Reinforcement Learning for robot navigation in constrained environments

M. (Marta) Barbero

MSc Report

Committee:
Prof.dr.ir. S. Stramigioli
Dr.ir. J.B.C. Engelen
N. Botteghi, MSc
Dr. M. Poel

September 2018

035RAM2018
Robotics and Mechatronics
EE-Math-CS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

UNIVERSITY OF TWENTE.

MIRA CTIT
BIOMEDICAL TECHNOLOGY AND TECHNICAL MEDICINE

Abstract

Making a robot arm able to reach a target position with its end-effector in a constrained environment implies finding a trajectory from the initial configuration of the robot joints to the goal configuration, avoiding collisions with existing obstacles. A practical example of this situation is the environment in which a PIRATE robot (i.e. Pipe Inspection Robot for AuTonomous Exploration) operates. Although the manipulator is able to detect the environment and obstacles using its laser sensors (or camera), this knowledge however is only approximate. One method for a robust motion path planner in these conditions is to use a learned movement policy by applying reinforcement learning algorithms. Reinforcement leaning is an automatic learning technique which tries to determine how an agent has to select the actions to be performed, given the current state of the environment in which it is located, with the aim of maximizing a total predefined reward. Thus, this project focuses on verifying whether an agent, i.e. a planar manipulator, is able to independently learn how to navigate in a constrained environment with obstacles applying reinforcement learning techniques. The studied algorithms are SARSA and Q-learning. To achieve that objective, a MATLAB-based simulation environment and a physical setup have been implemented, and tests were performed with different configurations. After a deep analysis of the obtained results, it has been proven that both algorithms allow the agent to autonomously learn the required motion actions to be able to navigate inside constrained pipe-like environments. Even though, SARSA has been demonstrated to be a more "conservative" approach with respect to Q-learning: if there is a risk along the shortest path towards the goal (e.g. an obstacle), Q-learning will probably collide with it and then learn a policy exactly along that risky trajectory to minimize the needed actions to reach the target. On the other hand, SARSA will try to avoid this path completely, preferring a longer but safer trajectory. Once a full path has been learned, this acquired knowledge can be easily applied to a similar but not equal configuration of the pipe in a transfer learning perspective. In this way, the algorithms have been demonstrated to be able to quickly adapt to different pipes layouts and to different goal locations.

Acknowledgements

At the end of this thesis period, it is my duty to place my heartfelt thanks to the people who supported me during this important period of my life, who have helped me to grow both professionally and humanly. It is very difficult in such a few lines to remember all the people who, in different ways, have made this journey better.

A special thanks goes to my daily supervisor MSc Nicolò Botteghi. My esteem for him is due, in addition to his experience and knowledge in the field of reinforcement learning, to the great humanity with which he was able to encourage me and guide me in the right direction. The enthusiasm and commitment that I have maintained during my thesis period finds justification in the wise direction that he lavished.

A special thanks goes also to another "supervisor", dr.ir. Johan Engelen, who, with patience and critical spirit, taught, advised and helped me throughout the course of the thesis, involving me also in preparatory activities for the final presentation. I am grateful for the openness he has shown me and the enthusiasm for the research he has communicated to me.

I also thank all the remaining committee, dr. Mannes Poel and prof.dr.ir. Stefano Stramigioli, and the whole RaM department, for the professionalism in the management of the thesis period and the availability in providing me with all the necessary supervision.

I cannot miss to acknowledge in this list of thanks all those people with whom I started and spent my studies, with whom I shared memorable moments, establishing a sincere friendship and a deep collaboration.

I did not mention the rest of my friends one by one, but you know that you are all here with me. Despite the distance, you have always been present in my days, making me understand that I could do it, encouraging me to "never give up".

I do not know if I can find the right words to thank my parents, mum and dad, but I would like that my achieved goal, as far as possible, is a reward for them and for the sacrifices they made. First of all, I would like to thank them for allowing me to make this experience, far from home, despite all the difficulties and hardships that my choice entailed. Infinite thanks for always being here, for supporting me, for teaching me what is "right" and what is not. Without you I certainly would not be the person I am. Thanks for your advice, for your criticism that made me grow. Thank you for your love.

A thanks from the bottom of my heart still goes to my boyfriend Nicola, who, with his love, his patience, his words, his ability to always make me smile, was able in every moment of this path and not only to encourage me to go on, showing his proud for myself and being satisfied of my achievements, even when I was having trouble noticing them. This degree is also a bit yours!

Last but not least, I would like to thank myself for having demonstrated sufficient determination to face this situation, away from the people I love the most.

Thanks to everyone.

Marta

Ringraziamenti

A conclusione di questo lavoro di tesi, è doveroso porre i miei più sentiti ringraziamenti alle persone che mi hanno sostenuto durante questo importante periodo della mia vita, che mi hanno aiutato a crescere sia dal punto di vista professionale sia umano. È molto difficile in così poche righe ricordare tutte le persone che, in diverso modo, hanno contribuito a rendere questo percorso migliore.

Un ringraziamento speciale va al mio "daily supervisor" MSc Nicolò Botteghi. La mia stima nei suoi confronti è dovuta, oltre che alla sua esperienza e conoscenza nel campo dell'apprendimento per rinforzo, alla grande umanità con la quale ha saputo incoraggiarmi e guidarmi nella giusta direzione. L'entusiasmo e l'impegno che ho mantenuto durante il mio lavoro di tesi trovano giustificazione nella sapiente direzione da lui profusa.

Un ringraziamento particolare va anche ad un altro "supervisor", dr.ir. Johan Engelen, che con pazienza e spirito critico mi ha insegnato, consigliato e aiutato durante tutto lo svolgimento della tesi, coinvolgendomi anche in attività preparatorie alla discussione finale. Gli sono grata per la disponibilità che mi ha dimostrato e l'entusiasmo per la ricerca che mi ha trasmesso.

Ringrazio anche tutta la rimanente commisione di laurea, dr. Mannes Poel e prof.dr.ir. Stefano Stramigioli, e tutto il dipartimento RaM, per la professionalità nella gestione del percorso di tesi e la disponibilità nel fornirmi tutta la supervisione necessaria.

Non possono mancare da questo elenco di ringraziamenti tutte quelle persone con cui ho iniziato e trascorso i miei studi, con le quali ho condiviso momenti memorabili, instaurando una sincera amicizia e una profonda collaborazione.

Non cito uno ad uno il resto delle mie amiche, ma sappiate che siete tutte qui. Malgrado la distanza, siete sempre state presenti nelle mie giornate, facendomi capire che potevo farcela, incoraggiandomi a "non mollare mai".

Non so se trovo le parole giuste per ringraziare i miei genitori, mamma e papà, però vorrei che questo mio traguardo raggiunto, per quanto possibile, fosse un premio anche per loro e per i sacrifici che hanno fatto. Innanzitutto vorrei ringraziarli per avermi permesso di fare questa esperienza, lontano da casa, malgrado tutte le difficoltà e disagi che questa mia scelta ha comportato. Un infinito grazie per esserci sempre, per sostenermi, per avermi insegnato ciò che è "giusto" e ciò che non lo è. Senza di voi certamente non sarei la persona che sono. Grazie per i vostri consigli, per le vostre critiche che mi hanno fatto crescere. Grazie per il vostro amore.

Un grazie dal profondo del mio cuore va ancora al mio fidanzato Nicola, che con il suo amore, la sua pazienza, le sue parole, la sua capacità di farmi sempre sorridere, è riuscito in ogni momento di questo percorso e non solo a spronarmi ad andare avanti, dimostrandosi orgoglioso e soddisfatto dei miei traguardi anche quando io facevo fatica a notarli. In fondo questa laurea è anche un po' sua!

Infine vorrei ringraziare me stessa, per aver dimostrato sufficiente determinazione per affrontare questo percorso lontano dalle persone che più amo, con tutte le difficoltà che ne sono conseguite.

Grazie a tutti.

Marta

Contents

1 Introduction

1.1 Context and relative problem statement

Nowadays, robots are more and more autonomously performing jobs that are deemed dangerous, monotonous or unacceptable to humans. Innovative systems such as pipe inspection robots are used all over the world to get high accuracy in damage detection. There are even inspection robots capable of climbing 90 meters on a wind blade to inspect the rotor blades of the plant (1). The kilometers of underground pipe systems are not less complex. These systems must always operate reliably, therefore regular inspections are absolutely necessary to prevent damage caused by corrosion, cracks and mechanical wear. However, some points in the pipe system, which are narrow and tortuous, are often unattainable: in these cases the only solution is to rely on specific technical solutions.

Under these circumstances, learning-based navigation approaches are advantageous to make robots able to autonomously move inside (partially) unknown environments, like a pipe. One learning methodology that has been proven to be efficient in navigation tasks is reinforcement learning. Reinforcement learning is an automatic learning technique that aims at actuating systems able to learn and adapt to the changes in the environment in which they are immersed through the distribution of a "prize", called reward, which evaluates their performance. The cited approach is able to run without any previous knowledge of the dynamic model of the system itself, called agent, and without an accurate knowledge of the environment in which it is placed. Consequently, it should be appropriate for making a PIRATE robot (i.e. Pipe Inspection Robot for AuTonomous Exploration) able to autonomously learn the pipe network environment in which it should operate. The integrated hardware (e.g. laser or torque sensors, camera etc.) may detect part of the environment, but a reinforcement learning approach should allow the robot to interact with different pipes configuration in a more productive and goal-oriented way, without being affected by model inaccuracies.

Thus, summarizing, reinforcement learning algorithms will be analyzed to verify whether an automatic learning technique can be beneficial in making a robot arm able to autonomously navigate in a constrained environment with a different obstacles configurations.

1.2 Project goals and expectations

As mentioned beforehand, the primary goal of this project is to make a robot arm able to autonomously navigate in an unknown constrained environment with obstacles, e.g. a pipe network, applying reinforcement learning algorithms to learn an optimal and robust movement policy. Furthermore, the algorithms to be tested should be chosen and tuned in such a way that they can be easily adapted to different circumstances, reducing computational time required to learn new tasks and new environments.

In particular, two RL algorithms will be tested: Q-learning and SARSA, both with discretized state-space and with continuous state-space. For the discretized state-space case, the agent implements SARSA/Q-learning methodologies as proposed in the literature, i.e. creating a table to estimate the action-value function (see sections 2.1.5 and 2.1.6). On the other hand, in the continuous-state space situation, the cited table is replaced by a neural network addressed to approximate the action-value function and, consequently, able to figure out a more detailed representation of the state of the environment (see section 2.1.8). After the realization of the elements required by the reinforcement learning approach, the performances of the agents in the learning phase are evaluated. Thus, depending on the algorithm selection, the configuration of the environment and the tuning of the learning parameters, different conclusions will be drawn.

1.3 Report outline

This report presents the different reinforcement learning algorithms that have been analyzed and tested both in simulation and on the real setup. Eventually, experimental results will be discussed and assessed.

In particular, in chapter 2, reinforcement learning approach is presented together with its applications in robotics domain. Moreover, the mechanical and software configuration of the existing setup that has been taken as a reference for the actual setup (2) is described. At this point, visual-guided manipulator state estimation is investigated in order to figure out the image processing strategy that is more efficient for real-time applications. In chapter 3, the solutions proposed in chapter 2 are deeply analyzed from different points of view so that the most appropriate approach can be employed to satisfy the requirements that are presented herein. Furthermore, chapter 4 focuses on the actual design choices and correspondent implementation of the chosen strategy in terms of software and control architectures as well as setup and simulation development. According to the cited implementation, chapter 5 shows the relative results and evaluate them based on the parameters presented in section 1.2. Eventually, chapter 6 draws the conclusions about the project and the possible recommendations for future works.

2 Background

In the next chapter, fundamentals of reinforcement learning are introduced, describing its key elements, e.g. Markov decision process and the relative definition of environment as well as the different existing approaches. Moreover, in section 2.2, the manipulator proposed in (2) is described in all its components and architectures, since it is taken as a reference and modified to fit project goals. Eventually, section 2.3 focuses on vision-guided state estimation techniques that can be applied to track robot configurations.

2.1 Reinforcement Learning

2.1.1 General introduction

An explanatory definition of what reinforcement learning is has been provided in a psychology book in 1898 (3): "Applying a reward immediately after the occurrence of a response increases its probability of reoccurring, while providing punishment after the response will decrease the probability". Therefore, after defining a goal to be achieved, reinforcement learning tries to maximize the received reward of executing the action or set of actions, in order to allow reaching the cited goal. The system does not know in advance which action is best to choose, as usually happen in most of machine learning approaches, but it should discover through repeated trials which actions will allow it do get the maximum reward (4). Thus, the basic idea is to capture the most important aspects of a real problem, by interaction between a learning agent with the environment (see figure 2.1), representing the real problem, in order to achieve a target. To reach this objective, it is advisable for the agent to be able to interact with the environment and observe its state at any time. The agent must be able to perform actions that will affect the environment by changing its state. Indeed, the formulation of the learning problem is based on these three aspects: observation, action and goal. Through the observation, the agent receives some feedback signals (i.e. state and reward) from the environment thanks to the available sensors. At this point, it decides which action to take in order to maximize the expected cumulative reward.

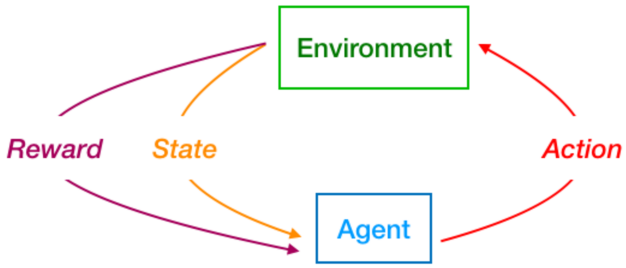

Figure 2.1: Environment-agent interaction in RL approach

Reinforcement Learning differs from other types of learning as specified in (4). For example, supervised learning is a learning-based method, which learns from pre-classified examples, provided by an external supervisor. Supervised learning is one of the most important learning methods, but by itself, it is not adequate for learning according to environment interaction, because it is usually impossible to obtain examples of desired behaviors that are both correct and representative of all possible situations in which the agent has to choose which action to take. In unexplored areas, where learning is expected to be the most important and beneficial tool, an agent must be able to learn from its own experience.

One of the main challenges that characterizes reinforcement learning and not other types of learning is the balance between the exploration of new situations and the exploitation of al-

ready learned information (4). To get a high reward, an agent must prefer the actions experienced in the past, which allowed it to produce a good reward. Therefore, to discover such actions, the agent must choose to perform actions that it has never experienced before. The agent must exploit what it already knows in order to maximize the final reward, but at the same time must explore in order to choose better actions in future executions. The dilemma is that neither the exploration nor the exploitation of experience, chosen exclusively, allow to complete the task without failing. The agent will therefore have to try a large set of actions and progressively choose the ones that have appeared more beneficial.

Another key feature of reinforcement learning is that it explicitly considers the whole problem of the interaction between agent and environment, without focusing on sub-problems (4). All RL agents are able to observe the environment and then choose which action to take to influence the environment. Moreover it is assumed from the beginning that the agent will have to operate and interact with the environment despite the considerable uncertainty in the choice of actions.

2.1.2 RL elements

In addition to the agent and the environment, four other elements can be identified which are relevant for RL algorithms analysis (4): a policy, a reward function, a value function and, if needed, a model of the environment.

The policy defines the behavior that the agent will have at a given moment during the learning phase. In general, the policy can be defined as the mapping between the observed states of the environment and the actions to be chosen when the agent is in these states. This corresponds to what in psychology is called conditioning or a set of stimulus-reaction associations (3)(4). In some cases, the policy can be a simple function or a look-up table, while in other cases it may result in more challenging computations, such as a search process. In any case, the policy represents the nucleus of the agent, in the sense that it alone is sufficient to determine its behavior.

The reward function in a reinforcement learning problem defines its objective or goal. It maps each state-action pair (or rather every action taken from a given state) with a single number, called reward, which intrinsically indicates how desirable is to undertake a certain action in a given state. The agent's goal is to maximize the total cumulative reward received over the entire period of training. The reward function defines the goodness of events for the agent. The rewards obtained in the state-action pairs represent for the agent the immediate characteristics of the problem it is facing (4). For this reason, the agent does not have to be able to alter the reward function but it can use it to alter its behavioral policy. For example, if an action selected by the policy is followed by a low reward, then the policy may change in order to choose different actions in the future in that same situation.

While the reward function indicates what is good immediately, the value function specifies what is good in the long run (4). The value of a state represents the cumulative reward that the agent can expect to get in the future, starting from the current state and following a certain policy. While the reward determines the immediate desire to achieve a state of the environment, the value-function indicates the long-term desire considering not only the state achieved in the immediate time, but also all the possible following states and the consequent rewards obtained by reaching those states. For example, a state could always be characterized by a low reward but, at the same time, it could allow to visit other states, which cannot be visited otherwise, which allow to get a high reward. Similarly for humans, a high reward represents pleasure while a low reward pain (3). Value-functions, on the other hand, represent a more refined and forward-looking judgment of how they will be satisfied or dissatisfied if the environment is in a particular state. The rewards therefore represent a primary reward while value-functions represent the prediction of the total reward. Without the rewards, value-functions would not exist,

since the only purpose of estimating value-functions is to obtain higher total rewards. However, decisions will be made according to the estimated value-functions, because the agent's goal is to maximize the total reward and not the immediate rewards. Unfortunately, determining the value-functions is more complicated than determining the rewards, since the latter are supplied to the agent directly from the environment, while the former must be estimated and re-established by the agent's observations (4). Precisely for this reason, the most important component of most reinforcement learning algorithms is the methodology which permits an effective estimation of the value-functions.

Most reinforcement learning methods are therefore structured around the estimate of the value-function, even if it is not strictly necessary to solve some RL problems. For example, meta-heuristic algorithms such as genetic and other functional optimization methods, like policy gradient methods, have been used to solve reinforcement learning problems (5),(7),(4). These methods search directly in the policy space optimizing locally around existing policies parametrized by a set of policy parameters. Consequently, they do not even consider the estimation of value-functions. This type of algorithms takes the name of evolutionary methods (or, sometimes, policy search (9),(10)) because their modalities follow biological evolution. If the policy space is sufficiently small, or it can be structured in a way to make the process of getting good policies easier, the evolutionary methods can be valid. Furthermore, evolutionary methods have advantages in problems in which the agent is not able to accurately observe the state of the environment. However, methods based on learning through interaction with the environment in many cases are more advantageous than evolutionary methods. This consideration is due to the fact that, unlike interaction-based methods, evolutionists ignore most of the formulation of the problem of RL: they do not exploit the fact that the policy they are looking for is a function that maps the observed states into actions to be taken. When the agent is able to perceive and observe the state of the environment, interaction methods allow a more efficient search. In order to make use of the advantages of both value-function based and policy-based algorithms, another type of algorithms has been implemented under the name of "actor-critic" approaches. These methods have the characteristic of separating the memory structure to make the policy independent from the value function. The block of the policy is known as actor, because it chooses actions, while the block of the estimated value-function is known as a critic, in the sense that it makes a critique of the actions performed by the policy that is being followed (4). From this explanation, it is possible to understand that this approach is a combination of the previous two methodologies.

The fourth and last element of some reinforcement learning systems, is the model of the environment (4). A model is an entity able to simulate the behavior of the environment. For example, given a state and an action, the model is able to predict the result of the next state and the next reward. Models are used for planning, where planning means any decision mode based on possible future situations, before they actually occurred.

2.1.3 Markov decision processes

As already mentioned in the previous paragraphs, the learning agent bases its decisions on the state perceived from the environment. In this section, a property of the environments and their states is defined: the Markovian property. To maintain simple mathematical formulas, the states and reward values are assumed to be finite (4). This allows to define the formulas in terms of probability sums instead of integrals and probability densities.

In the general case, the state of the environment at time step $t+1$ after executing action A_t at time step t, can be defined as a probability distribution (4):

$$P\{R_{t+1} = r, S_{t+1} = s' | S_0, A_0, R_1, ..., S_{t-1}, A_{t-1}, R_t, S_t, A_t\} \tag{2.1}$$

where R corresponds to the reward, S to the state and A to the action and R_{t+1} and S_{t+1} are jointly determined.
If the finite set of environment states satisfies Markov property, then the response given by the environment at the instant $t+1$ depends exclusively only on the state and action representing the instant t. In this case, the dynamics of the environment can be defined by the probability distribution:

$$P\{R_{t+1} = r, S_{t+1} = s' | S_t, A_t\} \tag{2.2}$$

for every r, s', S_t, A_t. In other words, an environment satisfies Markov property if and only if equation 2.1 is equivalent to equation 2.2 for each r, s' and every possible sequence of past events $S_0, A_0, R_1, ..., S_{t-1}, A_{t-1}, R_t, S_t, A_t$.
If an environment satisfies Markov property, then, thanks to equation 2.2, it is possible to predict the next state and the next reward by knowing only the state-action pair of the previous instant. For this reason, it is possible to consider equation 2.2 as the basis for the choice of actions to be taken. In short, the best policy that chooses the action following Markov property is as good as the best policy that chooses the actions considering the whole sequence of past events.

Even when the environment does not satisfies Markov property, it is still appropriate to think of the state in reinforcement learning as an approximation of a Markov state. Markov property is important in reinforcement learning because all the methods related to learning by reinforcement base their choices by assuming that the values provided by the environment are functions only of the state and the action taken at a previous instant. A reinforcement learning system that satisfies Markov property is called Markov Decision Process (MDP). If the set of states and the set of actions are finite, it is called finite Markov Decision Process (finite MDP). Finite MDPs are very important for the theoretical definition of reinforcement learning. A finite MDP is defined by a set of states and actions and the single-step dynamics of the environment. Given a state-action pair, s and a, the probability of any possible next state s' is:

$$p(s', r | s, a) = P\{S_t = s', R_t = r | S_{t-1} = s, A_{t-1} = a\} \tag{2.3}$$

The conditional property 2.3 is called transitional property.
In the same way, given a state-action pair (s, a) combined together with any next state s', the expected reward can be defined as:

$$r(s, a, s') = \mathbb{E}\{R_t | S_{t-1} = s, A_{t-1} = a, S_t = s'\} \tag{2.4}$$

These values, $p(s'|s, a)$ and $r(s, a, s')$, provide a complete definition of one of the most important aspects concerning the dynamics of a finite MDP.

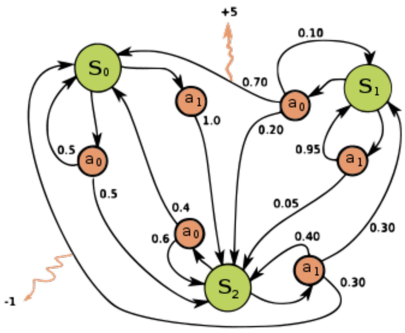

Figure 2.2: Example of finite MDP with three states $\{s_0, s_1, s_2\}$ and two actions $\{a_0, a_1\}$ (18)

Figure 2.2 shows a finite MDP. At each state it is possible to execute one of the possible actions, a_0 or a_1. The execution of each action from a given state is followed by one or at maximum i

transitions, with $i = 3$ in the analyzed MDP. A value representing $p(s_{i+1}|s_i, a_j)$ is associated to each transition. Transitions can also have another associated expected value which represents $r(s_i, a_j, s_{i+1})$.

2.1.4 RL algorithms classification

Several algorithms have now been implemented for solving reinforcement learning problems. At the base of each apporach, it is possible to identify central ideas in common among all. By comparing the different methods, it is possible to notice that they differ in the way they learn the value-function, but, in any case, the underlying idea remains the same for all methods and is called Generalized Policy Iteration (GPI). GPI represents the iterative approach aimed at approximating policy and value-functions: the value-function is repeatedly altered to approximate the value-function relative to the policy, and the policy is repeatedly improved with respect to the current value-function. In the next subsections, an accurate definition of GPI is provided as well as the correspondent algorithms implementation.

Generalized Policy Iteration

The policy iteration consists in the mutual influence of two processes (4): the first performs the task of creating the value-function V, consistent with the current policy π (process called *policy evaluation*), while the second has the task of modifying the policy, in *greedy* mode, following the values extracted from the value-function, $\pi \rightarrow greedy(v)$ (process called *policy improvement*). In the generalized policy iteration, these two processes alternate, the second starts when the first one ends, even if this is not necessary. There are variants in which the processes partially terminate before starting the following ones. For example, in Temporal Difference methods, the policy evaluation process updates the value of a single state-action pair at each iteration before terminating and allowing the policy improvement process to run. If the two processes iteratively update all the states, the final result is equivalent to the convergence with the optimal value function v^*, following the optimal policy π^*.

Thus, through the GPI, it is possible to describe the behavior of all the algorithms treated in this project, and most of the existing reinforcement learning methods. This means that, in most methods, it is possible to identify a policy according to which actions are selected and a value function, where the former is always improved compared to the values estimated by the second, and the second is always guided by the first, for the calculation of the new value-function. When both processes stabilize, then the value-function and the obtained policy will result in being optimal. This is because the value-function will only stabilize when it is consistent with the policy, and the policy will only stabilize when it has a behavior that follows the current value-function. If the policy changed the behavior with respect to the new value function, then consequently the value function would also be modified in the following iteration, to better model the behavior of the policy. For this reason, if the value function and the policy stabilize, they will be both optimal and consistent with each others. The evaluation process of the value-function and the process of improving the policy are simultaneously competing and cooperating with each others. They are competing because, by making the greedy policy in relation to the value-function, the value function is made incorrect for the new policy, while making the value-function consistent with the policy, the policy is made greedy than the new calculated value-function. In the long run, in any case, these two processes interact to find a solution that coincides with each others, which is equivalent to obtaining optimal results.

In order to fully understand the role of the policy and the value-function, it is good to briefly summarize the elements of the problem of RL. The agent and the environment interact in a sequence of discrete steps over time. The actions taken in the environment are chosen by the agent. The states are the basis on which the agent chooses the actions to be taken, and the rewards are the basic information to determine the goodness of the action performed by the

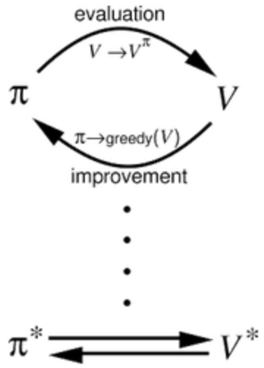

Figure 2.3: GPI: interaction between policy and value-function up to convergence to optimal solution (4)

agent in a given state of the environment. Everything inside the agent is completely controllable from it, while what is outside is not controllable by the agent and may be partly unknown.

