only depends on the environment. Using the greedy policy from Q-learning, $\mu(s_t) = \underset{a}{\text{argmax}} Q(s_t, a_t)$ and representing the Q-function as a function approximator parameterized by θ_Q, the mean-squared error can be used as a loss function in the same way actor-critic method approach (see 3.2.2).

The optimization problem is by minimizing the loss function:

$$L(\theta_Q) = E_{s_t \rho^\beta, a_t\sim\beta, r_{t+1}\sim E}[(y_t - Q(s_t, a_t)|\theta_Q)^2], \tag{3.19}$$

where

$$y_t = r_{t+1} + \gamma Q(s_{t+1}, \mu(s_{t+1})|\theta_Q), \tag{3.20}$$

and ρ^β is the discounted state transition for the policy β. y_t is often called the *target value.*
DDPG uses an actor-critic approach where the actor is parameterized approximation of a deterministic policy $\mu(s_t|\theta_\mu$, and the critic is parameterized approximation of the action-value function, $Q(s_t, a_t|\theta_Q)$, and they are both represented by deep neural networks. The critic $Q(s_t, a_t)$, is learned using the Bellman equation as Q-learning (ref. to 3.17), while the actor is learned by using the policy gradient. Dr. Silver et al. [20] showed that for a deterministic policy, the policy gradient is simply the expected gradient of the action-value function:

$$\nabla_{\theta_\mu} J \approx E_{s_t\sim\rho^\beta}[\nabla_{\theta_\mu} Q(s_t, \mu(s_t|\theta_\mu)|\theta_Q)] \tag{3.21}$$

$$= E_{s_t\sim\rho^\beta}[\nabla_{a_t} Q(s_t, \mu(s_t)|\theta_Q)\nabla_{\theta_\mu}\mu(s_t|\theta_\mu)] \tag{3.22}$$

3.4.1 Replay Buffers

Most optimization algorithms used for training neural networks assume that the samples used are independently and identically distributed. In RL, where the samples are generated from sequentially interacting with the environment, this assumption does not hold. Additionally, to take advantage of hardware optimizations, it is essential to learn in mini-batches, rather than online.
One way to deal with this issue, is to use an experience replay buffer, which was introduced by Dr. Lin [12]. As in Deep Q-learning method, different experience tuples $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$ are generated, and saved in a *replay memory.* This set contains a finite amount of previous experiences, and at each time step, both the actor and critic are updated using a uniformly sample mini-batch of tuples from this buffer, yielding uncorrelated sample for training. When the buffer is full, old samples are discarded to make place for new ones.

3.4.2 Target networks

The DDPG algorithm uses another trick to achieve stable learning with the deep neural networks by using target networks. The critic network $Q(s_t, a_t|\theta_Q)$ is being updated while also being used in the target value defined in eq. 3.20 of the loss function, which means the parameter update depends on the parameters θ which are being updated. This causes the Q-function update to be prone to divergence, and makes learning unstable. To avoid this, copies of both the actor and critic networks are created, and denotes as $\mu^{'}(s_t|\theta_{Q_\mu})$ and $Q^{'}(s_t, a_t|\theta_{Q^{'}})$. They are used for computing the target values, hence their names.

The idea is that the weights of the target networks are initialized as copies of the weights of the actor and critic networks, but updated more slowly to keep them fixed for some time steps. The target networks are updated with "soft" updates:

$$\theta_{Q^{'}} \leftarrow \tau\theta_Q + (1-\tau)\theta_{Q^{'}} \tag{3.23}$$

$$\theta_{\mu^{'}} \leftarrow \tau\theta_\mu + (1-\tau)\theta_{\mu^{'}} \tag{3.24}$$

where $\tau \in (0, 1)$ is a hyperparameter, usually with a small value (e.g. 0.001). This causes the target networks to change slowly, which slows the learning phase but greatly improves learning stability.

3.4.3 Exploration

As discussed earlier, the trade-off between *exploration* and *exploitation* is an important problem in reinforcement learning, especially in continuous action spaces. Due to the fact that DDPG is off-policy the problem of exploration can be completely separated from the learning algorithm itself. In order to make the DDPG agent explore the environment, noise sampled from a noise process N is added to the actor policy when selecting an action during training:

$$\pi(s_t) = \mu(s_t|\theta_\mu) + N \tag{3.25}$$

This is called action noise. Different noise processes can be used, the original DDPG paper [19] suggest an *Ornstein-Uglenbeck process* explained in [18], which is time correlated. In order to reduce the failure rate of training, it is used to use a the variance of the noise between *1 %* and *10 %* of the maximum control input action.

3.5 Conclusion: discussion on RL

At this point it is possible to think that only setting up an environment, placing the RL agent in it and then let the computer solve all the problems while the Engineer is doing other things. Unlucky, even if a perfect agent is setup and the RL algorithm converges on a solution, there are still drawbacks such as the need of experience to know how to interpret correctly the results and how to improve them.

There challenges come down to two main questions:

- How is known the solution is going to work?
- Could be improved even if the solution seems good?

The first issue regards the fact that a policy is made up of a neural network with very high number of weights and biases and non linear activation functions. The contribution of these values and the structure of the network create a complex function that maps high-level observations to low-level actions. This function is a black box to the designer. It is possible to have a sense of how this function operates but it is possible to know the reason behind those values. Another problem concerning this type of approach is the computing power and memory available in the personal computer. The timing of the application of these algorithms strongly depends on the computing power and can go from a few minutes to a few hours with the same hyperparameters but with two different computers. On the other hand, traditional control approaches can quickly become difficult or impossible to achieve when the system is hard to model, is highly nonlinear, or has large state and action spaces (e.g walking robot) but they can be applied when the plant is not too complicated to model and its dynamics are perfectly known. Due to all these problems mentioned above, the implementation of the DDPG algorithm that will be deepened in the next chapter, took a long time to adapt the tuning of the hyperparameters both from the point of view of the time required for training and for the high number of hyperparameters to be tested to obtain good performance by the agent on the system to be controlled.

Chapter 4

DDPG implementation using RL MATLAB Toolbox

Reinforcement Learning Toolbox provide functions and blocks for training policies using RL algorithms including Deep Deterministic Policy Gradient. These policies can be used to implement controllers and decision-making algorithms for complex systems. It is also possible to implement the policies using deep neural networks, polynomials or look-up tables. This toolbox gives the possibility to train policies by enabling them to interact with an environment modelled using *Matlab* scripts or *Simulink* blocks. The evaluation of the algorithms among with experimentation with hyperparameter setting as well as monitor training progress is also provided.

In this section the implementation of DDPG algorithm on a linear and non linear dynamical system for optimal control will be presented in order to evaluate the algorithm performances.

The implementation will be divided analyzing step-by-step the following flow-chart:

1. **Formulate Problem**: define the task for the agent to learn, including how the agent interacts with the environment from the set of observations and actions point of view and any primary and secondary goals the agent must achieve.
2. **Create Environment**: define the environment within which the agent operates, including the interface between agent and environment and the environment dynamic model.
3. **Define Reward**: design the reward signal that the agent uses to measure its performance with respect to task goals and how this signal is calculated from the environment.
4. **Create agent**: define the agent, which includes defining a policy representation and configuring the agent learning algorithm and its hyperparameters.
5. **Train agent**: train the agent policy representation using the defined environment, reward and agent learning algorithm.
6. **Validate agent**: evaluate the performance of the trained agent by simulating the agent environment together.

4.1 Example: DC Motor

There are various uses of Direct Current machines (DC motors) in the industry. DC motors are used in both high and low power applications as well as fixed and variable speed electric drives. For example, their applications range from low power toys, spinning and weaving machines, vacuum cleaners, elevators, electric traction as well as spacecrafts and so on. The speed of a DC motor can be controlled easily, by changing voltage and current depending on the type of the DC motor used.

In this thesis, a linear DC Motor speed control using the voltage applied to the motor as control input is considered.

4.1.1 Speed control problem

Consider the following state space representation of a DC motor second-order continuous time model:

$$x_{t+1} = f(x_t, u_t) = Ax_t + Bu_t \tag{4.1}$$

$$\rightarrow \frac{d}{dt}\begin{bmatrix}\omega \\ i\end{bmatrix} = \begin{bmatrix}\frac{-b}{J} & \frac{K}{J} \\ \frac{-K}{L} & \frac{-R}{L}\end{bmatrix}\begin{bmatrix}\omega \\ i\end{bmatrix} + \begin{bmatrix}0 \\ \frac{1}{L}\end{bmatrix} V$$

where:

- $x_t \in \mathbb{R}^n$ is the measured state DC Motor speed $[\frac{rad}{sec}]$, $x_{max} = 10$ $[\frac{rad}{sec}]$,$x_{min} = 0$ $[\frac{rad}{sec}]$;
- $u_t \in \mathbb{R}^m$ is the control input Voltage applied to the DC motor [V] , $u_{max} = 10$ $[V]$, $u_{min} = 0$ $[V]$;
- $y_t = y_{t_{meas}} \in \mathbb{R}^p$ is the output assumed to be equal to the state: $y_t = x_t$.
- A is the state matrix with appropriate dimensions representing the system dynamics by taking into account the evolution of the speed
- B represents the contribution of the input u in vector form with appropriate dimension
- C is the output matrix with appropriate dimension

It is assumed that the DC Motor can only rotate counterclockwise with a saturated range of
$u_t \in [0, 10][V]$.

For the infinite-horizon LQT problem, the goal is to design an optimal controller for the system which ensures that the output $y_{t_{meas}}$ tracks a step reference speed trajectory $y_{t_{ref}}$ and acts on the difference between the two defined as control error e_t.

The optimal control problem can be seen as regularization of the form:

$$y_{t_{ref}} - y_{t_{meas}} = 0 \rightarrow e_t = 0 \tag{4.2}$$

4.1.2 Designing reference speed signal

The regularization control problem is set with respect to a *time-varying step function* characterized by a period of $T = 60$ $[s]$ designed in Figure 4.1.

