


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. Juan ESCARENO (ENSIL-ENSCI)

September 30, 2022

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 supervisor Prof. Juan ESCARENO 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 to an inverted pendulum will be presented based on Reinforcement Learning MATLAB Toolbox.

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 functional that capture desired design objectives. One of the main principle for solving such problems is the dynamic programming (DP) principle. DP provides a sufficient condition for optimality by solving a partial differential equation, known as the Hamilton-Jacobi-Bellman equation. Classical optimal control solutions 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. Particularly, the area of AI concerned with creating computer programs that can solve problems, which requires intelligence by learning from data, and this is called Machine Learning (ML)[19]. ML is grouped in three main categories depending on the amount and quality of feedback about the system or task:

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

In *Supervised learning*, the feedback information provided to learning algorithms is a labeled training data set, and the objective is to build the system model representing the learned relation between the input, output and system parameters in order to enable the learning agent with the capability to generalize its responses to cases not included in the training set.
While in *Unsupervised learning*, no feedback information is provided to the algorithm and it uses unlabeled data as input and detects hidden patterns in the data 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 agents apply actions, also called decisions, to a system considered as environment over an extended period of time in order to

achieve the desired goal. 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. Rewards are provided in order to evaluate the one or more step decision-making performance, and the goal is to optimize the long-term performance, measured by the total reward accumulated over the course of interaction between the agent and the system. Such 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?

- Training phase is not done with a large (labeled or unlabeled) pre-collected static data-set. Rather, controller's "data" is provided to him dynamically via feedback from the real world environment in which the controller is interacting.
- Interactively decisions are made over a sequence of time-steps. In a classification problem, the controller runs inference once on the input-data to produce an output prediction. With RL, the controller runs inference repeatedly, navigating through the real-world environment as he goes.

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 this, 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 Matlab.
In particular, the thesis is structured as follows:

1. **Introduction**: presents the background and goals of the thesis

2. **Reinforcement Learning**: presents a detailed description about Reinforcement Learning state-of-the-art in order to provide the reader useful tools to enter in this research field and a comparison with traditional control from a nomenclature point of view

3. **Deep Reinforcement Learning**: presents *Deep* approach to Reinforcement Learning, providing an outline of *Deep Learning* and presenting *Deep Deterministic Policy Gradient (DDPG) algorithm*

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 in simulated environment using Reinforcement Learning Matlab Toolbox

5. **Conclusions**: presents a summary of the results obtained from experiments with some considerations

6. **Perspectives**: presents a conclusions from a possible future work point of view

Chapter 2

Reinforcement Learning

Reinforcement learning (RL) offers powerful algorithms to search for optimal controllers for both linear and non linear systems with deterministic or even stochastic dynamics that are unknown or highly uncertain. Reinforcement learning (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 the control engineer point of view. First, some general concepts of reinforcement learning with respect to control theory will be presented, and the rest will mostly focus on reinforcement learning specifically.

2.1 Reinforcement learning for optimal control applications

Broadly speaking, the goal of a control system is to determine the correct control inputs (actions) into a system that will generate the desired system behavior. Figure 2.1 shows that with feedback control systems, the controller uses measured output to improve performance and correct random disturbances and errors. Engineers use that feedback, along with a model of the plant and environment, to design the controller to meet the system requirements.

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, sequence of actions or control inputs (i.e voltage applied to a DC Motor) which will allow to control a dynamic environment to maximize some objective function. Despite the simplification, this formulation is constructed mainly to catch the attention of anyone with experience in the field of control theory, as this should seem very similar to what their own field of study attempts to accomplish. However, the two fields have different approaches to the problem of controlling dynamic systems, with control theory being mainly model-driven, and machine learning being mainly data-driven.

The schema block shown in Figure 2.1 represents a general control system block representation which can be translated to a Reinforcement Learning representation:

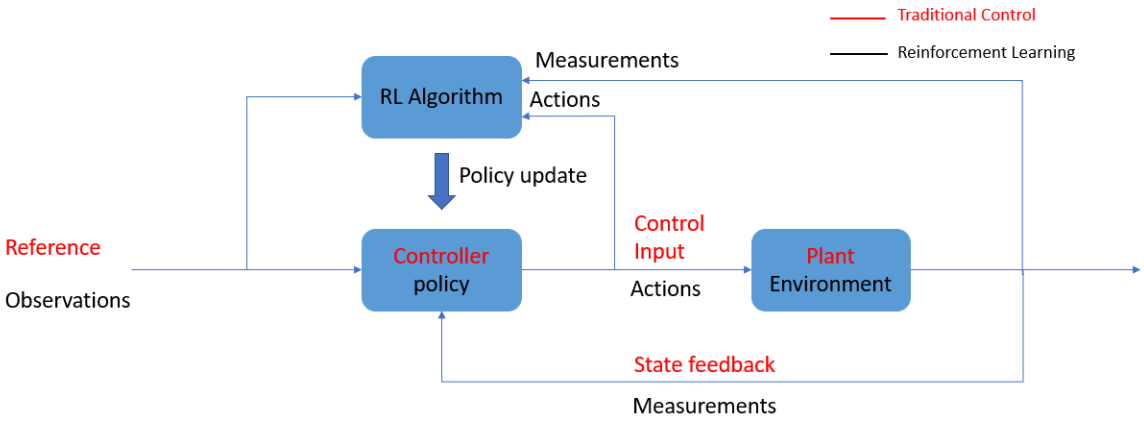

Figure 2.1: Reinforcement Learning vs Traditional Control

From RL point of view, the *environment* is everything which is not included by the controller: the plant, the reference signal, and the error computation. 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 and so on, while 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. For 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 from the web. 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.

2.2 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 and need 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.

2.2.1 Trial-error-learning

In all the approaches and algorithms that solve the RL problems, the interactions between the agent and the environment go on for several cycles because the RL agents will improve their behavior through trial-and-error learning cycles. The *environment* is represented by a set of variables related to the problem. The combination of all the possible values this set of variables can take is referred to as *state space*. A *state* is a specific set of values the variables take at any given time t.

Agents may or may not have access to the *actual* environment's *state*; however, one way or another, agents can *observe* something from the environment. The set of variables the agent perceives at any given time t is called an *observation*. The combination of all possible values these variables can take is the *observation space*. Know that 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, the environment makes available a set of *actions* 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 actions in all states is referred to as the *action space*. Figure 2.2 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* $s_t \in$ *state space* S, then selects an *action* a_t by applying the control input $\in$ *action space* A, according to a policy, which is a rule used by the agent to decide which control input to apply to the system. The policy might be *deterministic*, in which case it is usually denoted as $\pi(s_t)$, or *stochastic*, in which case it is usually denoted as $\pi(a_t|s_t)$:

$$\pi(s_t, a_t) = P[a_t|s_t] \tag{2.1}$$

where the policy π returns a probability $P[a_t|s_t]$ of taking an action in presence of stochastic environments otherwise directly the action is taken for each possible state.

Then the agent receives a new observation from the environment, which contains bot a *scalar reward* r_{t+1} defined in eq 2.2, 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.

$$r_{t+1} = \rho(x_t, u_t) \tag{2.2}$$

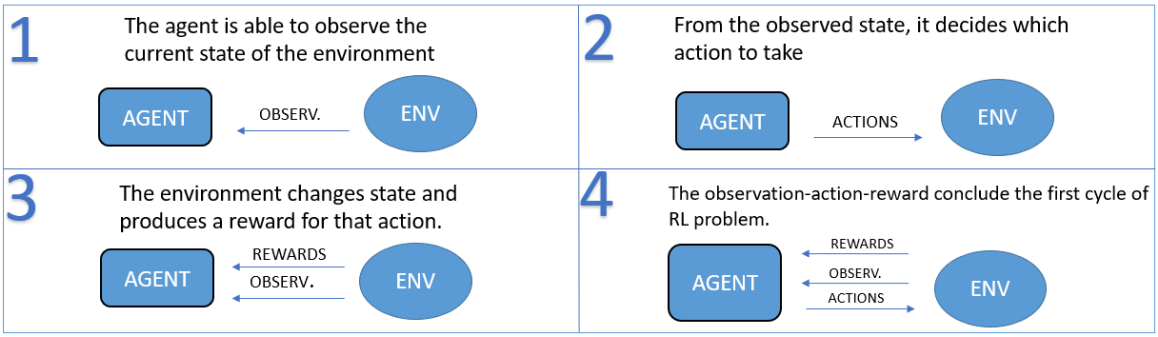

Figure 2.2: Trial-error-learning cycle

The set of the state, the action, the reward, and the new state is called *experience*:

$$e_t =< s_t, a_t, r_{t+1}, s_{t+1} > \tag{2.3}$$

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 game, 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 sacrifice immediate reward to gain greater ones later.

2.2.2 Return

The return value g_t which is defined in eq. 2.4 is the accumulated reward

$$g_t(\tau) = r_{t+1} + \gamma r_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \quad \gamma \in [0, 1] \tag{2.4}$$

where γ is the discount factor. This parameter shows a preference for immediate rewards rather than for the future ones but also mathematically is necessary in order to be sure that this quantity converge to a finite value in case of an infinite-horizon sum of rewards. The main goal of the agent is to *maximize the cumulative reward*. On the other hand, τ is a trajectory which is a sequence of states and actions in the environment defined in eq. 2.5:

$$\tau = (s_0, a_0, s_1, a_1, ...) \tag{2.5}$$

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.