The policy is a stochastic rule by which the agent selects the action according to the current state. Since the agent's goal is to maximize the total amount of reward accumulated over time, the policy will have to be greedy with respect to the value-function, preferring the choice of actions considered better by the value-function, where an action is considered better when it presents a higher value in a given state.

The value function assigns to each state, or to each state-action pair, the expected return. The expected return corresponds to the total amount of rewards obtainable as a result of a state visit. Upon reaching the optimal value-function, each state, or state-action pair, will have assigned the highest obtainable expected return following the optimal policy.

It is possible to estimate the value-function from the agent's experience. For example, if an agent follows a policy π and keeps in memory the rewards obtained for each encountered state, it can combine them to get the expected return (total reward) of the state. If the agent keeps an average of the total reward for all the times it has visited such state, with the number of times tending towards infinity, the average will equal the maximum obtainable expected return by visiting the state (i.e. expected value).

For MDPs, $v_\pi(s)$ is easily definable as (4):

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{2.5}$$

where $\mathbb{E}_\pi[G_t|S_t = s]$ represents the expected value obtained by the agent when following the policy π, at each time step t. Equation 2.5 is said state-value function for policy π.

Similarly is possible to define a value-function which takes into account the value of performing an action a from a certain state s following policy π:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{2.6}$$

where $q_\pi(s,a)$ is called action-value function for policy π.

It is possible to estimate the expected return, obtainable after visiting one state, through the following sum:

$$G_t = R_{t+1} + R_{t+2} + ... + R_T \tag{2.7}$$

where T represents the final step. It is assumed that in the environment there is a natural notion of final step, and, therefore, that the sequences of interaction between environment and agent are divided into episodes which term coincides with the achievement of one of the possible terminal states of the environment, e.g. reaching a final goal.

Since the environment can be of stochastic nature, it is not possible to be sure that in the following episode, by visiting s, one always obtains the same G_t. A discounting concept can be added:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma R_{t+3} = \sum_{k=0}^{T} \gamma^k R_{t+k+1} \tag{2.8}$$

with γ between 0 and 1, called the discount rate. The discount rate permits to consider with less weight choices taken in the future with respect to choices made at time step t. As you move away from state s, the obtained rewards have a lower and lower weight in the calculation of the expected return.

Thanks to the definition of expected return and value function, it is now possible to determine a fundamental property of the value functions which shows that they satisfy particular recursive relationships. For any policy π and any state s, the following consistency condition is satisfied between the value of s and the value of the possible successive state s':

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[\sum_{k=0}^{T} \gamma^k R_{t+k+1} | S_t = s] = \mathbb{E}_\pi[R_{t+1} + \gamma \sum_{k=0}^{T} \gamma^k R_{t+k+2} | S_t = s] \\ &= \mathbb{E}_a \pi(a|s) \mathbb{E}_{s'} p(s'|s,a)[R_{t+1} + \gamma v_\pi(s')] \end{aligned} \tag{2.9}$$

The equation in 2.9 is said Bellman equation and expresses the relation between the value of a state and the value of the states succeeding it (4). The Bellman equation averages among all the possibilities, weighing each of them with respect to the relative probability. It states that the value of the state s must be equivalent to the value of the following state, reduced by a parameter γ, added to the reward obtained by executing the transition. In equation 2.9, $\pi(a|s)$ represents the probability of choosing action a given the state s. $p(s'|s,a)$ is equivalent to equation 2.3. Since the consistency property is satisfied by all possible policies π, it is necessary to generalize the equation as:

$$\mathbb{E}_\pi[\cdot] = \sum_a \pi(a|s) \sum_{s'} p(s'|s,a)[\cdot] \tag{2.10}$$

Summarizing, the value of a state is given by the sum of the expected returns, weighted according to the probability of the combination of the policy choice of a and possible following states s', deriving by the stochastic nature of the environment.

Bellman equation represents the basis for calculation, approximation and learning of the value-function.

Dynamic Programming

The family of algorithms called dynamic programming (DP) was introduced by Bellman (1954), who showed how these methods can be used to solve a wide range of problems. The following is a summary of how dynamic programming approaches the decision-making process of Markov.

The DP methods deal with the solution of Markov decision-making processes through the iteration of two processes called policy evaluation and policy improvement, as defined in the previous paragraph on GPI. DP methods operate through the entire set of states assumable by the environment, following each complete iteration for each state. Each update operation performed by the backup updates the value of a state based on the values of all possible successor states, weighed for their probability of occurrence, induced by the policy and by the dynamics of the environment. Full backups are closely related to the Bellman equation 2.9, they are nothing more than the transformation of the equation into assigned instructions. When a complete backup iteration does not bring any change to the state values, convergence is obtained and then the final state values fully satisfy the Bellman equation 2.9. The DP methods are applicable only if there is a perfect model of the environment (4), which must be equivalent to a

Markov decision process. Precisely for this reason, the DP algorithms are of little use in reinforcement learning, both for their assumption of a perfect model of the environment, and for the high and expensive computation, but it is still opportune to mention them because they represent the theoretical basis of reinforcement learning. In fact, all the methods of RL try to achieve the same goal of DP methods, only with lower computational cost and without the assumption of a perfect model of the environment. Although DP methods are not practical for large problems, they are still more efficient than methods based on direct search in the policy space, such as the genetic algorithms mentioned in paragraph 2.1.2. DP methods converge to the optimal solution faster with respect to methods based on direct policy search (4).

The DP methods update the estimates of the values of the states based on the estimates of the values of the successive states, or update the estimates on the basis of past estimates. This represents a special property, which is called bootstrapping. Several methods of RL perform bootstrapping, even methods that do not require a perfect model of the environment, as required by the DP methods. In the following section, a summary of the dynamics and characteristics of methods that do not require an environment model is reported, without the need of bootstrapping. These two characteristics are separate, but the most interesting and functional algorithms, such as Q-Learning and SARSA, are able to combine them.

Monte-Carlo methods

Despite DP, Monte Carlo methods do not require the presence of a model of the environment (4). They are able to learn through the use of the agent's experience alone or from samples of state sequences, actions and rewards obtained from the interactions between the agent and the environment. The experience can be acquired by the agent in line with the learning process or emulated by a previously populated data-set. The possibility of gaining experience during learning (*on-line learning*) is interesting because it allows to obtain excellent behavior even in the absence of prior knowledge of the dynamics of the environment. Even learning through an already-populated experience data-set can be interesting, because, if combined with online learning, it makes automatic policy improvement induced by others' experiences possible.

In order to solve RL problems, Monte Carlo methods estimate the value function on the basis of the total sum of rewards, obtained on average in the past episodes. This assumes that the experience is divided into episodes and that all episodes are composed of a finite number of iterations. This is because in Monte Carlo methods only once an episode is completed the estimate of the new values and the modification of the policy take place. Like GPI, Monte Carlo methods iteratively estimate policy and value function. In this case, however, each iteration cycle is equivalent to completing an episode, i.e. the new estimates of policy and value function occur episode by episode. Usually the term Monte Carlo is used for estimation methods, which operations involve random components; in this case, the term Monte Carlo refers to RL methods based on total reward averages. Unlike DP methods that calculate the values for each state, Monte Carlo methods calculate the values for each state-action pair, because in the absence of a model, the only state values are not sufficient to decide which action is better to perform from a certain state. It is necessary to explicitly estimate the value of each action to allow the policy to make the choices. For this reason, in Monte Carlo methods, it is necessary to obtain the value function $q^*(s,a)$. The evaluation process of the action-state values is based on the estimate of $q^\pi(s,a)$ or the expected return obtained starting from the state s, choosing action a, following the policy π. There are two main Monte Carlo methods, which differ in terms of estimated expected returns:

- **Every-visit** method MC: it estimates the value of a state-action pair $q(s,a)$ as the average of the expected returns obtained after each visit to the state s and choice of the action a.
- **First-visit** method MC: it estimates $q(s,a)$ as the average of the expected returns obtained just after the first visit of the state s and action a in a given episode.

Compared to DP methods, the idea behind Monte Carlo methods is much simpler and more deterministic. However, by simplifying the calculation dynamics of the value-function, a complication arises. It is possible that different state-action pairs relevant to the achievement of the objective are never visited by following the simple policy that selects the action with higher expected return at each state (4). In fact, if the pure policy greedy was followed, the behavior of the agent would become deterministic reducing to zero the exploration factor. Maintaining sufficient exploration is a problem in Monte Carlo methods. It is not enough to choose the estimated action considered the best in that situation, because, doing so, would not make the agent acquire the knowledge about the expected returns obtainable by following alternative actions, which could lead to learn a better policy.

An approach that can solve the problem of insufficient exploration is based on the use of an ϵ-greedy policy, where ϵ represents the probability of randomly choosing the action instead of following the best defined choice from the value-function, as possible to see in the next scheme:

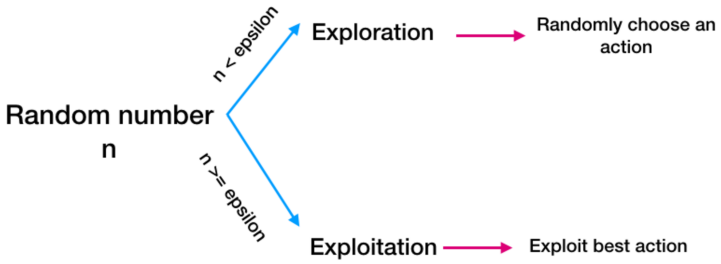

Figure 2.4: ϵ-greedy exploration-exploitation strategy

One of the main advantages of Monte Carlo methods, compared to DP methods, is the feature of focusing estimates on a smaller subset of states that the environment can acquire. A region of special interest can be accurately assessed without the need to accurately evaluate the whole set of states. In addition, Monte Carlo methods differ from DP methods for two reasons. First, MC methods learn directly from experience samples and then it is possible to learn the dynamics of the environment simultaneously with the acquisition of agent experience in the absence of a model. Finally, MC methods do not perform bootstraps, they do not estimate expected return based on other estimates.

In the following section, methods that learn from experience, such as Monte Carlo methods, but also perform bootstraps, such as DP methods, will be considered.

Temporal Difference methods

Learning by TD is a combination of ideas which stand behind Monte Carlo methods and Dynamic Programming (4). TD methods learn directly from experience in the absence of a model of the dynamics of the environment, an idea which reminds of Monte Carlo methods. In addition, like DP methods, TD methods update their estimates partly according to past estimates (executing bootstrap). As already verified for the other methods families, TD methods also follow the idea defined by GPI for the policy iteration.

Unlike MC methods, that wait for the end of an episode before estimating the expected return according to the agent's experience, TD methods update the expected return at each time step. At time step $t+1$, they are already able to update the state S_t, observing the reward R_{t+1} and applying the value of the new visited state $V(S_{t+1})$. In the most simple case, the update of the value-function is performed according to (4):

$$V(S_t) = V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{2.11}$$

where the parameter $\alpha \in [0,1]$ reflects how is the new value estimate weighted against the old. From equation 2.11, it is possible to notice that the evaluation of the value of a state depends on reward obtained at time step $t+1$ and on the value of the new visited state S_{t+1}, reduced by a discount factor γ, already introduced in equation 2.8. If $\alpha = 1$, the equation is equivalent to Bellman's equation (equation 2.9).

As required in MC methods, the value-function to be estimated is $q(s,a)$, since the complete value of a state $v(s)$ is not sufficient to determine the best movement policy, in absence of a model of the environment. On the other hand, unlike MC, the estimates of the values occur step-by-step, considering, instead of the state, the state-action pair.

In order to provide further details, it is necessary to analyze TD methods according to their classification into:

- *On-policy TD learning*, e.g. SARSA.
- *Off-policy TD learning*, e.g. Q-learning

It has been chosen to address full sections to SARSA and Q-learning because they are the algorithms that have been applied in this project.

2.1.5 On-policy learning - SARSA

On-policy algorithms, such as SARSA (acronym for State-Action-Reward-State-Action), update the policy according to the taken actions. This means that, when making moves, they follow a control policy and use it to update the Q-values, following the equation (4):

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{2.12}$$

Thus, in the simplest case, SARSA employs a table to store each state-action pair. SARSA estimates q_π according to the behavior of policy π and, in the meanwhile, it modifies the greedy behavior of the policy according to the updated estimates of q_π. The convergence of $SARSA$ and, more generically, of all on-policy TD methods, depends on the nature of the policy. In TD methods, ϵ-greedy policies are used, as possible to notice in figure 2.5.

Sarsa: An on-policy TD control algorithm

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Figure 2.5: Pseudo-code SARSA algorithm (4)

2.1.6 Off-policy learning - Q-learning

Q-Learning is a simple method of Temporal Difference learning. It allows an agent to learn the optimal behavior in a MDP. Q-Learning, like SARSA, estimates the value-function $q(s,a)$ in incremental mode, updating the state-action pair at each time step, following the logic that stands behind the update rule in equation 2.11. As already anticipated, Q-Learning, unlike SARSA, has off-policy features, that is, while the improvement of the policy occurs according to the values estimated by $q(s,a)$, the value-function updates the estimate following a strictly

greedy secondary policy: given a state, the action choice is always the one that maximizes the value $max_a q(s,a)$. Anyway, the policy π has still an important role in values estimates since it is used to determine the state-action pairs to be visited and updates (4).

$$Q(S_t,A_t) = Q(S_t,A_t) + \alpha[R_{t+1} + \gamma max_a(Q(S_{t+1},a)) - Q(S_t,A_t)] \tag{2.13}$$

The equation in 2.13 represents the approach for estimating the state-action pairs in Q-learning algorithms. As SARSA algorithm in section 2.1.5, Q-learning is characterized by two parameters: γ, which is the discounting factor and allows to assign less weight to reward obtained in the future, while α represents the learning rate.

As SARSA, also Q-learning makes use of a ϵ-greedy policy. Thanks to the Q-Learning off-policy nature, the analysis of the algorithm is simpler because it assumes a behavior independent from the chosen policy. As a matter of fact, Christopher J.C.H. Watkins, the creator of Q-Learning (1989), in collaboration with Peter Dayan in 1992, demonstrated the mathematical convergence of Q-Learning (6), unlike SARSA, for which in the literature the demonstration of convergence is not yet present.

In the simplest case, Q-learning employs a table to store each state-action pair. At each step the agent observes the current state of the environment and, following the policy π, it selects and executes the action. By executing the action, the agent obtains the reward R_{t+1} and achieves a new state S_{t+1}. At this point, the agent is able to calculate $Q(S_t,A_t)$ updating the estimate (see figure 2.6 for further details).

Q-learning: An off-policy TD control algorithm

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state},\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$
  until S is terminal

Figure 2.6: Pseudo-code Q-learning algorithm (4)

2.1.7 SARSA vs Q-learning

SARSA and Q-learning algorithms have been analyzed in the previous sections, but it is also relevant to highlight which are the differences between the two approaches. In this perspective, based on the definition of on-policy and off-policy algorithms, SARSA is a more "conservative" algorithm with respect to Q-learning: if there is a risk across the optimal path (e.g. an obstacle) represented by a negative reward, Q-learning will probably collide while exploring, whereas SARSA will try to avoid it, since it updates the policy looking at the value of performing a new action from the next state. An emblematic example of this risk handling across SARSA and Q-learning is represented by the so called Cliff-Walking experiment (11), which shows that SARSA is characterized by a lower chance of risky exploration. On the other hand, if the goal is to find an optimal policy in simulation or if a slow-moving robot is adopted (and consequently the damage risk decreases), Q-learning is a better choice because it learns an optimal policy directly, while SARSA requires a more sophisticated strategy to converge towards it.

2.1.8 Deep Reinforcement Learning

In the previous paragraphs, the estimates of the value-function have been performed by using a table, in which each cell represents a state, or a state-action pair. The employment of a table

to represent the value-function allows the creation of simple algorithms and, if the environment conditions are Markovian, it permits to accurately estimate the value-function, because it assigns to each possible configuration of the environment the expected return learned during policy iterations. The use of the table, however, also leads to limitations: in fact, the tabular action-value methods are applicable only to environments with reduced number of states and actions. The problem is not limited only to the large amount of memory required to store the table, but also to the large number of data and time required to estimate each state-action pair accurately. In other words, the main problem is generalization (4). A solution to the problem must be found, generalizing the experience gained on a subset of state-action pairs, so as to approximate a broader set. Fortunately, generalization based on examples has already been extensively studied and there is no need to completely invent new methods to be used in reinforcement learning. The solutions to generalization are based on the combination of RL approaches with methods of function approximation. Based on a subset of examples of behavior of a given function, function approximation methods try to generalize with respect to them to obtain an approximation of the whole function (4). As with table-based methods, there are various techniques of function approximations. In order not to make the treatment too complex, in the following section, just Deep Q-Learning and Deep SARSA are introduced and are referred as Deep RL. Both are the evolution of Q-learning and SARSA explained respectively in sections 2.1.6 and 2.1.5.

The term Deep reinforcement learning identifies a RL method based on function approximation (4). It therefore represents an evolution of the basic RL method since the state-action table is replaced by a neural network, with the aim of approximating the optimal value function q^*. With respect to the standard approaches, in which the network was structured in a way to use as input both states and actions and getting as output the correspondent expected reward, Deep RL analyzed in this report revolutionizes the structure, to require only the state of the environment as input and supplying as output as many state-action values as there are actions that can be performed in the environment.

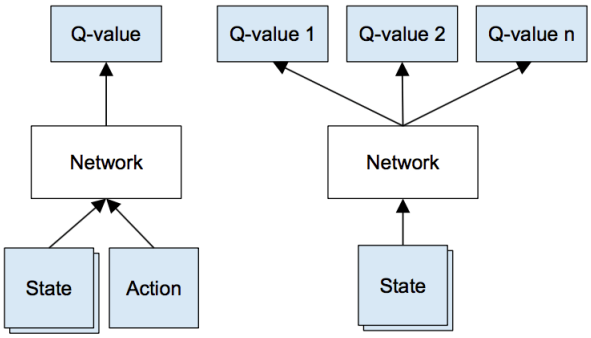

Figure 2.7: On the left: naive structure. On the right:Deep RL structure (19)

Since for each value update it is necessary to determine $max_a q(s,a)$ or $q(s',a')$ for Q-learning and SARSA respectively (see equations 2.13 and 2.12), with the naive configuration shown in figure 2.7, for each step it is necessary to execute n forward steps. On the other side, in deep RL, the number of forward steps is always equal to 1, whatever the number of executable actions, because the output of the network is composed of as many neurons as the number of actions, and the value contained in them represents the expected return of the related actions.

In the simple method with table, the learning is done by accessing the row-column representing the state-action pair and updating the expected return based on the new estimate following formula 2.13 or 2.12. This learning method is not applicable to a neural network because the only way to modify its behavior is through the adjustment of the weights, by performing a backward step. The learning of the value-function in the Deep RL method is based on the

adjustment of the net weights according to the loss function, which corresponds to the mean squared error between the target and the current value function:

$$L_t = [Target - Q(S_t, A_t)]^2 \tag{2.14}$$

where $Q(S_t, A_t)$ corresponds to the value estimated by the network and the targets, representing the optimal expected return, are respectively for Q-learning and SARSA:

$$Target = [R_{t+1} + \gamma max_a Q(S_{t+1}, a)] \tag{2.15}$$

$$Target = [R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})] \tag{2.16}$$

Clearly, the optimal expected return must be estimated. Its estimation can be done with techniques already used in MC methods or using directly the network. In the second case, note that the target values depend on the configuration of the network weights, to which changes will be made at each step. Since a loss function is applied, Deep RL treats the estimate of the value-function as a regression problem. The errors calculated by the loss function will be propagated backwards in the network through a backward step, following the descent logic of the gradient with the intent of minimizing the error.

With the made analysis, it is now possible to define a first basic Deep RL algorithm, where the state-action pairs table is replaced by a neural network initialized with random weights and the learning of the value-function is obtained by minimizing the errors calculated by the loss function.

Algorithm 1: Basic Deep Q-learning

```
Init Q(s, a) with random weights w;
while episode != final episode do
    Init and observe S;
    while step != final step do
        Choose A from S using ε-greedy π
          derived from Q;
        Take action A, observe R and S';
        Calculate target T;
        if condition then
            S' terminal state then T = R;
        else
            T = R + γ max_a Q(S', a);
        end
        Train the Q-network using
          (T − Q(S, A))^2 as loss function;
        S = S';
    end
end
```

Algorithm 2: Basic Deep SARSA

```
Init Q(s, a) with random weights w;
while episode != final episode do
    Init and observe S;
    Choose A from S using ε-greedy π derived
      from Q;
    while step != final step do
        Take action A, observe R and S';
        Choose A' from S' using ε-greedy π derived
          from Q;
        Calculate target T;
        if condition then
            S' terminal state then T = R;
        else
            T = R + γ Q(S', A');
        end
        Train the Q-network using (T − Q(S, A))^2
          as loss function;
        S = S';
        A = A';
    end
end
```

Applying the basic algorithms shown in 1 and 2, it turns out that the approximation of the value-function through a neural network is not stable. To achieve convergence, the basic algorithm should be modified by introducing techniques to avoid oscillations and divergences. The most important technique is called experience replay (14), (4), (8). During the episodes,

at each step, the agent's experience, i.e. $e_t = (s_t, a_t, r_t, s_{t+1})$ and $e = (s_t, a_t, r_t, s_{t+1}, a_{t+1})$ for Q-learning and SARSA respectively, is stored in a dataset $D_t = \{e_1, ..., e_t\}$ called replay memory. In the cycle inside the algorithm, instead of performing the network training based on only the transition just performed, a subset of transitions is selected randomly from the replay memory, and the training takes place as a function of the loss (e.g. of the quadratic error) calculated on the subset of transitions. This type of update takes the name of minibatch update and brings a significant advantage over the basic method. First of all, every step of experience is potentially used in various network weight updates, and this allows data to be used more efficiently. Furthermore, learning directly from consecutive transitions is inefficient due to the strong correlation between them. The experience replay technique, by randomly selecting transitions from replay memory, eliminates the problem of correlation between consecutive transitions and reduces variance between different updates. Finally, the use of the experience replay makes possible avoiding converging into a local minimum or to diverge catastrophically, since the update of the weights is based on the average of several previous states, smoothing the learning and avoiding oscillations or divergences in the parameters. In practice, the modified algorithm stores the last n experiences in the replay memory D and randomly select a subset of experiences from D each time it performs an update of the network weights (see algorithms 3 and 4).

Algorithm 3: Full Deep Q-learning

Init replay memory D to capacity N;
Init $Q(s,a)$ with random weights w;
while *episode != final episode* **do**
  Init and observe S;
  while *step != final step* **do**
    Choose A from S using ϵ-greedy π derived from Q;
    Take action A, observe R and S';
    Store experience (S, A, R, S') in D;
    Sample random transition (S_s, A_s, R_s, S'_s) from D;
    Calculate target T_s;
    if *condition* **then**
      S' terminal state then $T_s = R_s$;
    else
      $T_s = R_s + \gamma max_a Q(S'_s, a)$;
    end
    Train the Q-network using $(T_s - Q(S_s, A_s))^2$ as loss function;
    $S = S'$;
  end
end

Algorithm 4: Full Deep SARSA

Init replay memory D to capacity N;
Init $Q(s,a)$ with random weights w;
while *episode != final episode* **do**
  Init and observe S;
  Choose A from S using ϵ-greedy π derived from Q;
  while *step != final step* **do**
    Take action A, observe R and S';
    Choose A' from S' using ϵ-greedy π derived from Q;
    Store experience (S, A, R, S', A') in D;
    Sample random transition $(S_s, A_s, R_s, S'_s, A'_s)$ from D;
    Calculate target T_s;
    if *condition* **then**
      S' terminal state then $T_s = R_s$;
    else
      $T_s = R_s + \gamma Q(S'_s, A'_s)$;
    end
    Train the Q-network using $(T_s - Q(S_s, A_s))^2$ as loss function;
    $S = S'$;
    $A = A'$;
  end
end

2.2 Existing setup

In this section, the configuration of the already existing setup that will be modified to fit project goals is presented. The existing setup is deeply described in (2). It consists of a three degrees of freedom robotic arm, actuated by three servo-motors and controlled by an Arduino board. In the next subsections, its mechanical and software architectures are presented.

2.2.1 Mechanical architecture

The manipulator has been manufactured using laser-cut technology provided by RaM protolab to simplify the production process. In particular, the structure consists of the following components (see figure 2.8):

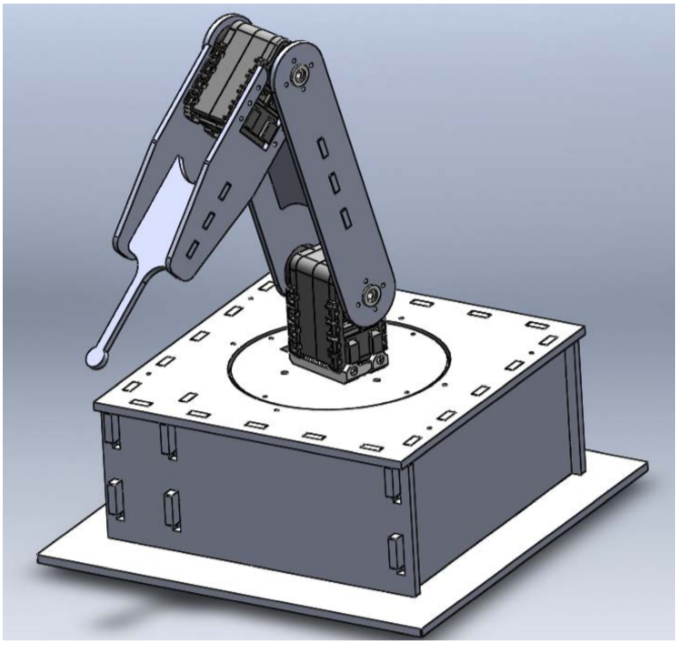

Figure 2.8: Existing design of the robotic arm (2)

- *Box-like base*, which supports the robotic arm and in which a servo-motor is placed to allow the upper plate and, consequently, the first link to rotate. All the electronic components lies inside the box as well. The box is assembled using four bolts (to connect the top of the box) and hooks and slots strategy (see figure 2.9).These hooks slide into place constraining all the motions expect one (in sliding direction); if that motion is constrained externally, the connection is strong and reliable.
- *Rotation plate*, which is connected to the base through a hollow bearing (i.e. slewing ring), to make the base more rigid. This rotation plate allows the first link to rotate upwards and downwards through a servo-motor which is settled on its top.
- *First link*, which is attached to the rotation plate. On its tip a third servo-motor is placed to make the second link to rotate.
- *Second link*, which is connected to the first thanks to the third servo-motor. Its tip is smaller to easily represent the end-effector position.