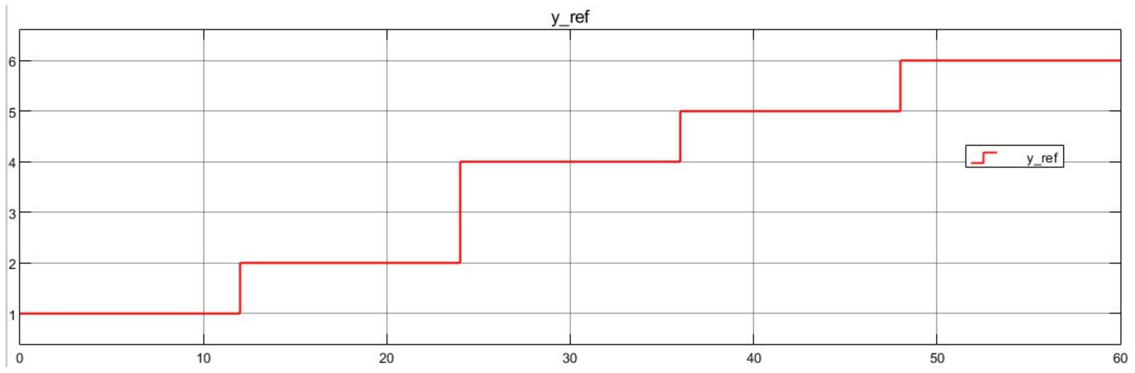

Figure 4.1: *Reference speed signal*

The design of the step function reference signal with different amplitudes was carried out in order to allow the neural network, and therefore the agent to become aware of all the frequencies present in its operating range and to verify after training whether the agent was able to track any step function or other shapes such as *sin* and *ramp* with different amplitudes until saturation.

4.1.3 Create Environment

In a Reinforcement Learning scenario, the environment *model* the dynamics with which the agent interacts and the one that the agent must learn. In particular, the dynamics include the plant modeling and also the disturbances. Practically speaking, the environment is everything except the controller defined by the RL algorithm, reward and policy.

Usually what is done to solve a RL problem is to train in a simulated environment which contains the plant dynamics and then improve its behavior by continuing to train the agent in the real environment so that it is also able to face off against a real environment characterized by uncertainty and disturbances.

4.1.3.1 Plant modelization

The DC motor is a typical actuator in control systems. When combined with wheels, drums, and cables, it may also provide translational motion in addition to rotational motion.

Figure 4.2 displays the rotor's free-body diagram as well as the armature's electric circuit.

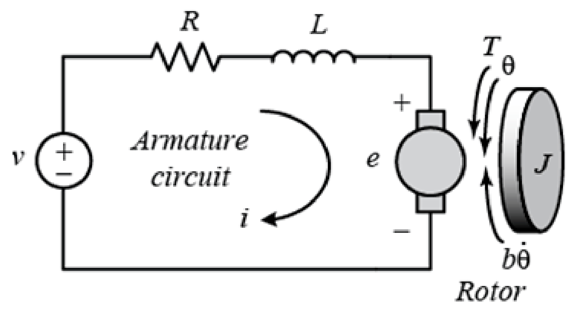

Figure 4.2: *DC Motor electric circuit*

It is assumed that the input of the system is the voltage source (V) applied to the motor's armature, while the output is the rotational speed of the shaft $\dot{\theta}$. The rotor and shaft are assumed to be rigid. Finally, it is also assume a viscous friction model, that is, the friction torque is proportional to shaft angular velocity.

The following table shows the physical variables values:

DC Motor Physical Setup	
Moment of Inertia	0.01 $[Kg.m^2]$
Motor Viscous Friction	0.1 $[Kg.m.s]$
Electromotive Force Costant	0.01 $[V/\frac{rad}{sec}]$
Motor Torque Constant	0.01 $[\frac{N.m}{A}]$
Electric Resistance	1 $[\Omega]$
Electric Inductance	1 $[H]$
Sampling Time	0.01 $[sec]$

Table 4.1: *DC Motor physical parameters*

In general, the torque generated by a DC motor is proportional to the armature current and the strength of the magnetic field. It is assumed that the magnetic field is constant and, therefore, that the motor torque is proportional to only the armature current i by a constant factor K_t as shown in eq. 4.3. This is referred to as an armature-controlled motor.

$$T = K_t i \tag{4.3}$$

The back emf denoted as emf is proportional to the angular velocity of the shaft by a constant factor K_e as shown in eq. 4.4.

$$emf = K_e \dot{\theta} \tag{4.4}$$

In SI units, the motor torque and back emf constants are equal resulting in $K_t = K_e = K$.

This system will be modeled by summing the torques acting on the rotor inertia and integrating the acceleration to give velocity. Additionally, the armature circuit will be subject to Kirchoff's laws.

Finally, by applying Newton's law to the motor system, its dynamics is obtained:

$$J\frac{d^2\theta}{dt^2} = T - b\frac{d\theta}{dt} \rightarrow \frac{d^2\theta}{dt^2} = \frac{1}{J}(K_t i - b\frac{d\theta}{dt}) \tag{4.5}$$

$$L\frac{di}{dt} = -Ri + V - emf \rightarrow \frac{di}{dt} = \frac{1}{L}(-Ri + V - K_e\frac{d\theta}{dt}) \tag{4.6}$$

The plant's dynamics will be defined as the environment of the RL problem using SIMULINK blocks as shown in Figure 4.3.

It is also necessary to create the environment MATLAB script object which will interact with the agent by generating the instantaneous rewards signals and observations in response to agent actions.

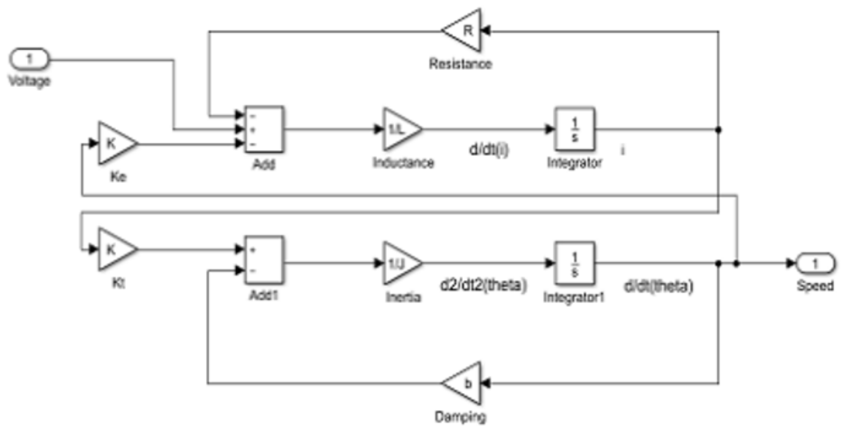

Figure 4.3: *Plant modelization using Simulink*

4.1.4 Observation and Action signals

The observation signal S which consists of the system feedback is defined in such a way as to make the agent able to learn all frequencies covered by the various in time reference speed signal designed previously: $S \in [y_{t_{meas}} \quad y_{t_{ref}} \quad \dot{y_{t_{meas}}} \quad \dot{y_{t_{ref}}}]$.

The action set A, which is the control input range, is continuous and is determined by the action range of the voltage given to the DC Motor since the DDPG agents must be trained with continuous sets: $A \in [0\ 10]\ [V]$ which will be normalized in the same scale as the observation set.

4.1.5 Reward signal

The reward signal r_{t+1} is one of the most important designing function used to guide the learning process of the agent because this signal measures the performance of the agent with respect to the control problem and system performance. In other words, for a given observation or state s_t, the reward measures the effectiveness of taking a particular action. During training, an agent updates its policy π based on the rewards received for different Q-function combinations in order to maximize the total outcome of the reward function representing the main goal of the agent in a RL-based framework.

At each time step t, the reward is defined as follows:

$$r_{t+1} = r(x_t, u_t) = \; -Q|e_t| - q11 \; |\dot{y}_{meas}| \tag{4.7}$$

where $e_t = y_{t_{ref}} - y_{t_{meas}}$.

Since this is a maximization problem of a negative quantity, the agent must learn how to make the reward function converge to zero as close as possible. The first term is responsible for tracking the reference signal by minimizing the error while the second one for reducing the oscillating behaviour once it achieved the tracking of speed

4.1.6 Normalization

In this example a *min-max normalization* defined in eq. 4.8 was implemented for both observation and action spaces in order to improve speed as well as performance of neural networks during the training.

$$x_{k_{norm}} = \frac{x_{k_{meas}} - x_{min}}{x_{max} - x_{min}} \; (h - l) + l \tag{4.8}$$

- $x_{k_{meas}}$ is the original data with no normalization,
- $x_{k_{norm}}$ is the normalized data,
- x_{max}, x_{min} are respectively the maximum and minimum values of the quantity to be normalized
- h, l are respectively the upper and lower values of the new range for the normalized data: $x_{norm} \in [l = -1; h = 1]$.

4.1.7 Training configuration

This section will first demonstrate the control schema that was implemented on the *Simulink platform* before presenting the settings and hyperparameters that were used to train the agent after learning the optimal policy π^*.

4.1.8 Control schema

The DDPG implementation for Motor Speed control is performed on *Simulink platform* is shown in Figure 4.4.

As can be noticed, it is divided into 5 main blocks:

1. Reference Signal Design block representing the *time-varying step* speed reference signal design known as y_{ref};
2. Plant/Environment Model block representing the DC Motor dynamics discussed in 4.1.3.1. In addiction, a white noise was designed and added to the control input in order to simulate the tolerance of the DC Motor;
3. Observation Set block representing the observation set defined as $S \in [y_{t_{meas}} \quad y_{t_{ref}} \quad \dot{y}_{t_{meas}} \quad \dot{y}_{t_{ref}}]$;
4. State and Input Normalization block where *min-max normalization* 4.1.6 technique is implemented for both states x_t and control input u_t at each time step t.

5. Agent/Controller block where the reward signal r_{t+1} designed in 4.1.5 is implemented as well as the DDPG agent.

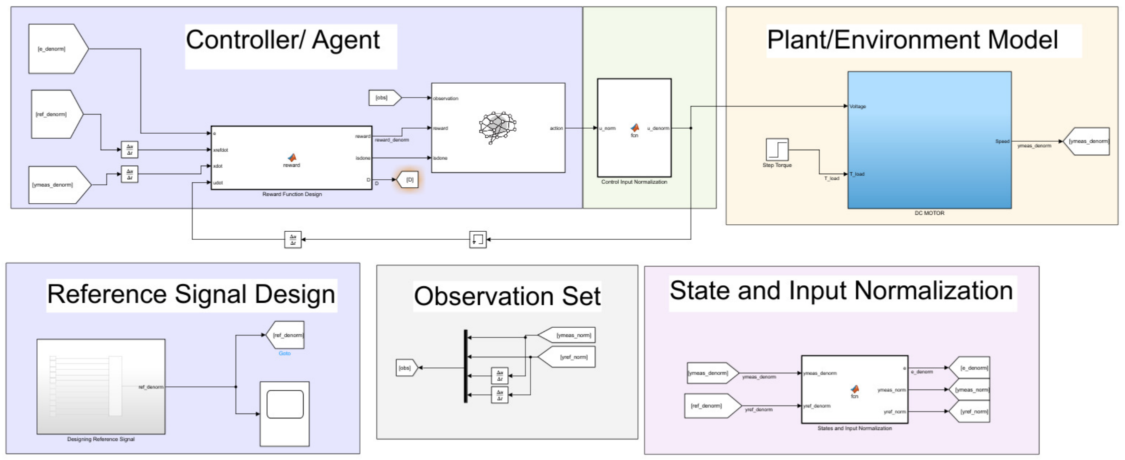

Figure 4.4: *DC Motor speed control implementation*

Notice that the control input action $u_t = a_t$ at each time step t from the controller is passed back to the reward function Matlab function via a unit delay to align them with the new state observations they cause.