2.2.3 Markov Decision Process

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

An MDP framework is made by:

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 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.
4. 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.

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, 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*. In this thesis fully observable environments will be used. Different environments and systems allow different kinds of action and a set of actions is called the *action space*. Eg, in the DC motor environment the action space could be the the voltage range that can be physically applied to the system. This example is characterized by an environment which could be with both discrete or continuous action space 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. The main mathematical quantities will be discussed in the following sections.

2.2.4 Value function

The value function $V_\pi(s_t)$ defined in eq. 2.6 is the expected (ref. to eq. 2.17) E_π return value of state s_t when following policy π. 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 in to account only the state while the value function provides a broader view of future rewards: it is a *sort of prediction of rewards*.

$$V_\pi(s_t) = E_\pi[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ...|s_t] = E_\pi[r_{t+1} + \gamma G_{t+1}|s_t] \tag{2.6}$$

In other words, the value function estimates how good a state is and it is represented by the expected value E_π under policy π and also does not takes into account actions but only the goodness to be in that particular state $V_\pi(s_t)$. 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. 2.7 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:

$$Q_\pi(s_t, a_t) = E_\pi[G_t|s_t, a_t] = E_\pi[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ...|s_t, a_t] = E_\pi[r_{t+1} + \gamma G_{t+1}|s_t, a_t] \tag{2.7}$$

In both of these value functions, $g_t(s_t)$ is the return value as defined in eq. 2.4. The optimal value function denoted as $V^*(s_t)$ and defined in eq. 2.9, is the one which yields maximum value compared to all other value function:

$$V^*(s_t) = \max_\pi V_\pi(s_t) \quad \forall s \in S \tag{2.8}$$

$$V^*(s_t) = \max_a \sum_{s_{t+1} \in S} P(s_{t+1}, r_{t+1}|s_t, a_t) + [r_{t+1} + \gamma V^*(s_{t+1})] \quad \forall s \in S \tag{2.9}$$

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. 2.10 is the one which yields the maximum value compared to all other action-value functions:

$$Q^*(s_t, a_t) = \max_\pi Q_\pi(s_t, a_t) \quad \forall s \in S, a \in A \tag{2.10}$$

$$Q^*(s_t) = \max_a \sum_{s_{t+1} \in S} P(s_{t+1}, r|s_t, a_t) + [r_{t+1} + \gamma * \max_{a_{t+1}} Q_\pi(s_{t+1}, a_{t+1})] \quad \forall s \in S, a \in A \tag{2.11}$$

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:

$$a^*(s_t) = \operatorname*{argmax}_a Q^*(s_t, a_t) \tag{2.12}$$

2.2.5 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), which 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:

$$V^*(s_t) = \max_{\pi} V_\pi(s_t) \quad \forall s \in S \tag{2.13}$$

$$V^*(s_t) = \max_{a} \sum_{s_{t+1} \in S} P(s_{t+1}, r | s_t, a_t) + [r_{t+1} + \gamma V^*(s_{t+1})] \quad \forall s \in S \tag{2.14}$$

$$Q^*(s_t, a_t) = \max_{\pi} Q_\pi(s_t, a_t) \quad \forall s \in S, a \in A \tag{2.15}$$

$$Q^*(s_t) = \max_{a} \sum_{s_{t+1} \in S} P(s_{t+1}, r_{t+1} | s_t, a_t) + [r_{t+1} + \gamma \max_{a_{t+1}} Q_\pi(s_{t+1}, a_{t+1})] \quad \forall s \in S, a \in A \tag{2.16}$$

where $P(s_{t+1})$ is the *transition state probability* which is the probability of reaching state s_{t+1} from state s_t performing action a_t while r_{t+1} is the reward function.

2.2.6 Expectation

The *expectation* defined in eq. 2.17 is used because the goal is to optimize long term future rewards and it represents the weighted sum of all possible outcomes multiplied by their probabilities (expected reward). 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:

$$E[f(x)] = \sum_{x} P(x) f(x) \tag{2.17}$$

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.

2.2.7 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.
There are some solutions such as naive explorations and the most used one is called *ϵ-greedy*, which is fairly straight-forward:

$$\text{let } \epsilon \in (0,1): \begin{cases} a^*(s_t) = \underset{a}{\text{argmax}} Q^*(s_t, a_t) & \text{with probability } (1-\epsilon) \\ \text{random action} & \text{with probability } \epsilon. \end{cases} \tag{2.18}$$

The action $a^*(s_t)$ expressed in eq. 2.18 is taken by the agent according to the best current policy π.

2.2.8 Approaches to RL

Every Agent consists of an RL algorithm that exploits to maximize the reward it receives from the environment. Each type of algorithm has its own peculiarity and understanding differences between them is useful 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.

2.2.8.1 Learning Components

The first worthy distinction among RL algorithms can be made by analyzing how the algorithms exploit the different components of the agent: indeed it is possible to explain the main strategies in RL using *policy*, *model* and *value function* defined in Section 2.2.4. One of the most crucial aspects of an RL algorithm is the question of whether the agent has access to or learns 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, does computations on them 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 2.2.4. 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.

2.2.8.2 Learning approaches

The most basic approach is to learn no politics and to use pure planning methods such as predictive model control (MPC), which is widely used in control theory. In MPC, an optimal plan with respect to the model is computed over some finite time-horizon, and then discarded in the next time step, and computed again. 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 gather new information to use from the system, while the second one progresses toward learning using limited data. Generalization becomes a critical problem in the latter approach because the agent is not able to interact anymore with the environment. 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 understand because it is relatively straightforward and it contains all the main ideas and frame bases used by most RL algorithms.

2.2.9 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. This technique provides a practical framework to solve MDP based problems by observing what is the best result achievable from them and assuming *full knowledge* about the system of the specific problem. For this reason, it applies primarily to model-based problems. Furthermore, 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 update step define the DP methods which mainly are *policy iteration* and *value iteration*.

2.2.9.1 Policy Iteration

The policy iteration aims to find the optimal policy π^* by directly manipulating the starting random policy in order to do a proper evaluation of the current policy and then by iteratively following the *policy iteration* algorithm in Figure A.1 iteration where θ is the parameter which defines the accuracy: the more the value is closer to 0, the more the evaluation would be precise. Subsequently, *policy improvement* represents the second step towards policy iteration where a new better policy than previous one is sought by changing the action to select in a specific state 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 2.7, this function returns the value of taking action a_t in the current state s_t and, after that, following the existing policy π.
If:

$$Q_{\pi'}(s_t, a_t) > Q_\pi(s_t, a_t) \tag{2.19}$$

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, reward and consequently, the new policy would be better overall. 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 which not modify the initial policy, returning the optimal policy found.

2.2.9.2 Value Iteration

The second approach used by DP to solve MDPs is *value iteration*. As it can be see in Figure A.2, value iteration algorithm is an iterative technique that alternate evaluation and improvement until it converges to the optimal policy. On the other hand, value iteration uses a modified version of policy

evaluation to determine the V-function $V(s_t)$ and then it computes the policy from it.

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 2.2.5. 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.

$$V(s_t) \leftarrow E_\pi[r_{k+1} + \gamma V_\pi(s_{t+1})] \tag{2.20}$$

2. **Policy Improvement** = improve the policy by acting greedily with respect to $V_\pi(s_t)$ of the current evaluated policy π. 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 it is 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.

$$\pi(s_t) \leftarrow \underset{a}{argmax} E_\pi[r_{t+1} + \gamma V_\pi(s_{t+1})] \tag{2.21}$$

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].

2.2.10 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).*

2.2.11 Temporal Difference Learning

Temporal Difference (TD) learning is a very central topic in reinforcement learning. TD learning methods are model-free and they 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 estimates are updated in part by using other learned estimates without waiting for the actual outcome. TD learning often refers to the *prediction problem* with an update rule for the value function given as:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{2.22}$$

where α is known as the learning rate $\in$ (0,1) while γ is the discount factor. Notice that, the agent takes into account more recent reward for $\alpha = 1$ and learns nothing when $\alpha = 0$. $r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$ defined in 2.22 is known as the *TD error*:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{2.23}$$

and arises in various forms in many areas of reinforcement learning.
The term,

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) \tag{2.24}$$

is denoted as *TD Target*.
The TD learning method for prediction is used in two different methods for doing TD control, which are discussed below:

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

2.2.11.1 SARSA learning

SARSA is an *on-policy* and *online* techniques TD learning and it gets its name from the quintuple representing a transition from one state-action pair to the next: $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$. It works by taking the principle of TD prediction and applying it in order to learn a Q-function $Q(s_t, a_t)$, instead of a V-function $V_(s_t)$. It is an on-policy method, because it estimates $Q_\pi(s_t, a_t)$ *for the current policy* π, and in simultaneously update the π greedily with respect to the estimated $Q_\pi(s_t, a_t)$. The Q-function is updated by bootstrapping (ref. to eq. 2.22 is given as:

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

and is used after every transition to a non-terminal state s_t. 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 an optimal Q-function $Q^*(s_t, a_t)$ and policy when all state-action pairs are visited an infinite number of times, and the policy converges to be purely greedy. In other words, an agent interacts with the given environment and performs policy update based on actions selected.