The servo-motors that have been selected to move the mechanical structure are DYNAMIXEL AX-12A from Robotis (see figure 2.10). They are digital servo-motors provided with different features, such as half-duplex communication, control parameter tuning, tracking of speed, temperature, shaft position, voltage and load. A complete data-sheet of those motors is reported in the appendix A.1.

2.2.2 Software architecture

The control architecture of the robotic arm is implemented in MATLAB and is based on screw theory. Screw theory is a technique which describes the motion of rigid bodies in 3D space

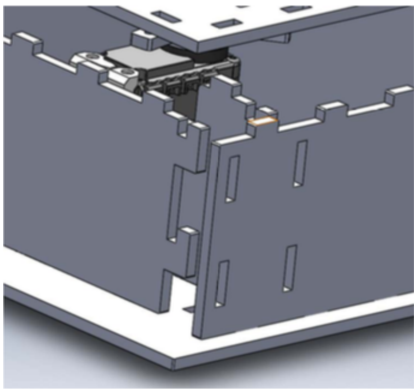

Figure 2.9: Exploded view of hooks-slots connection of the base (2)

Figure 2.10: Servo-motors AX12A produced by Robotis (20) have been selected to actuate the joints of the manipulator

through a vector approach. In this perspective, the velocity of the considered body is represented as a screw-like motion along a line, such that translation and rotational components are considered together in one vector called twist. After evaluating the twists for the considered frame of reference, a Jacobian matrix can be extrapolated from which parameters of interest can then be calculated, i.e. joints velocities and, consequently, joints positions. The control algorithm applied to the robot arm is shown in figure 2.11.

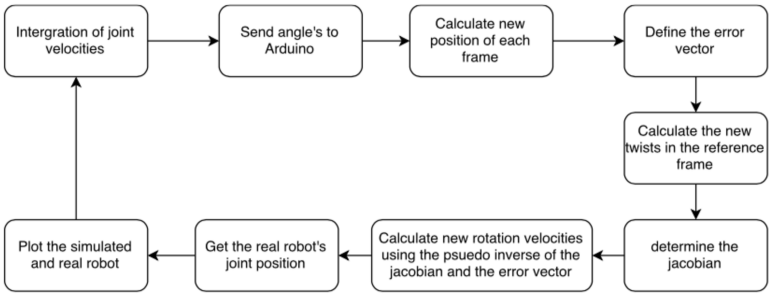

Figure 2.11: Control architecture as implemented in MATLAB (2)

As possible to notice from 2.11, the communication between MATLAB and the motors is made through an Arduino board. Because of half duplex communication of the cited motors (see section 2.2.1), RX and TX ports of Arduino have to be connected to the same communication line. This is done by using a tri-state buffer which acts as a switch between the Arduino and the DYNAMIXEL. In addition, the tri-state protects the Arduino ports.

Even if the motors are provided with encoders, which are able to evaluate motors position and speed, vision guided configuration estimation strategies are analyzed in the next section in

order not to rely on other hardware like torque sensors and/or laser scanners to evaluate the distance between the manipulator structure and other points of interest (e.g. goal and obstacles).

2.3 Vision guided state estimation

In recent years, interest in vision systems has considerably increased. This is because vision techniques present themselves as the best compromise between cost, flexibility and provided information. The development of robotic applications in unstructured, dynamic and rapidly changing environments requires the use of robust and reliable vision systems that are able to perceive the events that occur in the environment and monitor their evolution. Thus, the advantage of using vision systems for localization and tracking purposes lies essentially in the amount of information that can be obtained even without the use of special and expensive hardware, such as position sensors or torque sensors. Unfortunately, extracting reliable and accurate information from images is not an easy task and it becomes even less so if the sequence of images should be taken and elaborated as much as possible real-time, as in the case of configuration estimation of robotic arms.

The problem linked to the location and tracking of a body is a problem that is difficult to solve, especially if, as in this case, the only source of information is a sequence of two-dimensional images. So, there is the need to find a solution for:

- Extract the elements of interest from the set of pixels that constitute a digital image (*classification*).
- Calculate their position in the environment (*localization*).
- Associate the elements previously identified with the current ones in order to identify the trajectory followed by each of them (*tracking*).

What makes localization and tracking a problem of not easy solution is the fact that it contains within it a large number of sub-problems, such as the extraction of the elements of interest from the images, the calculation of their position in space, the resolution of the ambiguities due to the various occlusion situations, the location of the same object in two different frames and so on. All these sub-problems have to be taken into account when selecting the most appropriate algorithm.

In this section, an analysis of some of the most interesting applications of vision-guided state estimation which can be found in literature is provided, in order to find out which is the most appropriate technique for real-time implementations.

2.3.1 Localization and tracking techniques

One of the problems encountered in the realization of a vision system, which allows localization and tracking of the movements of robots, is related to how to calculate their position and their trajectories within the environment. The localization process has as its main objective the determination of the position and the possible orientation of the elements of interest. In the tracking process, on the other hand, it is important to identify the correspondences between the previous and the current frame, which allow agents to be followed over time. In other words, it is a question of extracting the elements of interest that characterize the frame at the instant $t-1$, such as points, lines, shapes, etc., determine their position in space (localization) and identify their presence in the frame captured at time t by determining a displacement trajectory (tracking). The techniques used to identify and extract these elements of interest are substantially two and are distinguished by the employment or not of particular devices called markers.

Localization and tracking with markers

According to this approach, markers and/or devices of various types are fixed on the robot's structure (e.g. see figure 2.12) (21)-(26). The signals emitted by these devices can be of different kinds, luminous, electro-mechanical, etc., and are captured by the appropriate receiving device, which has the task of converting these signals into two/three dimensional information. This technique is widely used in Virtual Reality applications and has the main advantage of obtaining the position and orientation of the robot in real time. On the other hand, it presents the following disadvantages:

- Moving the sensors from their original position causes situations of uncertainty in the results.
- Particular difficulty in positioning such devices on certain region of the body, such as narrow areas.

Figure 2.12: AprilTag markers for navigation of mobile robots (26)

However, different types of reliable markers have been identified and tested on different robots. In (21), fiducial markers developed in the ARToolkit library are attached to the links of a small manipulator and then tracked through a monocular camera for visual servoing control applications. Visual servoing control permits to obtain a reliable state estimation and control without using measurements acquired with the encoders placed in the joints of the manipulator. In order to detect these special markers, first an edge detection is performed, identifying discontinuities through Laplacian operator and looking for connections between group of pixels having similar gray tone. Once edges are extracted, four lines forming a frame are considered as potential markers. The detection has been proved to be fast but not so robust to changes in illumination (26). The same markers library have been used, detected and tracked through a monocular camera also in (22), where the kinematic model of a six degrees of freedom manipulator is learned together with the geometrical relationships between its body parts as a function of the joint angles. Furthermore, the predicted internal kinematic model could be used to adapt it when the robot body changes due to fatigue or failure. The central idea that stands behind these concepts is learning through non-parametric regression a large set of local kinematic models and then look for the best arrangement of these models to represent the full robotic system. In this perspective, a large sequence of random motor commands are given to the robot and, after each movement, the new configuration of the manipulator is checked, detecting the new location of the markers. But, since arbitrary motion patterns (just constrained by the geometry of the manipulator) are set, full visibility of the markers is not guaranteed and, in that case, the configuration is rejected. Thus, the work follows the idea of *learning by explanation*, i.e. the search for the kinematic structure is guided by the accuracy of observations and, consequently, depend on how well those observations could explain the model.

Markers similar to ARTag have been developed by (26). AprilTags are fiducial markers which use 2D bar code style "tag" (see figure 2.12), allowing full localization of features from a single image. With respect to ARToolkit, this library is completely open and well documented. In order to detect these markers, a graph-based image segmentation algorithm based on local gradients has been implemented. As specified in (25), image segmentation is the process of partitioning an image into meaningful regions. More precisely, segmentation is the process of classifying image pixels that have common characteristics, so each pixel in a region is similar to the others in the same region for some property or characteristic (color, intensity, or texture). Adjacent regions are significantly different than at least one of these features. The result of a segmented image is a set of segments that, collectively, cover the entire image. Thus, applying the local gradient means computing the gradient direction and magnitude at every pixel and then cluster the pixels with similar gradient directions and magnitudes. Moreover, a quad extraction is performed in order to find line segments that form the quad itself and, once the quad is found, a 2D barcoding algorithm is applied to extract the digital code of the marker. The AprilTag detection has been proved to be fast and robust (26).

Following different approaches with respect to the previous ones, other types of markers have been employed in (23) and (24) and detected by a single monocular camera. Both the researches made use of color markers which are placed on the features of interest of the robots (in (23) they are placed on the hands of Nao robot, while in (24) they are settled on the joints of the used manipulator and on its end-effector). In order to detect these kinds of markers, a colour segmentation technique can be applied. Once the colour has been detected through image segmentation, a process of blob analysis (25) is employed to detect the contours of the markers and classify them according to their area. This approach have been proved to be accurate and fast in real-time applications (23), (24).

Localization and tracking without markers

Methods that do not use markers in the phase of localization and tracking are able to obtain an estimate of the position of the tracked robot processing only the sequences of images from video capture systems. The sequence of images can come from a single camera (monocular vision system), or from two or more cameras (multi-camera vision systems).

In monocular vision systems, the position of the robot is tracked by first extracting the profile of it and then trying to find the correspondences with a 3D model. An example of this implementation is described in (27), where a virtual visual servoing algorithm has been applied to track several parts of a six degrees of freedom manipulator with the use of a single monocular camera. Then, the obtained information are employed together with the kinematic model of the robot to estimate its configuration.

This technique, associated with a geometric model of the camera, allow the transition from two-dimensional image coordinates to three-dimensional ones. It should be noted that the geometric model of the camera is not sufficient to determine the position of a point in space. In fact, in addition to knowing the coordinates (u, v) in the image domain, to get a single solution the distance d that separates the point of interest from the camera is needed. An estimate of this distance d is provided by applying the methodology mentioned above.

Another possible way to reconstruct the third dimension is based on the use of stereo cameras, as proposed in (28). In (28), a seven degrees of freedom manipulator is tracked with a binocular camera. Firstly, an HSV segmentation of a planar patch placed on the end-effector is made (25). Afterwards, a region of interest is selected, extracting feature points and tracking the latter in all the video frames. Finally, feature points are used to estimate the homography between world reference frame and image frame.

These techniques, in which the use of markers or devices of various types is not required, allow the robot to move freely. This advantage is paid, however, with greater difficulty in the

reconstruction of the third dimension. In previously mentioned monocular vision systems, the estimate of the third dimension turns out to be even more difficult to obtain compared to vision systems that employ two or more cameras. This is because the formation of a two-dimensional image is constituted by the superimposition of more three-dimensional information that govern the scene. As a result, the inverse problem, given a two-dimensional image determining the three-dimensional scene from which it derives, does not have a single solution.

3 Analysis

This chapter mainly focuses on analyzing the most appropriate solutions to satisfy project goals and expectations through the definition of specific requirements.

In the literature, many researches have already been done concerning the integration of RL algorithms in the control architecture of robotic manipulators (see section 3.1), but none of them actually compare different algorithms with different learning parameters tuning (e.g. different reward functions). Such a comparison would be of great interest when dealing with a complex and over-constrained environment like in the case of a pipe: the most appropriate algorithm could be selected depending on the configuration of the environment that the manipulator should learn at that time. Thus, in order to test these methods in an efficient and effective way, it is necessary to develop a setup which is easy to be used and simply modifiable, so that it can be adapted effortlessly to fit user requirements.

Summarizing, this chapter is organized as follow: in section 3.1, the state-of-the-art of RL in robotics applications is analyzed together with its correspondent main challenges. Moreover, in section 3.2, the domain of interest is inspected, in terms of RL algorithms selection, manipulator configuration and vision-based setup layout. Then, in section 3.3 all the requirements concerning RL algorithms, setup and tests are specified. Based on the requirements and on the domain analysis, the methodology adopted in this project is outlined so that conclusions concerning the feasibility of the chosen alternatives can be assessed.

3.1 State-of-the-art of RL in robotics applications

Reinforcement Learning has become of great importance in robotics applications since it permits to fill the gap towards autonomous robots, providing the necessary data to make a robot able to perform a specific task without the need of an exact model of the environment around it (9), (10). Thus, in this section, some of the most relevant applications of RL into robotic arms domain are analyzed, together with their correspondent approach and tuning choices. Eventually, in the last paragraph of this section, the main challenges in robotics-RL environment are described.

3.1.1 RL and robotics manipulators

As previously stated, in the last decade, many researches have been done towards the integration of RL algorithms in robotics applications. This trend is due to the fact that RL is strictly related to the theory of classical optimal control (9), since both the approaches try to find an optimal policy (i.e. a controller) which is able to maximize an objective function, often called cost or reward function. However, optimal control approaches require complete knowledge of the model of the system, i.e. a function which is able to describe, starting from the current state, which will be the next state if a certain action is performed. On the other hand, RL does not require this kind of knowledge, because the learning procedure operates through direct interaction between the agent and the environment according to measured data (see section 2.1.1). Precisely for this last aspect, RL has been increasingly used in arm planning applications. Arm planning relates to all those sets of solutions which provide the robot arm with the ability of navigating in an environment, avoiding collisions with possible obstacles and, eventually, determining the best trajectory to be followed to achieve a predefined objective, e.g. grasping an object or reaching a certain goal with the end-effector.

Many of the already implemented researches make use of (Deep) Q-learning algorithm to learn different kinds of tasks. In (12), a two-link manipulator is trained to move the end-effector to a defined position avoiding obstacles, applying compositional Q-learning together with a

reward function, which is higher when destination is approached with low velocities, negative when high velocities are reached and really negative when collision occurs. Q-learning is also applied in (15), in which a two links rigid manipulator learns how to stabilize at 0 with zero velocity in minimum time. The state-space has been discretized as well as the action-space, which corresponds to three torque values for each joint. The reward function is zero when the goal is reached and -0.5 otherwise. A similar situation is described in (13), where a six degrees of freedom robot arm learns how to reach a target without any previous knowledge of the constrained environment in which it is placed. A double neural network together with a Q-learning algorithm is used as learning approach and combined with a reward function which is positive when a target is reached without collisions and negative otherwise. On the other hand, in (14), Deep Q-learning with convolutional neural network is applied to make a six degrees of freedom robot arm with two grippers and a camera set on the end-effector to learn how to pick up an object. A trivial linear reward function is applied, which takes into account of the discounted exponential distance between the end-effector and the object.

With respect to Q-learning, SARSA has not been used a lot in the literature of robotic arms. However, (16) proposes an interesting application on a three links planar robotic arm: a multi-agents SARSA algorithm (i.e. each link has is proper Q-table, which update independently) is tested on making the cited robot arm to reach first a fixed goal and then track a random goal. A penalty of -1 is assigned at each time step, while a huge reward of +1000 is achieved when the end-effector reaches the target. Both the action-space and the state-space is considered to be discrete. Moreover, in (17), a seven degrees of freedom robotic arm is used to learn the so called "ball-in-a-cup" game applying SARSA with discretized state and action spaces. The learning goal is to swing the ball with the desired angle and the desired velocity. Each state is rewarded based on the angle and negative rewards are assigned to prevent the robot to overcome its stroke-ends. This approach has been compered with a supervised learning technique, based on dynamic motion primitives.

3.1.2 Challanges in RL-robotics domain

As possible to notice also from the reported literature, one of the main issue concerning RL algorithms in robotic arms domain is related to treat high dimensional state-spaces. Robots operate in high-dimensional state-spaces constituted of both internal states (e.g. joints and links position and velocity) and external states (e.g. obstacle locations, presence of other robots, wind conditions, etc.). Under these circumstances, the robot selects its motors commands (i.e. the actions) according to a certain control policy π. The motors commands will then alter the state of the robot and its environment, based on the value function V^{π}.

(9) and (10) reported the main challenges of this domain in a good order:

- *Curse of Dimensionality*: high dimensional continuous state-action pairs, which require the implementation of either environment discretization (i.e. just some state-action pairs are allocated to memory) or value-function approximation through deep-learning techniques, e.g. neural-networks (see section 2.1.8).
- *Curse of Real-World Interactions*: Reinforcement Learning algorithms have to operate in real-time, selecting actions in real-time, so they should be able to deal with delays in the sensing and execution typical of physical systems.
- *Model errors*: assumption of complete knowledge of the robot and its environment is unrealistic. Thus, uncertainties and noise have to be considered when designing the model.
- *Shape a proper reward function*: it is necessary to design a proper reward function, which takes into account the available knowledge of the environment as well as learning convergence rate.

Thus, when implementing a new RL approach in robotics domain, all the cited challenges should be taken int account and appropriately managed.

3.2 Domain Analysis

This section describes the domain of the project, paying a particular attention towards RL algorithms selection 3.2.1, manipulator configuration 3.2.2 and vision-based setup definition 3.2.3.

3.2.1 RL algorithms selection

The selection of RL algorithms is discussed herein. In particular, the advantages and disadvantages of the different algorithms are analyzed together with their relative simplicity in the implementation.

The section is divided into two subsections: the first presents a comparison between value-function based and policy based RL algorithms, while the latter focuses on the curse of dimensionality issue (see section 3.1) and on the two approaches that can deal with it: discretized state space and continuous state space algorithms.

Value-function based vs policy based

As already specified in section 2.1.2, RL approaches can be split into (figure 3.1):

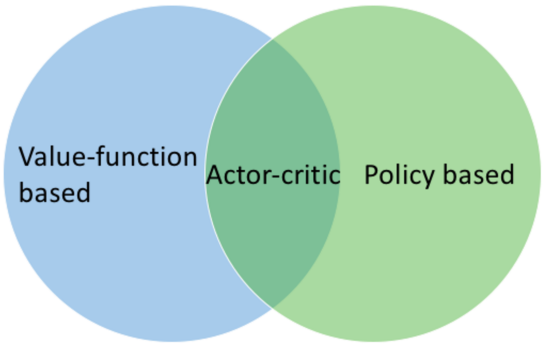

Figure 3.1: Venn diagram representing RL algorithms classification

- *Value-function based approaches*, such as Monte-Carlo methods and Temporal Difference methods, like Q-learning (section 2.1.6) and SARSA (section 2.1.5).
- *Policy based approaches*, such as policy gradient algorithms as well as meta-heuristic or genetic algorithms (4),(5),(7).
- *Actor-critic approaches*, which can be considered as a combination of the previous two methodologies.

All the methodologies present pros and cons: if on one hand value-function based approaches are easier to be implemented and straightforwardly converge to a global (sub)optimal policy, while policy search approaches usually converge faster but just to a local π, the latest better deal with high dimensional spate-action pairs without the need of implementing a value-function approximation. Moreover, policy based methods can present a very large variance in gradient estimators and this happen also in actor-critic approaches (4) and they are both much more difficult to be implemented since they are based on probabilistic models of the environment.

Thus, after analyzing the state-of-art of RL algorithms in robotics applications (section 3.1), it has been noticed that both Q-learning and SARSA obtain good results when dealing with robotic manipulators in both discrete and continuous time (9)-(17) even though they are

model-free and they require function approximations to deal with continuous state spaces (see section 2.1.8).

Exactly for these reasons, besides the fact that their implementation is more straightforward and similar, both the algorithms will be tested and assessed, in order to check which of the twos presents faster convergence to a (sub)optimal policy and which is more reliable in dealing with over-constrained environments.

Discretized RL vs deep RL

As already mentioned in section 3.1, the Curse of Dimensionality is an important issue to be solved. In particular two different approaches could be adopted:

- *State-space discretization*: as explained in sections 2.1.5 and 2.1.6, the state-space can be discretized reducing the possible states that the manipulator can assume in the environment to a finite number of states. For example, applying this approach, the joints could be able to rotate only of some predefined degrees declared in the discretization sample and based on the action vector. Thus, the algorithm makes use of a Q-table to store each state-action pair.
- *Continuous state-space*: as mentioned in section 2.1.8, the state-space can be kept continuous by making use of function approximations. According to this approach, the state-action pairs that have to be allocated to memory in value-function based approaches are substituted by a function approximator (e.g. neural networks) which approximates the value of as many Q-functions as there are actions in the action vector.

If on one side Deep RL approaches are able to handle huge state-spaces, their implementation is not straightforward and they need some extra tuning before working in an appropriate way. On the other hand, discretized RL algorithms are easy to be used when the available state-action pairs are not too much, but, when the environment becomes larger, they cannot be applied accurately anymore: a too high discretization level will necessarily bring to less accurate results, because the agent would not be able anymore to reach any state of the environment, but just the discretized states that have been considered. Thus, in a first moment, the state-space will be discretized not to over-complicate the algorithm structure and, therefore, the interpretation of hypothetical failures. When satisfactory results will be obtained in this context, the algorithm will be extended to the continuous state-space case through the application of deep learning techniques as specified in section 2.1.8.

RL parameters tuning

Appropriately tuning the RL parameters is of fundamental importance in order to make an agent to converge to a (sub)optimal policy as fast as possible applying value-function based algorithms. In particular, the main parameters that have to be taken into account are the following:

- Number of *iterations per episode*: a specific number of iterations per episode should be selected, because they are proportional to the number of actions the agent is allowed to select (see pseudo-codes in figures 2.5 and 2.6). Infinite iterations, or iterate up to the goal is reached, can make the agent to fall in some local minima in value-function estimate, never reaching the goal, but just following a policy which is thought to be optimal even if it is not. On the other hand, not enough iterations could correspond to a never reaching the goal situation because more actions could be required to achieve the objective. Thus, it is important to proportion the number of iterations according to the size of the environment and, consequently, the available state-action pairs.
- ϵ-greedy exploration and exploitation (figure 2.4): with ϵ probability a random action is chosen, while with probability $1-\epsilon$ the best action is chosen, according to the estimated

value-function. Thus, ϵ should be tuned such that a trade-off between exploration and exploitation phases is figured out. In this way the agent is encouraged not to waste time exploring non-interesting areas of the environment, but, at the same time, it explores sufficiently to acquire knowledge of the surroundings (i.e. location of points of interest s.t. obstacles and goal).

- α (look at equations 2.12 and 2.13), which is called learning rate and determines how is the new value estimate weighted against the old.

- γ (look at equations 2.12 and 2.13), which corresponds to the discount factor. If γ is set equal to 1, it means that future rewards are more relevant when evaluating the value of the current state-action pair. On the other hand, if $\gamma = 0$, future rewards do not play any role during the update.

- *Reward function*: the reward function is the most important factor in RL, because according to it the agent select the most appropriate actions. As a matter of fact, the actions are chosen so that the cumulative reward is maximized. Reward function can be a simple bonus when reaching a goal and, consequently, a penalty otherwise (as described in (15), (13) and (16)), or, on the other hand, it can be more sophisticated and depend on the distance between the agent and the goal (as described in (14) and (17)).

Thus, an accurate selection of the cited parameters has to be performed in order to understand their relation with the convergence rate and consequently optimize it.

3.2.2 Manipulator configuration analysis

As specified in section 2.2, the setup implemented in (2) has been taken as a reference for designing the three degrees of freedom planar manipulator that will be employed in this project. In this section, the cited configuration is analyzed in order to determine which modifications should be made to its mechanical design and software architecture to allow the integration of RL approaches.

Current mechanical architecture analysis

The mechanical configuration of the setup presented in (2) is non-planar. As a matter of fact, the manipulator presents the following degrees of freedom (see figure 3.2):

1. A first degree of freedom, which allows the first link to rotate clockwise and counter-clockwise around its z-axis (red arrow in figure 3.2).

2. A second degree of freedom, which allows the second link to rotate upwards and downwards (yellow arrow in figure 3.2).

3. A third degree of freedom, which allows the third link to rotate upwards and downwards (green arrow in figure 3.2).

Thus, its design should be modified in order to make the links completely planar and consequently parallel to the work-surface (see section 1.2). The decision to create a planar manipulator is due mainly to the greater simplicity of design and verification of the prototype. Furthermore, a planar manipulator is more easily employed for pipe inspection purposes, since it can better fit inside small section pipes. On the other hand, a three degrees of freedom configuration is selected to have more motion flexibility and, consequently, to be able to test more sophisticated, constrained and challenging trajectories, which can better simulate applications inside a pipe environment. Therefore, the SOLIDWORKS design of the manipulator will be modified to get a planar configuration. However, the same servo-motors, DYNAMIXEL AX12A (see A.1), could be employed since they present an integrated position controller, which allows

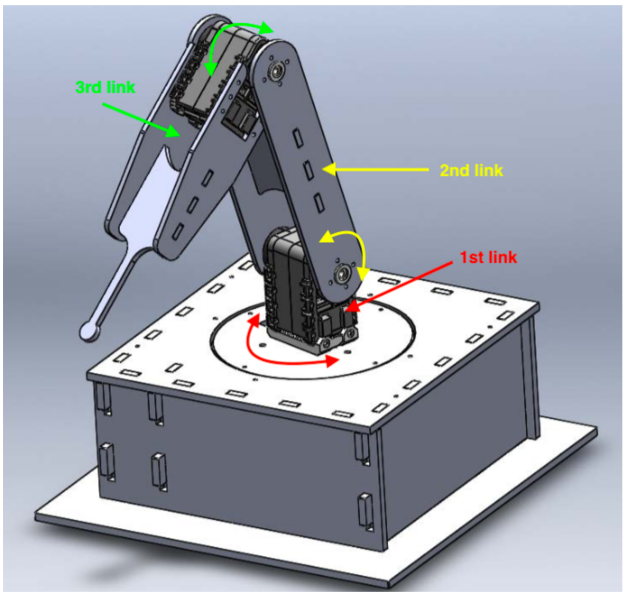

Figure 3.2: Degrees of freedom existing setup (2)

to directly send them accurate position commands. This would not be the case if DC motors would have been used, because the latter require an external control architecture to get the required current commands.