4.1.8.1 RL design process

Training an RL agent presents a different set of challenges:

- Selecting the reward signal;
- Tuning many design variables;
- Long training times;

Finding a successfully reward signal is the most challenging part of the RL design. It is an iterative process which depends on multiple design parameters, and it cannot be validated quickly due to the long training times. That is one of the main reasons a absolute value or quadratic-based reward signal is most used.

The parameters tuning approach concerning critic and actor neural networks, reward function r_{t+1} as well as *simulation time* and *sampling rate* approach will be discussed below.

Training a reinforcement learning agent is lengthy process ranging approximately between 30 [min] to 2 [h] for optimal DC Motor speed control. Thus, unlike model-based design, with RL its takes time to validate if the agent can successfully could achieve the goal depending also on how much the agent will explore the main frequencies of the system and how much data it collects around the main frequencies.

It is very important to document how changing certain parameters (i.e, design variables) affects the response of the system. One way is to have a table with the various design parameters and the corresponding outcome.

These are the main design parameters that were changed in the training sessions:

- **Reward signal**: quadratic weights used in the reward function and penalization terms
- **Critic and Actor networks**: number of hidden layers, neurons, activation functions
- **Sample time**: sample time of the Simulink model which represents the environment and the dynamics to be study for both the environment and the agent.
- **Training Episodes**: the number of episodes it took for the RL algorithm to create the policy for the agent to be trained.

4.1.8.2 Hyperparameters and network architectures

The table following table shows the hypermerameters used for training the agent after several iterations of the RL design process:

Hyerparameters	
Reward Function Weights	Q=6; q_{11}=0.01

Fully Connected Actor Network	Architecture:
	Feature input Layer: 4 neurons;
	Hidden Layers + Activation Function: 3 Hidden Layers with 16 neurons each with Relu as activation function;
	Output Layer: 1 neuron corresponding to the selected action with tanh as activation function order to normalize it between -1 and 1
	
	Learning rate: 1e-3;
	L^2 Regularization Factor: 2e-4;
	Discount Factor: 0.995;
	Sample Time: 0.01;
	Mini Batch Size: 128;
	Experience Buffer Lenght: 1e6;
	Noise Variance: 1 * 0.1/$\sqrt{T_s}$ = 10% of Control Input Voltage

Fully connected Critic Network	Architecture:
	Feature Input Layer: 4 neurons (dimension of Observation Space);
	Hidden Layers + Activation Function: 4 Hidden Layers with 16 neurons each and leaky Relu with 0.5 slope as activation function. The choice of leaky Relu is done in order to take into account also negative observations.
	
	Learning rate: 1e-4;
	L^2 Regularization Factor: 1e-4;
	Discount Factor: 0.995;
	Sample Time: 0.01;
	Mini Batch Size: 128;
	Experience Buffer Lenght: 1e6;

Training Options	Max Episodes: 1000;
	Max Steps Per Episode: 600;
	Episode Duration: Sampling Time * Max Steps per Episode = 60 *sec*

The adjustment of the hyperparameters has a moderate impact on the DDPG algorithm. Even though DDPG turned out to be very sensitive to the reward functions, they were not mentioned in the original paper. Minor adjustments are typically not problematic or, in some situations, can result in faster learning. As can be seen in the table, among the several reward functions were tried, the most reliable one is the one with $Q = 6$ and $q_{11} = 0.01$ (ref. to eq. 4.7) which make the control error term e_t defined in eq. 4.2 with the highest influence while pushing the agent to track the convergence of the error to zero.

Notice also that the number of neurons comprising the feature input layer for each neural network is equal to the dimension of the observation state while the number of neurons of the output layer corresponds to control input action space. The number of hidden layers and their neurons depend on the complexity of the dynamics to be learned for the agent, in this case 4 hidden layers with 16 neurons each. In order to push the agent explore in an important way, *10 %* of the control input voltage range is selected as variance.

4.1.9 Performances

Once successful agent is trained, the policy π performance for tracking the reference speed is tested in *Simulink platform.*

The purpose of the simulation is to verify that the policy π learned by the agent is able to correctly represent the dynamics of the DC Motor respecting the performance required to track the reference speed signal.

We also wanted to verify that the agent has also learned the frequencies involved in the signal and any saturations so that he is able to adapt to tracking reference speed signals of different shapes such as *sin* or *ramp.*

The following plots show the closed-loop step agent responses:

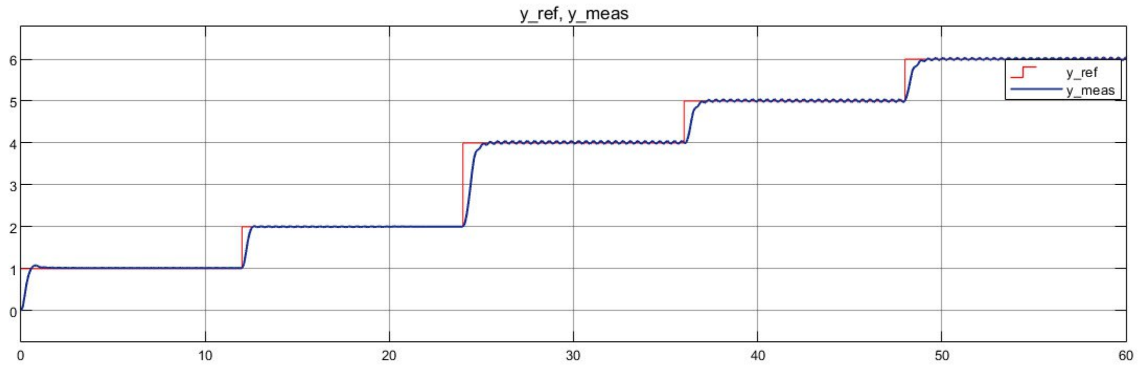

Figure 4.5: *Speed controlled trajectory w.r.t* time varying step *reference signal*

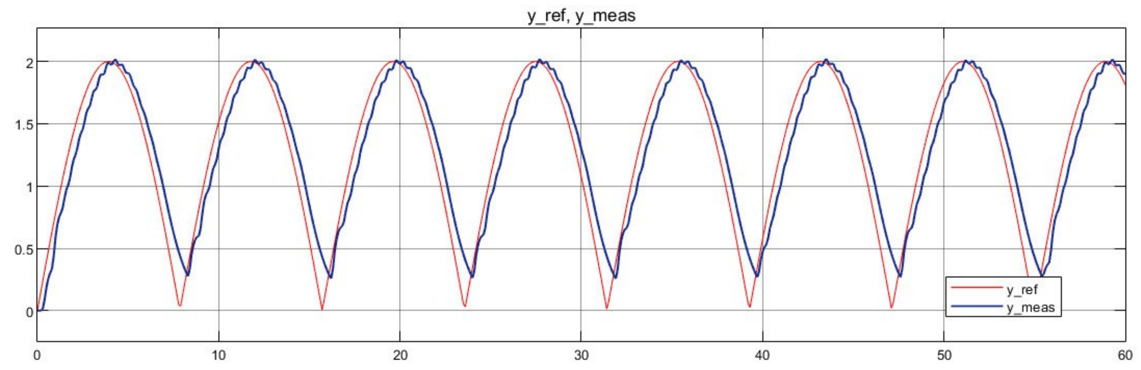

Figure 4.6: *Speed controlled trajectory w.r.t* sin wave *reference signal*

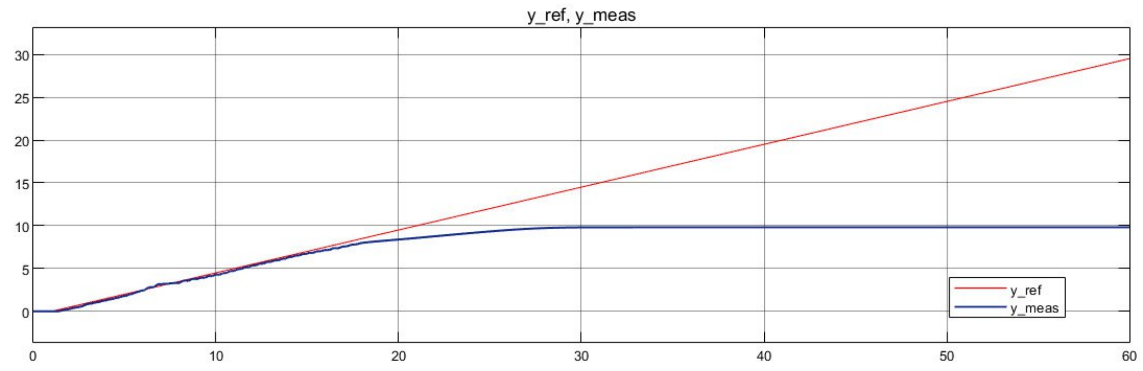

Figure 4.7: *Speed controlled trajectory w.r.t* ramp *reference signal*

The plot in figure 4.5 demonstrates that the agent was able to learn the unknown dynamics of the DC Motor and track the designed reference signal with a *rise time* $t_r = 0.3$ $[s]$ managing to rotate the motor with a the steady state error e_{ss} characterized by a*zero mean* oscillating response converging close to zero:

$$e_{ss} = \lim_{t \to +\infty} e_t = 0.0008 \; [\frac{rad}{sec}] \tag{4.9}$$

where e_t is the error defined as the difference between the reference signal and the control input defined in eq. 4.2 but considering the control input after the transient phase.

The maximum number of steps for each episode was setted to $s_{max} = 6000$, which means an episode duration of $t_{sim} = 60$ $[sec]$ in order to give the agent the time to track the entire reference speed signal:

$$t_{sim} = Ts * s_{max} \tag{4.10}$$

where $Ts = 0.01$ is the sampling time.

The agent after *581 episodes* $= 20[min]$ of training was able to converge to the optimal policy π^* and also to adapt its behaviour to track also different signals characterized with the same frequencies as the one used for train.

From the plot in Figures 4.7 and 4.8 it can be see that the agent is also able to track a *sin wave* and also a *ramp* reference signals which have different dynamics as it was trained. In particular, the agent is able to track the *ramp* signal until physical saturation while for the positive *sine wave*, the agent tracks the signal with a small time offset. Taking into account that the agent was unaware of these shape of signals during the training phase, this is an impressive result.