2.2.11.2 Q-learning

One of the early breakthroughs in reinforcement learning is the *off-policy* control algorithm *Q-learning*. It takes the Q-function $Q(s_t, a_t)$ 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. 2.26.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \underset{a}{max} Q(s_{t+1}, a_{t+1})] \tag{2.26}$$

where α is the learning rate and γ is the discount factor. 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 $u(s_t)$	1. Move one step selecting a_t from $\pi(s_t)$
2. Observe r_{t+1}, s_{t+1}	2. Observe $r_{t+1}, s_{t+1}, a_{t+1}$
3. Update the state-action function $Q(s_t, a_t)$	3. Update the state-action function $Q(s_t, a_t)$
4. (optional) Update the policy $\pi(s_t) \longleftarrow \text{argmax}_a(s_t, a_t)$	4. Update the policy $\pi(s_t) \longleftarrow \text{argmax}_a Q(s_t, a_t)$

Figure 2.3: SARSA vs Q-Learning

2.3 Conclusion: limitations of Reinforcement Learning

The algorithms explained so far work with systems characterized by well-defined and time-discrete states and actions. In this context, a state-value function or Q-value function 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. It is easy to see that there may be some non-negligible problems as the value-state function V 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 (e.g. pendulum) 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 approximation functions 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 about 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 [5] which have already achieved some extraordinary accomplishment. 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:

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

The use of *Artificial Neural Networks* as estimators reduces the training time for high dimensional systems, and it requires less space in memory. This point represents the bridge between traditional reinforcement learning 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 DeepMind with their pioneering paper [14] and the consequent [13]. Because of the nature of this work, the focus of this chapter will be on model-free algorithms. The following sections aims to explain the state-of-the-art and the leading theory behind Deep RL framework together with an overview about deep learning and the presentation of two deep actor-critic algorithms used in the experiments of 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:

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

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):

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

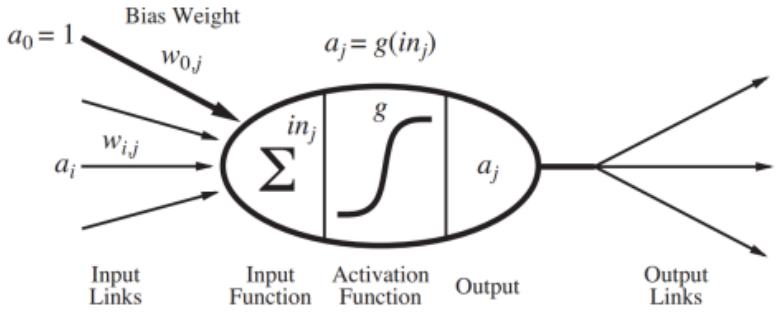

Figure 3.1: Example of a simple model for a neuron. From [19]

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 input layer, where the input data is fed to the network, and one output layers, which produces the resulting specific output based on the network's task. Networks can have any number of hidden layers, including zero and its number depends on the complexity of the function to be approximate. In a *feed forward* network, nodes in each layer are connected to all the nodes in the next layer (*fully connected layers*), see Figure 3.2.

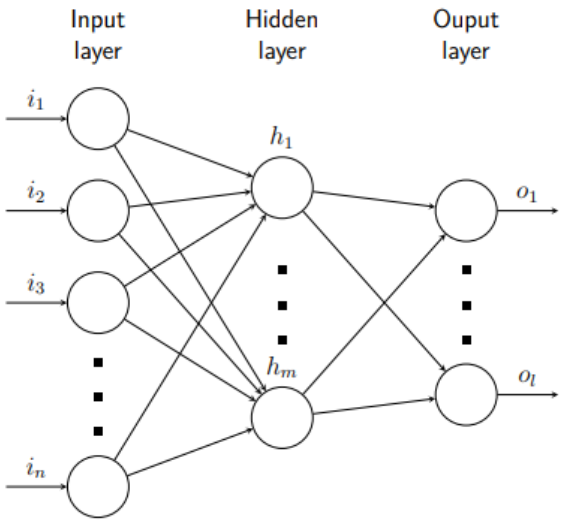

Figure 3.2: Simple example showing a deep neural network with four layers. From [19]

Each node implements a linear classifier, but the network as a whole can approximate nonlinear functions as well. Dr. Russel [19] showed that two hidden layers can represent any function, and later proved that a single hidden layer is enough to represent any continuous function, an early version of the universal approximation theorem. 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.

3.1.2 Learning process

The learning process aims to seek the set of parameters θ which results in the best possible function approximation for a specific objective. The learning process consists in many 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 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 3.5, which computes derivatives of composed function by multiplying the local derivatives:

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

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. This process will bring the function approximation process closer to the minimum at each iteration. The following equation describes the update rule based on the *gradient descent*:

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

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 3.9 and .3.9 show two examples of regularization terms. The first one is L^2*-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.

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

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

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. Equation 3.10 shows the most common activation functions:

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

where α is an hyperparameter network generally set to 0.01.

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

These elements are the basis of the learning phase of the neural networks used for estimates the interested RL function. In particular, if the neural network is used for estimating and providing directly the control input, the algorithm is called *Policy Gradient*. On the other hand, if the neural network is used for estimating and providing the Q-function in order to extract in a second step the action, the algorithm is called *value-based*.

In the following section both of them will be discussed in detail with a particular attention for the *Deep Deterministic Policy Gradient algorithm*, the one used in this thesis.

3.2 Policy Gradient methods

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:

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

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

It is possible to expand the expectation as:

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

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}.
Differentiate 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 because random samples of episodes will be taken.
By using the definition of return (2.4 is possible to obtain:

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

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. Since a log of probabilities is computed, it is easy to have noisy gradients and this could cause unstable learning or convergence to a sub-optimal policy. The flow of Gradient Policy Methods algorithm is:

1. Perform a trajectory roll-out using the current policy;
2. Store log probabilities (of policy) and reward values at each step;
3. Compute discounted cumulative future reward at each time step;
4. Compute policy gradient and update policy parameter;

5. Repeat 1-4 until the optimal policy π^* is reached.

The main advantage of policy gradient approaches consists in the stability of their convergence: these methods work updating their policy directly at each time step t instead of renewing value functions from which to derive the policy like value-based methods. 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, useful in uncertain contexts or partially observable environments. Despite the presence of the advantages just mentioned, policy gradient methods have a substantial disadvantage: they tend to converge to a local maximum instead of the global optimum. In this context, a better mix approached called *Actor-Critic method* was introduced in order to solve this disadvantage.

3.2.2 Actor-Critic architecture

Pure policy gradient methods tent to learn slowly due to estimates with high variance, and are inconvenient to implement for online problems. However, the TD methods discussed in chapter 2.2.11 can be used to oppose these problems. Actor-critic methods combine these two approaches to learn both a policy and a state value function simultaneously, and use the value function for bootstrapping. The policy is the *actor* neural network, and controls how the agent behaves and act, while the learned value function is 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 and the environment at each time step t and it is explained in details in B This idea of combining policy-based and value-based method 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 Deterministic Policy Gradient

Deep deterministic policy gradient (DDPG) is a model-free, 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* Q-function based defined in eq. 3.16 in order to learn the Q-value function thanks to which it will subsequently learn policy:

$$Q_\pi(s_t, a_t) = E_{s_{t+1}\sim E}[r_{t+1} + \gamma E_{a_{t+1}\sim\pi}[Q_\pi(s_{t+1}, a_{t+1})] \tag{3.16}$$

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:

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

Due to the fact that this expectation only depends on the environment, Q_μ can be learned *off-policy*, by using transitions generated by a different policy β. 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:

$$L(\theta_Q) = E_{s_t\rho^\beta, a_t\sim\beta, r_{t+1}\sim E}[(y_t - Q(s_t, a_t)|\theta_Q)^2], \tag{3.18}$$

where

$$y_t = r_{t+1} + \gamma Q(s_{t+1}, \mu(s_{t+1})|\theta_Q), \tag{3.19}$$

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 (see above), 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:

$$\nabla_{\theta_\mu} J \approx E_{s_t\sim\rho^\beta}[\nabla_{\theta_\mu} Q(s_t, \mu(s_t|\theta_\mu)|\theta_Q)] \tag{3.20}$$

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

3.3.1 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]. Following some policy, different experience tuples $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$ are generated, and saved in a cache. 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.

3.3.2 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. 3.19) of the loss function, which means the parameter update depends on the parameters θ which are being updated. This causes the Q 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:

$$\theta_{Q'} \leftarrow \tau\theta_Q + (1-\tau)\theta_{Q'} \tag{3.22}$$

$$\theta_{\mu'} \leftarrow \tau\theta_\mu + (1-\tau)\theta_{\mu'} \tag{3.23}$$

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.