Current software architecture analysis

As already mentioned in section 2.2.2, the current software architecture of the existing setup proposed in (2) makes use of screw theory, implemented in MATLAB, together with an Arduino board. In particular, motor position commands are obtained from the integration of joint velocities acquired from the Jacobian matrix and they are then send to the Arduino board, which converts them to appropriate signals for the motors.

The connection between the Arduino board and the motors is not straigthfoward, because the motors are only able to make half-duplex communication and, consequently, RX and TX ports of Arduino have to be connected to the same communication line through a tri-state buffer, appropriately configuring the respective Arduino scripts. This communication procedure could be simplified employing another device produced by Robotis (supplier company of the motors) which allows direct connection between the motors and a PC USB port: USB2Dynamixel (29) (see figure 3.3).

Figure 3.3: USB2Dynamixel: device which directly connects DYNAMIXEL motors to PC USB port (29)

As specified in (29), on one side USB2Dynamixel have a USB connection which allows it to directly connect to a serial USB port of a PC, while also 3P and 4P connectors are installed so that different DYNAMIXEL motors can be connected. This device does not supply power to the motors but an external power supply can be adopted.

The main advantages of employing this device instead of Arduino are that it is simple to be used and a full open source library called DYNAMIXEL SDK has already been implemented by Robotis team (31) and it can operate with different programming language (i.e. C#, C++, C, Java, Python, LabVIEW, MATLAB) together with many external environments, such as ROS and Arduino.

Once the motor are appropriately connected, another point to take into account is the integration of RL approach into the control architecture of the robotic arm. In this perspective, the control architecture should be simplified as much as possible in order for it to be flexible and easily adjustable to combine different RL algorithms. Thus, using the open libraries cited beforehand could be of great benefit since they are open and, consequently, modifiable to satisfy project expectations.

3.2.3 Vision-based setup

In section 2.3, different vision-based localization and tracking techniques have been reported. The stated techniques have been subdivided into two main categories:

- *Localization and tracking with markers* (21)-(26), which can be easily attached to the manipulator structure and to relevant features of the environment, such as obstacles and goal. These approaches can accurately work in real-time applications and allow to get precise pixel location of the points of interest, simply employing an uncalibrated monocular camera. Even though, the main disadvantage of this approach is that these markers should be visible in most of the video frames not to increase uncertainty in the model. Thus, their original location should be accurately chosen such that change in the surroundings, i.e. motion of the robot or alteration of light conditions, do not affect or overcomplicate their detection.

- *Localization and tracking without markers* (27),(28) requires an accurate 3D model of the environment in order to get correspondences between the actual environment as seen by the camera and the model. These approaches are more tricky, in particular when adopting a single camera, because they all need the reconstruction of the depth dimension (i.e. third dimension) to evaluate the distance between the features of interest and the camera itself, so that a homography matrix can be generated.

Based on this analysis, it is evident that applying an approach which require the use of markers is more handy in the current situation, because there is no need to implement a specific model of the environment as required applying the other technique. The use of a model would also go against the central idea of Q-learning and SARSA which are both model-free algorithms.

The use of a planar manipulator, which is parallel to the working environment, as well as of a fixed single camera placed at a specific height above the environment should guarantee that the markers are always detectable in the scene and should so overcome the main disadvantage of this approach. Beside that, it is important to place the setup in a location with good light conditions in order to avoid the presence of shadows that could obscure the camera field of view and, consequently, impede the markers detection.

3.3 Requirements

This section reports the requirements for the project. In particular, five main topics are covered: in section 3.3.1, Reinforcement Learning algorithm requirements are described, paying attention to convergence and tuning issues. Moreover, the setup requirements are shown in section 3.3.2, while section 3.3.3 deal with tests and simulation requirements. In section 3.3.4, the requirements concerning the documentation are presented. Finally, section 3.3.5 illustrates the non-functional requirements that have not been specified beforehand.

3.3.1 RL requirements

1. *Employ value-function based RL algorithms*

   Value-function-based RL algorithms will be adopted in this project, since they have been proven to be efficient in robotic navigation and manipulation applications (9)-(17). Furthermore, their implementation is more straightforward and completely model-free, so no previous knowledge of the environment is required. In this perspective, the tested value-function based RL algorithms have to guarantee convergence to a sub-optimal policy in at maximum 8 hours, so that they can be fully tested in one working day.

2. *Handle large state-space efficiently*

   If the state-space is discretized, a proper discretization level should be found out so that the manipulator will still be able to accurately reach specific goal location. If the discretization level is too low, the considered state-space representation of the environment would be too rough and would not allow the end-effector to reach precise locations, while high discretization level would correspond to not-allocable Q-table. Thus, the discretization level should be chosen in such a way that the end-effector can reach states in the neighborhood of the goal (maximum 5 millimeters away).
   On the contrary, if continuous state-spaces are adopted, the Q-table will be substituted by a neural network (see section 2.1.8), which should be appropriately tuned and trained to manage the whole state-space. In particular, the training period of the neural network should not take more than 0.5 seconds so that the manipulator motion remains fluent.

3. *Find an optimal trade-off between exploration and exploitation*

   A trade-off between exploration and exploitation phases should be figured out appropriately tuning ϵ, to encourage the agent to learn the environment in a smart and goal-oriented way. In particular, the agent should not waste time in exploring areas far away from the goal, but, at the same time, it should be able to learn a suboptimal policy in less than the available 8 working hours.

4. *Make the algorithm as much as possible environment independent*

   The algorithm should be able to operate without requiring too many information on the environment in which the robot is placed. If this requirement is satisfied, the algorithm becomes flexible and easily adaptable to new conditions and constraints. To test if this requirement is satisfied, the acquired knowledge (i.e. Q) can be utilized in a transfer learning perspective, to learn similar environments initializing the Q in the same way. In particular, the transfer learning approach has to be at least 50% faster than the standard learning algorithm.

5. *Improve convergence rate appropriately tuning RL parameters*

   As mentioned beforehand , RL parameters actually affect the convergence of the algorithms. Thus, their tuning has to be justified such that convergence to a (sub)optimal policy is always guaranteed.

6. *Smart collisions management*

   A proper interaction between the robot arm and the obstacles should be figured out to speed up the learning phase. E.g. reset the robot to its initial pose when a obstacle is hit. Furthermore, the environment could be over-constrained to assess the efficiency of the algorithm. This requirement is satisfied in the moment in which the number of collisions is minimized without affecting the learning rate.

3.3.2 Setup requirements

1. *The setup should be manufactured with RAM facilities.*

   The setup should be designed in such a way that RaM Group laser cut or 3D printing prototyping technology could be used to manufacture it.

2. *The necessary components should be low-cost and commercial.*

   The components required for both the manipulator and the camera should be cheap and easy to find, considering a maximum available budget of 1000 €.

3. *The communication between different hardware components have to be as fast as possible*

   The communication between the different systems (camera, motors, control architecture with RL algorithm) should be fast and reliable, so that signals can be exchanged as much as possible real-time to avoid implementing complex synchronization procedures. To satisfy this requirement it is advisable to make use of already tested open-source libraries.

4. *The camera has to localize the markers efficiently*

   As already stated, markers will be placed in the points of interest to realize visual-guided localization and tracking. Thus, markers should be selected such that they are easy to be detected and tracked real-time. The detection algorithm should not take more than 0.5 seconds to detect all the markers present in the scene.

3.3.3 Tests requirements

1. *Simulation is the first-step for valuable tests.*

   Before performing tests on the real setup, evaluate the code in simulation is important to become sure of its outcome. The simulation has to be as much as possible representative of the real conditions under which the robot will operate.

2. *Employ virtual obstacles to bypass setup damages.*

   To avoid actual damages to the setup during possible collisions, virtual obstacles has to be employed to make the test effective and safe at the same time.

3. *Prioritize tests on more performing algorithms.*

   Since RL algorithms require some time to converge, it is necessary to prioritize tests on more efficient algorithms, such that the full work-ability of such algorithms can be explored.

3.3.4 Documentation requirements

1. *The documentation has to be up-to-date.*

   The project documentation has to be frequently updated, such that all the meaningful developments and results are always underlined and real-time reviewed. In this way, no information loss should occur.

2. *Code documentation has to be well-structured.*

   In order to make the code user-friendly, the scripts should be clearly commented and organized to improve readability and maintenance.

3.3.5 Non-functional requirements

1. *The code should be user-friendly.*

   The code should be quickly readable and effectively understandable from both a user and programmer point of view. It can also be provided of an intuitive Graphic User Interface (GUI) to simplify the interaction with the code itself.

3.4 Methodology

As already mentioned in section 1.2, the goal of this project is the implementation of a reinforcement learning based control architecture aimed at making a robotic arm able to autonomously navigate in a constrained pipe-like environment. In order to test the performances of the selected reinforcement learning algorithms, which should be value-function based, two experimental environments have to be built:

- MATLAB-based **simulation environment**, in which the kinematic model of the robot arm is taken into account to simulate its RL-based motion.
- **Physical setup**, which identifies a series of constrained pipe-like environments characterized by the presence of a goal and a planar three degrees of freedom manipulator, whose state is estimated through a visual-guided system: a fixed monocular camera has to be placed on the top of the setup, so that it can be used both to track the joints and end-effector location and to detect distances between points of interest (e.g. obstacles and goal location) and the manipulator structure. The robotic arm should be designed and built within RaM laboratory and it should be made of commercial and low-cost components. The hardware should be selected in a way that the communication with the employed RL architecture is reliable and almost real-time.

These two environments are taken as a reference to test two RL algorithms: Q-learning and SARSA, both with discretized state-space and with continuous state-space. If the discretized state-space is considered, a proper discretization level should be figured out to make the manipulator motion accurate and fast. On the other hand, if the continuous state-space is adopted, the deep neural network applied as a function approximation of the full environment state should be quick to be trained and able to provide a good estimate of the complete state-space. After the realization of the elements required by the reinforcement learning approach, the performances of the agents in the learning phase can be evaluated. The agents' performances are calculated by means of the following parameters:

- **Convergence rate**: required time before reaching the target with the same amount of actions and consequently with the maximum cumulative reward.
- **Obstacles avoidance**: ability in avoiding obstacles and, consequently, reduce the amount of collisions between the robotic structure and the obstacles markers.
- **Tuning of learning parameters**: convergence rate comparison based on the correspondent selection of learning elements, including exploration-exploitation trade-off and deep neural network design.
- **Adaptability to new configuration of the environment**: applied the acquired knowledge of the environment to quickly learn a similar but not equal configuration in a transfer learning perspective.

Thus, analyzing these evaluation criteria for each implemented algorithm, it is possible to assess their learning abilities, so that the most appropriate approach for autonomous navigation in a pipe-like environment can be found out.

3.5 Conclusions

This section focuses on summarizing the domain analysis and relative requirements in order to draw the conclusions about the design and implementation approaches that will be selected.

As already mentioned, value-function based RL approaches, i.e. Q-learning and SARSA algorithms, will be validated both in simulation and on a real setup. Their convergence rate and

their ability in dealing with over-constrained environments will be of central importance in assessing their performance and efficiency, both with discretized and continuous state-space. Beside that, learning parameters and reward function shaping will be appropriately investigated in order to find out the correlation between valuable accomplishments and correspondent tuning.

The setup that will be used to test the algorithms will consist of a planar three degrees of freedom robotic arm which will be placed in a constrained environment with virtual obstacles, so that actual damages to the setup will be circumvented. The manipulator will be actuated by three DYNAMIXEL AX12A motors, that will be connected to the USB serial port of a PC through a USB2Dynamixel device. All the features of interest (e.g. joints, obstacles, goal, etc.) will be provided with markers. These markers will be localized and tracked real-time through an external monocular camera, that will be settled at a certain height to record the whole scene.

Eventually, a full documentation on how the setup works will be provided such that every new user will not face problem in dealing with it. Beside that, the code will be provided with comments in all the sections and, more importantly, with a GUI, so that, without looking directly to the code, the full setup can be used.

4 Design and Implementation

As already specified, in this project a physical setup has been developed to effectively test reinforcement learning algorithms. To allow the interaction between the environment and the agent, which characterizes reinforcement learning methods, the environment is recorded by an external monocular camera, which is able to detect the features of interest (i.e. obstacles, goal, joints and end-effector of the manipulator). In this way, the distance between those features can be easily evaluated depending on their pixel location in the frame: e.g. collisions are detected when the manipulator markers are touching the obstacles markers, or, on the other hand, a success is identified when the end-effector marker has the same location of the goal marker. This approach permits to avoid the use of different hardware: actual obstacles that can damage the structure of the manipulator, position sensors to detect the distances and/or torque sensors which can determine a collision.

In order to properly design the required mechanical and software architectures, it is necessary to define RL elements presented in section 2.1 for the considered context. Thus, the following components should be clarified (see figure 2.1):

- **Environment**: the environment corresponds to all the setup components which are present in the considered system, i.e. manipulator structure, including motors and encoders, obstacles and goals markers as well as camera signals. In this context, the environment as well as the agent are described by a Markov Decision Process (see equation 2.3) and, consequently, they consist of a set of states (i.e. robotic arm joint positions), a set of actions (i.e. joint position displacements), a transitional function, which assigns a probability distribution to each state-action pair, and a reward function, which assigns a numerical value to each transition.

- **Agent**: the agent is able to receive some feedback signals from the environment thanks to the available sensors and, based on those feedback signals, it decides which action to take and, eventually, it translates those actions into actual motion for the robotic arm motors, trying to maximize the cumulative reward. In a complex robotic system like the considered one, the agent consists of the following components: a component which is able to make high level decisions according to the received input from the environment and a second component that implements those decisions in the environment. When selecting an action a, in each time step the agent follows a policy which associates to each state-action pair a probability to take action a from state s.

Summarizing, the boundary between the considered agent and environment can be specifically defined only in the moment in which states, actions and rewards are determined together with a decision making strategy (4).

This chapter presents the design and implementation phases aimed at creating the setup environment, whose intent is to test and compare four reinforcement learning algorithms:

- SARSA with discretized state-space.
- Q-learning with discretized state-space.
- Deep SARSA with continuous state-space.
- Deep Q-learning with continuous state-space.

All these algorithms will be tuned differently in order to figure out the connection between RL parameters tuning and actual performance of the methodology (see section 3.2.1).

In particular, the first section 4.1 deals with the actual design of the setup, in which the environment and its dynamics are globally defined. At this point, in section 4.2, the implemented RL frameworks are described and the differences between the simulation and the real-time environment are highlighted (section 4.2.3). Moreover, the experimental design approach is described (section 4.3) in order to give further details about how the different experiments are performed. In section 4.4, a Graphical User Interface is presented to make the setup easier to be utilized by different users. Eventually, in section 4.5, the design is assessed according to the requirements reported in section 3.3.

4.1 Setup configuration

In this section, the configuration of the setup is described in all its details. Firstly, in section 4.1.1, the design and implementation of the manipulator are presented together with its selected hardware components. At this point, the complete setup environment is described both in real-time (section 4.1.2) and its correspondent simulation (section 4.1.4).

4.1.1 Manipulator design and implementation

SOLIDWORKS has been used to modify the existing setup design (see figure 2.8) and to convert it into a planar three degrees of freedom robotic arm. In order to do that, the configuration in (2) has been modified to get the design in figure 4.1. In particular, the structure of the manipulator has been tilted of 90 degrees so that a planar layout is obtained. Moreover, the manipulator has been extended with a third link to get a three planar degrees of freedom layout.

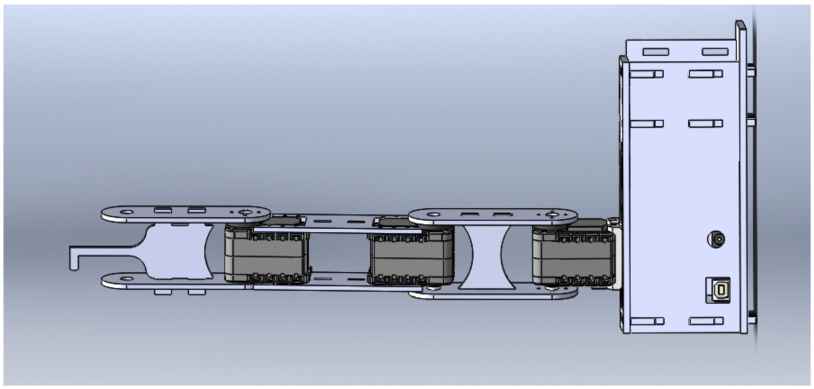

Figure 4.1: New design of the robotic arm

In the reported configuration, the manipulator is constituted by the following components:

- *Base*: a base is designed on which the robot links are connected. Inside the base all the electronic cables are hidden. On one side the base is fixed to a metallic structure provided with ducts through M6 bolts, to avoid possible tilt of the setup and to counterbalance the manipulator load. As a matter of fact, on the other side of the box the manipulator will be attached through the first servo-motor, which allows the first link to rotate clockwise and counter-clockwise. The box is assembled using four bolts and hooks and slots strategy as depicted in figure 2.9. These hooks slide into place constraining all the motions expect one (in sliding direction); if that motion is constrained externally, the connection is strong and reliable.

- *First link*: the first link is assembled to the base through the servo-motor and 9 bolts. On the tip of the link a second servo-motor is located to make the second link to rotate clockwise and counter-clockwise.

- *Second link*: a second link is attached to the first one and rotates thanks to the servo-motor placed at its bottom. A third servo-motor is collocated at its tip to allow the third link to rotate.
- *Third link*: a third link with the same configuration of the first one is connected to the second one. Its clockwise and counter-clockwise rotations are supported by the servo-motor placed at its bottom. At its tip, a simple end-effector is attached to have a better visualization of its location. This end-effector have been designed in different lengths (i.e. 80 mm, 100 mm and 120 mm) so that the robot arm can also reach further locations.

Due to their simple design, all the components can be easily laser-cut with the laser-cut equipment of RaM laboratory. In particular the box as well as the first link are laser-cut from a row panel 4 mm thick, while all the other components are 3 mm thick. This decision has been taken because the box and the first link are the ones which should support all the load and, consequently, should be more robust and ductile. Therefore, the obtained manipulator is shown in figure 4.2.

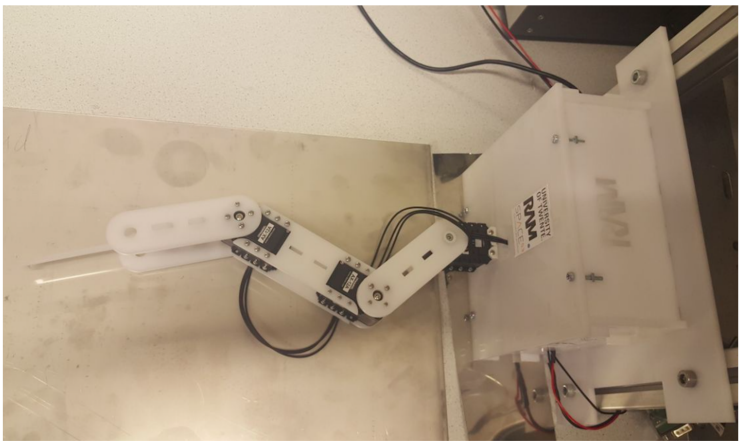

Figure 4.2: Final layout of the three degrees of freedom manipulator

As possible to notice from figure 4.2, the manipulator presents three degrees of freedom which are actuated by three DYNAMIXEL AX12A servo-motors (A.1). The motors make the robotic links able to rotate clockwise and counter-clockwise. As already mentioned in section 4.1, the interface between the servo-motors and the PC is implemented through a simple device called USB2Dynamixel (see figure 3.3 and check (29)). USB2Dynamixel is provided by 3P connectors and a USB input, that allow the PC to directly send position signals to the motors without the need of any other external control board. Since USB2Dynamixel cannot supply power to the motors, an external power supply has to be used (see figure in appendix A.2), to which the ground and power wires of the first motor are connected. The remaining motors are connected to the first in series through 3P cables, which transmit power, voltage and data signals.

As already specified in section 4.1, Robotis team (29) implemented a full open source library called DYNAMIXEL SDK (31) to create a communication interface between the motors and the PC serial port using USB2Dynamixel device. This library has been programmed with different programming languages (i.e. C#, C++, C, Java, Python, LabVIEW, MATLAB), so that each user can choose the one that is more suitable for his objective. Based on the fact that the control architecture for the robot arm designed by Berndsen in (2) has been developed in MATLAB, in this project MATLAB language (30) will be adopted as well to facilitate the integration between the existing architecture and the reinforcement learning algorithms. Even if MATLAB is not considered a pure programming language, but a tool for open-source engineering solutions, it is widely used in robotics applications because of its simplicity and its built-in toolboxes.

It is provided by machine learning and deep learning libraries as well as graphical interfaces and image processing solutions. Other than that, MATLAB permits to design representative and emblematic simulations of the setup that will be adopted, so that the implementation of the experimental phase of the project can be more straightforward (see section 3.3.3). Last but not least, MATLAB codes are easily understandable and modifiable by programmers of different backgrounds, not only programming experts, satisfying the requirement defined in section 3.3.5.

Therefore, MATLAB library provided by Robotis team will be adopted to communicate with the servo-motors. The procedure to install the library in different operating systems (i.e. Linux, MacOSX, Windows) is well documented in (31). In this project, in particular the following functions have been applied in order to connect the PC with the motors and interchange data:

- `openPort(port_number)`, which open the serial communication between the motors and the USB port according to the port path specified in `port_number`.
- `write1ByteTxRx()`, which orders to the DYNAMIXEL motor identifiable with a specific `DXL_ID` and with a predefined `PROTOCOL_VERSION` to communicate with the `port_number`, writing 1 byte to activate its torque at the correspondent address `ADDR_AX_TORQUE_ENABLE` and, consequently, set its status as being ready to move.
- `write2ByteTxRx()`, which orders to the DYNAMIXEL motor identifiable with a specific `DXL_ID` and with a predefined `PROTOCOL_VERSION` to communicate with the `port_number`, writing 2 bytes for two main purposes:
  - send the position value specified in `DXL_GOAL_POSITION` at the correspondent address `ADDR_AX_GOAL_POSITION` and, consequently, move to that position. The motor can get position values in the range 0-1023, where each unit corresponds to 0.29 degrees (20).
  - send the velocity value specified in `DXL_GOAL_VELOCITY` at the correspondent address `ADDR_AX_GOAL_VELOCITY` and, consequently, move with that velocity. The motor can get velocity values in the range 0-1023, where each unit corresponds to 0.111 rpm (20).
- `read2ByteTxRx()`, which orders to the DYNAMIXEL motor identifiable with a specific `DXL_ID` and with a predefined `PROTOCOL_VERSION` to communicate with the `port_number`, reading 2 bytes for two main purposes:
  - read the current position value specified at the address `ADDR_AX_GOAL_POSITION`.
  - read the current velocity value specified at the address `ADDR_AX_GOAL_VELOCITY`.

The full "read-write" code is provided in the appendix A.4 together with the control table of DYNAMIXEL AX12A servo-motors (A.3) in which all the features addresses are defined.

4.1.2 Setup environment design

As already mentioned, a setup should be designed in order to test RL approaches on the real environment. First of all, the environment must allow the interaction foreseen by reinforcement learning methodologies, as observable in figure 2.1: the environment must be designed in such a way that an external agent is allowed to perform an action and consequently return to the agent the relative reward and the new state. Since the environment must be episodic in nature (see pseudo-codes in 2.5 and 2.6), the agent should eventually reach a terminal state. Each episode must end whether the goal is achieved or otherwise. Guided by the nature of the objective of the environment, the terminal state is achieved only in the moment in which the agent succeeds in reaching the goal. Applying this approach, however, makes the agent not

able to end an episode if the target has not been reached. For this reason, it is advisable to add another terminal state which ends the episode after some interaction steps, e.g. if the agent takes too much time to reach the goal it can be reset to its initial configuration and start exploring from the beginning. Another aspect that should be taken into account when defining terminal states is collision. When the robotic arm collides with an environment obstacle, two choices can be made:

1. The manipulator continues exploring the environment avoiding collisions as much as possible.
2. The manipulator reset to its initial configuration just after colliding.

Both the situations will be analyzed in more details in section 4.2. In any case, at the end the manipulator should learn to reach the goal completely avoiding obstacles that can be present along its trajectory. A good reward function is an important factor for prioritizing trajectories that make the agent able to achieve the target, because for an agent the ultimate goal remains to get the most total reward at the end of the episode. For this reason the reward function must be defined by favoring the achievement of the final objective, avoiding convergences into local minima. Providing a single positive reward only when the goal has been reached would make it more complicated and would slow the progress down. For this reason, it is smarter to add smaller intermediate rewards to direct the agent to the choice of actions that drive it to reach the final goal. In addition to facilitate correct intermediate actions, it is also necessary to discourage actions that diverge from the objective by giving back negative rewards to the agent. A challenging reward function could be the negative distance between the end-effector and other points of interest (14), (17). In this way, if the end-effector is approaching the goal, it would get a higher reward than when it is far away from it. In the latter case, the reward would be more negative the further the end-effector is from the goal.

Therefore, since it is relevant to evaluate the distances between different features in the scene, the employment of easily detectable and reliable markers is of great importance. MATLAB has been selected as main software to implement the experiments, so the proposed markers (see sections 2.3 and 3.2.3) should be tested in MATLAB to evaluate which is more efficient in real-time applications. Since ARToolkit library is not available for MATLAB, just AprilTag and color markers have been assessed (see figure 4.3).