4.1.10 Conclusions

It has been demonstrated that the implementation of the DDPG algorithm in LQT control of a first-order linear system was successful with excellent performance and the training phase lasted relatively shortly. This is due to the not overly complicated dynamics that the agent had to learn. However, before applying the trained agent to the real system, it is recommended to finish training it on the real real system in order to allow the agent to learn the uncertain effects due to real hardware systems and compensate them with a suitable voltage.

In the next section, the implementation of the DDPG algorithm on a non-linear system will be tested in order to make the system agent learn more complex dynamics and analyze its performance with respect to the linear one.

4.2 Example: Inverted Pendulum

The balance control of a *Furuta* Inverted Pendulum based on the DDPG algorithm has been implemented.

Katsuhisa Furuta developed the Furuta pendulum at the Tokyo University of Technology in 1992. It is now thought of as one of the most well-known systems that enables the application and understanding of non-linear control theory.

The QUBE-Servo 2 Furuta Inverted Pendulum system is considered. As can be seen in Figure 4.8, a Furuta Inverted Pendulum is a system with two arms, one of which is attached to a powered motor. One degree of freedom is shared by both arms, which allows for rotation around each axis.

In spite of its seeming simplicity, the pendulum's mechanics involve a degree of complexity that necessitates the use of lengthy equations to describe its motion.

Encoders are used to measure the position of the rotary arm (i.e., the DC motor angle) and the pendulum link as well as their rate of change defining the system output:

$$y_t = y_{t_{meas}} = [\theta, \qquad \alpha, \qquad \dot{\theta} \qquad \dot{\alpha}]. \tag{4.11}$$

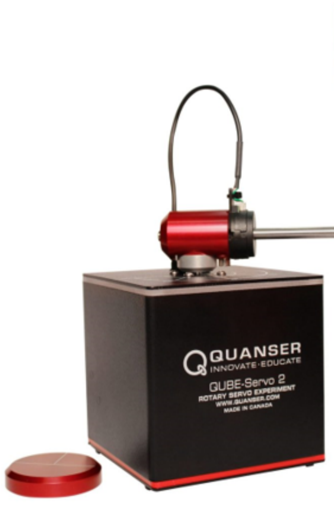

Figure 4.8: *QUBE-Servo 2 Inverted Pendulum*

4.2.1 Position control problem

The control of a pendulum has been one of fundamental problems in control field. As a control strategy to stabilize at the up-right position, it is well known that a linear quadratic technique is effective. The goal of the implementation of the model-free DDPG algorithm is aiming not only to make an agent learn its dynamics but also to stabilize it and also to swing up the pendulum in the vertical position positioned at the origin of the reference system: $\alpha_0 = 0\ [rad]$ and $\theta_0 = 0\ [rad]$

In this context, for the infinite-horizon LQT problem, the goal is to design an optimal controller for the system which ensures that the output: $y_t = y_{t_{meas}} = [\theta, \qquad \alpha, \qquad \dot{\theta}, \qquad \dot{\alpha}]$ tracks the reference position trajectory $y_{t_{ref}} = [\theta, \quad \alpha, \quad \dot{\theta} \quad \dot{\alpha}] = [0, \quad 0, \quad 0, \quad 0]$ and acts on the difference between the two defined as control error e_t:

$$e_t = y_{t_{meas}} - y_{t_{ref}} \tag{4.12}$$

4.2.2 Create Environment

The environment used to train the agent is a nonlinear dynamic model of the rotational inverted pendulum QUBE-Servo 2 system and is defined in the Simulink QUBE-Servo 2 Pendulum Model block provided by MATLAB which contains all the real dynamics equations of the Inverted Pendulum and its physical parameters of the hardware. The system consist of a motor arm, which is actuated by a DC servo motor, with a swinging pendulum arm attached to its end. This system is challenging to control because it is under-actuated, highly non linear and non minimum phase.

4.2.2.1 Plant modelization

The rotary inverted pendulum model is shown in Figure 4.9. The rotary arm pivot is actuated by a DC Motor. The arm has a length of L_r, a moment of inertia of J_r defined in eq. 4.13, and its angle, θ, increases positively when it rotates *counter-clockwise (CCW)*. In particular, both the pendulum and the arm turn in the CCW direction when the control applied voltage is positive, i.e., $V_m > 0$.

$$J_r = \frac{m_r L_r^2}{3} \tag{4.13}$$

where m_r is the total mass of the rotary arm.
The pendulum link is connected to the end of the rotary arm and it has a total length of L_p with the center of mass located at $\frac{L_p}{2}$. The moment of inertia about its center of mass is J_p defined in eq. 4.13

$$J_p = \frac{m_p L_p^2}{3} \tag{4.14}$$

where m_p is the total mass of the pendulum link. The inverted pendulum angle α, is zero when it is perfectly upright in the vertical position and increases positively when rotated CCW.

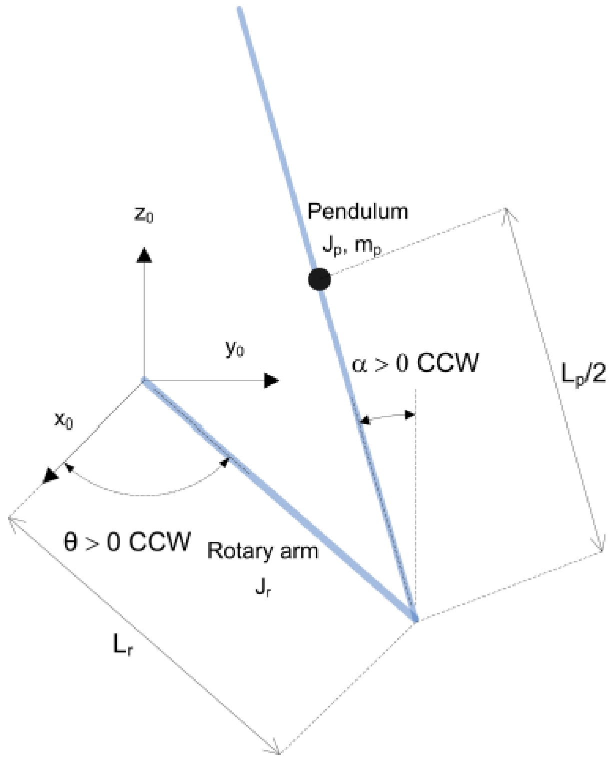

Figure 4.9: *Rotary inverted pendulum conventions*

4.2.2.1.1 Nonlinear Equations of Motion

The *Lagrange method* is used to find the equations of motion of the system which is a systematic method often used for more complicated systems such as robot manipulators with multiple joints. More specifically, the equations that describe the motions of the rotary arm and the pendulum with respect to the motor voltage will be obtained using the *Euler-Lagrange equation* described in eq. 4.15

$$\frac{d^2L}{dtd\dot{q}_i} - \frac{dL}{dq_i} = Q_i \tag{4.15}$$

where L is the *Lagrangian* of the system while Q_i are the *generalized forces.* The interested variables q_i are called *generalized coordinates.* For this system:

$$q(t)^T = [\theta(t) \quad \alpha(t)] \tag{4.16}$$

where, as shown in Figure 4.9, $\theta(t)$ is the rotary arm angle and $\alpha(t)$ is the inverted pendulum angle. Their corresponding velocities are represented in eq. 4.17

$$\dot{q}(t)^T = [\frac{d\theta(t)}{dt} \quad \frac{d\alpha(t)}{dt}] \tag{4.17}$$

Notice that the dot convention for the time derivative will be used. The time variable t will also be dropped from θ and α, i.e., $\theta = \theta(t)$ and $\alpha = \alpha(t)$.
By using the generalized coordinates expressed in 4.16, the *Euler-Lagrange* equations for the rotary

pendulum system are defined in eq. 4.18.

$$\begin{aligned} \frac{d^2L}{dtd\dot{\theta}} - \frac{dL}{d\theta} = Q_1 \\ \frac{d^2L}{dtd\dot{\alpha}} - \frac{dL}{d\alpha} = Q_2 \end{aligned} \tag{4.18}$$

where the *Lagrangian* of the system is described in eq. 4.19.

$$L = T - V \tag{4.19}$$

where T is the *total kinetic energy* of the system and V is the *total potential energy* of the system. The generalized forces Q_i are used to describe the *non-conservative forces* such as friction applied to the system with respect to the generalized coordinates. In this case, the generalized force acting on the rotary arm is defined in eq. 4.20:

$$Q_1 = \tau - B_r\dot{\theta} \tag{4.20}$$

where B_r is the viscous damping of the rotary arm which defines the viscous friction torque $-B_r\dot{\theta}$ opposing the applied torque generated τ from the input servo motor voltage, V_m.
The generalized force acting on the pendulum is defined in eq. 4.21:

$$Q_2 = -B_p\dot{\alpha} \tag{4.21}$$

Recalling that the pendulum is not directly actuated, B_p is the viscous damping coefficient of the pendulum which is opposing the pendulum rotation.
By exploiting the Euler-Lagrange's terms (see eq. 4.15) it is possible to obtain the equations of motion of the system:

$$\begin{aligned} \left(m_pL_r^2 + \frac{1}{4}m_pL_p^2\cos^2(\alpha) + J_r\right)\ddot{\theta} - \left(\frac{1}{2}m_pL_pL_r\cos(\alpha)\right)\ddot{\alpha} + \left(\frac{1}{2}m_pL_p^2\sin(\alpha)\cos(\alpha)\right)\dot{\theta}\dot{\alpha} + \\ + \left(\frac{1}{2}m_pL_pL_r\sin(\alpha)\right)\dot{\alpha}^2 = \tau - B_r\dot{\theta} \end{aligned} \tag{4.22}$$

$$-\frac{1}{2}m_pL_pL_r\cos(\alpha)\ddot{\theta} + \left(J_p + \frac{1}{4}m_pL_p^2\right)\ddot{\alpha} - \frac{1}{4}m_pL_p^2\cos(\alpha)\sin(\alpha)\dot{\theta}^2 - \frac{1}{2}m_pL_pg\sin\alpha = -B_p\dot{\alpha} \tag{4.23}$$

The relationship between the torque applied at the base of the rotary arm generated by the DC motor with respect to the control voltage applied is defined in eq. 4.24

$$\tau = \frac{k_m\left(V_m - k_m\dot{\theta}\right)}{R_m} \tag{4.24}$$

where k_m is the back-electromagnetic force, V_m is the control input voltage and R_m is the internal motor resistance.
By solving eq. 4.22 and 4.23 for the acceleration terms yields:

$$\ddot{\theta} = \frac{J_pK_1\left(-m_pLrL\cos(\alpha)\dot{\alpha}^2\right)K_2}{J_t} \tag{4.25}$$

$$\ddot{\alpha} = \frac{\left(-m_pL_rL\cos(\alpha)\right)K_1 + \left(J_r + J_p\sin^2(\alpha)\right)K_2}{J_t} \tag{4.26}$$

where $L = \frac{L_p}{2}$ while J_t, K_1 and K_2 are defined in eq. 4.27, 4.28, 4.29 respectively.