3.3.3 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:

$$\pi(s_t) = \mu(s_t|\theta_\mu) + N \tag{3.24}$$

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.

3.4 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 drawback to this method and it need experience to know how to interpret correctly the results and how to improve them. There challenges come down to two main question

- 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 reinforcement learning 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 environment represented by *Matlab* or *Simulink*. The evaluation of the algorithms, experimentation with hyperparameter setting, and 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 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 and so on. The speed of a DC motor can be adjusted easily, by changing voltage and current depending on the type of the DC motor used.

4.1.1 Speed control problem

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

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

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

where:

- $x_t \in R^n$ is the measured state DC Motor speed $[\frac{rad}{sec}]$, $x_{max} = 10$ $[\frac{rad}{sec}]$,$x_{min} = 0$ $[\frac{rad}{sec}]$;
- $u_t \in R^m$ is the control input Voltage applied to the DC motor [V] , $u_{max} = 10$ $[V]$, $u_{min} = 0$ $[V]$;
- $y_t \in R^p$ is the output and A,B,C are dynamical matrices of the system.

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

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

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

In Figure C.1 can be see the physical setup of the DC Motor.

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 *60* $[s]$ designed in Figure 4.1.

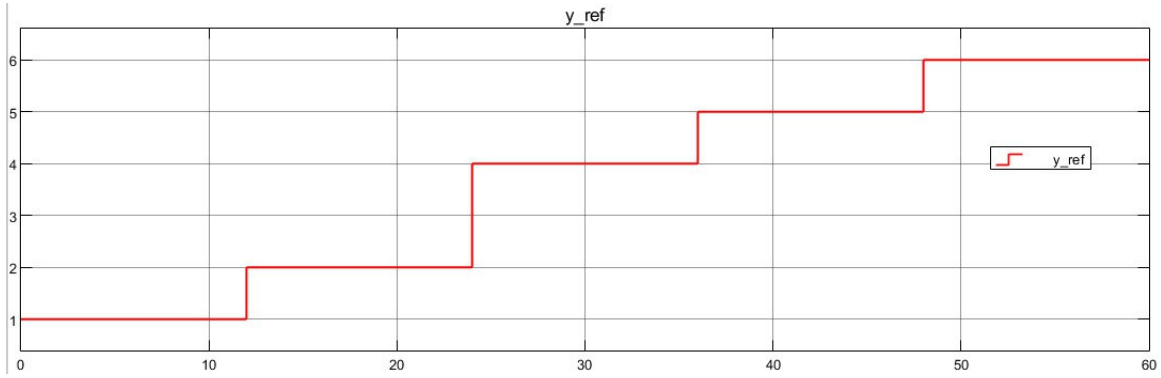

Figure 4.1: Reference speed signal

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 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.
The dynamics of the system can be modelled as:

$$J\frac{d^2\theta}{dt^2} = T - b\frac{d\theta}{dt} \rightarrow \frac{d^2\theta}{dt^2} = \frac{1}{J}(K_t i - b\frac{d\theta}{dt}) \tag{4.3}$$

$$L\frac{di}{dt} = -Ri + V - e \rightarrow \frac{di}{dt} = \frac{1}{L}(-Ri + V - K_e\frac{d\theta}{dt}) \tag{4.4}$$

The plant's dynamics will be defined as the environment of the RL problem, it is necessary to create the environment MATLAB object which will interact with the Agent by generating the instantaneous rewards signals and observations in response to agent actions.

4.1.4 Observation and Action signals

The observation signals which consist of the feedback from a control point of view is defined in such a way as to make the agent able to learn all of the frequencies covered by the various in time reference speed signal designed in the previous paragraph: $S \in [y_{k_{meas}} \quad y_{k_{ref}} \quad \dot{y_{k_{meas}}} \quad \dot{y_{k_{ref}}}]$. The action

set which is the control input range, on the other hand, is defined as continuous since the DDPG can be trained in continuous set, is defined by the action range of the voltage applied to the DC Motor: $A \in [0\ 10]V$ which will be normalized in the same scale as the observation set.

4.1.5 Reward signal

The reward signal 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 specific task. In other words, for a given observation or state, the reward measures the effectiveness of taking a particular action. During training, an agent updates its policy based on the rewards received for different state-action combinations in order to maximize the total outcome of the reward function which is the goal of the Reinforcement learning problem.
At each time step t, the reward is defined as follows:

$$r_{t+1} = r(x_t, u_t) = \ -Q|e_t| - q11\ |\dot{y_{meas}}| \tag{4.5}$$

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* normalization defined in eq. 4.6 was implemented for both observation and action spaces in order to improve the speed and the performance of the training.

$$x_{k_{norm}} = \frac{x_{k_{meas}} - x_{min}}{x_{max} - x_{min}}\ (u - l) + l \tag{4.6}$$

- $x_{k_{meas}}$ 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
- u, l are respectively the upper and lower values of the new range for the normalized data: $x_{norm} \in [l = -1; u = 1]$.

Notice that the control input action 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). This means they can be included in the reward calculation. In addiction, a white noise was designed

and added to the agent action in order to simulate the tolerance of the actuator and to improve the exploration of the Agent itself.

4.1.7 Training configuration

This section will present the settings and hyperparameters used to train the agent after learning the optimal policy π^*

4.1.7.1 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 approach used to select the correct weights is discussed below.Besides the reward signal, there are many other design variables involved in training the agent such as the *simulation time*, *sampling rate*, *number of nodes* in both critic and actor networks, defining a reset function and so on. 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.

4.1.7.2 Hyperparameters and network architectures

The table presented in Figure C.3 shows the hypermerameters used for training the agent after several iterations of the RL design process.

The DDPG algorithm is fairly sensitive to hyperparameter tuning. Small changes are generally unproblematic or can lead to faster learning in some cases and even though DDPG turned out to be highly sensitive to the reward functions, they have not been included in the original paper. 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. 4.5) making the error term with the highest influence and 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.

4.1.8 Performances

Once successful agent is trained, how well the policy is for tracking the reference speed is tested in simulation on Simulink platform.

By plotting the closed-loop step Agent response response:

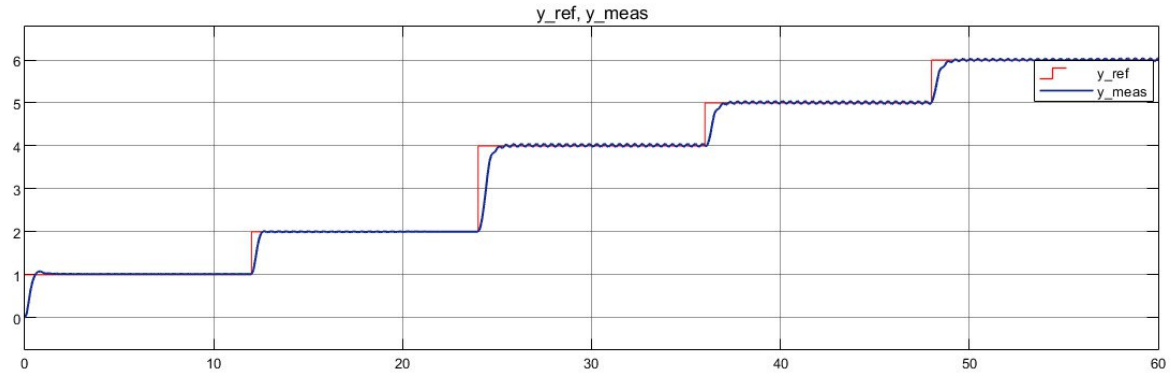

Figure 4.2: Speed controlled trajectory from *time varying step function*

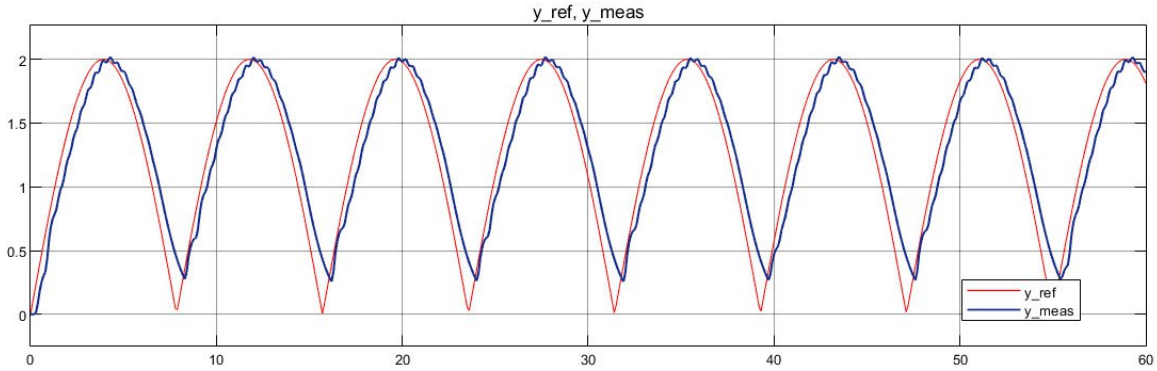

Figure 4.3: Speed controlled trajectory from *sin wave*

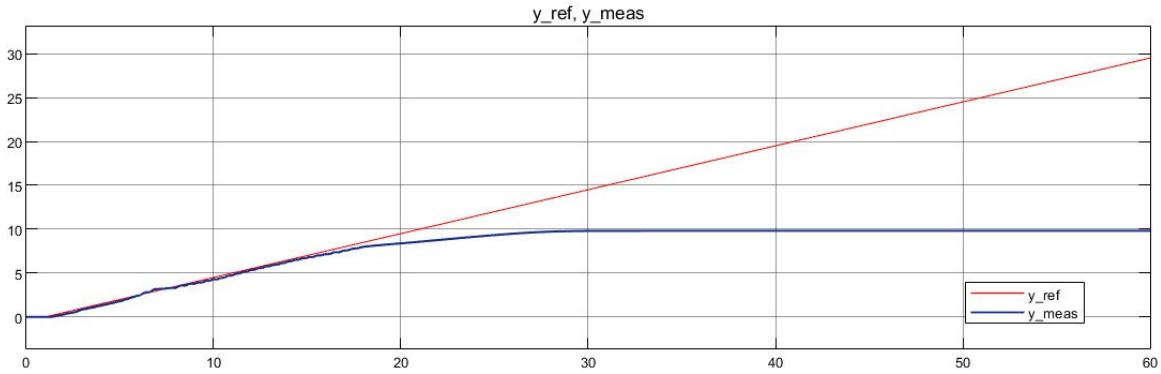

Figure 4.4: Speed controlled trajectory from Ramp

The plot in figure 4.2 shows how the agent was able to learn the unknown dynamics of the DC Motor and track the the designed reference signal with a *rise time* = 0.3 [s] managing to rotate the motor making the steady state error e_ss the with an acceptable *zero mean* oscillator behavior converging close to zero:

$$e_{ss} = \lim_{t \to +\infty} e_t = 0.0008 \tag{4.7}$$

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 6000 which means an episode duration of *60 sec* in order to give the agent the time to track the entire reference speed signal:

$$ep_t = Ts * maxsteps \tag{4.8}$$

where ep_t is the episode duration expressed in sec while $Ts = 0.01$ 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 4.4 and 4.5 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.

4.1.9 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

In this section, the implementation of the DDPG algorithm on a Inverted Pendulum system is analyzed. The agent first will train in a simulated environment and then the goal will be to apply the optimal policy and the system dynamics learned during training on the real hardware system in order to verify if the agent will be able to adapt to the dynamics of the real system and in non-simulated conditions such as air friction or with some physical variables which change in time such as mass.
The QUBE-Servo 2 Inverted Pendulum system, shown below, has two encoders to measure the position of the rotary arm (i.e., the DC motor angle) and the pendulum link and a DC motor at the base of the rotary arm.

Figure 4.5: 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.

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.
The table presented in Figure D.1 shows the physical parameters of the Inverted Pendulum.
The pendulum system was 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 is 0 [rad] when the arm is oriented horizontal and forward as shown in the diagram (counterclockwise is positive)
- The pendulum angle is 0 [rad] when the pendulum is oriented vertically downwards (counter-clockwise is positive)

- Input of the plant model is a DC voltage signal for the motor. The voltage values range from -12 to 12 V.
- The pendulum and motor angles and angular velocities are measured by sensors.

In the Simulink model, a change of reference system was applied in order to have a zero rad. value angle ($\alpha = 0[rad]$) in vertical equilibrium position in order to treat the problem as *regularization control problem*. This technique makes the unstable vertical equilibrium point as a single value ($\alpha = 0$) without putting the agent in a position to understand if the direction of rotation is counterclockwise ($\phi = -\pi[rad]$) or clockwise ($\alpha = \pi[rad]$).

4.2.3 Action and Observation signals

For the rotary inverted pendulum there are four continuous observation signals: $S \in [\theta,\ \alpha,\ \dot{\theta},\ \dot{\alpha}]$. 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 when applied to the hardware. Both the observation and action state are normalized between -1 and 1 using the following procedure and formula of the previous implementation.

4.2.4 Reward signal

The reward signal was designed as follows:

$$r_{t+1} = -\left(q_{11}\theta_t^2 + q_{22}\alpha_t^2 + q_{33}\dot{\theta}_t^2 + q_{44}\ \dot{\alpha}_t^2 + \ r_{11}u_t^2\right) \tag{4.9}$$

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 radians 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.

4.2.5 Hyperparamers and network architectures

The table presented in Figure D.2 shows the hypermerameters used for training the agent after tuning.

4.2.6 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 hours* 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* scope in Figure 4.6 and the corresponding voltage applied to the motor is shown in the Voltage Vm (V) scope in Figure 4.7. As shown in the responses below, the pendulum is balanced in approximately *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, as shown in Figure D.2 the maximum number of steps for each episode was setted to *2000*, which means an episode duration of *20 sec*:

$$ep_t = Ts * maxsteps \tag{4.10}$$

where ep_t is the episode duration expressed in sec while $Ts = 0.01$ is the sampling time.

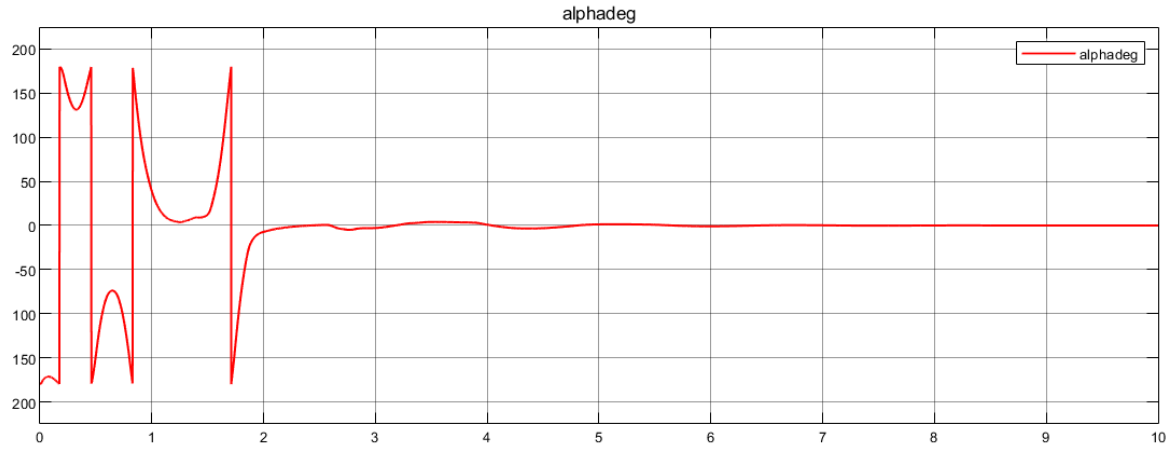

Figure 4.6: Inverted Pendulum Angle (deg)

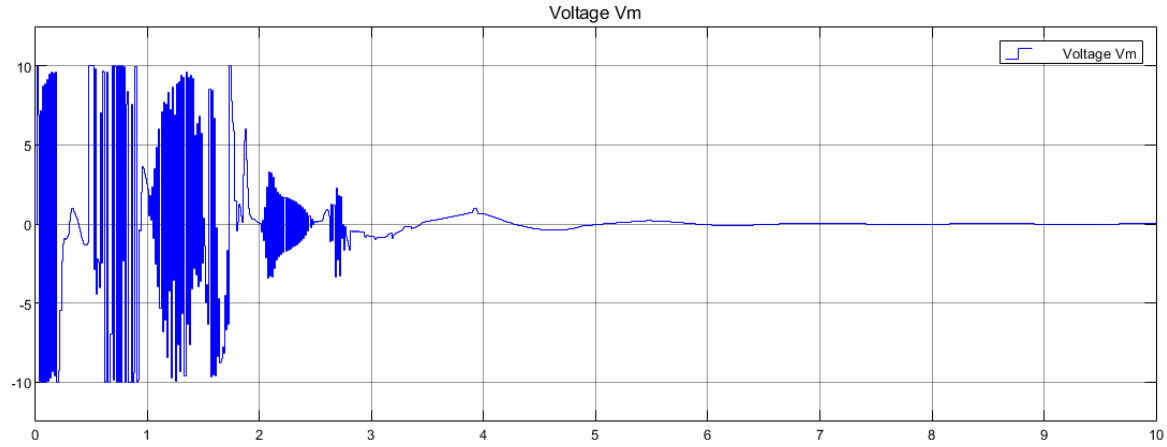

Figure 4.7: Voltage Vm

4.2.7 Conclusions

While some improvements can be made, this approach demonstrates that reinforcement learning 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 has 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.

Chapter 5

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 carried out for the implementation will be provided in appendix section (see The DRL algorithm displays an impressive flexibility in a variety of control domains. 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. 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.