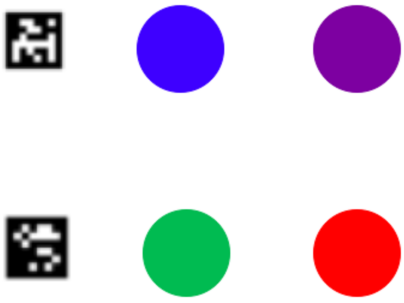

Figure 4.3: Test image for detection assessment of AprilTag and color markers in MATLAB

AprilTag markers

To evaluate AprilTag markers, it is sufficient to install the library which can be found in (32). This library provides a mex-function called `apriltags.mex`, which reads and detects AprilTag markers from an input grey-scale image, giving as output the pixel location of the centre and the corners of each AprilTag. Thus, the following pseudo-code can be applied to identify AprilTags in the image (5):

Algorithm 5: AprilTag detection

Read the input image;
Convert the input image into gray-scale through `rgb2gray`;
Detect the markers with `tags = apriltags(image);`
Check the euclidean distance between each tag coordinates.

Color markers

To test color markers, two different approaches have been executed. First of all, the MATLAB example proposed in (33) has been adopted, in which an input un-distorted image is converted to HSV color space to get its saturation channel and, consequently, threshold to detect all the colorful regions on white background, i.e. the color markers. Once those regions are obtained, a blob analysis is performed such that the largest connected components in the segmented image are classified as markers and, consequently, surrounded by rectangles which identify their centre and borders (see figure 4.4). The coordinates of the cited rectangles are used to evaluate the distances between each marker.

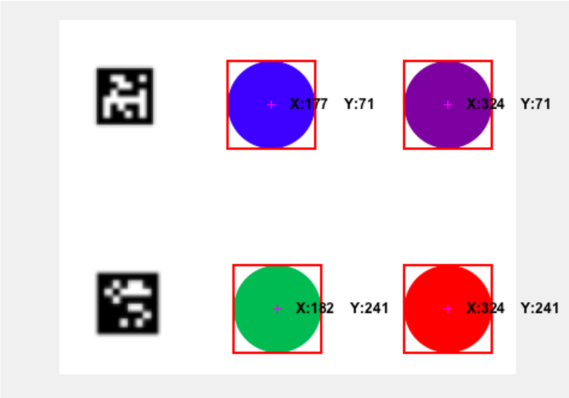

Figure 4.4: Color markers detection and pixel location of their centres

The second approach is similar to the previous one, but, instead of detecting makers of all possible colors, just RGB markers can be detected (e.g. in figure 4.4 just the blue, green and red marker). As a matter of fact, the input image is first converted to gray-scale and then each RGB component is subtracted to it and then combined together with the other obtained images to get a final image from which red, green and blue features can be easily extracted (see algorithm in 6 and full code in the appendix B.1).

Algorithm 6: RGB markers detection

Read the input image;
Convert the input image into gray-scale through `rgb2gray`;
Subtract the gray-scale image to each RGB image according to its channel, applying `imsubtract(input_image(:,:,RGB_channel), gray_image)` to obtain an image with just the selected RGB component;
Apply a medial filter to filter out noise with `medfilt2`;
Convert the resulting gray-scale image into a binary image with `im2bw` ;
Apply a blob analysis, removing all those pixels less than 300 pixels with `bwareaopen`;
Label all the connected components in the image with `bwlabel`;
Bound the labelled components into a rectangular box to get boundaries and centroids;
Check the euclidean distance between each marker coordinates.

The three approaches have been evaluated in terms of computational time, i.e. time required to detect the markers and evaluate the correspondent distance, obtaining the following average results (4.1):

Table 4.1: Assessment markers detection algorithms

Types of marker	Elapsed time (seconds)
AprilTag	0.3669
Color markers	1.4766
RGB markers	0.1356

As possible to notice, the faster of the three methodologies is the last one, which requires just 0.1356 seconds on average to detect the red, blue and green markers and evaluate their relative distance. Thus, these markers have been adopted in the real setup in order to real-time detect the location of the features of interest in the scene.

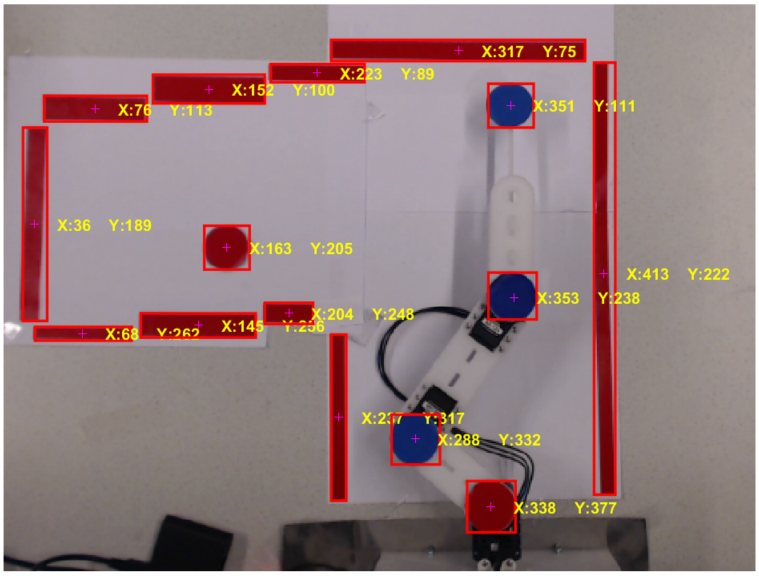

Figure 4.5: Final layout of the setup including markers as seen by the external camera

In this perspective, the setup acquires the configuration shown in figure 4.5. Due to the light conditions of the lab in which the setup is placed, green markers could not be tracked easily, so just the following blue and red markers have been employed:

- Two red round-shape markers, one to represent the goal position that the end-effector should reach and another one is placed at the base of the robotic arm to represent the origin of the frame of reference of the robotic arm.
- Red rectangular markers, which designate the obstacles that the manipulator should avoid. As a matter of fact, their location is chosen in a way to simulate a pipe-like environment as required.
- Three blue round-shape markers, which are settled on the manipulator joints and end-effector to visualize the location of each link in the cited frame of reference.

The location of the red markers is kept constant during the learning, so the latter are detected before starting the learning procedure itself and then saved in the workspace for future employments. On the other hand, the blue markers are real-time detected at each movement of

the robotic arm to evaluate their new location and consequently the distance between them and the static parts of the environment (i.e. goal and obstacles).

In order to avoid any calibration of the camera to get real-world distances (i.e. distances in the metric system), the location of the markers as well as their relative distances are evaluated in pixels. In this way, the camera application becomes plug-and-play and no further analysis have to be performed on the images taken by it.

4.1.3 Final setup architecture

In this section the final design of the full setup architecture is shown (see figure 4.6).

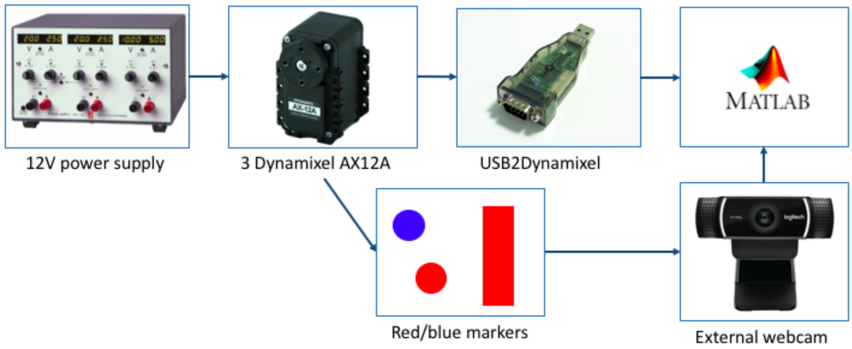

Figure 4.6: Final design of the setup architecture

As noticeable from figure 4.6, the final architecture of the setup consists of a power supply which provides 12 Volts to three DYNAMIXEL motors connected in series. The first DYNAMIXEL is also connected to a USB2Dynamixel device, which allows the interaction with MATLAB. On the other hand, markers are placed in the environment to represent obstacles and goal (i.e. red markers) and on the manipulator structure (i.e. blue markers) to obtain end-effector and joints positions. An external camera evaluates the coordinates of these markers in the shown reference frame as well as the distances of interest and it sends back these features as signals to the software architecture implemented in MATLAB. The latter is so the fulcrum of all the implementation, because the software architecture is what makes all the components to appropriately interact with each others and to reach the objective defined by the user. In this perspective, the setup environment can be defined and initialized according to the following pseudo-code:

Algorithm 7: Pseudo-code setup environment implementation and initialization

Connect the motors to the serial port;
Enable motor torque as in A.4;
Set motor velocity to a constant value as in A.4;
Move the robot to initial configuration `DXL_INITIAL_POSITION` as in A.4;
Connect with camera with function `webcam`;
Take picture of initial configuration with function `snapshot`;
Detect the markers in the picture with algorithm (6);
Classify the markers in origin, goal and obstacles.

Thus, the pseudo-code in 7 permits to integrate all the different components of the setup, in particular the encoders and the camera signals, to move the robotic arm to a specific initial po-

sition and get the details of that configuration with the camera, which can detect the obstacles, goal and manipulator location thanks to the red and blue markers (see picture 4.5).

4.1.4 Setup in simulation

After implementing a working setup, it is important to design a simulation on which reinforcement learning algorithms can be tested before switching to the setup itself. As specified in section 3.3.3,the simulation environment should be as much as possible representative of the real conditions in which the robot arm will operate, such that, when real experiments will be carried out, results will not be completely unpredictable.

The simulation environment have been implemented in MATLAB as well as all the control architecture. In order to simulate the motion of the robotic arm, it is necessary to make use of its forward kinematic model and plot the new configuration of the robot once an action is performed. Since the robotic arm is provided by three degrees of freedom and is planar, its forward kinematic model can be derived according to figure 4.7.

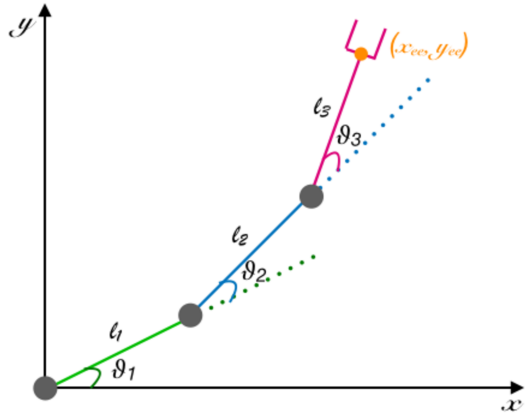

Figure 4.7: Three degrees of freedom planar manipulator

The forward kinematic model allows to evaluate the position of the end-effector (x_{ee}, y_{ee}) according to the pose of the whole manipulator, i.e. to the angles of the motors which are placed in the joints (gray dots in figure 4.7). Based on the nomenclature and frame of reference of figure 4.7, the following geometric relations can be derived:

$$\begin{cases} x_{ee} = l_1 cos(\theta_1) + l_2 cos(\theta_1 + \theta_2) + l_3 cos(\theta_1 + \theta_2 + \theta_3) \\ y_{ee} = l_1 sin(\theta_1) + l_2 sin(\theta_1 + \theta_2) + l_3 sin(\theta_1 + \theta_2 + \theta_3) \end{cases} \tag{4.1}$$

Where θ_1, θ_2 and θ_3 are respectively the motor position of the first, second and third joints, while l_1, l_2 and l_3 are the link lengths of the first, second and third links as possible to see in figure 4.7.

Thus, according to the joints position, it is possible to get the end-effector location. This knowledge is of fundamental importance also when distances of interest have to be evaluated. Of course in simulation no camera signals are used and, consequently, the distances between the manipulator structure and static objects in the environment (i.e. goal and obstacles) are obtained based on the forward kinematic model of the manipulator itself. In this perspective, the simulation model in figure 4.8 has been obtained and it is an accurate representation of the actual setup in figure 4.5.

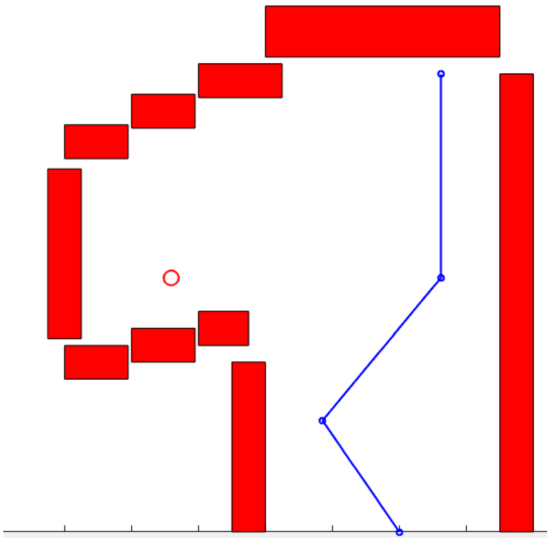

Figure 4.8: Representative setup in simulation environment (MATLAB). The goal is represented by a red circle and the obstacles are red rectangles. The manipulator structure is simply the combination of three lines representing the links and four circles, i.e. one circle for the end-effector and three circles for the manipulator joints.

In figure 4.8, the origin of the frame of reference is placed at the base of the robot exactly as in figure 4.7, so that the forward kinematic model can be easily applied to obtain the end-effector location. To get the cited layout, the following pseudo-code can be employed:

Algorithm 8: Pseudo-code simulation environment implementation and initialization

Init origin `init_point`, goal `goal_point` and obstacles `(obs_x, obs_y);`
Init initial configuration of the robotic arm `theta_start;`
Apply `fkine_3DOF(init_point,L,theta_start);`
Plot the robot as collection of blue lines (links) and circles (joints);
Hold on ;
Plot the obstacles as red rectangles and the goal as red circle.

In the pseudo-code in 8, the origin and the goal are identified by an array of two elements corresponding to the x and y coordinates of the points. On the other hand, the obstacles are determined by two matrices, one for the x-coordinates `obs_x` and one for the y-coordinates `obs_y`, in which each line corresponds to each obstacle relative x or y corner coordinates. Moreover, the forward kinematic model is collected in `fkine_3DOF` function (see equation 4.1), which gets as input the origin `init_point` where the robot should be located, the length of the links $L = [l_1, \quad l_2, \quad l_3]$ and the initial joint configuration $theta\_start = [\theta_1, \quad \theta_2, \quad \theta_3]$ and gives as output the $x-y$ coordinates of all the joints, of the centre of mass of the links and of the end-effector.

4.2 RL architecture design

This section focuses on the design of the RL architecture, i.e. SARSA and Q-learning in both discretized and continuous state-space, to make the presented robotic arm able to autonomously learn the environment in which it operates (simulation or setup environment).

For each algorithm, the tuning choices are described in details as well as their correspondent motivations. In the discretized algorithms (section 4.2.1), i.e. Q-Learning and SARSA with the realization of Q-table, it was necessary to discretize the values of the environment states and

actions and reduce their number; as a matter of fact, each possible state-action pair estimated value should be allocated in a cell inside the Q-table during the learning phase. By discouraging and reducing the number of parameters that represent the state of the environment, information is lost and consequently the states may lose Markov property since the same state parameters could result in different environment configurations. Because of the discretization and reduction of inputs, discretized methods may not be able to achieve excellent results; for this reason a second more advanced method has been implemented, Deep RL (section 4.2.2), which replaces the Q-table with a neural network to approximate the value-function. As already defined in detail (see section 2.1.8), thanks to the use of the neural network, there is no real materialization of the value of every possible state-action pair, but the estimation of the behavior of the value-function is done through the neural network by modifying its internal weights. The neural network receives continuous input values and does not require any discretization.

Although different algorithms have been implemented, the mode of interaction between the agent and the environment remains of episodic nature for all the situations (see pseudo-codes in 2.5, 2.6 and algorithms 3 and 4). In particular, all the methods present two nested loops, since each episode (external loop) is constituted of a particular amount of iterations/steps (internal loop). As already mentioned in section 3.2.1, each episode must be characterized by a predefined number of iterations: each iteration corresponds to an action that the agent is allowed to perform. Due to this correlation, the number of iterations should be well tuned to avoid the following circumstances:

- Infinite iterations, which could make the agent to follow non-optimal policies related to local minimum areas.
- Not enough iterations, which could make the agent not to have sufficient available actions to reach the goal, since the number of iterations is equal to the number of available actions per episode.

According to this analysis, the number of iterations have been selected to be 250, considering that the minimum number of actions required to reach a goal placed as in figure 4.5 corresponds on average to 80.

At the beginning of each new episode, the manipulator is reset to an initial configuration which is defined in advance, initializing in this way the initial state of the agent in the environment. In order to end an episode, the agent should reach the goal avoiding the obstacles. Although, if a collision occurs before reaching the goal, two strategies have been tested:

- Reset after collision: the episode ends if a the manipulator hits an obstacle and so the agent returns to its initial state before ending the iterations.
- Move one step backwards: if a collision occurs, the iterations continue and the manipulator moves back to its previous state, i.e. the state before the collision, and it should select a new action from that state.

Summarizing, this section presents the design of the RL architecture, both with discretized state-space (section 4.2.1) and continuous state-space (section 4.2.2). Eventually, the differences between the two implementations on the real setup and in simulation (section 4.2.3) are highlighted.

4.2.1 Discretized RL algorithms

As already mentioned in sections 2.1.5 and 2.1.6, SARSA and Q-learning make use of a Q-table to store the estimated values of all the possible state-actions pairs. In this perspective, a smart discretization approach should be figured out in order to circumvent the curse of dimensionality issue typical of the robotics domain (see section 3.1 and (9)(10)).

State discretization

Servo-motors with position control are adopted and, consequently, the state parameters correspond only to the possible joint angles, i.e. the three motors positions that the manipulator can present in the considered environment. The state-space dimension depends mainly on two parameters:

- **Size of the workspace**: the definition of a specific workspace in which the robot arm can operate is of great importance to have a better visualization of the motion the robotic arm can perform and of the regions that the end-effector can actually reach in the environment. Thus, all the servo-motors mounted on the robotic arm have been limited to specific position ranges to avoid undesired configurations and possible collisions between different parts of the manipulator structure.
- **Discretization level**: to select a proper discretization level, it is necessary to look at the environment in which the robotic arm will operate. In a L-shaped pipe-like environment (see e.g. figure 4.5), the manipulator should be able to reach quite far targets with its end-effector, maybe rotating the latter of 90 degrees with respect to the second link and, at the same time, the first two links should allow the end-effector to enter and explore the other curve as fast as possible. In this context, both the first motor and the third motor should have a high resolution so that they can make accurate movements, while, on the other hand, the second motor, since it is attached to a longer link with respect to the first one, should present less discretization to allow the end-effector to enter the curve faster.

According to the considerations previously made, the states of the agent can be selected as follow, obtaining the workspace shown in figure 4.9:

Table 4.2: Definition of the states of the agent

Parameter	Range [motor units]	Discretization sample [motor units]	State size
θ_1	550-900	5	71
θ_2	400-1000	8	76
θ_3	180-900	5	145
		Total state size	*782420*

As possible to notice from table 4.2, the three states are combined together to get a final state matrix of dimension 782420x3, in which the rows corresponds to all the possible combinations of the three column elements, i.e. the states θ_1, θ_2 and θ_3. These states are in motor units and each unit is equivalent to approximately 0.29 degrees (20); thus, the first and the third state present a discretization sample of around 1.45°, while the second is sampled every 2.32°, allowing the end-effector to move faster towards the right/left side of the pipe depending on the goal location.

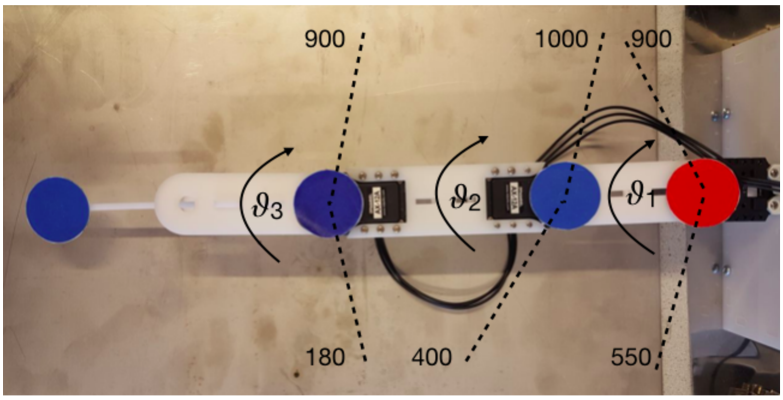

Figure 4.9: Definition of the robotic arm workspace. The lines on the joints represent the motion range of each joint and, consequently, of each servo-motor of the manipulator, according to the state definition in table 4.2.

Discretized action-space

According to the described discretization approach, an action vector should be selected to make the robot arm able to move inside the environment. For each joint of the robot, some actions have to be defined to increase or decrease their angle. In this way, the manipulator acquires the ability to move clockwise or counter-clockwise depending on the angle sign and on its state range. In this context, the action vector is discretized as well so that a finite number of state-action pairs can be identified and then stored in the Q-table. The selection of the actions depends mainly on two parameters:

- **Discretization level:** a selection of actions whose values are smaller than the discretization sample would make the agent not to change its current state. So, the actions should be selected to be at least equal or bigger than the discretization sample for each state.
- **Workspace:** a selection of actions whose value would bring the manipulator out of its predefined state ranges is not preferable, because in those situations the robot would not move and would stop at the current state.

Based on the previous analysis, the agent state in the environment should be interpolated such that, when the robot performs an action, it ends up in a new environment state depending on the discretization. To implement the stated interpolation, it is necessary to figure out which is the state closer to the current robot configuration. In this perspective, the euclidean distance between the current configuration and the available discretized states is evaluated with:

$$current\_state = min[(state\_matrix - theta\_matrix)^2] \tag{4.2}$$

In the reported equation, *state_matrix* corresponds to the full state matrix (see table 4.2), while *theta_matrix* is the matrix containing the current configuration of the robot, i.e. $theta = [\theta_1, \ \theta_2, \ \theta_3]$ repeated as many times as the available number of states. In this way, each element of the state matrix can be actually compared to the current configuration of the agent, calculating the euclidean distance between each element of the state matrix and the current *theta*. The minimum obtained value of that distance will actually identify the closer environment state for the current configuration of the manipulator.

Actions selection strategy

In order to choose the action to be implemented, the ϵ-greedy action selection strategy (see section 2.4) have been partially modified. The standard ϵ-greedy algorithm is as follow:

Algorithm 9: ϵ-greedy action selection algorithm

```
if rand() > ε then
    Select greedy action derived from Q;
else
    Select a random action;
end
```

Even if ϵ-greedy strategy presents a good balance between exploration and exploitation during the action selection, it has one main disadvantage in the exploration phase: it selects equally among all the available actions (4), i.e. all the actions are equally attractive for the agent. This scenario could be undesirable in the moment in which some actions would bring the agent to reach environment states which should be avoided or when a full environment exploration should be guaranteed. In order to solve those issues, the action selection strategy has been modified as described in algorithm 10.

Algorithm 10: Modified ϵ-greedy action selection algorithm

```
if rand() > ε then
    Select greedy action derived from Q;
else
    if iteration == 1 then
        Select a random action;
    else
        Check the most and less selected actions;
        if The most selected action is unique then
            Select randomly among all the actions except the most selected;
        else
            if The less selected action is unique then
                Select that action;
            else
                Select randomly among all the less selected actions;
            end
        end
    end
end
```

According to the modified ϵ-greedy action selection proposed in (10), all the selected actions are stored in MATLAB workspace, such that, once the agent is exploring, instead of selecting a random action uniformly among all the available actions, it selects a random action uniformly among all the actions except for the most selected action. If more than one action have been selected the most, the agent chooses the less selected action or, if even the latter is not unique, it selects randomly among all the less selected actions.

In this way, the agent is allowed to explore the environment in a more uniform way and, since RL algorithms have an episodic nature, problem concerning resets of the random generator in MATLAB are avoided. This latter issue makes the MATLAB function `rand()` to returns the same result any time it is executed at the beginning of each episode. Thus, since a completely random exploration is not possible in a PC but it is always pseudo-random, it should be at least supervised such that the agent can also explore the whole environment with a more homogeneous actions selection.

In this context, it is so necessary to discover an optimal exploration-exploitation trade-off (see requirement in 3.3.1), so that the agent does not waste time in exploring remote areas far from the goal, but, at the same time, it does not exploit too much a specific policy, because, if not, it could converge to a local optimal policy, which is not actually the generalized optimal one. In particular, as mentioned in (4), to obtain higher rewards, the agent has to select actions that it discovered in previous trials and that allowed it to get positive reward in the past, but, to determine those actions, the agent must test actions that it has never tried beforehand. Therefore, to learn that some actions may be better than others, the agent has to sufficiently explore the environment. To converge to optimal policy, the exploration period must be gradually reduced, such that the algorithm can behave greedy letting ϵ converging to zero and, once the last episodes are reached, the agent is allowed to just exploit the optimal policy it discovered. Thus, at the beginning of the learning phase, a bigger $\epsilon = 0.3$ has been selected (see results in section 5.1.1 and figure 5.4 for comparison about tested ϵ-values), such that the agent can explore 30% of the time in each episode and, at the end of each episode, its value is decayed by a factor of 0.998. In this way, $\epsilon \rightarrow 0$ when episode $\rightarrow$ final episode.

Moreover, during the training period, convergence to the (sub)optimal policy is identified in the moment in which the agent reaches the goal for 50 successive episodes with the minimum number of actions. If this situation is reached, the exploration phase is completely stopped, i.e. $\epsilon = 0$, and the agent is allowed to exploit just the learned policy.

In order to find the greedy action specified in (10), a Q-table must be available in MATLAB workspace which stores the value of all the available state-action pairs according to the discretization approach. Thus, this Q-table should have a number of rows equals to the available states and how many columns as the available actions. As a result the implemented Q-table would be 782420x6.

As specified in both 2.5 and 2.6, at the beginning of the algorithm, $Q(s,a)$ should be initialized and then updated according to the equations 2.12 and 2.13 for SARSA and Q-learning respectively. In the literature, the Q-table is usually initialized arbitrarily (4), e.g. $Q = zeros(states, actions)$, but, to make the initial exploration phase more goal-oriented, it may be initialized based on the reward function. In this way, states which bring the agent closer to the goal present higher values with respect to the states which make the agent to move further from the goal. Of course, since the value of each state-action pair has not been estimated yet during the initialization of the algorithm, each action presents the same value for each state, i.e. the resulting Q-table is just the repetition of the reward matrix in all its columns (see equation 4.3 and code in appendix B.2).

$$Q = \begin{bmatrix} R^1_{nx1} & \dots & R^m_{nx1} \end{bmatrix} \tag{4.3}$$

Where n is the number of available states and m is the number of available actions.