$$J_t = J_r J_p + J_p^2 \sin^2(\alpha) - m_p^2 L_r^2 L^2 \cos^2(\alpha) \tag{4.27}$$

$$K_1 = \tau - B_r\dot{\theta} + 2J_p \sin(\alpha)\cos(\alpha)\dot{\theta}\dot{\alpha} + m_p L_r L \sin(\alpha)\dot{\alpha}^2 \tag{4.28}$$

$$K_2 = -B_p\dot{\alpha} - J_p \sin(\alpha)\cos(\alpha)\dot{\alpha}^2 - m_p g L \sin(\alpha) \tag{4.29}$$

The following table shows the physical variables values:

Environment/Plant Parameters	**Variable Value**
Motor Resistance $[\Omega]$	R_m = 21.7
Damping of Motor Shaft $[\frac{Nm}{rad/s}]$	μ_m = 3.08e-6
Constant Torque $[\frac{Nm}{A}]$	K_t = 0.042
Back EMF constant $[\frac{V}{Rad/s}]$	K_b = 0.0392
Back EMF constant $[\frac{V}{Rad/s}]$	K_{b2} = 0.182
Motor Shaft Inertia $[kg * m^2]$	J_m = 4e-6
Motor Inductance $[H]$	L_m = 4.98e-3
Arm rod radius $[m]$	rod_{rad} = 0.003
Pendulum radius $[m]$	P_{rad} = 0.0045
Length of Pendulum $[m]$	L_p = 0.126
Length of Arm rod $[m]$	L_r = 0.103
Mass of Pendulum $[kg]$	m_p = 0.024
Mass of Arm rod $[kg]$	m_r = 0.095
Inertia of Pendulum $[kg * m^2]$	$J_p = m_p(\frac{p_{rad}^2}{4} + \frac{L_p^2}{12}) + m_p(\frac{L_p}{2})^2$
Inertia of Arm rod $[kg * m^2]$	$J_r = m_r(\frac{rod_{rad}^2}{4} + \frac{L_r^2}{12}) + m_r(\frac{L_r}{2})^2$
Damping of Arm rod $[\frac{Nm}{rad/s}]$	D_r = 0.001
Damping of Arm rod $[\frac{Nm}{rad/s}]$	D_{r2} = 1.88e-04
Damping of Pendulum $[\frac{Nm}{rad/s}]$	D_p = 8e-6
Gear ratio	n = 1
Gravity $[\frac{m}{s^2}]$	G = 9.81

4.2.3 Control schema

The pendulum system was then modeled in Simulink using *Simscape Electrical* and *Simscape Multibody* components.

For this system:

- $\theta \in [-\frac{\pi}{2} \quad \frac{\pi}{2}]$ is the motor arm angle and $\alpha \in [-2\pi \quad 2\pi]$ is the pendulum angle.
- The motor arm angle θ is 0 $[rad]$ when the arm is oriented horizontal and forward.
- The pendulum angle α is 0 $[rad]$ when the pendulum is oriented vertically downwards.
- Control input $u_t = a_t \in [-12\ 12]\ [V]$ is the DC voltage signal applied to the motor.

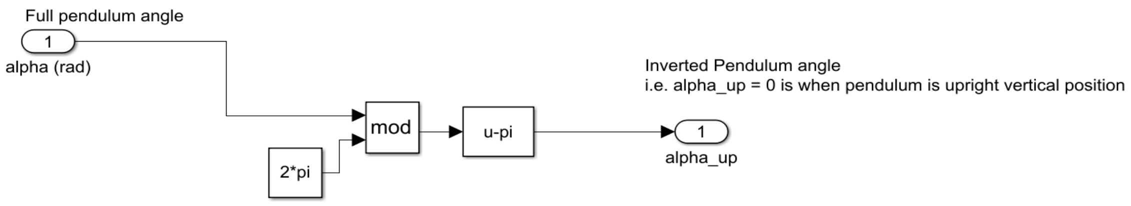

Figure 4.10: *Inverted Pendulum reference system conversion*

This technique makes the unstable vertical equilibrium point as a single value ($\alpha = 0\ [rad]$) without putting the agent in a position to understand if the direction of rotation is counterclockwise ($\phi = -\pi\ [rad]$) or clockwise ($\alpha = \pi\ [rad]$). The overall control schema is shown in Figure 4.11

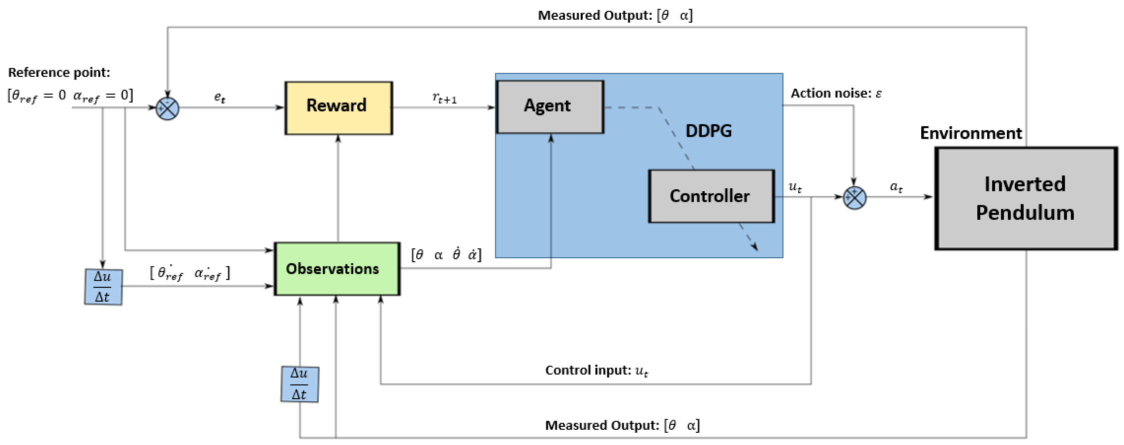

Figure 4.11: *Overall control schema*

In the Simulink model, a change of reference system was applied in order to have a 0 $[rad]$ value angle ($\alpha = 0\ [rad]$) in vertical equilibrium position in order to treat the problem as *regularization control problem* as shown in the Simulink schema in Figure 4.10.

4.2.4 Action and Observation signals

For the rotary inverted pendulum there are four continuous observation signals which are the motor angle θ, the pendulum angle α and their rate of change respectively: $S \in [\theta,\ \alpha,\ \dot{\theta},\ \dot{\alpha}]$. On the other side, even if the voltage range applied is in between -12 [V] and 12 [V], the continuous action set is in between -10 [V] and 10 [V] in order to improve the robustness in case of hardware implementation: $A \in [-10\ 10]\ [V]$.

Both the observation and action state are normalized using *min-max normalization* technique as defined in 4.1.6.

4.2.5 Reward signal

The reward signal was designed as follows:

$$r_{t+1} = -\left(q_{11}\theta_t^2 + q_{22}\alpha_t^2 + q_{33}\dot{\theta}_t^2 + q_{44}\ \dot{\alpha}_t^2 + \ r_{11}u_t^2\right) \tag{4.30}$$

The agent's purpose is to maximize the reward function. In other words, the weighted quadratic function rewards the agent when the rotary arm stays within the 0 $[rad]$ for both the angles. The terms concerning both rotational speeds and control input effort are introduced in order to penalize high oscillations and the control motor voltage does not go too high.

It was found that quadratic-based reward signals are easier to tune and increase the likelihood of training a successful policy with this system.

4.2.6 Hyperparamers and network architectures

The table shows the hypermerameters used for training the agent after tuning.

Hyerparameters	
Reward Function Weights	q_{11}=10; q_{22}=20; q_{33}=0; q_{44}=1; r=1

Fully Connected Actor Network	**Architecture**:
	Feature input Layer: 4 neurons;
	Hidden Layers + Activation Function: 3 Hidden Layers with 300 neurons each with Relu as activation function;
	Output Layer: 1 neuron corresponding to the selected action with tanh as activation function order to normalize it between -1 and 1
	
	Learning rate: 1e-4;
	L^2 Regularization Factor: 2e-4;
	Discount Factor: 0.995;
	Sample Time: 0.01;
	Mini Batch Size: 128;
	Experience Buffer Lenght: 1e6;
	Noise Variance: 1 * $0.3/\sqrt{T_s}$ = 30% of Control input

Fully connected Critic Network	**Architecture**:
	Feature input Layer: 4 neurons (dimension of Observation Space);
	Hidden Layers + Activation Function: 4 Hidden Layers with 16 neurons each and leaky Relu with 0.5 slope as activation function. The choice of leaky Relu is done in order to take into account also negative observations.
	
	Learning rate: 1e-4;
	L^2 Regularization Factor: 1e-4;
	Discount Factor: 0.995;
	Sample Time: 0.01;
	Mini Batch Size: 128;
	Experience Buffer Lenght: 1e6;

Training Options	Max Episodes: 10000;
	Max Steps Per Episode: 2000;
	Episode Duration: Sampling Time * Max Steps per Episode = 20 *sec*

Table 4.2: *Hyperparameters and network architectures*

4.2.7 Performances

Once a successful agent is trained, how well the policy balances the pendulum is tested in simulation, on the Simulink platform using the QUBE-Servo 2 block. Training a reinforcement learning agent is a lengthy process ranging approximately between 1.5 and 5 [h] for the rotary pendulum. Thus, unlike model-based design, with RL its takes time to validate if the agent can successfully balance the pendulum and assess its response. It is important to document how changing certain parameters (i.e., design variables) affects the response of the system.

The response of the rotary arm and pendulum when it starts at approximately 0 deg from the upright balance position is shown in the *Inverted Pendulum (deg)* shown in *alphadeg* plot in Figure 4.12 and the corresponding voltage applied to the motor is shown in the *Voltage Vm (V)* plot in Figure 4.13.