Chapter 6

Perspectives

This report summarizes half of the period spent in internship at Polytech Nancy where my goal was to learn the basic concepts and main tools and algorithms of Deep and Reinforcement Learning and then learn how to implement them in dynamic systems that are more used in traditional control as a tool. The goal that we set ourselves for the end of the internship is to apply this acquired knowledge to realize the path-planning of a hexarotor composed of a cascade controller entirely using the reinforcement learning applied to a hexarotor. The cascade controller will consist of an outer loop controller with the aims of learning the dynamics present between the inertial reference system and the body frame, the so-called position control, generating in output the angles yaw, pitch, roll that will be then fed into the controller / inner loop agent to directly generate the torque of each motor at a low level to maneuver the hexarotor and bring it to its predetermined destination.

Appendix A

Reinforcement Learning

A.1 Policy Iteration

Algorithm A.1: Policy Iteration for estimating $\pi \sim \pi^*$

Input: π the policy to be evaluated; a small threshold θ which defines the accuracy of the estimation

Initialise $V(s)\ \forall s \in \mathcal{S}$ arbitrarily, except that $V(terminal) = 0$
$is\_policy\_stable \leftarrow true$
repeat
  /* Policy Evaluation */
  repeat
    $\Delta \leftarrow 0$
    for *each* $s \in \mathcal{S}$ **do**
      $v \leftarrow V(s)$
      $V(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} P(s', r|s, a)\Big[r + \gamma V(s')\Big]$
      $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
    end
  until $\Delta < \theta$
  $V_\pi \leftarrow V(s)$
  /* Policy Improvement */
  while *true* **do**
    for *each* $s \in \mathcal{S}$ **do**
      $old\_action \leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} P(s', r|s, a)\Big[r + \gamma V_\pi(s')\Big]$
      if $old\_action \neq \pi(s)$ **then**
        $is\_policy\_stable \leftarrow false$
      end
    end
  end
until $\neg\ is\_policy\_stable$

Output: V^* and π^*

Figure A.1: Policy Iteration Algorithm

where $V(s) = V(s_t)$ 2.6 and V^* is the optimal V-function defined in 2.9, $V_\pi = V_\pi$ is the V-function under the current policy π, π^* is the optimal policy to be found, S is the state space, $s^{'} = s_{t+1}$

A.2 Value Iteration

Algorithm A.2: Value Iteration, for estimating $\pi \sim \pi^*$

Input: A small threshold θ which defines the accuracy of the estimation

Initialise $V(s)\ \forall s \in \mathcal{S}$ arbitrarily, except that $V(terminal) = 0$

repeat

  $\Delta \leftarrow 0$

  for *each* $s \in \mathcal{S}$ **do**

    $v \leftarrow V(s)$

    $V(s) \leftarrow \max_a \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} P(s', r|s, a)\Big[r + \gamma V(s')\Big]$

    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$

  end

until $\Delta < \theta$

Output a deterministic policy, $\pi \sim \pi^*$, such that

$$\pi(s) = \arg\max_a \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} P(s', r|s, a)\Big[r + \gamma V(s')\Big]$$

Output: V^* and π^*

Figure A.2: Value Iteration Algorithm

Appendix B

Actor-Critic learning

1. The Actor is a network that is trying to take the best action a_t given the current state s_t which maximize his output.

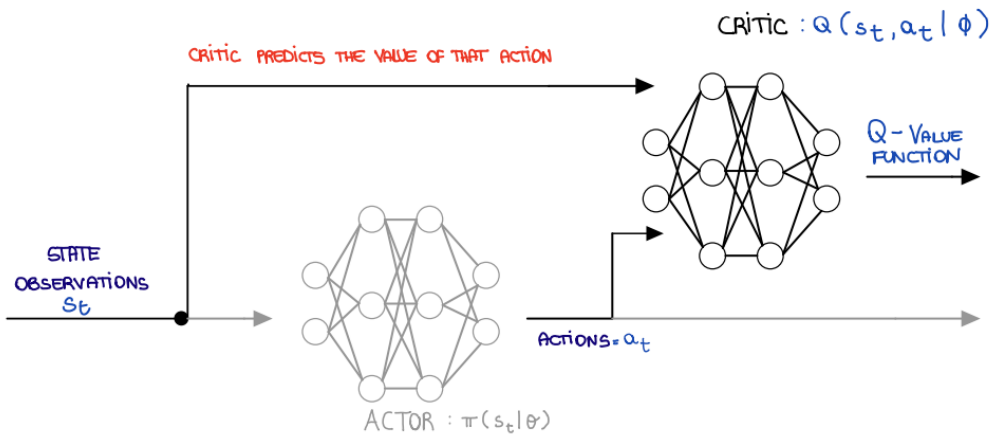

Figure B.1: Actor-Critic learning phase: Step 1

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}.

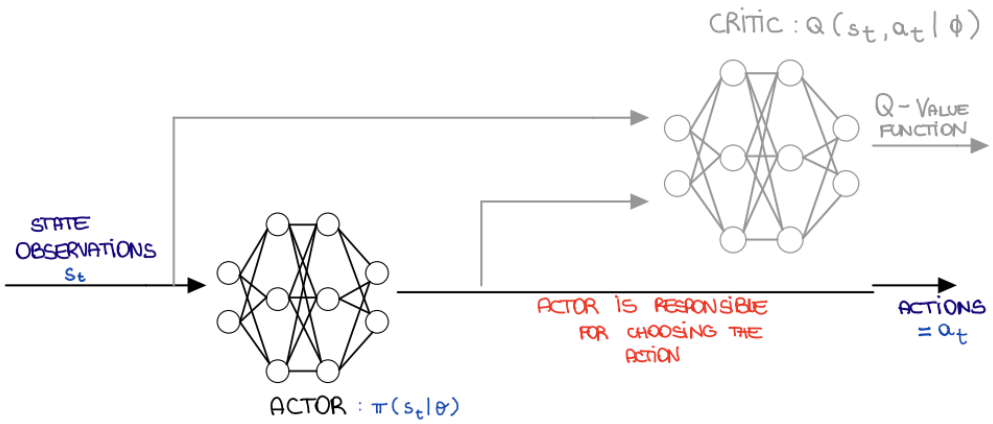

Figure B.2: Actor-Critic learning phase: Step 2

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. 2.23 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.

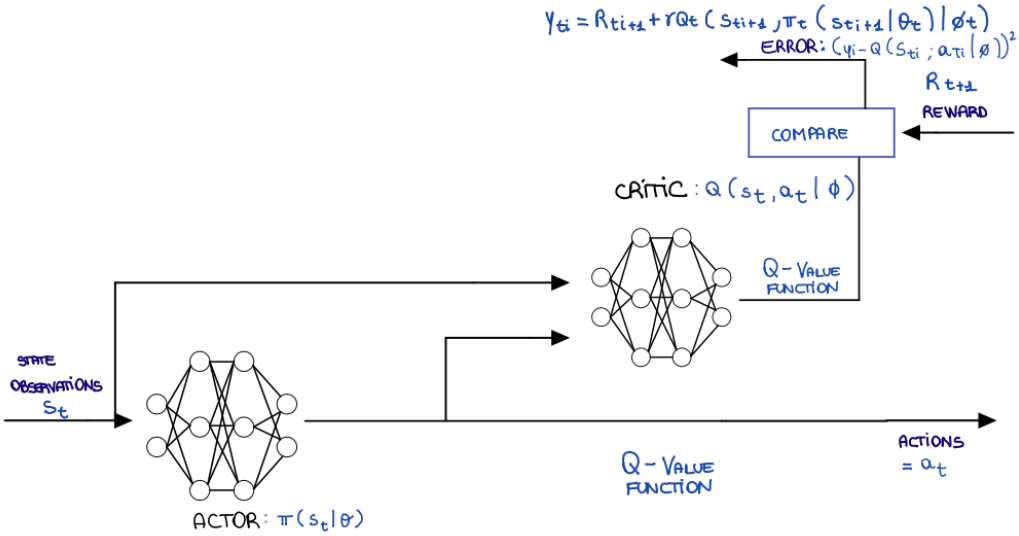

Figure B.3: Actor-Critic learning phase: Step 3

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.

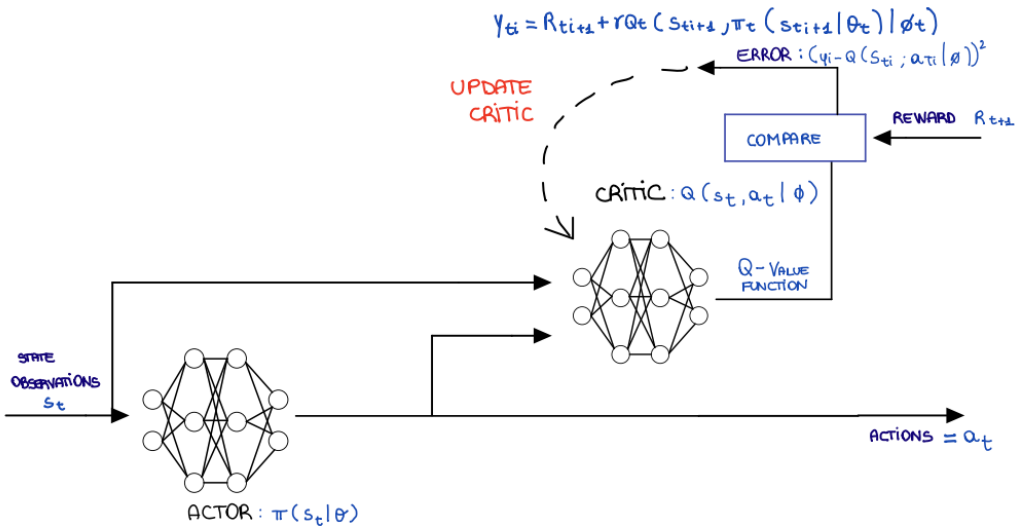

Figure B.4: Actor-Critic learning phase: Step 4

5. The Actor also updates itself with the response from the critic in order to adjust its probability of taking that action again in the future.