Consequently, each action for each state is initialized equally and, so, this aspect should be taken into account during the exploitation phase: when the agent in state S has to select the greedy action A^* derived from Q (see algorithm 10), it should not always select the first action for that state, i.e. A_1, but it should select randomly among all the available actions with the highest Q-value, so that, after some iterations, the optimal action A^* for that state S can be actually highlighted by a higher value estimate. Following the described analysis, a second version of the modified ϵ-greedy algorithm is provided in 11.

According to algorithm 11, in the exploitation period the greedy-action is selected only when at the current state S the available actions have different value estimates in the correspondent Q-table. Eventually, at this point, the index of the chosen action is returned so that the manipulator can actually move the correspondent motor to the specified position.

Algorithm 11: Second modified ϵ-greedy action selection algorithm

```
if rand() > ε then
    if all(Q(S,:) == Q(S,A_1)) then
        Check the most and less selected actions;
        if The most selected action is unique then
            Select randomly among all the actions except the most selected;
        else
            if The less selected action is unique then
                Select that action;
            else
                Select randomly among all the less selected actions;
            end
        end
    else
        Select greedy action derived from Q;
    end
else
    if iteration == 1 then
        Select a random action;
    else
        Check the most and less selected actions;
        if The most selected action is unique then
            Select randomly among all the actions except the most selected;
        else
            if The less selected action is unique then
                Select that action;
            else
                Select randomly among all the less selected actions;
            end
        end
    end
end
```

In section 5.1.1, a graph is reported which compare the action selection distribution among standard ϵ-greedy algorithm 9 and modified ϵ-greedy algorithm 11 (see figure 5.3) to check if the modified strategy actually provides a more uniform selection of the actions in each episode.

RL parameters selection

Once the manipulator has moved, the new state S' of the agent and the reward received for performing that action are observed. As already specified beforehand in section 3.2.1, the reward could be a simple bonus when reaching the goal and a penalty when a collision occurs, i.e. binary/sparse reward, or it can depend on the distance between the end-effector and the goal. This reward function is ideal when the manipulator has to learn an environment like the pipe one shown in figure 4.5, because it guides the agent towards the goal in a more efficient way. If the manipulator was rewarded only when the goal has been reached, as in the sparse reward case, it would not receive any feedback concerning the distance with respect to the goal. In this research, it is assumed that the agent knows the Euclidean distance between the end-effector and the goal. This is a realistic assumption because of the fact that the agent makes use of the markers detection signals provided by the camera. In this perspective, the following distance reward functions have been investigated and tested:

- **Euclidean distance**: simple Euclidean distance between the end-effector and the goal is evaluated according to equation 4.4:

$$Euclidean\_reward = -\sqrt{(x_{ee} - x_{goal})^2 + (y_{ee} - y_{goal})^2} \tag{4.4}$$

- **Exponential Euclidean distance**: exponential Euclidean distance between the end-effector and the goal can be evaluated as:

$$Exponential\_reward = \alpha * e^{\gamma * (|-Euclidean\_distance|)} - offset \tag{4.5}$$

where γ is the decay rate of the exponential, while α and $offset$ are a multiplication factor and an offset respectively used to fit exponential units to desired values. The decay rate can be selected looking at the following plot:

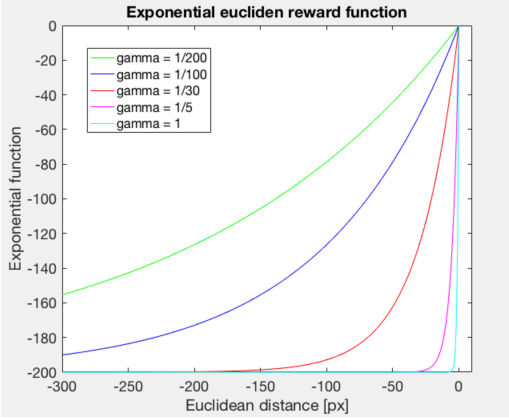

Figure 4.10: Comparison between different exponential decays. The x-axis represents the euclidean distance between the end-effector and the goal in pixels, while the y-axis corresponds to the relative exponential reward value.

Thus, the smaller the decay the higher the slope of the exponential curve. In this perspective, all the states closed to the goal would present much higher reward than the one far away from it. In this case, a decay factor of 0.01 has been selected (blue line in figure 4.10).

- **Gaussian Euclidean distance**: similar to the exponential Euclidean distance, the Gaussian Euclidean distance presents the following form:

$$Gaussian\_reward = \frac{\alpha}{\sigma\sqrt{2\pi}} e^{-\frac{|-Euclidean\_distance|}{2\sigma^2}} \tag{4.6}$$

In this case, two parameters have to be tuned, i.e. the variance σ^2 and the multiplication factor α. The multiplication factor is needed just to adjust the order of magnitude of the function, while, on the other hand, the variance imposes the slope of the Gaussian function as shown in figure 4.11.

Thus, as noticeable, decreasing the variance increases the slope of the Gaussian function. As a result, in reward context, just the states really closed to the goal would be characterized by larger rewards. In this case a variance of 100 has been selected.

After defining all these parameters, it is important to highlight how the two algorithms, i.e. SARSA and Q-learning, estimate the value of each state-action pair. SARSA algorithm updates the Q-table according to the equation in 2.12, while Q-learning follows equation 2.13. Thus, the main difference is that Q-learning estimate $Q(S_t, A_t)$ using the Q-value of the next state

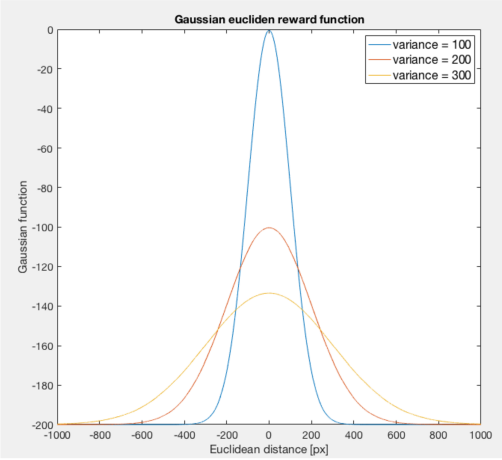

Figure 4.11: Comparison between different variances in Gaussian function. The x-axis represents the euclidean distance between the end-effector and the goal in pixels, while the y-axis corresponds to the relative Gaussian reward value.

S_{t+1} and the greedy action a, so it assumes to follow the optimal policy even if it is not. On the other hand, SARSA updates the Q-table according to the Q-value of the next state S_{t+1} and the next action from that state A_{t+1}, so it always assume to follow the current policy. This difference disappears in the moment in which the current policy corresponds to the optimal one. Even though, both the update rules consist of the following parameters that should be tuned:

- α (equations 2.12 and 2.13), which is the learning rate. It determines how the new learned information will be weighted against the old information. A factor of 0 would prevent the agent from learning, on the contrary a factor of 1 would cause the agent to be interested only in recent information. Since the $Q-$table is initialized as the reward function in a goal-oriented perspective and the initial phase of the learning is not complete exploration, it is meaningful to select a learning rate which is high (e.g. 0.99) so that the learned information is prioritized.

- γ, which is the discount rate parameter (see equations 2.8, 2.12, 2.13 and section 3.2.1). It represents the contribution given by the values of the successive states in the evaluation of the value of the current state-action pair. Thus, $\gamma = 1$ means that only the cumulative future rewards play a role in updating $Q(S_t, A_t)$, while $\gamma = 0$ takes in to account only the immediate rewards. As specified in the literature (4), discounted rewards are useful in the moment in which the algorithm does not present episodic nature and, so, the interaction between the agent and environment is continuous and does not stop when a specifid situation is faced. On the other hand, undiscounted rewards can be applied when the algorithm is episodic as in this case. Exactly for this reason, a discount factor of 0.9 has been selected, because it has been found in the literature (4) to be a good trade-off between the intrinsic episodic nature of the algorithm and the discretization and reduction of the environment, that partially affects the Markovian property.

All the tuning choices are summarized in table 4.3:

Table 4.3: Summary of implemented tuning choices for learning parameters

Parameter	Value	Description
Iterations	250	Number of iterations per episode. An episode can last as maximum 250 iterations.
ϵ	0.3	Initial exploration index. It represents the probability of selecting a random action with respect to selecting the greed-action (see section 5.1.1).
ϵ decay	0.998	Decay of ϵ at the beginning of each new episode. So that almost full exploitation is reached in the last episodes. As a matter of fact, $0.3 * (0.998^{2000}) = 0.0055$, which is almost equal to zero exploration after 2000 episodes.
α	0.99	Learning rate. It represents the updating index of the value of a state-action pair.
γ	0.9	Discount index. When assessing the value of a state-action pair, it declares how important is the value of the future states.

The full MATLAB code of both SARSA and Q-learning is provided in the appendix B.3 and B.4.

4.2.2 Deep RL algorithms

Since Markov property is affected by the reduction and discretization of the environment, it is not easy to obtain excellent results in those conditions. Unfortunately, discretization is inevitable, since in its absence, the high memory requirement for storing all the state-action pairs would be prohibitive, but, above all, the number of data and time required to accurately estimate each single state-action pair would be too high. The solution to the problem is to combine reinforcement learning, both Q-Learning and SARSA, with a function approximation method. The implementation of Deep RL with experience replay has been selected (see section 2.1.8). In the Deep RL method, the Q-table is replaced with a neural network for the approximation of the value-function. The adoption of a neural network allows to use the complete state provided by the environment without reductions and discretizations. Of course, since the encoders of the motors have precise resolution of 1 motor unit, i.e. 0.29 degrees, the full state space is intrinsically discretized with a sample of 1 motor unit.

Concerning the actions choice, Deep RL adopts an ϵ-greedy policy, similar to the policy used by the first implementation (algorithm 11). In this case, however, the values of the state-action pairs are obtained by performing a forward step in the neural network. During the learning phase, the implemented method estimates the target values for the calculation of the loss function (see equation 2.14), following the logic proposed by Monte Carlo methods (see section 2.1.4) which provides the estimate of the expected return of a state-action pair, using the information stored in a class called Replay Memory.

In Deep Q-Learning, at each iteration the agent stores a new tuple in replay memory containing: the previous state, the performed action, the obtained reward and the new visited state. In Deep SARSA also the next action from the new visited state is stored (i.e. A_{t+1}). See table 4.5 to get further details.

At this point, by randomly choosing a replay memory tuple, it is possible to estimate the expected return (equation 2.8). If the tuple selected in the Replay Memory belongs to a not yet completed episode, the calculation of the expected return is totally or partly done using the estimates of the neural network. In this perspective, instead of training the network with only the just performed transition, as implemented in algorithms 1 and 2, the training is based on

Table 4.5: Values contained in each tuple of the Replay Memory

Class object	Parameter	Description
obj.S	state 1	State of the environment at instant t
obj.A	action 1	Index of the performed action at instant t
obj.R	reward	Obtained reward when action 1 is performed
obj.S_new	state 2	New successive state at $t+1$, reached executing action 1 at state 1
obj.A_new	action 2	New action selected from state 2 (only in Deep SARSA)

a subset of randomly selected transitions from the Replay Memory. This type of update takes the name of updating with mini-batch (see section 2.1.8) and brings a considerable advantage with respect to the basic method, since, in the learning phase, fewer episodes are required to reach convergence.

To create the neural network, MATLAB Neural Network toolbox has been employed. This package offers a wide variety of architectures and training functions for modeling complex non-linear systems in a simple way, using artificial neural networks of different type. MATLAB implements different optimization algorithms for gradient descent learning, including `traingdm` (35), which is a gradient descent function with momentum back-propagation and has been proved to be 70% more accurate with respect to the standard gradient descent back-propagation functions (36). As a matter of fact, since momentum back-propagation is applied, the network becomes able to respond both to local gradients and to current trends in the error surface (35); consequently, the momentum behaves as a low-pass filter which filters out all the small features in the error surface, allowing the network to bypass possible local minimum points. `traingdm` makes use of the following formula to change the weights w of the neural network (35):

$$d(w) = mc * d(w_{prev}) + \frac{lr * (1 - mc) * d(perf)}{d(w)} \tag{4.7}$$

In equation 4.7, the next parameters are present:

- $d(w)$, which represents the change of the NN weights.
- $d(w_{prev})$, which is the previous change to the weight.
- mc, which corresponds to the momentum constant. $mc = 0$ means no momentum, while $mc = 1$ means that the relative network would not learn from the local gradient.
- lr is the learning rate of the network. A too high learning rate cannot ensure convergence of the network, because the changes of weight values could be so huge that the gradient descent can overshoot the minimum, making the loss function even worse. On the other hand, a small learning rate is more reliable but it can make the training slower because the steps required to minimize the loss function are usually of infinitesimal order. In this case a learning rate of 0.012 has been selected since the resulting network can be trained sufficiently fast (see results in table 5.2).
- $\frac{\partial(perf)}{\partial w}$ corresponds to the derivative of the performance with respect to the weight w. The performance is evaluated according to the performance function, which in this case is the mean-squared error as specified in equation 2.14.

To initialize the Deep RL agent, in addition to the action-space attribute, containing all the possible actions that the agent is allowed to perform, and the learning parameters ϵ, ϵ-decay, α and γ (see table 4.3), additional parameters must be provided:

- Maximum dimension of the Replay Memory class, i.e. `memory_size`.
- Maximum dimension of the mini-batch, i.e. `batch_size`.

As already mentioned, to perform an action selection, Deep RL implements an ϵ-greedy policy in a way similar to the methodology applied for the discretized state-space case (see algorithm 10). In this case, the estimates of the values are obtained through the neural network (see algorithm 12).

Algorithm 12: Deep RL ϵ-greedy action selection algorithm

```
if rand() > ε then
    Select greedy action derived from Q applying argmax_a' Q(S, a');
else
    if iteration == 1 then
        Select a random action;
    else
        Check the most and less selected actions;
        if The most selected action is unique then
            Select randomly among all the actions except the most selected;
        else
            if The less selected action is unique then
                Select that action;
            else
                Select randomly among all the less selected actions;
            end
        end
    end
end
```

As possible to notice from algorithm 12,if the action with higher expected return (i.e. greedy-action) is selected, the deep ϵ-greedy algorithm invokes an `argmax` function that requests as input parameter the state of the environment S, used internally as input to the neural network Q, in order to obtain the values related to every possible action that can be performed from that state. By executing that function, it is possible to interact with the modeled neural network, making a forward step with the environment state as input. The obtained output corresponds to a vector of length equal to the number of executable actions. Once the vector is evaluated, the function returns the index of the action with a higher value (greedy-action).

Once the selected action is performed, the new state and the relative reward are observed. Thus, the new tuple (i.e. (S_t, A_t, R_t, S_{t+1}) for Deep Q-learning and $(S_t, A_t, R_t, S_{t+1}, A_{t+1})$ for Deep SARSA) is added to the Replay Memory by calling the function `store_transition`. `store_transition` concatenates each parameter of the tuple to the previous ones applying the MATLAB code provided in appendix B.5. In case the insertion of the new tuple exceeds the maximum size of the Replay Memory `memory_size`, the maximum size is maintained by eliminating the exceeding tuples following FIFO logic.

At this point, the action-value function Q is learned based on a randomly selected subset of tuples from the Replay Memory. The amount of samples in the subset are determined based on the chosen batch size. Then, for each selected tuple, the target value is calculated (see equa-

tions 2.15 and 2.16) in order to determine the error of the network outputs to be minimized by adjusting the weights. The error is simply the mean-squared error between the evaluated target and the current value-function $Q(S_t, A_t)$, as specified in the loss function (equation 2.14). Eventually, the network can be trained using the `train` function in MATLAB, providing as parameters the state vector and the target.

The modeled neural network, represented graphically in figure 4.13, is a deep feed-forward neural network and is made of three components, one of which is a fully connected hidden layer:

- **Input**: the input is made up of three neurons, one for each state parameter of the environment (see table 4.2).
- **Hidden layer**: the hidden layer consists of 300 neurons with hyperbolic tangent activation, so following the equation:

$$f(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{4.8}$$

  The cited equation results in the plot reported in figure 4.12.

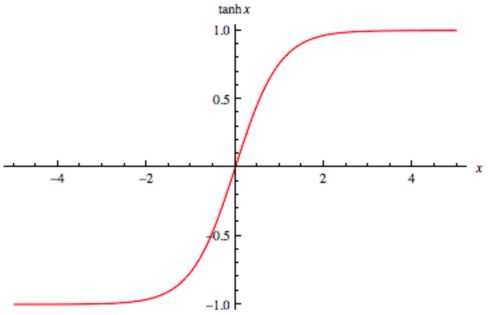

Figure 4.12: Hyperbolic tangent function behavior

  This activation function gives output in the range [-1, 1], so negative inputs to the layer will correspond to negative outputs, while positive inputs to positive outputs. In this way, the sign is preserved.

  The number of neurons in the hidden layer can be selected according to the considered number of inputs and outputs ((34) and see results in section 5.1.1).

- **Output layer**: the output layer provides a neuron for every possible action (in figure two actions for each joint, i.e. six neurons) with linear activation, since the outputs will have to estimate the values of the state-action pairs. The linear activation function combined with nonlinear activation functions in the hidden layer, hyperbolic tangent in this case, provides good results for nonlinear regression.

The resulting neural network is considered to be deep because the input and the output layers are connected through another hidden layer. Both the hidden layer neurons and the output layer neurons are biased, with biases initialized at 0 (b in figure 4.13 are initialized at zero). The values assumed by the net weights (w in figure 4.13) are regularized by using a momentum constant of 0.9 to avoid local minima, as mentioned beforehand, and they are initialized at zero.

All the selected parameters for the neural network are summarized in the table below:

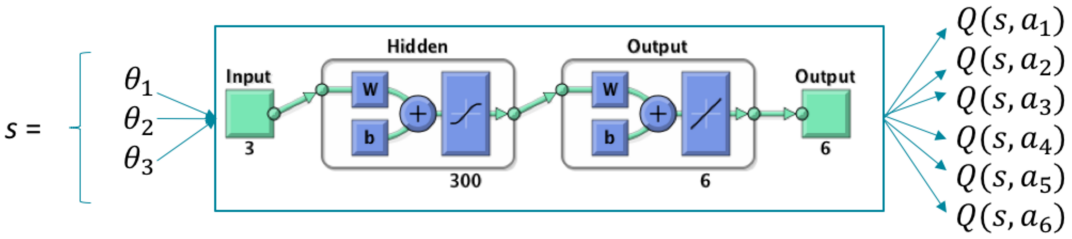

Figure 4.13: Graphical representation of the adopted feed-forward neural network

Table 4.7: Summary of implemented tuning choices for deep NN parameters

Parameter	Value	Description
Dimension input layer	3	Number of neurons that constitute the first layer of the neural network, amount equivalent to the number of available states of the environment.
Dimension hidden layer	300	Number of neurons that constitute the hidden layer of the neural network, single hidden layer of the network (see section 5.1.1 for further details).
Dimension output layer	6	Number of neurons that constitute the output layer, amount equivalent to the number of available actions, in figure 4.13 corresponds to 6 (see equation 5.1).
Learning rate	0.012	Learning rate of the neural network, given as an input.
Momentum constant	0.9	Parameter needed in `traingdm` function. It acts as a filter of local irregularities to avoid convergence to local minima.

Concerning the parameters of the Deep RL algorithms with experience replay, the same choices summarized in table 4.3 remain valid. The only parameters that still have to be tuned are the ones related to the Replay Memory. As a matter of fact, the maximum dimension of the Replay Memory has been selected to be `memory_size = 250'000`; this means that, in the worst case, the Replay Memory can store 1000 episodes, since each episode consists of 250 iterations (i.e. $250 * 1000 = 250'000$). The updates of the weights of the network are executed with mini-batch of dimension `batch_size = 75`. Since the network is trained based the available mini-batches, this number has been selected making the following considerations:

- Small `batch_size` corresponds to a not accurate training. Since the batches are randomly selected, if a small sample size is considered it can be related just to a specific portion of the environment and, consequently, the network would be train only for specific state-space.
- Large `batch_size` can avoid the previous problem, but it makes the training period much longer, since the network should be trained for more data.

Thus, a good trade-off of 75 has been selected not to elongate the training period and to ensure training on larger areas of the environment state-space.

The full MATLAB codes of the implemented Deep SARSA and Deep Q-learning are provided in the appendix B.6 and B.7.

4.2.3 RL architecture on real setup vs simulation environment

As already specified in section 4.1.2, the distance between the end-effector and the goal is evaluated through the camera, which detects the location of the goal and manipulator markers. On the other hand, in simulation environment, since no camera signal is available, the end-effector location is calculated according to the kinematic model in 4.1 and, based on that, its distance with respect to the predefined goal location in $x-y$ plane can be determined. Thus, since the reward function is based on the Euclidean distance between the cited points of interest, it is evaluated in a different way according to the employed environment.

Even though, in the discretized state-space case, the Q-table should be initialized based on the selected reward function, as specified in section 4.2.1. In this perspective, camera signals cannot be utilized because it is inefficient to assess in advance all the possible configurations of the manipulator and, consequently, evaluate the Euclidean distance between the end-effector and the goal in all the stated configurations. Therefore,the kinematic model is also employed in the setup environment in order to initialize the reward function and, consequently, the Q-table, making the necessary adjustments for the unit of measures: to apply forward kinematics, the joints positions should be in degree, so a conversion is applied to make the motor units conforming to the kinematic model; at the same time, the links lengths are kept in pixel coordinates, knowing that each pixel corresponds to 0.8824 millimeters in the employed PC display, since the distances with the camera are evaluated in pixels, which do not require camera calibration.

4.3 Experimental design

In order to test the different RL algorithms, it is necessary to design and plan the series of experiments that will be performed.

The first configuration of the environment that will be considered is the one shown in figure 4.5, on which all the four implemented algorithms will be tested, i.e. discretized SARSA, discretized Q-learning, Deep SARSA and Deep Q-learning. On this environment, according to the methodology proposed in section 3.4, each algorithm will be assessed through the following parameters:

- **Convergence rate**: required episodes before reaching the target with the same amount of actions and consequently accumulating the same maximum reward. If convergence rate is reached, (sub)optimal policy is guaranteed to be found.
- **Obstacles avoidance**: ability of avoiding collisions with obstacles.
- **Number of successes**: average number of successes per learning period. An episode is considered successful if the goal has been achieved. This number is a good performance measure because it allows to understand how many times during the all available episodes the agent is able to reach the goal.
- **Reward function selection**: performance evaluation based on the chosen reward function, which can be:
  - Euclidean distance (equation 4.4).
  - Exponential Euclidean distance (equation 4.5).
  - Gaussian Euclidean distance (equation 4.6).

For each experiment, the performance of each algorithm will be analyzed through two types of graphs:

1. **Average cumulative reward** obtained by the agent at the end of each episode evaluated as:
$$\bar{R} = \frac{\sum_{n=1}^{N} R(n)}{N} \tag{4.9}$$
where $\bar{R}$ is the average cumulative reward at the end of the episode, N is the number of iterations that the agent experience per episode (with a maximum of 250) and $R(n)$ is the reward at considered iteration n.
Reinforcement learning tries to maximize the cumulative reward per episode, so this plot is useful to verify whether the agent is able to correctly estimate the optimal policy. The agent should follow this optimal policy to reach the goal without collisions and with the least amount of actions. In this way, also convergence rate can be assessed, because complete convergence is achieved in the moment in which the same policy is followed and, consequently, the average cumulative reward is maximum and constant for the remaining episodes. At the same time, since the average cumulative reward depends on the selected reward function, it is possible to analyze which reward function provides the best results.

2. **Bonus behavior**. A sparse reward will be added to the reward function to prioritize actions that make the agent reach the goal and penalize actions that make the agent colliding. In particular, a bonus of +200 is assigned when the agent reach the goal, while a penalty of -100 is obtained when a collision occurs. Looking at the bonus behavior allows to understand whether an agent has actually reached the target location at the end of the episode, maximizing the average cumulative reward, or collided with an obstacle. This sparse reward is necessary in particular when obstacles are located close to the goal: reward would be high at those points, since the distance between the end-effector and the goal would be small, even if the goal has not been reached yet.

Based on the results obtained through the proposed assessment, just the most goal-oriented algorithms will be tested in other configurations of the environment during the later experimental phase. This choice have been made because of the long training period of each algorithm on the real setup, in the range 1-3 hours per experiment, depending on the selected number of training episodes and on the chosen approach. In this way, further analysis or modification to the selected algorithms can be performed to evaluate their efficiency in a more accurate way. In particular, the environment layouts shown in figure 4.14 will be also taken into consideration with different goal locations, i.e. goal on the left, goal on the right and straight goal.

These layouts have been chosen because they are representative of some possible standardized configurations of a pipe network, as shown in figure 4.15. Configuration 4.14a, configuration 4.14d and configuration 4.14e are similar concerning the location of the goal and, therefore, RL algorithm adaptability to new configurations of the environment can be investigated, i.e. the acquired knowledge of the already learned environment is applied to learn a similar but not equal configuration, adopting a transfer learning approach. If adaptability is proved to work efficiently, the same trained Q can be adopted to similar pipe networks, ensuring a faster convergence to the (sub)optimal policy.