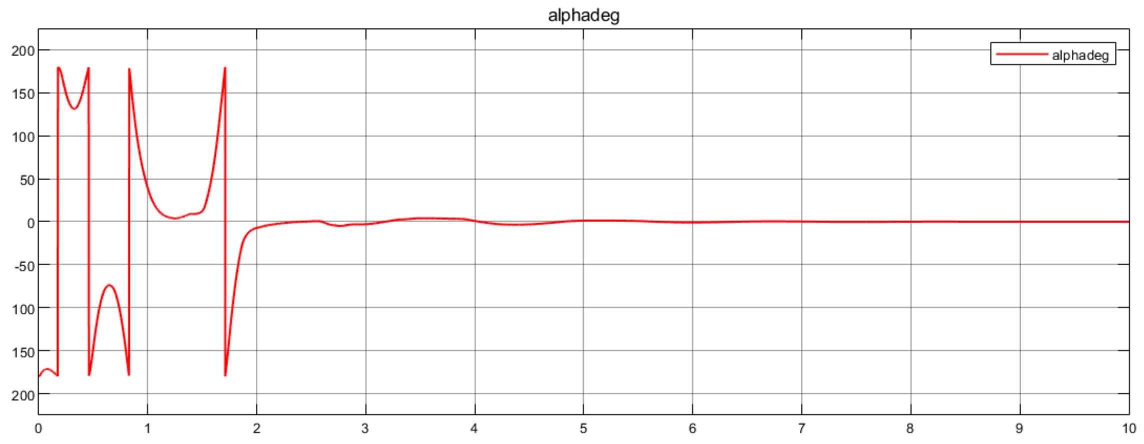

Figure 4.12: *Inverted Pendulum angle [deg]*

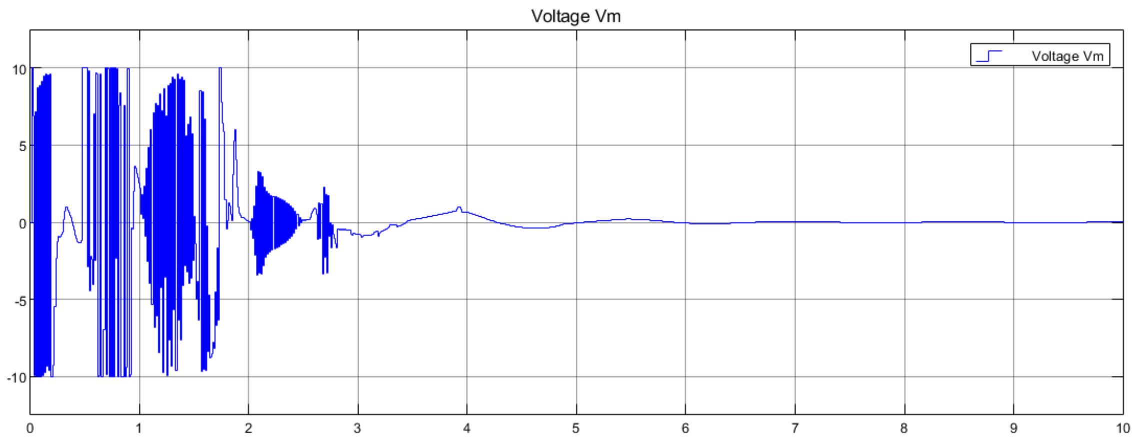

Figure 4.13: *Motor voltage [V]*

As can be noticed, the pendulum is balanced with a settling time $t_s = 2\ [sec]$. This time could be decreased by decreasing the number of maximum steps which will decrease the time available for the agent to train in a single episode. In this case, the maximum number of steps for each episode was setted to $s_{max} = 2000$ resulting in an episode duration of $t_{sim} = 20\ [sec]$:

$$t_{sim} = Ts * s_{max} \tag{4.31}$$

where $Ts = 0.01$ is the sampling time.

4.2.8 Conclusions

While some improvements can be made, this approach demonstrates that RL can successfully be used for advanced control tasks in a non linear electromechanical system and is realizable on actual hardware. There have already been a host of examples showing how RL can be used to balance and swing-up pendulum systems. While using RL for control system applications as already shown some benefit, it is still in its infancy and will become a more powerful and easy-to-use technique in the next few years.

4.3 Discussion on RL training

- Observe the agent's behavior during training, such as with scopes or other visualization blocks in the Simulink model. With this, it is possible to observe evolutions in the policy and episode reward. Is the agent getting stuck in a bad policy? Is it learning to exploit the reward in unintended ways?
- Training takes time. Agents can go through periods of better and worse performance as they try different policies. Even if your agent is not yet performing well, unless it's clearly no longer learning anything useful, let it keep training. If the the maximum number of training episodes is reached while the agent is still making progress, increase the number of episodes.
- Exploration is critically important. If an agent doesn't explore enough, it will settle on a poor policy. If the agent seems to have stopped learning, try experimenting with the exploration options to promote better exploration.
- Ultimately, agent's learning is dictated by the reward function. Check that the agent isn't learning to exploit a "loophole" in the reward, such as the robot driving into the shelves to terminate the episode early to avoid negative rewards. Try shaping the reward function to guide the agent towards desirable states. Relying only on sparse rewards (such as a bonus when a task is successfully achieved) can make training difficult because the agent may never achieve the reward through random exploration.
- If the learning rate is too low, training may take a long time. However, a learning rate that is too high may cause unstable learning. Try to use as large a learning rate as you can, but if agent's policy seems to be changing randomly without any improvement to the average reward, the actual learning rate may be too high.
- With enough neurons, a network can represent an extremely complicated function. But more neurons means more parameters, which requires more training. Start with a simple network – use the default network or copy the architecture from a similar example. But if the agent doesn't seem to be able to learn, no matter what else you try, it is used to try increasing the number of hidden-layer neurons in the networks.
- Normalization of the action and observation state in the same scale range helps the neural network to improve its performance and time from a training point of view.

In addiction, the sample efficiency of data collection process during the learning phase is still not well addressed. The convergence rate to the optimal policy as well as the time of the learning process are strongly influenced by the efficiency of the data collected by the agent during the learning phase.

Chapter 5

Complementary reward function based learning enhancement for Deep Reinforcement learning

This chapter proposes a method to generate efficient sample data which allows the agent to collect high reward trajectories more frequently, decreasing the learning phase time. The proposed method consists of Complementary reward (CR) function augmented to the traditional reward function. The CR tends to infinity when the control input leads to the system performance that meets the given requirements very accurately. Consequently, the control policy which maximizes the reward function can render the system to optimal performance. The main contributions of this study include the following aspects: (1) a new proposed Complementary reward that is augmented to the reward function which improves performance of the reinforcement learning based controller in terms of system requirements; (2) speed-up of training phase via generation of more efficient data resulting in a better learned policy.

5.1 Complementary reward function method

In RL problems, the agent learns the optimal control policy which maximizes the reward function by exploring the range of control strategies and selecting the optimal control strategy which maximizes the reward function. In this context, a new method is proposed to speed-up the control learning process and favouring the convergence to the optimal policy by generating more trajectories with high rewards. Such trajectories are generated by a CR included in the reward function which tends to infinity when the system response leads to optimal performance making the agent collect a higher reward during the learning phase. In this sense, in the case of balancing control problem defined in section **??**, the CR defined in Eq. 5.1 is proposed:

$$\mathrm{C}(s_t, e_\alpha, \dot{\alpha}) : \begin{cases} q_{55} e^{\left(\frac{1}{\gamma|\alpha ref - \alpha|}\right)} + q_{66} e^{\left(\frac{1}{\gamma|\dot{\alpha}|}\right)} & \text{if } \theta < \pm \text{ A and } \alpha < \pm \text{ B and } \dot{\alpha} < \pm \text{ D} \\ 0 \quad \text{otherwise} \end{cases} \tag{5.1}$$

where γ is the damping coefficient defining the speed divergence of the function with respect to the control error e_t, q_{55} and q_{66} are weighted parameters while A and B and D are CR conditions to be fixed based on the control problem or system requirements.

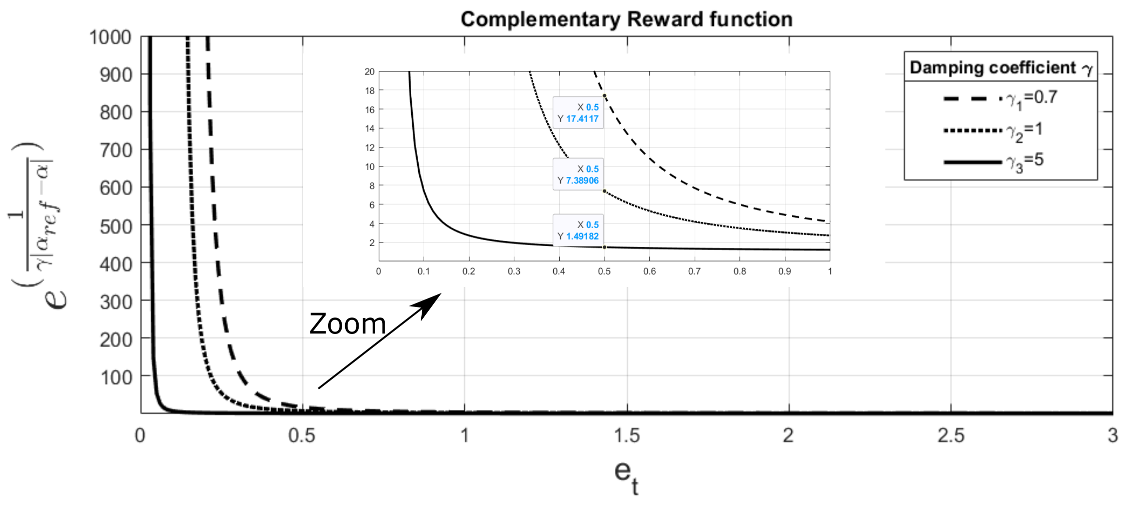

Figure 5.1: *Complementary function with different damping coefficient*

As shown in Fig. 5.1, the proposed CR is inversely proportional control error (see Eq. **??**) tending to infinity when e_α is very close to zero. The rate of convergence of the CR is dictated by the damping coefficient as shown in the zoom sub-figure. The choice of γ is very important and must be chosen according to system requirements and control problem.
The CR is included in the DDPG agent reward function defined in Section **??** as shown in Eq. 5.2:

$$R_{t+1} = r_{t+1} + C(s_t, e_\alpha, \dot{\alpha}) \tag{5.2}$$

where r_{t+1} is the traditional reward function and $C(s_t, \alpha, \dot{\alpha})$ in Eq. 5.1 respectively.
The traditional reward function becomes negligible when at least e_α is close to zero:

$$R_{t+1} = r_{t+1} + C(s_t, e_\alpha, \dot{\alpha}) \simeq C(s_t, e_\alpha, \dot{\alpha}) \quad \text{if at least } e_\alpha \simeq 0 and \dot{\alpha} \simeq 0 \tag{5.3}$$

The corresponding pseudo-algorithm concerning the proposed method implemented with DDPG algorithm is given below 3 with M, T and n being the total number of episodes, episode duration and number of samples respectively.