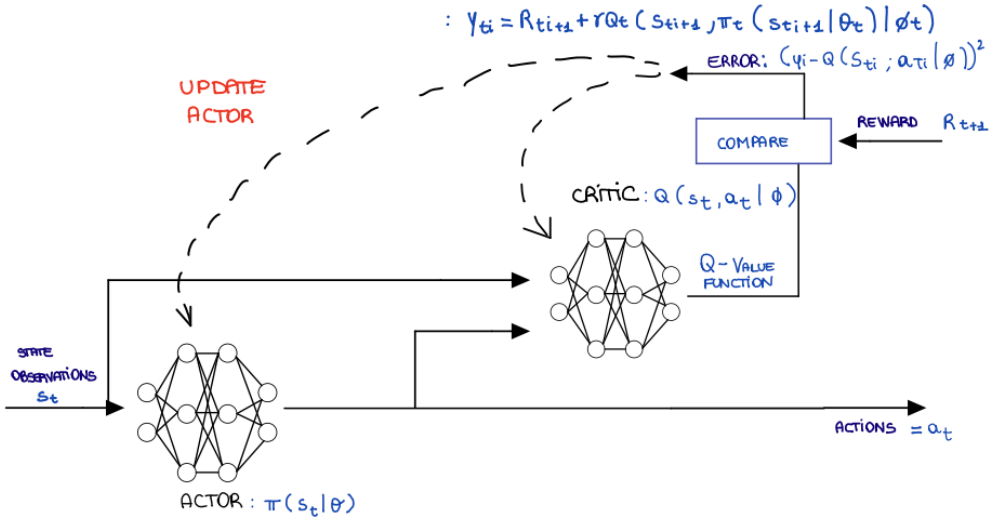

Figure B.5: Actor-Critic learning phase: Step 5

Appendix C

DC Motor

Physical setup:

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

Figure C.1: DC Motor physical setup

Plant modelization:

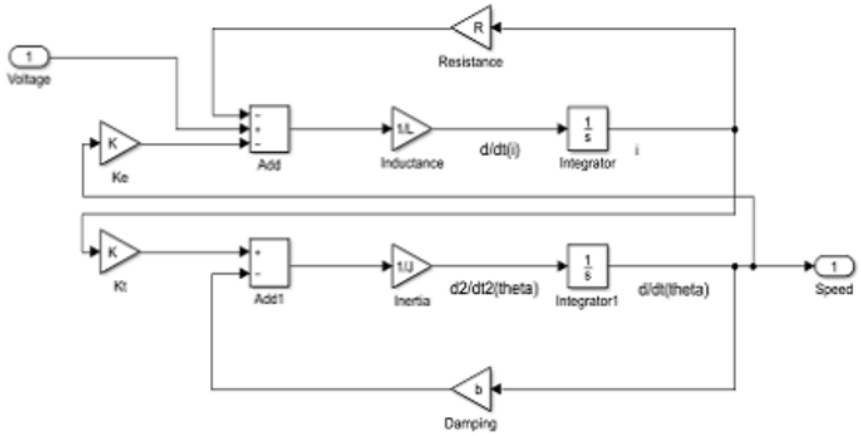

Figure C.2: DC Motor dynamics

Tuned hyperparameters:

Hyerparameters	**Value**
Reward Funtion Weights	Q = 6; q11 = 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; L2RegularizationFactor: 2e-4 Discount Factor: 0.995; Sample Time = 0.01; Mini Batch Size: 128; Experience Buffer Lenght: 1e6; Noise Variance: 1*0.1/sqrt(Ts) = 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 choise of leaky Relu is done in order to take into account also negative observations. Learning Rate: 1e-4 L2RegularizationFactor: 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]

Figure C.3: Tuned hyperparameters

Appendix D

Inverted Pendulum

Physical setup:

Environment/Plant Parameters	Variable Value
Motor Resistance [Ohm]	R_m = 21.7;
Damping of Motor Shaft [Nm/rad/s]	Mu_m = 3.08e-6;
Constant Torque [Nm/A]	K_t = 0.042;
Back EMF constant [V/rad/s]	K_b = 0.0392;
Back EMF contant [V/rad/s]	K_b_2 = 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*(p_rad^2/4+L_p^2/12) + m_p*(L_p/2)^2;
Inertia of Arm rod [kg*m^2]	J_r= m_r*(rod_rad^2/4+L_r^2/12) + m_r*(L_r/2)^2
Damping of Arm rod [Nm/rad/s]	D_r = 0.001;
Damping of Arm rod [Nm/rad/s]	D_r_2 = 1.88e-04;
Damping of Pendulum [Nm/rad/s]	D_p = 8e-6;
Gear ratio	n = 1;
Gravity [m/s^2]	G = 9.81;

Figure D.1: QUBE-Servo 2 Physical Setup

Tuned hyperparameters:

Hyerparameters	
Reward Funtion Weights	q11 = 10; q22 = 20; q33 = 0; q44 = 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; L2RegularizationFactor: 2e-4 Discount Factor: 0.995; Sample Time = 0.01; Mini Batch Size: 128; Experience Buffer Lenght: 1e6; Noise Variance: 1*0.3/sqrt(Ts) = 30% of Control
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 choise of leaky Relu is done in order to take into account also negative observations. Learning Rate: 1e-4 L2RegularizationFactor: 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]

Figure D.2: Tuned hyperparameters

Appendix E

Hardware

E.1 Implementation

The agent was interfaced to the actual QUBE-Servo 2 system hardware using the Quanser QUARC Real-Time Control software and the following Simulink model. The purpose of the hardware implementation was to verify how a pre-trained agent in the simulated environment reacts to the real variables.

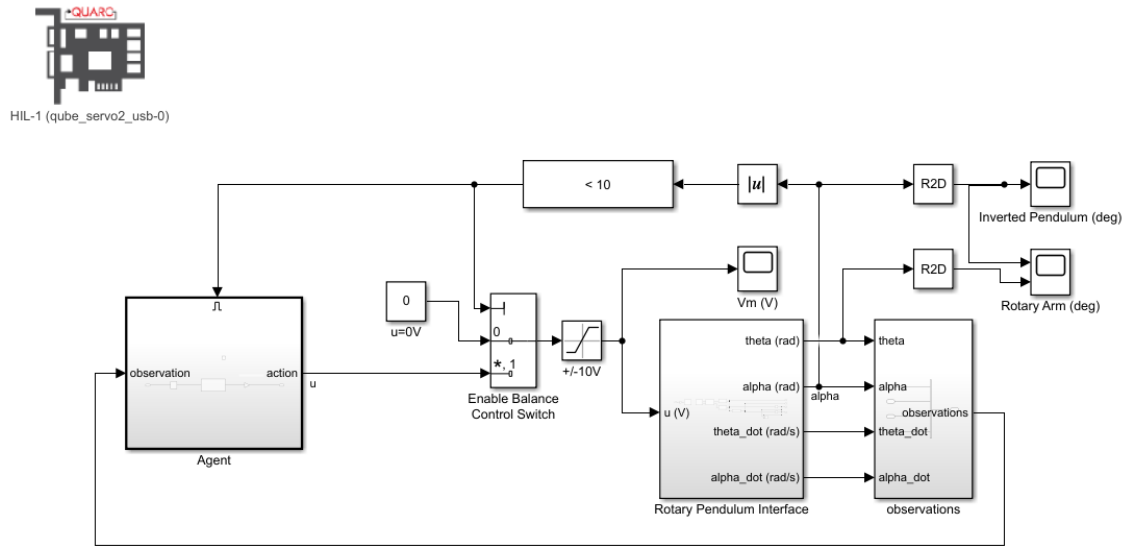

Figure E.1: Hardware Simulink block

In this application, the QUARC Real-Time Control Software creates a QUARC executable for 64-bit Windows using code generation from Simulink Coder™ and MATLAB Coder™. The QUARC executable is then run through the Simulink interface in full External mode. However since the RL Agent block does not support code generation in R2020a, the version that was used at the time of writing, we will deploy the trained agent on the 64-bit Windows target with QUARC using the geneterePolicyFuntion function from MATLAB which takes as input the trained Agent objects and creates a file containing the policy (agentData) which can be called during the online phase and hardware implementation block:

```
function action1 = evaluatePolicy(observation1)
%#codegen

% Reinforcement Learning Toolbox
% Generated on: 15-Jul-2022 15:11:41

persistent policy;
if isempty(policy)
    policy = coder.loadRLPolicy("agentData.mat");
end
% evaluate the policy
action1 = getAction(policy,observation1);
```

Figure E.2: Code generation policy

The Quanser hardware is interfaced using the blocks from the QUARC Targets library. As shown below, the QUARC HIL Write Analog applies a voltage to the DC motor and the HIL Read Encoder Timebase measures the angles of rotary arm and pendulum through the encoders.