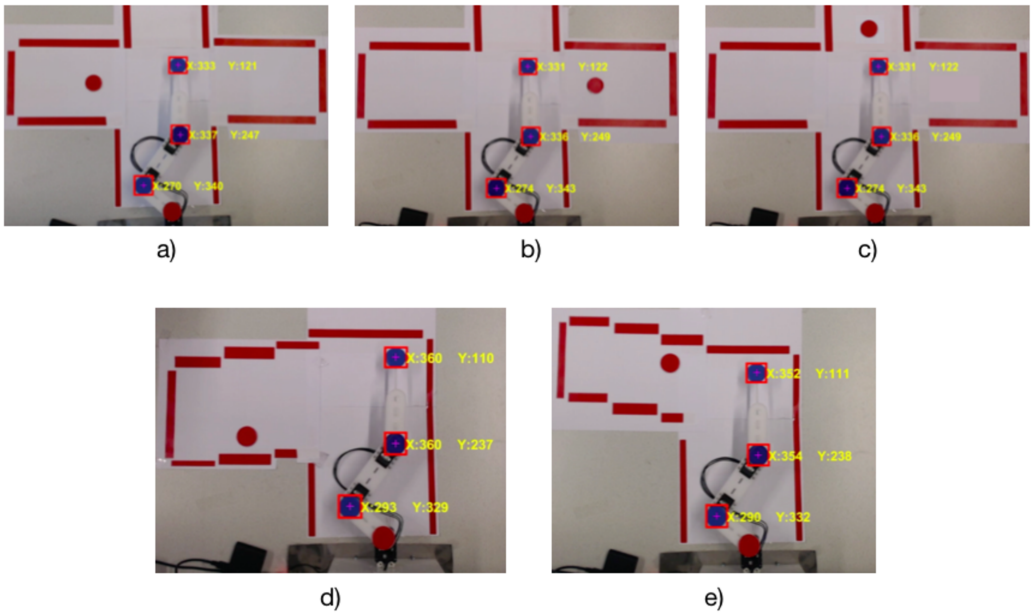

Figure 4.14: Possible setup layouts. a) Cross-environment with goal on the left. b) Cross-environment with goal on the right. c) Cross environment with goal straight. d) Acute curve with goal at the bottom. e) Obtuse curve with goal at the top.

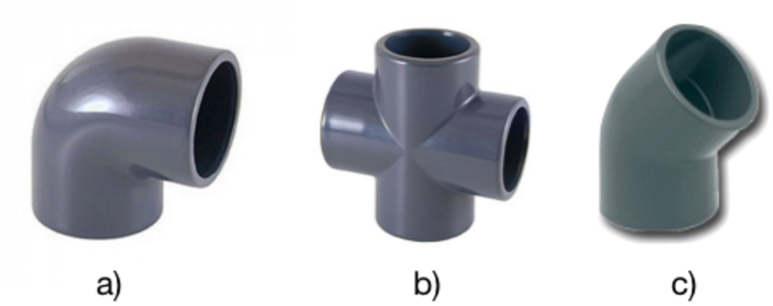

Figure 4.15: Standardized pipes configuration. a) 90 degree curve. b) Crossroad. c) Obtuse curve.

4.4 GUI-based software architecture

In order to make the software architecture more user-friendly as required in section 3.3.5, a Graphical User Interface (GUI) has been developed for the setup environment so that a new user can easily deal with the implemented algorithms and with the correspondent setup, without actually analyzing the code line-by-line. In this context, MATLAB provides the possibility to create GUI with point-and-click control, eliminating the need to completely understand the code or to type commands to run the applications.

First of all, the user should select which kind of environment is using, i.e. simulation or actual setup from the menu shown in figure 4.16.

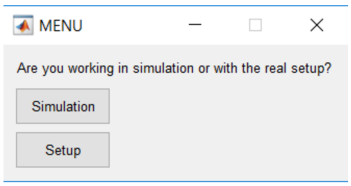

Figure 4.16: MATLAB GUI for selection of simulation or setup environment

If the simulation environment has been decided, the user is asked to select one of the sample environments shown in figure 4.17. These three environments have been considered as sample because they represent the main three types of pipes configurations (see figure 4.15). In this perspective, also the animation option can be activated to have a better visualization of the learning procedure.

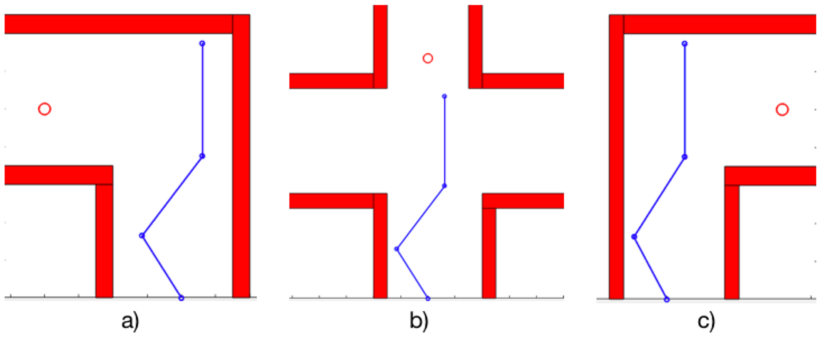

Figure 4.17: Sample simulation environment configurations. a) left curve, b) crossroad, c) right curve

On the other hand, if the actual setup has been chosen, the user will be asked to complete the following steps;

- Check the ID of each motor.
- Connect the servo-motors to the USB2Dynamixel device and the latter to the serial port of the PC (see section 4.1).
- Check the name of the selected PC serial port.
- Switch on the power supply for the motors and set it to 12 V.
- Connect the external camera to the PC USB port to record the overall scene.

If all these steps are completed correctly, the user could see the manipulator moving to its initial configuration.

At this point, the user should declare which kind of RL algorithm he wants to execute. Two multiple-choice menus will appear, one for selecting the algorithm itself (see figure 4.18a) and another one to select if discretized state-space or continuous state-space should be considered (see figure 4.18b).

Once the RL algorithm has been completely defined, the user can freely choose which reward function he would like to test based on the alternatives proposed in section 4.2.1.

Finally, according to the implemented choices, the chosen algorithm will start to run automatically.

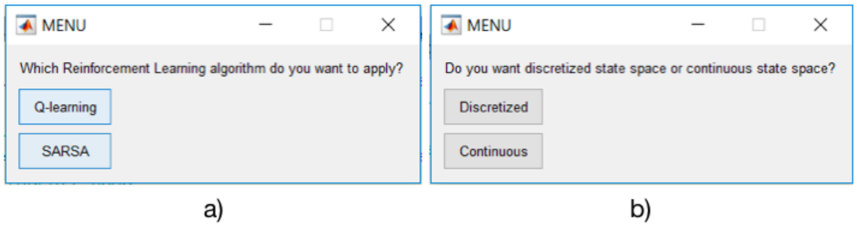

Figure 4.18: MATLAB GUI for selection of RL algorithm

4.5 Final design assessment

The requirements specified in section 3.3 can be used as evaluation criteria to assess the design proposed in the previous paragraphs. Since not all the requirements are related to design choices but to a greater extent to the results reported in the following chapter (5), just the design requirements are herein discussed.

4.5.1 Setup requirements

1. *The setup should be manufactured with RAM facilities.*

   The robotic manipulator adopted in this research has been laser cut with RaM Group technology.

2. *The necessary components should be low-cost and commercial.*

   The employed components for the manipulator and the camera are cheap and easy to find online. Each servo-motor costs approximately 53€, the USB2Dynamixel device is sold for 50€, while the employed camera 115€, for a total cost of 324€. Thus, the total amount is in the available budget of 1000€.

3. *The communication between different hardware components has to be as fast as possible.*

   The communication between different hardware components is fast and reliable thanks to the open-source library developed by (31). This library allows to connect all the components to MATLAB and communicate with them straightforwardly.

4. *The camera has to localize the markers efficiently.*

   The adopted marker detection algorithm proposed in 6 is available to operate accurately and to localize the present RGB markers in only 0.1356 seconds, which is much less than the maximum permitted time of 0.5 seconds.

4.5.2 Tests requirements

1. *Simulation is the first-step for valuable tests.*

   A representative simulation environment has been designed to implement valuable tests. Instead of making use of the camera signals, the simulation environment is characterized by a forward kinematic model (see figure 4.7 and equation 4.1) that allows the evaluation of distances between the simulated robotic structure and the simulated obstacles/goal. To maintain a correspondence between the two environments, the obstacles and goal location in simulation are in pixels as in the case of the physical setup, so that all the simulated markers can be placed in the same pixel location of the physical environment.

2. *Employ virtual obstacles to bypass setup damages.*

   Red rectangular markers have been adopted to represent the obstacles in the scene. In this way, no actual collision between the robotic structure and the obstacle can occur.

3. *Prioritize tests on more performing algorithms.*

   As specified in section 4.3, all the algorithms are first tested in the environment shown in figure 4.6 to assess the most performing ones. At that point, just the most efficient algorithms will be tested in other configurations of the environment (see figure 4.14).

4.5.3 Non-functional requirements

1. *The code should be user-friendly*

   In order to make the code more user-friendly, a MATLAB-based graphical user interface has been developed (see section 4.4), so that new users can easily interact with the implemented RL-based navigation system without the need of directly look at the main algorithms.

5 Results

This chapter presents the results obtained by applying RL algorithms on different configurations of the environment. As already explain in section 4.3, the first configuration of the environment that has been considered is the one shown in figure 4.5, on which all the four implemented algorithms have been tested, i.e. discretized SARSA, discretized Q-learning, Deep SARSA and Deep Q-learning.

Based on the results obtained through the proposed first experiments, just the most performing algorithms have been tested in other configurations of the environment (see figure 4.14). Since configuration 4.14a, configuration 4.14d and configuration 4.14e are similar concerning the location of the goal, a transfer learning approach can be employed to verify algorithms adaptability to similar environment layouts.

Thus, this chapter is organized as follow: first of all, the early experimental phase of the project is presented, with a particular focus towards algorithms adjustments in simulation and tests of the four possible approaches on the setup environment in figure 4.5. Afterwards, in section 5.2, the later experiments are described, in which Q-learning and Deep Q-learning have been tested in different configurations of the environment. Eventually, in section 5.3, a final analysis of the different implementations is proposed, paying attention towards the strengths and weaknesses of each technique.

5.1 Early experiments and results

In this section, the first experiments that have been performed both in simulation (section 5.1.1) and on the setup in figure 4.5 are presented (sections 5.1.2 and 5.1.3).

The algorithms have first been tested in simulation environment (see section 5.1.1) in order to check their work-ability and to optimize the selection of RL and NN parameters (see table 4.3 and 4.7). These parameters cannot be all assessed on the actual setup because of the long training time. Their selection is correlated to the objective of the algorithm, the environment configuration and the agent behavior. Since the simulation environment together with the forward kinematic model of the robot is a good approximation of the real environment in which the robotic arm operates, the results obtained in simulation with specific parameters remain invariant with respect to the results that would have been achieved in real-time environments.

Once all the parameters are determined, the algorithms are assessed on the real setup (see sections 5.1.2 and 5.1.3) to verify which of them presents the best performance and so should be chosen for later and more detailed experiments (see section 5.2).

In all the experiments, the action vector has been defined as follow:

$$actions = \begin{bmatrix} -5, & 5, & -8, & 8, -5, & 5 \end{bmatrix} \tag{5.1}$$

The unit of measure of the action vector is motor units, so in degrees the vector would become:

$$actions = \begin{bmatrix} -1.45, & 1.45, & , -2.32, & 2.32, -1.45, & 1.45 \end{bmatrix} \tag{5.2}$$

The first two elements correspond to actions for the first joint, the third and the forth for the second joint, while the last two elements correspond to third joint actions. In this perspective, each joint can move clockwise and counter-clockwise depending on the selected action. As possible to notice, the actions values are equal to the discretization sample of each state, that has been actually discretized depending on the size of the workspace and on the motion each link should be able to perform (see section 4.2.1).

5.1.1 RL and NN parameters assessment in simulation environment

First of all, discretized Q-learning algorithm initialized with RL parameters in 4.3 has been taken as a reference to understand how the agent should deal with collisions. As already specified in section 4.2, once a collision occurs, the agent can behave in two different ways:

- It resets to its initial configuration, ending the episode at that point. The correspondent Q value function is updated at the end of each iteration.
- It moves back to the state before the collision took place, continuing the iterations. The correspondent Q value function is updated before moving back to the previous state.

Both the approaches have been tested on the environment in figure 4.8 to understand which technique makes the agent to reach the goal faster and with less collisions (see figure 5.1 and figure 5.2).

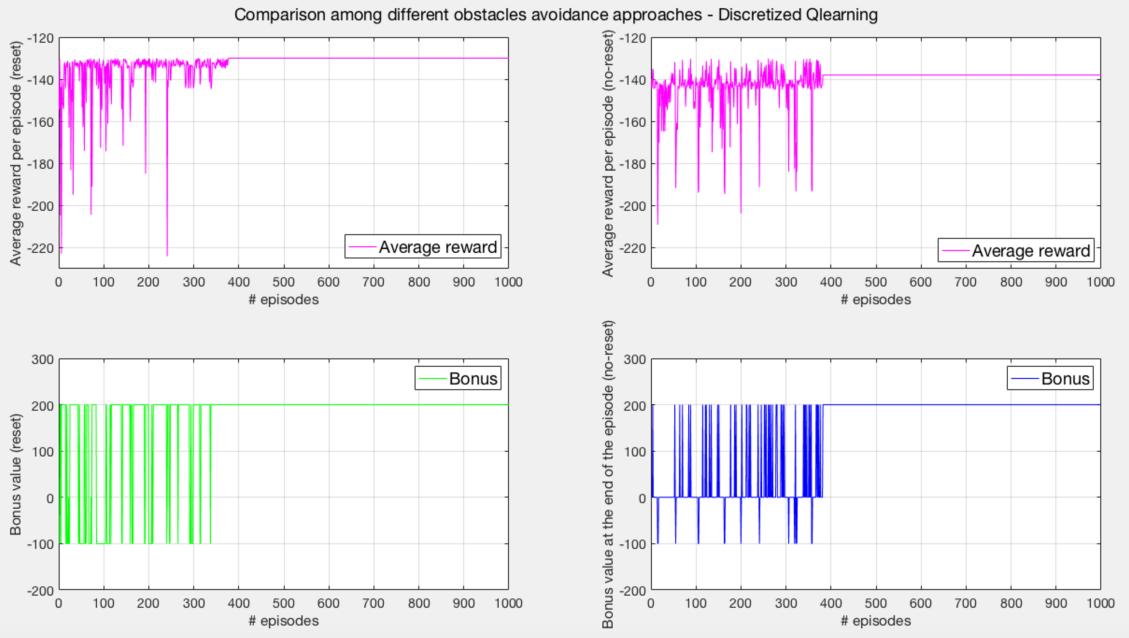

Figure 5.1: Comparison among different obstacles avoidance approaches for discretized Q-learning algorithm, i.e. reset after collision or no reset and, so, move to the previous state. The y-axis of each plot represents the average cumulative reward 4.9 that the agent gains at the end of each episode, applying a Euclidean reward function (equation 4.4) and an exploration/exploitation factor $\epsilon = 0.3$. A training time of 1000 episodes has been chosen.

Looking at the plots in figures 5.1, it is possible to notice that both the approaches converge to a (sub)optimal policy after around 400 episodes, i.e. the correspondent average cumulative reward is constant from that point onward. The main difference that can be immediately highlighted between the two methodologies is that, when the manipulator is reset to its initial configuration after a collision (plots on the left), its respective average cumulative reward is less negative, even in the moment in which convergence is reached. This situation is due to the fact that, once a collision occurs after few actions, the manipulator reset itself instead of continuing exploring states of the environment in which obstacles are present. On the other hand, if the manipulator is not reset after colliding (plots on the right), it continues moving and it stops only when either the goal has been reached, a collision occurs or the 250 available actions/iterations are over. This last situation is well represented in the bonus plot in the lower right-hand corner: at the end of the episode, the bonus value is often zero, i.e. the agent made use of all the available actions without neither reaching the goal nor colliding. Thus, the "no-reset" approach

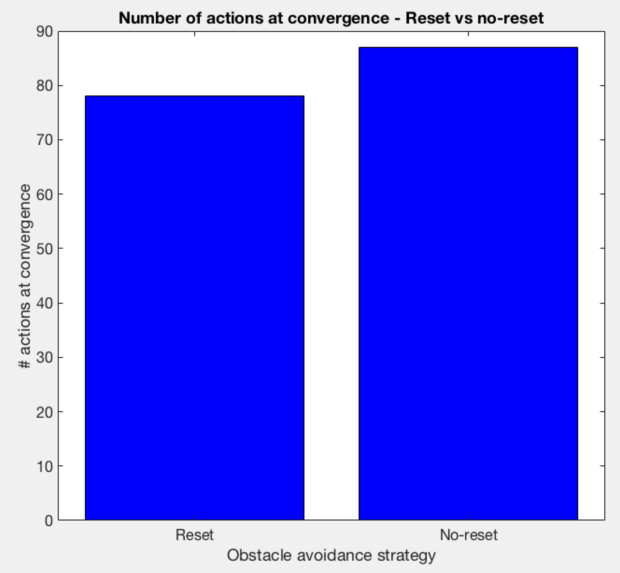

Figure 5.2: Comparison among average number of actions required to reach the goal at convergence for different obstacles avoidance approaches for discretized Q-learning algorithm, i.e. reset after collision or no reset and, so, move to the previous state.

actually accumulates more negative reward on average, because it lets the agent free to collide with obstacles from different sides and from different state configurations and, consequently, it gets stuck exploring areas that actually are not of interest. In this case, when convergence is reached, the result is that the agent is not able to completely minimize the amount of actions needed to reach the goal (the correspondent average cumulative reward has a value of -138 versus the value of -130 which characterizes the "reset" case) because either it may prefer to reach the goal colliding instead of looking for an obstacle free trajectory, if this path is shorter, or it does not present enough exploration to figure out the best policy. The agent tries to find out the shortest path (i.e. minimal number of actions) to the goal, but, since the available environment states are 782420, it would require more exploration to acquire knowledge about the obstacles states and the good states which would bring it faster and more safely towards the goal. On the other hand, the "reset" case has to find out the collision free trajectory for sure. As a matter of fact, looking at the histogram in figure 5.2, it is possible to notice that the "reset" algorithm requires 78 actions on average to reach the goal at convergence, while the "no-reset" algorithm figured out a longer policy which consists of 87 actions.

Last but not least, since the "no-reset" case usually makes use of most of the available actions in the first exploration phase, it requires more time to execute the whole training of 1000 episodes, i.e. 677 seconds on average, with respect to the much smaller learning period of the "reset" case, i.e. 387 seconds on average. This fact has to be taken into account in a real-time perspective: since the learning in real-time is expected to be much longer than the one in simulation, a solution that tries to minimize the learning period as much as possible should be selected. For all the aforementioned reasons, the experiments will be carried out resetting the robotic arm when a collisions takes place.

The learning period in both simulation and real-time is also affected by the implemented tuning choices (see sections 3.2.1 and 4.2). As already specified, while the choice of discount index γ and RL rate α is more straightforward and driven by what can be found in the literature (4),(12),(14),(9),(10), the tuning of ϵ parameter, which guides the exploration-exploitation phases, is more dependent on the target the agent has to achieve: if an agent has to explore most of the environment in which it is placed, higher values of ϵ should be selected to guarantee more exploration; on the other hand, if the agent must converge to a (sub)optimal policy as

fast as possible, the exploitation period should be prioritized with lower values of ϵ. In this case, an optimal trade-off between exploration and exploitation should be figured out to encourage the robotic arm to learn the environment in a smart and goal-oriented way. In this perspective, the ϵ is also discounted of a discount factor of 0.998 as specified in table 4.3, so that after 2000 episodes it always reach values closed to zero, to ensure exploitation of the learned policy. Moreover, when the goal has been reached for 50 successive episodes with approximately the same number of actions (i.e. number of actions oscillating between +2 and -2 with respect to the average value), ϵ is decreased to zero not to let the agent to explore the environment anymore and to just exploit the learned policy.

As already specified in section 4.2.1, the applied ϵ-greedy algorithm has been modified (see algorithm 11) to make the agent able to more uniformly select the available actions, obtaining the results in figure 5.3.

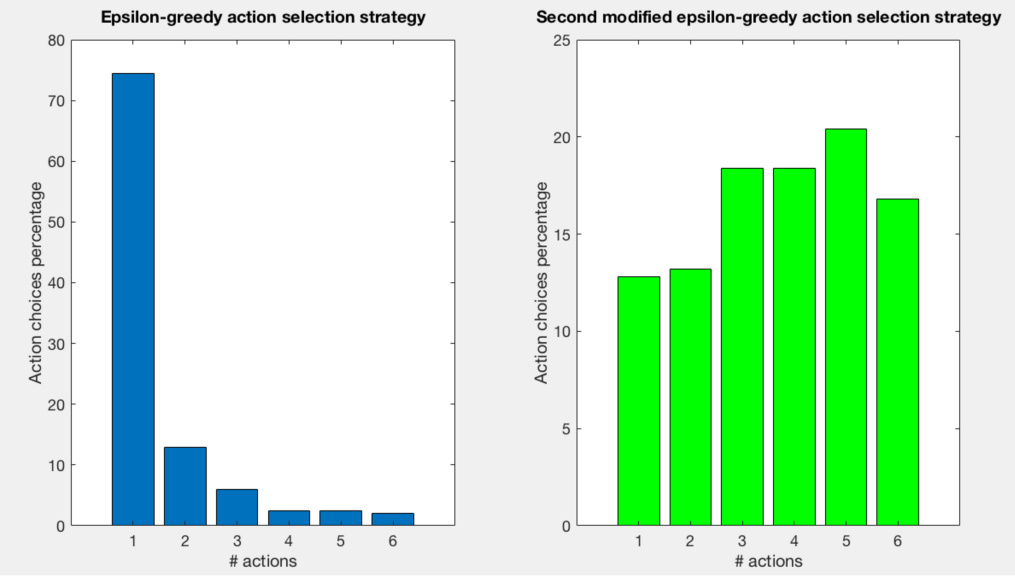

Figure 5.3: Comparison between action selection distribution in ϵ-greedy algorithm 9 and modified ϵ-greedy algorithm 11. Discretized Q-learning algorithm has been applied together in simulation environment 4.1.4 with an action vector of six actions, two for each joint. The y-axis represents the number of times each action (x-axis) has been selected in one episode of 250 iterations.

As easily noticeable from figure 5.3, the standard ϵ-greedy algorithm 9 (histogram on the left) favors the selection of the first action among all the others (74.4% of probability to be selected in one episode), since all the actions are inizialized equally as specified in equation 4.3. On the other hand, the modified ϵ-greedy algorithm 11 allows to obtain a more uniform distribution. As a matter of fact, action 1 is selected 12.8%, action 2 13.2%, action 3 and action 4 18.4%, action 20.4% and action 6 16.8%.

At this point, since uniform actions selection is guaranteed, it is possible to evaluate which exploration-exploitation trade-off is preferrable for this kind of applications. Three ϵ values have been investigated on discretized Q-learning in simulation, i.e. $\epsilon = 0.1$, $\epsilon = 0.3$ and $\epsilon = 0.6$, and the obtained results are shown in figure 5.4.

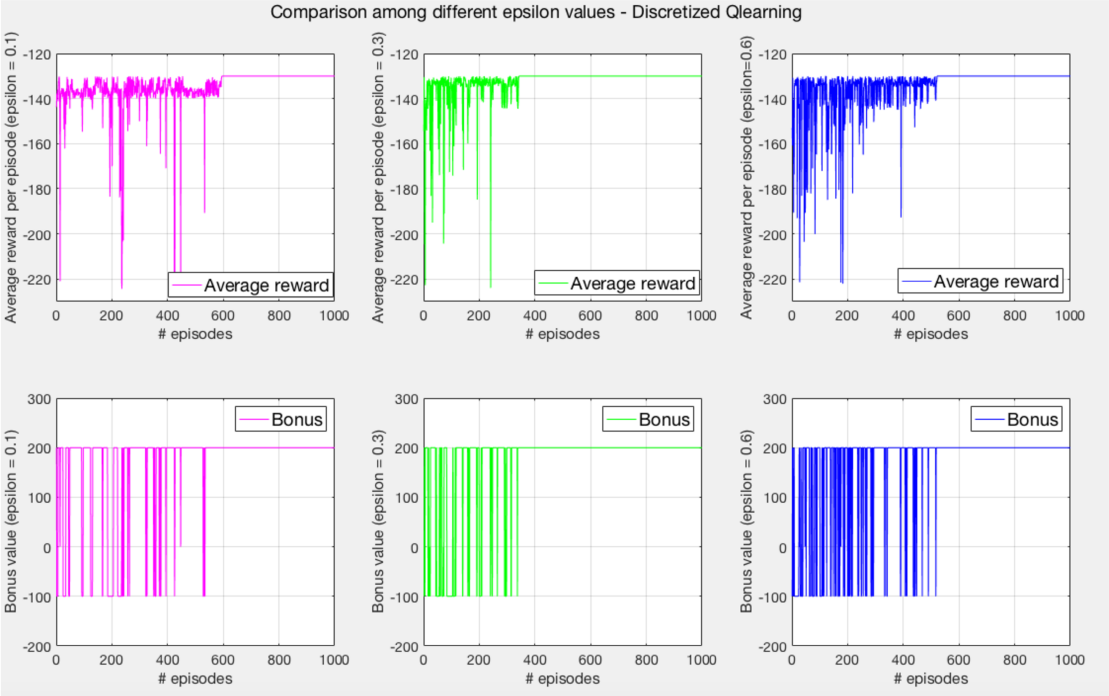

Figure 5.4: Comparison among different ϵ values in discretized Q-learning algorithm, i.e. $\epsilon = 0.1, 0.3$ and 0.6. In the upper plots, the y-axis of each plot represents the average cumulative reward that the agent gains at the end of each episode. The considered reward function is the negative Euclidean distance 4.4. A training time of 2000 episodes has been chosen, but since convergence is always reached before the end of all the training period, just the first 1000 episodes are considered (x-axis). In the lower plots, the behavior of the bonus is shown (y-axis) with respect to the number of available episodes.

As possible to notice from figure 5.4, all the proposed exploration-exploitation strategies are characterized by really negative average cumulative rewards in the first range of episodes, due to two main reasons:

- Collisions with obstacles far away from the goal: when a collision occurs, the robotic arm is reset to its initial configuration, i.e. the configuration in figure 4.8) and so, if the agent hits an obstacle far away from the goal, the average cumulative Euclidean distance becomes really low.
- The agent makes use of all the available actions for episodes (i.e. 250) without reaching the goal, obtaining zero as bonus.