Algorithm 3 DDPG using Complementary reward function

```
Step 1. Initialization
Randomly initialize actor network $\mu(s_t|\theta_\mu)$ and critic network $Q(s_t, a_t|\theta_\phi)$
with weights $\theta_\mu$ and $\theta_\phi$ respectively
Copy the weights to target network $\mu^{'}$ and $Q^{'}$, $\theta_{\mu'} \leftarrow \theta_\mu$, $\theta_{\phi'} \leftarrow \theta_\phi$
Initialize replay buffer
Step 2. Exploration
for episode = [1:M]
Initialize the noise process N ~ OU for exploration
for t = [1:T]
Obtain the current system state $s_t$ from the environment
Generate control input action $a_t = \mu(s_t|\theta_\mu) + N$ based on the current actor policy
exploration action noise
Execute action $a_t$, then if: $e_\theta < A$ and $e_\alpha < B$, receive reward $R_{t+1}$(5.2) otherwise $r_{t+1}$
Obtain the resulting system state $s_{t+1}$
Store experience $(s_t,\ a_t,\ R_{t+1}\ or\ r_{t+1},\ s_{t+1})$ into the replay buffer
Step 3. Update
Extract a mini-batch of $n$ experiences $(s_t,\ a_t,\ R_{t+1}$ or $r_{t+1},\ s_{t+1})$ from replay buffer
Compute the target value: $y_t = r_{t+1} + \gamma Q(s_{t+1}, \mu(s_{t+1})|\theta_\phi)$
Update critic network by minimizing the loss $L = \frac{1}{n}\sum_i (y_i - Q(s_{i+1}, a_{i+1}|\theta_\phi))^2$
Update the actor policy by using the policy gradient
$\nabla_{\theta_\mu} J \approx \frac{1}{n}\nabla_{\theta_{\mu i}} Q(s_i, \mu(s_i|\theta_{\mu i})|\theta_\phi)$
Update the target networks
$\theta_{\phi'} \leftarrow \tau\theta_\phi + (1-\tau)\theta_{\phi'}$
$\theta_{\mu'} \leftarrow \tau\theta_\mu + (1-\tau)\theta_{\mu'}$
end
end
```

To demonstrate the effectiveness of the proposed approach while using DDPG algorithm for optimal control, a comparison between traditional method and the proposed one has been carried out. The training settings, hyper and weighted parameters, neural networks configuration remain the same. The training phase duration is set to 600 episodes whose duration is defined by 2500 number of steps each resulting in 25 seconds since the sample time is set to 0.01. The learned policy by the agent is then tested on the same simulink model that was used to train the agent for both methods with simulation time of 10 [sec]. The position of the pendulum and motor angles are always initialized in the same way at the beginning of each training episode: $\alpha_0 = \pm\pi$ [rad] and $\theta_0 = 0$ which consist on vertically down position for the pendulum and motor angle positioned at the origin of the reference frame respectively. During each episode, the agent will generate online trajectories by observing the measured output and reward coming from the environment and will subsequently use them to improve and update their policy.

5.2 Traditional reward function approach

The trained agent performance without using CR function is first analyzed. Fig. 5.2 shows the output response of the inverted pendulum angle α with respect to the unstable equilibrium point $\alpha_{ref} = 0$. As it can be noticed, the trained agent is not able to balance the pendulum in up-right position but leads to oscillations around the stable equilibrium point which is positioned at $\alpha =$ +180 [deg] as well as $\alpha =$ -180 [deg] depending on the rotation sense. On the other hand, Fig. 5.3 shows that θ output response is respecting the system requirements of keeping the motor angle in between $\pm$ 30 [deg]. These performances demonstrate that the agent learnt a sub-optimal policy and it would need to be trained for more than 600 episodes for converging to a better policy. The sub-optimal convergence can be observed in the episode reward function as shown in Fig. 5.6 where the steady state response has an offset of -500 with respect to zero episode reward which is the maximized one.

5.3 Complementary function approach

The implementation of the proposed method via CR is carried out with the same training and structure of the neural networks and a damping coefficient $\gamma = 0.7$. Fig. 5.4 shows the inverted pendulum angle α with respect to the reference position $\alpha_{ref} = 0$ [deg]. As it can be noticed, the trained agent is able to keep the inverted pendulum in up-right position in less than 1 [sec]: $\alpha \pm$ 5 [deg]. However, the trained agent is not able to balance the pendulum for more than one 1.5 [sec] since the learned policy is not optimal. Fig. 5.5 shows that θ output response is respecting the system requirements of keeping the motor angle in between $\pm$ 30 [deg]. This improvement in terms of performance and convergence to a better policy in the same amount of episodes took place thanks to the introduction of the CR which allowed the agent to collect higher rewards with respect to the traditional approach resulting in a better performance. From episode reward shown in Fig. 5.7 is possible to notice that at the beginning of the training phase the reward function response is similar to the traditional approach until the agent is within the CR conditions.

As soon as the agent collects high trajectories, he focuses on generating others with higher reward by updating the policy. Consequently, CR speed-up the convergence rate to a better policy with respect to the traditional approach.