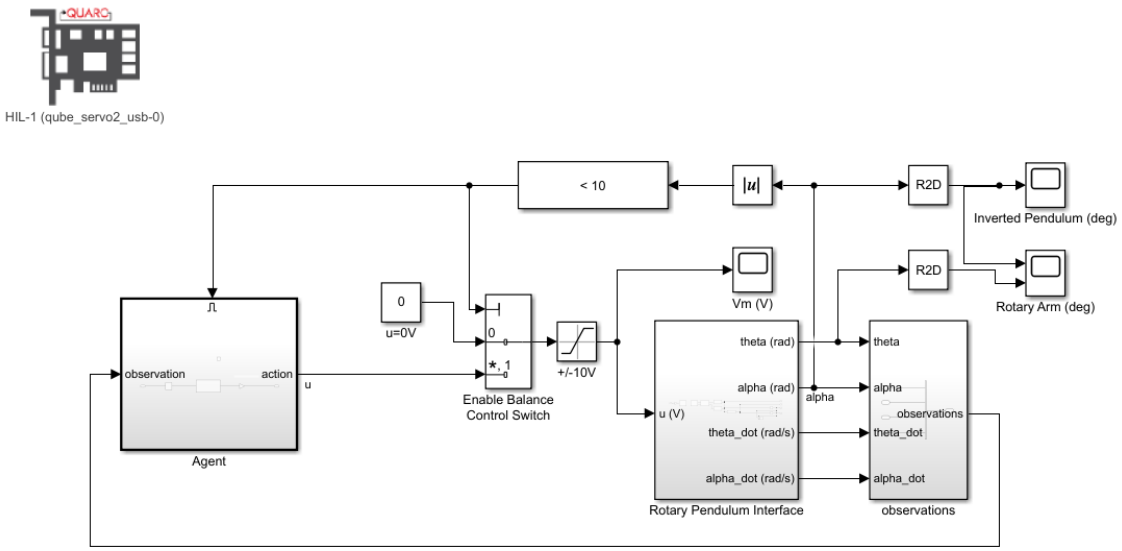

Figure E.3: HW and Simulink communication interface

E.2 Performances

The use of the trained agent in the last session on the same simulated system caused problems in terms of hardware as not taking into account the fact that the pendulum is not rigidly fixed to the base and applying a voltage of *-10 [V] or 10 [V]* to the motor, the pendulum detached from the support failing to stabilize the system in the vertical position continuously. For this reason, it was necessary to train the agent again by modifying the continuous action set from $A \in$ *[[-10 10] [V]* to *(-5 5) [V]* in order

to increase the robustness and avoid the detachment from the support for the pendulum.

The result will be presented in the QR scan put inside the poster of this thesis.

Appendix F

QUBE-Servo 2 Pendulum Model

The Peruta Inverted Pendulum's modelization using classical blocks on Simulink platform is required in order to make the user able to introduce some *varying in time* physical variables such as *mass*, *length* etc. since this procedure is not possible using *Simscape Electrical* and *Simscape Multibody* components once the physical setup of the Inverted Pendulum episode is setted. In this context, this section will provide the dynamic equations and their implementation on Simulink platform.

F.1 Background

F.1.1 Model Convention

The rotary inverted pendulum model is shown in Figure F.1. 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. F.1, 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., $V_m > 0$.

$$J_r = \frac{m_r L_r^2}{3} \tag{F.1}$$

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. F.1

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

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.

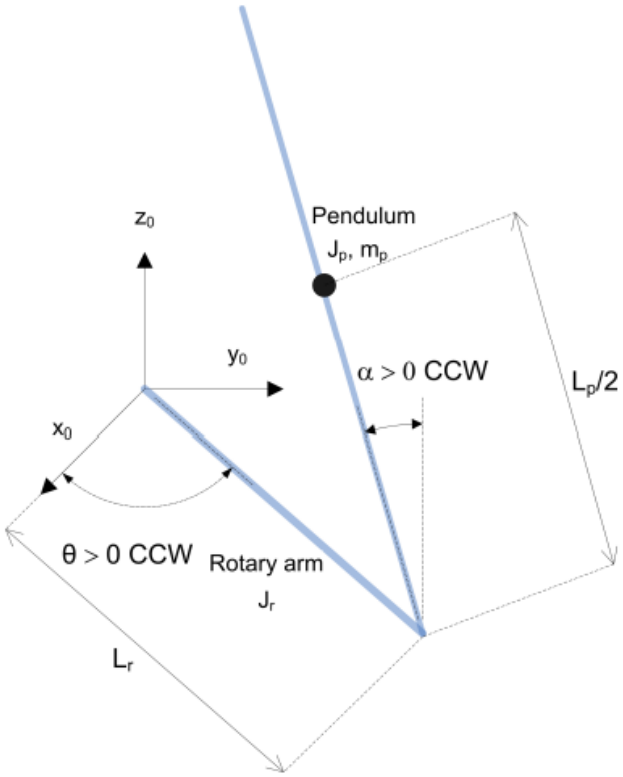

Figure F.1: Rotary inverted pendulum conventions

F.1.2 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. F.3

$$\frac{d^2L}{dtd\dot{q}_i} - \frac{dL}{dq_i} = Q_i \tag{F.3}$$

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:

$$q(t)^T = [\theta(t) \quad \alpha(t)] \tag{F.4}$$

where, as shown in Figure F.1, $\theta(t)$ is the rotary arm angle and $\alpha(t)$ is the inverted pendulum angle. Their corresponding velocities are represented in eq. F.5

$$\dot{q}(t)^T = [\frac{d\theta(t)}{dt} \quad \frac{d\alpha(t)}{dt}] \tag{F.5}$$

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 F.4, the *Euler-Lagrange* equations for the rotary pendulum system are defined in eq. F.6.

$$\begin{aligned} \frac{d^2L}{dtd\dot{\theta}} - \frac{dL}{d\theta} &= Q_1 \\ \frac{d^2L}{dtd\dot{\alpha}} - \frac{dL}{d\alpha} &= Q_2 \end{aligned} \tag{F.6}$$

where the *Lagrangian* of the system is described in eq. F.7.

$$L = T - V \tag{F.7}$$

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. F.8:

$$Q_1 = \tau - B_r\dot{\theta} \tag{F.8}$$

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. F.9:

$$Q_2 = -B_p\dot{\alpha} \tag{F.9}$$

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. F.3) it is possible to obtain the equations of motion of the system:

$$\begin{aligned} \left(m_pL_r^2 + \frac{1}{4}m_pL_p^2\cos^2(\alpha) + J_r\right)\ddot{\theta} - \left(\frac{1}{2}m_pL_pL_r\cos(\alpha)\right)\ddot{\alpha} + \left(\frac{1}{2}m_pL_p^2\sin(\alpha)\cos(\alpha)\right)\dot{\theta}\dot{\alpha} + \\ + \left(\frac{1}{2}m_pL_pL_r\sin(\alpha)\right)\dot{\alpha}^2 = \tau - B_r\dot{\theta} \end{aligned} \tag{F.10}$$

$$-\frac{1}{2}m_pL_pL_r\cos(\alpha)\ddot{\theta} + \left(J_p + \frac{1}{4}m_pL_p^2\right)\ddot{\alpha} - \frac{1}{4}m_pL_p^2\cos(\alpha)\sin(\alpha)\dot{\theta}^2 - \frac{1}{2}m_pL_pg\sin\alpha = -B_p\dot{\alpha} \tag{F.11}$$

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. F.12

$$\tau = \frac{k_m\left(V_m - k_m\dot{\theta}\right)}{R_m} \tag{F.12}$$

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. F.10 and F.11 for the acceleration terms yields:

$$\ddot{\theta} = \frac{J_p K_1 \left(-m_p L r L \cos(\alpha)\dot{\alpha}^2\right) K_2}{J_t} \tag{F.13}$$

$$\ddot{\alpha} = \frac{\left(-m_p L_r L \cos(\alpha)\right) K_1 + \left(J_r + J_p \sin^2(\alpha)\right) K_2}{J_t} \tag{F.14}$$

where $L = \frac{L_p}{2}$ while J_t, K_1 and K_2 are defined in eq. F.15, F.16, F.17 respectively.

$$J_t = J_r J_p + J_p^2 \sin^2(\alpha) - m_p^2 L_r^2 L^2 \cos^2(\alpha) \tag{F.15}$$

$$K_1 = \tau - B_r \dot{\theta} + 2 J_p \sin(\alpha)\cos(\alpha)\dot{\theta}\dot{\alpha} + m_p L_r L \sin(\alpha)\dot{\alpha}^2 \tag{F.16}$$

$$K_2 = -B_p \dot{\alpha} - J_p \sin(\alpha)\cos(\alpha)\dot{\alpha}^2 - m_p g L \sin(\alpha) \tag{F.17}$$

Appendix G

Discussion for 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.

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).