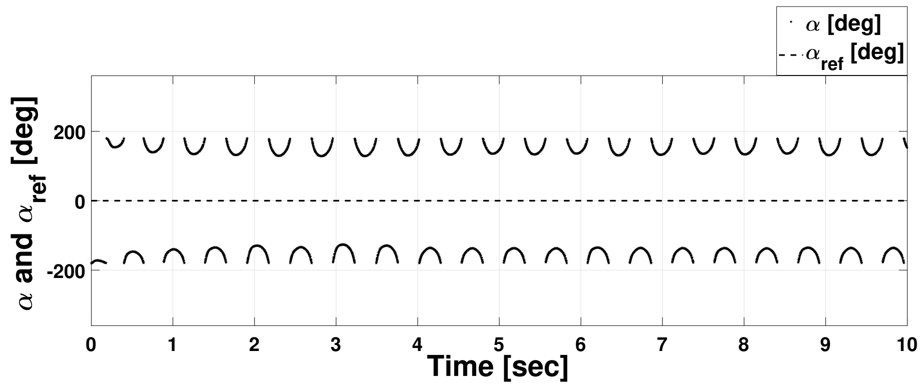

Figure 5.2: *Inverted Pendulum angle response with no CR*

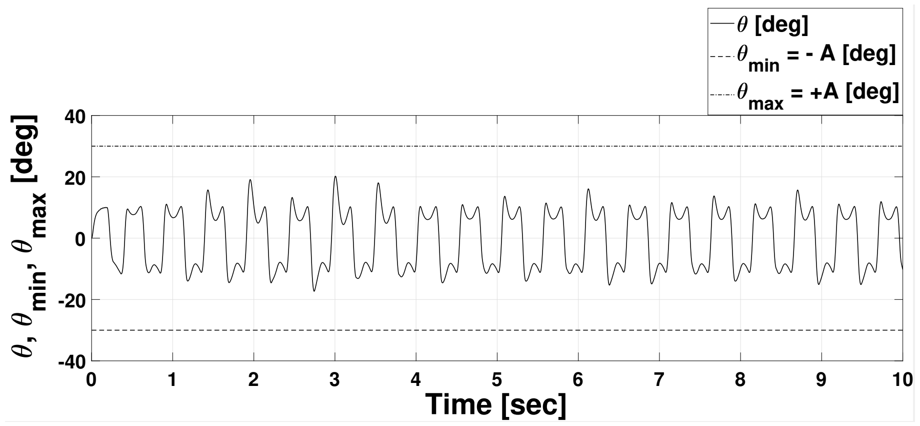

Figure 5.3: *Motor angle response with no CR*

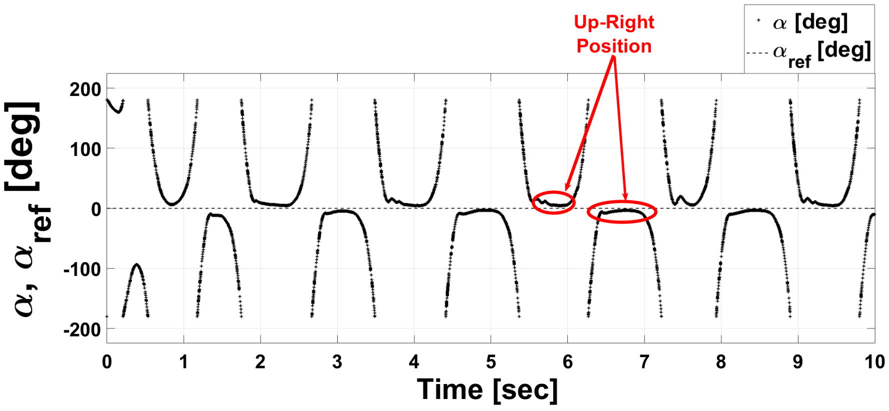

Figure 5.4: *Inverted Pendulum angle response with CR and* $\gamma = 0.7$

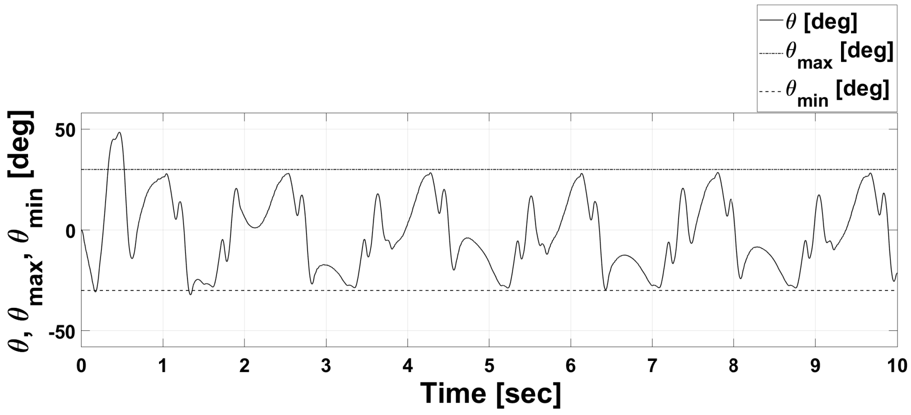

Figure 5.5: *Motor angle response with CR and* $\gamma = 0.7$

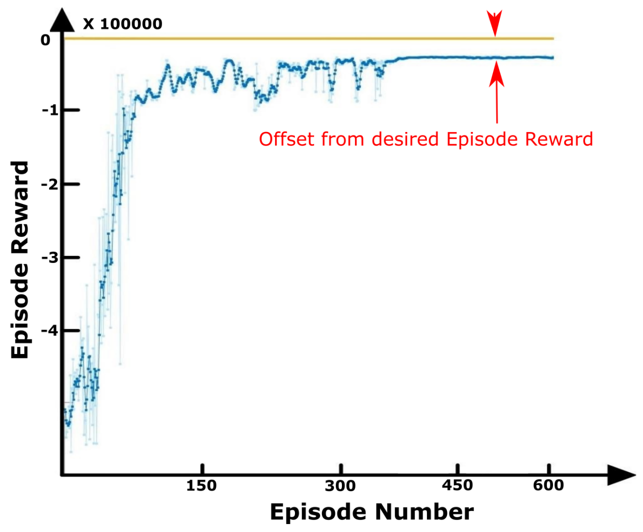

Figure 5.6: *no CR*

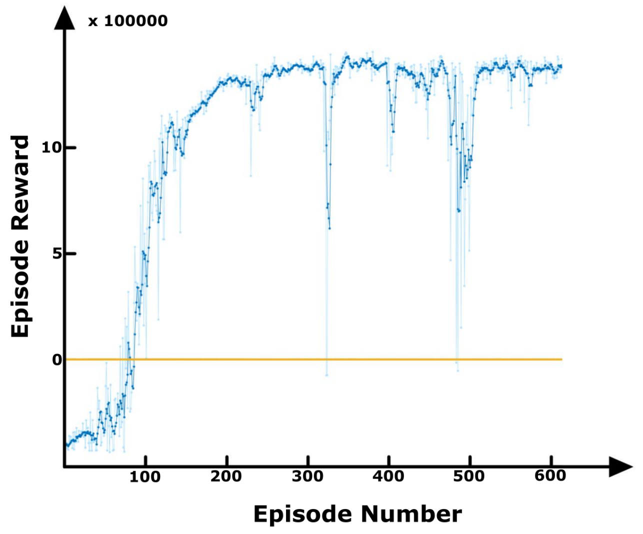

Figure 5.7: *CR with* $\gamma = 0.7$

Chapter 6

Conclusions

The DRL algorithm achieves fairly high performance in the demonstrated experiments, considering the general and generic implementation, short training time and little hyperparameter tuning. In this context, a list of tips that have been learned during numerous tests and experiments was provided in section 4.3.

The performance of the agent with a modified reward function demonstrates the power of using reward functions to design agent behaviour, and the level of generalization due to dynamic randomization is also notable.

However, there are substantial drawbacks to the DRL paradigm. In the control theory approach, robustness and stability are important aspects of any controller solution. When dealing with deep neural network approximators, there are few guarantees to be made in practice. In theory, given infinite simulation steps, the policy are guaranteed to converge, but this is not realizable when it comes to the real world. If something goes wrong in real systems, explanations and analysis are usually required, which can be considerably difficult or even impossible to obtain in black box systems. Even if DRL methods outperform the classic control theory methods, safety concerns and robustness and stability outweigh performance in most physical systems.

Sample inefficiency is one of the biggest problems in DRL. The best performing algorithms usually require millions of environment interactions to find good solutions for complex problems, and few methods exist for when the sample size is small. This is also a problem for Deep Learning in general. In order to solve this issue, the CR method was proposed. The proposed approach improves the policy learning by speeding up its convergence rate via Complementary reward function making the agent able to learn a better policy in less amount episodes thanks to its capability of rewarding the agent with high values when the control task achieves the required performance levels. The proposed approach can be implemented in any kind of RL problems since it acts at the level of reward function design which is a step inherent to all the RL algorithms. Thus, the proposed CR function concept can be viewed as a general contribution to RL based approach.

Numerical issues that might occur when the CR tends to infinity represent a mathematical constraint of this method. For this reason, the CR must be up bounded accurately based on the weighted parameter and damping coefficient so the CR function diverges to the up bound which can only be reached when the control error is very close to zero. However, the comparison between the traditional method and the proposed one demonstrates that CR function based approach can provide a better performance under similar conditions. The next step will be to tune the hyper parameters of the CR function, in particular, the damping coefficient in order to speed-up the convergence of the optimal policy in as few episodes as possible.

Moreover, purely data-driven approaches such as DRL might not be the leading solution for real-world applications yet, and they might never be, but there is no denying that these methods have achieved impressive feats. The attempt to combine the robustness and stability of control theory with the exciting performance of machine learning is definitely an interesting research area.

Bibliography

[1] Charu C. Aggarwal. *Neural Networks and Deep Learning: A Textbook*. en. Cham: Springer International Publishing, 2018. ISBN: 9783319944623 9783319944630. DOI: 10.1007/978-3-319-94463-0. URL: http://link.springer.com/10.1007/978-3-319-94463-0 (visited on 08/19/2022).

[2] Andrew G. Barto and Sridhar Mahadevan. "Recent advances in hierarchical reinforcement learning - discrete event Dynamic Systems". In: *SpringerLink* (). URL: https://link.springer.com/article/10.1023/A:1025696116075.

[3] Lucian Busoniu et al. *Reinforcement Learning and Dynamic Programming Using Function Approximators*. en. 0th ed. CRC Press, July 2017. ISBN: 9781439821091. DOI: 10.1201/9781439821091. URL: https://www.taylorfrancis.com/books/9781439821091 (visited on 08/19/2022).

[4] Pengzhan Chen et al. "Control Strategy of Speed Servo Systems Based on Deep Reinforcement Learning". en. In: *Algorithms* 11.5 (May 2018), p. 65. ISSN: 1999-4893. DOI: 10.3390/a11050065. URL: https://www.mdpi.com/1999-4893/11/5/65 (visited on 08/19/2022).

[5] Hao Dong, Zihan Ding, and Shanghang Zhang, eds. *Deep Reinforcement Learning: Fundamentals, Research and Applications*. eng. Singapore: Springer Singapore Pte. Limited, 2020. ISBN: 9789811540950 9789811540943.

[6] Vladimir Feinberg et al. *Model-Based Value Estimation for Efficient Model-Free Reinforcement Learning*. arXiv:1803.00101 [cs, stat]. Feb. 2018. DOI: 10.48550/arXiv.1803.00101. URL: http://arxiv.org/abs/1803.00101 (visited on 08/19/2022).

[7] Vincent François-Lavet et al. 2018.

[8] MinKu Kang and Kee-Eung Kim. "Analysis of Reward Functions in Deep Reinforcement Learning for Continuous State Space Control". en. In: *Journal of KIISE* 47.1 (Jan. 2020), pp. 78–87. ISSN: 2383-630X, 2383-6296. DOI: 10.5626/JOK.2020.47.1.78. URL: http://www.dbpia.co.kr/Journal/ArticleDetail/NODE09289741 (visited on 08/19/2022).

[9] Ben J.A. Kröse. "Learning from delayed rewards". In: *Robotics and Autonomous Systems* 15.4 (1995). Reinforcement Learning and Robotics, pp. 233–235. ISSN: 0921-8890. DOI: https://doi.org/10.1016/0921-8890(95)00026-C. URL: https://www.sciencedirect.com/science/article/pii/092188909500026C.

[10] Yuxi Li. *Deep Reinforcement Learning*. arXiv:1810.06339 [cs, stat]. Oct. 2018. DOI: 10.48550/arXiv.1810.06339. URL: http://arxiv.org/abs/1810.06339 (visited on 08/19/2022).

[11] Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning*. arXiv:1509.02971 [cs, stat]. July 2019. DOI: 10.48550/arXiv.1509.02971. URL: http://arxiv.org/abs/1509.02971 (visited on 08/19/2022).

[12] Long-Ji Lin. "Reinforcement Learning for Robots Using Neural Networks". PhD Thesis. Pittsburgh, PA, USA, 1992. UMI Order No. GAX93-22750.

[13] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". en. In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 0028-0836, 1476-4687. DOI: `10.1038/nature14236`. URL: `http://www.nature.com/articles/nature14236` (visited on 08/19/2022).

[14] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. arXiv:1312.5602 [cs]. Dec. 2013. DOI: `10.48550/arXiv.1312.5602`. URL: `http://arxiv.org/abs/1312.5602` (visited on 08/19/2022).

[15] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). arXiv: `1312.5602`. URL: `http://arxiv.org/abs/1312.5602`.

[16] Miguel Morales. *Grokking Deep Reinforcement Learning*. Shelter Island, New York: Manning Publications, 2020. ISBN: 9781617295454.

[17] Anusha Nagabandi et al. *Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning*. arXiv:1708.02596 [cs]. Dec. 2017. DOI: `10.48550/arXiv.1708.02596`. URL: `http://arxiv.org/abs/1708.02596` (visited on 08/19/2022).

[18] Matthias Plappert et al. *Parameter Space Noise for Exploration*. arXiv:1706.01905 [cs, stat]. Jan. 2018. DOI: `10.48550/arXiv.1706.01905`. URL: `http://arxiv.org/abs/1706.01905` (visited on 08/19/2022).

[19] Stuart J. Russell, Peter Norvig, and Ernest Davis. *Artificial intelligence: a modern approach*. 3rd ed. Prentice Hall series in artificial intelligence. Upper Saddle River: Prentice Hall, 2010. ISBN: 9780136042594.

[20] David Silver et al. "Deterministic policy gradient algorithms". In: *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*. ICML'14. Beijing, China: JMLR.org, June 2014, pp. I–387–I–395. (Visited on 08/19/2022).

[21] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. arXiv:1712.01815 [cs]. Dec. 2017. DOI: `10.48550/arXiv.1712.01815`. URL: `http://arxiv.org/abs/1712.01815` (visited on 08/19/2022).

[22] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction*. Second edition. Adaptive computation and machine learning series. Cambridge, Massachusetts: The MIT Press, 2018. ISBN: 9780262039246.

[23] Richard S. Sutton et al. "Policy gradient methods for reinforcement learning with function approximation". In: *Proceedings of the 12th International Conference on Neural Information Processing Systems*. NIPS'99. Cambridge, MA, USA: MIT Press, Nov. 1999, pp. 1057–1063. (Visited on 08/19/2022).

[24] Thanh Long Vu et al. *Barrier Function-based Safe Reinforcement Learning for Emergency Control of Power Systems*. 2021. DOI: `10.48550/ARXIV.2103.14186`. URL: `https://arxiv.org/abs/2103.14186`.

[25] Marco Wiering and Martijn van Otterlo, eds. *Reinforcement learning: state-of-the-art*. Adaptation, learning, and optimization v.12. OCLC: ocn768170254. Heidelberg ; New York: Springer, 2012. ISBN: 9783642276446 9783642276453.

[26] Ronald J. Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". en. In: *Machine Learning* 8.3-4 (May 1992), pp. 229–256. ISSN: 0885-6125, 1573-0565. DOI: `10.1007/BF00992696`. URL: `http://link.springer.com/10.1007/BF00992696` (visited on 08/19/2022).