In all the presented situation, the selected agent gradually improves its behavior until convergence is achieved. When convergence is reached, the correspondent average cumulative reward stays almost constant for the whole remaining episodes. This means that, from that point onward, the agent continues following the same trajectory, selecting the same actions to reach the goal. Thus, this trajectory is driven by the optimal policy which has been learned during the training and that, in practice, corresponds to an average cumulative reward of approximately -130, i.e. the average cumulative Euclidean distance between the end-effector and the goal becomes almost -130 pixels. The only plot that shows a behavior different with respect to the others is the one in which $\epsilon = 0.1$, because in that case the agent is not allowed to explore a lot and, consequently, it reaches the goal exploiting policies which are not the optimal ones, as possible to see in the episodes range 60-91, 264-321 and 448-527. As a matter of fact, the correspondent average cumulative rewards in those time frames remain almost constant but lower than -130 and so non optimal. On the contrary, when $\epsilon = 0.6$, the agent explores a lot and,

consequently, it takes more episodes to converge to the optimal solution, colliding more times with obstacles.

In order to assess which ϵ-greedy strategy is the most appropriate for the considered application, the different approaches are compared based on the episodes required to converge to the optimal actions selection and on the average number of successes or collisions in the training period before convergence is achieved. The following results are obtained for 10 experiments under the same conditions:

ϵ-value	Convergence episode	Probability of success	Probability of collision
$\epsilon = 0.1$	585	71.2%	28.4%
$\epsilon = 0.3$	377	70.6%	29.1%
$\epsilon = 0.6$	568	67.8%	31.5%

Table 5.1: Assessment of ϵ-values based on convergence episode and number of successes/collisions during the whole learning period

According to the results shown in table 5.1, the exploration-exploitation trade-off which guarantees faster convergence is the one with $\epsilon = 0.3$ (377 episodes): its convergence episode is smaller with respect to the ones obtained with $\epsilon = 0.1$ and $\epsilon = 0.6$. Anyway, the amount of collisions and successes are preferable with $\epsilon = 0.1$, but, as previously mentioned, having too small exploration period does not guarantee convergence to optimal policies but just to non-optimal ones. On the other hand, larger values of exploration like $\epsilon = 0.6$ make the agent to explore too much and consequently collide more frequently (31.5% of training period before convergence with respect to 28.4% and 29.1% of $\epsilon = 0.1$ and $\epsilon = 0.3$ respectively). Eventually, $\epsilon = 0.3$ has been chosen as exploration-exploitation strategy.

As possible to notice, the percentages of successes and collisions shown in table 5.1 do no add up to 100% because, in the remaining period, the agent ends an episode making use of all the available iterations (i.e. 250) without neither colliding nor reaching the goal.

At this point, all the RL parameters are completely defined, so it is possible to focus more on the choice of NN parameters applied in Deep RL algorithms. While learning rate and momentum constant choices have already been deeply analyzed in section 4.2.2, the dimension of the hidden layer should be assessed in simulation to figure out the best trade-off between accuracy of the results and training period required for the neural network.

As already mentioned in section 4.2.2, the number of neurons in the hidden layer depend on the size of input and output layers. For instance, if the input has a size of three, the output consists of six neurons and the hidden layer of 50 hidden neurons, the number of unknown variables to be estimated is equivalent to $(3 * 50 + 50 * 6) = 450$ and so at least 450 training examples are needed. In this case, the number of training episodes is huge since it corresponds to all the possible combinations of states and actions, so three different layouts of the hidden layer have been investigated on Deep Q-learning in simulation, i.e. 50, 150 and 300 neurons, and the obtained results are shown in figure 5.5.

As noticeable from figure 5.5, all the three implementations of the hidden layer guarantee convergence, but only if 300 hidden neurons are utilized the algorithm directly converges to the optimal policy with a average cumulative reward of approximately -130. This result was expected because of the fact that increasing the number of hidden neurons correspond to an increment in the number of combinations among the training examples and so an increment in the accuracy of the network. In the other cases, with 50 and 150 hidden neurons, the algorithm converges to optimal policies only after much more episodes. For instance, when 50 hidden neurons are applied (plot on the left), the number of combinations among training examples are not sufficient and, consequently, the agent reaches the goal frequently (bonus plot

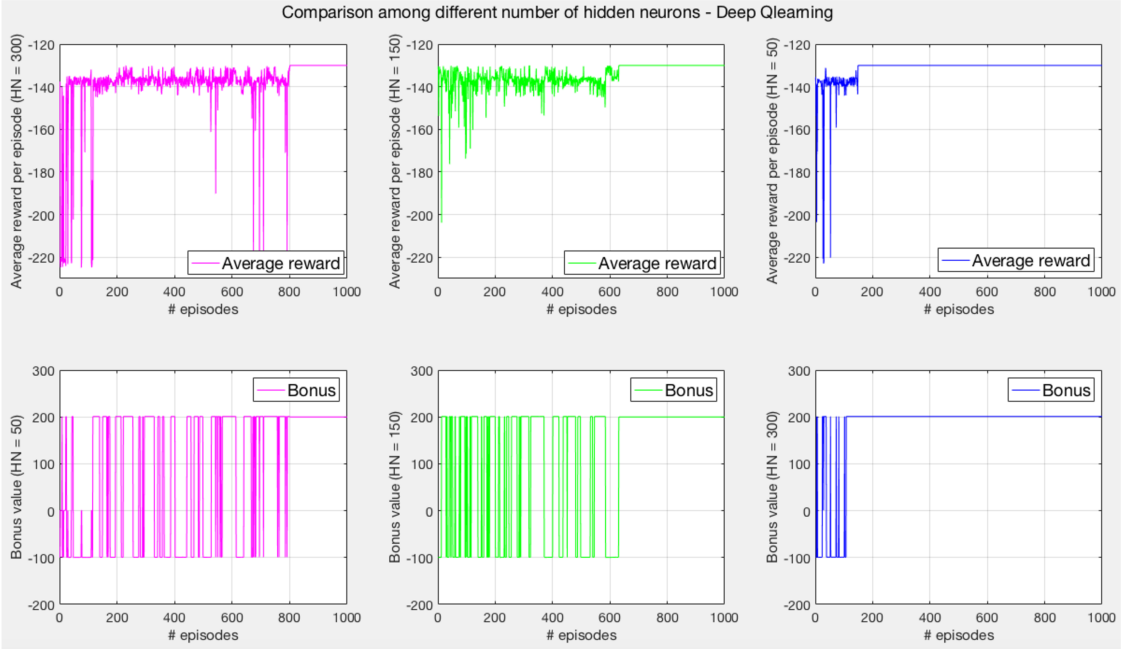

Figure 5.5: Comparison among different number of hidden neurons in Deep Q-learning algorithm, i.e. HN = 50, 150 and 300. In the upper plots, the y-axis of each plot represents the cumulative reward that the agent gains at the end of each episode. The considered reward function is the negative Euclidean distance (equation 4.4). A training time of 2000 episodes has been chosen, but since convergence is always reached before the end of all the training period, just the first 1000 episodes are shown (x-axis). In the lower plots, the behavior of the bonus is shown (y-axis) with respect to the number of available episodes.

in episodes range 300-350 is equal to 200) but without being able of finding the shortest path (average cumulative reward still oscillating in the same episodes range) and continuing making use of different number of actions. Therefore, to figure out which hidden layer layout is actually preferable for this kind of applications, an analysis has been made according to the convergence episode and average network training period for each batch, as reported in table 5.2.

Number of hidden neurons	Convergence episode	Training period for batch (seconds)
HN = 50	801	0.0735
HN = 150	631	0.093
HN = 300	149	0.1117

Table 5.2: Assessment of HN-values based on convergence episode and network training period

According to the results shown in table 5.2, the number of hidden neurons that ensures faster convergence is 300, which reach converge in only 149 episodes. As already mentioned, more hidden neurons correspond to more reliable results, but increasing the number of HN too much makes the training period longer. As a matter of fact, the faster network training is achieved with less neurons (0.0735 seconds with HN = 50) and its velocity decreases as the number of neurons increases, reaching a value of 0.1117 seconds when HN = 300. Nevertheless, the latter is the only one that ensures convergence to the optimal policy in a minimum number of episodes, so, even if the training period becomes larger, this hidden layer configuration has been selected for next experiments of Deep RL.

5.1.2 First experiments on real setup - Reward function assessment

Since all RL and NN parameters are now completely defined in simulation, first experiments can be performed on the real setup to assess each algorithm and each correspondent reward function. In order to implement this analysis, each approach has been tested in the setup environment in figure 4.5, which is considered to be a good representation of a pipe-like environment with many constraints and a goal quite hard to be reached due to its closeness to some obstacles. In this section, all the obtained results are reported and deeply examined to find out which algorithm is more efficient and which reward function should be selected to acquire a more goal-oriented learning plan. Each RL algorithm has been tested independently for each possible reward function (see section 4.2.1), so that the best reward can be figured out for each RL methodology (see figures 5.1.2, 5.7, 5.8 and 5.9).

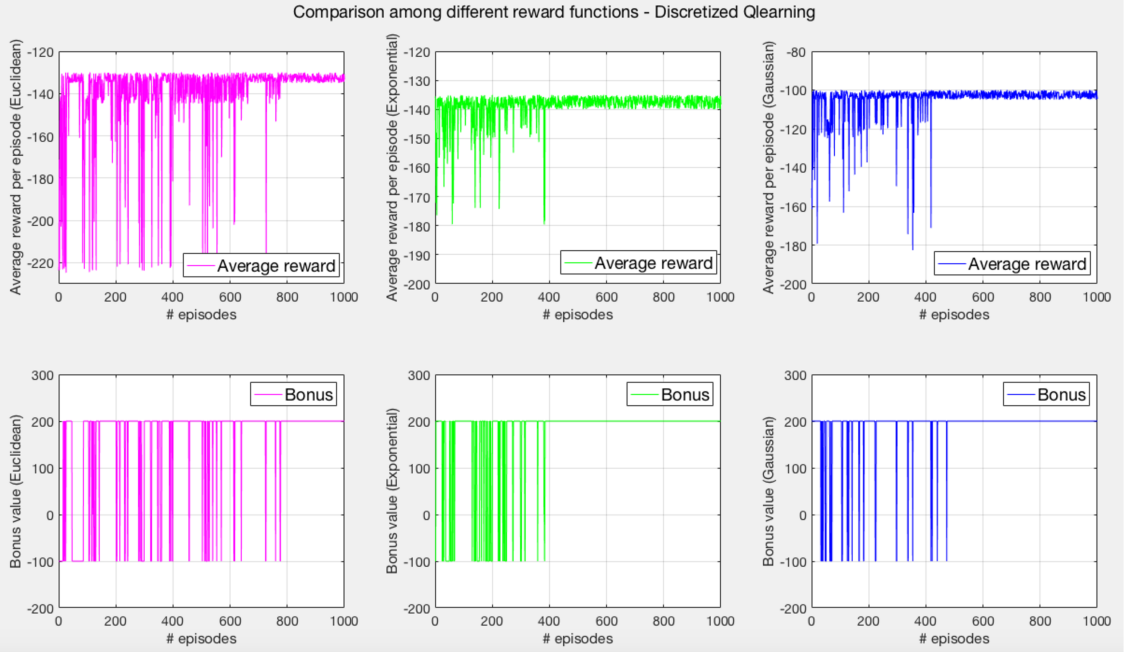

Figure 5.6: Comparison among different reward functions for discretized Q-learning algorithm, i.e. from the left Euclidean distance, exponential Euclidean distance and Gaussian Euclidean distance (see section 4.2.1). In the upper plots, the y-axis of each plot represents the average cumulative reward that the agent gains at the end of each episode. A training time of 1000 episodes has been chosen. In the lower plots, the behavior of the bonus is shown (y-axis) with respect to the number of available episodes.

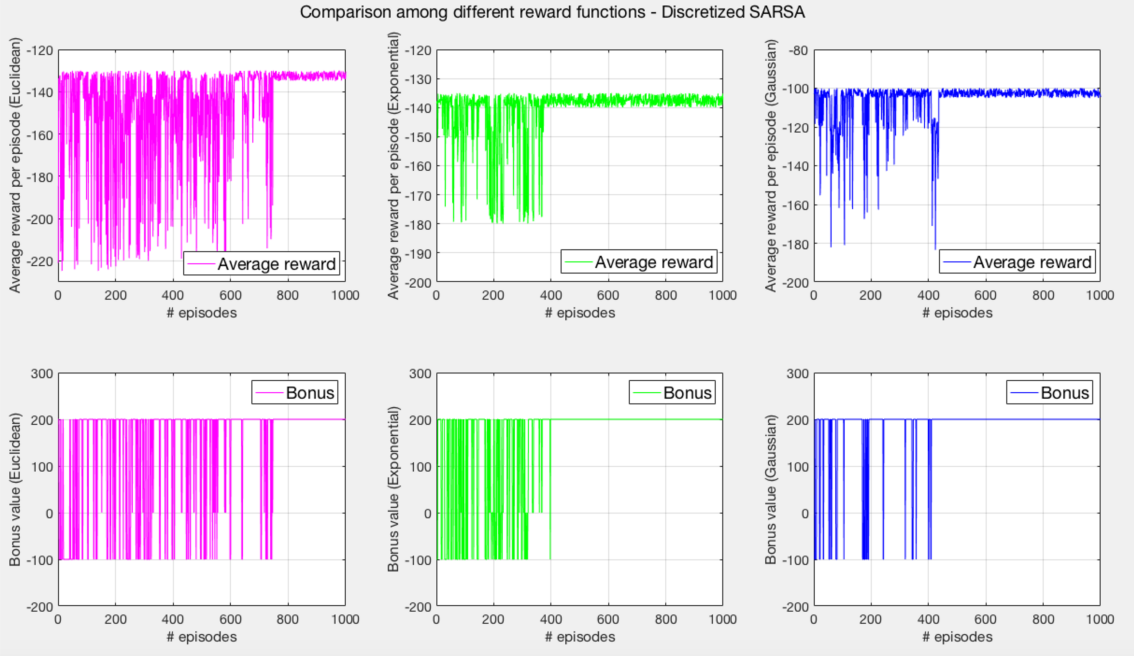

Figure 5.7: Comparison among different reward functions for discretized SARSA algorithm, i.e. from the left Euclidean distance, exponential Euclidean distance and Gaussian Euclidean distance (see section 4.2.1).. In the upper plots, the y-axis of each plot represents the average cumulative reward that the agent gains at the end of each episode. A training time of 1000 episodes has been chosen. In the lower plots, the behavior of the bonus is shown (y-axis) with respect to the number of available episodes.

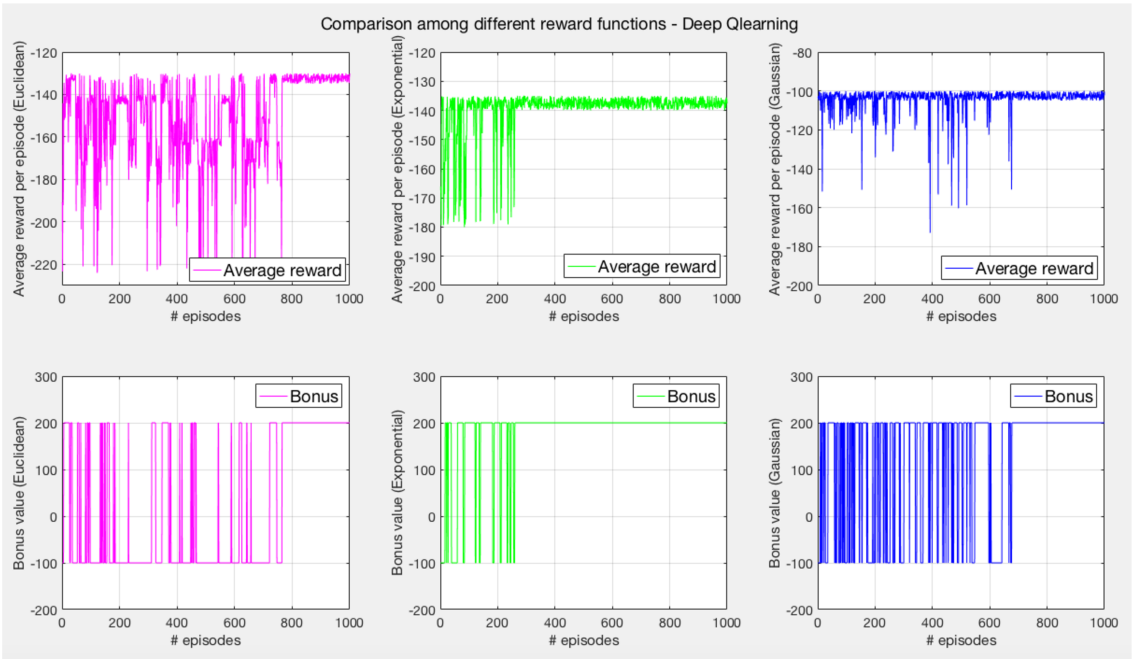

Figure 5.8: Comparison among different reward functions for Deep Q-learning algorithm, i.e. from the left Euclidean distance, exponential Euclidean distance and Gaussian Euclidean distance (see section 4.2.1). In the upper plots, the y-axis of each plot represents the average cumulative reward that the agent gains at the end of each episode. A training time of 1000 episodes has been chosen. In the lower plots, the behavior of the bonus is shown (y-axis) with respect to the number of available episodes.

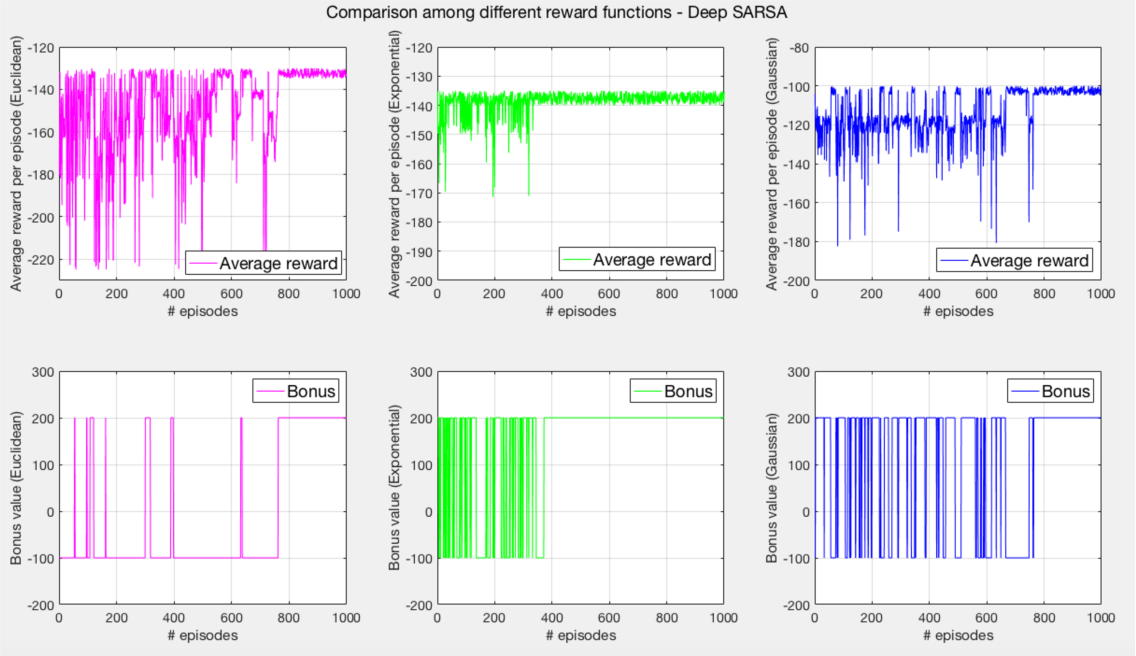

Figure 5.9: Comparison among different reward functions for Deep SARSA algorithm, i.e. from the left Euclidean distance, exponential Euclidean distance and Gaussian Euclidean distance (see section 4.2.1). In the upper plots, the y-axis of each plot represents the average cumulative reward that the agent gains at the end of each episode. A training time of 1000 episodes has been chosen. In the lower plots, the behavior of the bonus is shown (y-axis) with respect to the number of available episodes.

In all the graphs presented in figures 5.1.2, 5.7, 5.8 and 5.9, the reward function that provides the slower convergence to the (sub)optimal policy is the Euclidean distance reward (equation 4.4). Even if the Euclidean distance can be considered a challenging reward, it presents a significant problem, that is two states that are both really closed to the goal are characterized by almost the same reward, while the closer should be represented by a significantly higher reward to motivate the agent to move to that state. The same issue arises also with Gaussian Euclidean distance (see figure 4.11). Even though Gaussian function presents a average greater slope with respect to Euclidean distance, around the goal (see figure 4.11) the slope of the curve starts to decrease due to the selected variance and, consequently, the correspondent reward is no more increasing as fast as before. This problem has been completely solved applying exponential Euclidean distance reward in equation 4.5, which is characterized by an always increasing trend, in particular in proximity of the goal, where the exponential function continues growing even more, as possible to notice from figure 4.10; as a matter of fact, in mathematical terms this function is considered a strictly increasing function, i.e. a function where for each interval I, $f(b) > f(a)$ with $b > a$. In RL perspective, this property ensures always increasing rewards in proximity of the goal and always decreasing rewards as the agent moves away from it. Consequently, this behavior persuades the agent to select actions that bring it as closer as possible to the goal, reaching a faster convergence to (sub)optimal policies for all the presented RL approaches.

As already mentioned, the ultimate goal of each reinforcement learning agent is to maximize its cumulative reward, selecting those actions that reward it more in the long-term. With this idea in mind, the agent should be able to improve its behavior in time reaching the goal more often and with less amount of actions. Consequently, the average cumulative reward at the end of each episode should gradually increase with training up to a stable value when the optimal policy has been completely learned and no more exploration is performed. In the previously reported plots, this continuous improvement does not actually occur so clearly, because of the

presence of obstacles in the environment. While exploring, the agent can end up in a new environment state far away from the goal, that corresponds to a collision state. Thus, after the collision occurs, the agent reset to its initial configuration, accumulating a really negative total reward, due to the fact that the obstacle has been hit when the Euclidean distance between the end-effector and the goal was really high. This condition is well highlighted in the bonus graphs (lower plots of each figure), where it is possible to notice that the agent performance oscillates between successes, +200 when the goal has been reached, and failures, -100 when a collision takes place. Really negative cumulative rewards (upper plots) are most likely correlated to collisions in the bonus plots. This statement cannot be considered as a general rule, because sometimes the manipulator also collides with obstacles which are closed to the optimal trajectory towards the goal (in figure 4.5 the obstacles with centre pixel coordinates (204,248), (145,256) and (223,89)). Some failures in the bonus plots are related to higher average cumulative rewards, which could also be considered as successes even if they are not. This latter trend can be easily noticed in the Euclidean distance reward plots on the left of figure 5.9, in the episode range 500-600. In that case, the average cumulative reward of Deep SARSA algorithm seems to have almost reached convergence, but the correspondent bonus in that phase demonstrates that actually the agent was colliding with obstacles near the goal at that time.

Even though a sub-optimal convergence is achieved under all the reported conditions, the average cumulative reward of each algorithm continues to be of oscillatory nature also after convergence, situation that does not occur in simulation (see plots 5.4 and 5.5). These oscillations are due mainly to two factors:

- All the reward functions are in pixels. Since a pixel corresponds to 0.8824 millimeters with the predefined camera location and PC screen resolution, the goal is considered to be achieved in the moment in which the distance between its marker centre and the end-effector marker centre becomes smaller than 5 pixels, i.e. 4.412 millimeters. This trade-off has been chosen because, from an external observer point of view, the end-effector has reached the goal even if it is not perfectly on it and with the discretized action vectors it is hard to reach precise pixel locations. Consequently, the reward function value oscillates between -5 and 0 depending on how close the end-effector is with respect to the goal.

- The policy learned at convergence is just sub-optimal. In order to learn a global movement policy, it is necessary to train the agent for at least 5000 episodes. This training period is too high for real-time experiments, because it would require more than one working day. Thus, 1000 episodes have been considered sufficient to learn a sub-optimal policy, which is a good approximation of the global one.

Otherwise, this oscillation was not present in simulation (see plots in figure 5.4 and 5.5). As a matter of fact, in the simulation environment the locations of the markers as well as of the manipulator itself are more accurate since they are not evaluated through external signals measurements (camera and encoders signals) but through kinematic-based mathematical models.

In order to finally assess the more efficient reward function, it is interesting to analyze the convergence episode and the probability of success and collision before convergence of each RL algorithm (see table 5.3).

The results presented in table 5.3 are in line with what has been previously stated: exponential Euclidean distance reward function is the one that guarantees faster convergence for each algorithm (smaller convergence episode in the table). In this perspective, it has been noticed that having an always increasing reward trend in the proximity of the goal makes the agent to be more goal-oriented and able to discern between environment states really close to the goal and environment states which are just in the goal larger neighborhood. As a consequence, the

Discretized Q-learning			
	Convergence episode	*Probability of success*	*Probability of collision*
Euclidean distance	777	64.2%	34.7%
Exponential Euclidean distance	385	71.1%	28.4%
Gaussian Euclidean distance	474	68.5%	31.4%
Discretized SARSA			
	Convergence episode	*Probability of success*	*Probability of collision*
Euclidean distance	747	61.9%	34.5%
Exponential Euclidean distance	398	69.4%	28.3%
Gaussian Euclidean distance	437	66.8%	30.8%
Deep Q-learning			
	Convergence episode	*Probability of success*	*Probability of collision*
Euclidean distance	765	17.5%	82.4%
Exponential Euclidean distance	260	70.6%	29.3%
Gaussian Euclidean distance	677	50%	49.9%
Deep SARSA			
	Convergence episode	*Probability of success*	*Probability of collision*
Euclidean distance	762	6.6%	93.4%
Exponential Euclidean distance	372	62.8%	36.2%
Gaussian Euclidean distance	764	50.7%	44.3%

Table 5.3: Comparison among convergence episode, probability of success and collision for each reward function for each algorithm.

cited reward function is also the one that ensures higher probability of succeeding at the end of an episode instead of colliding (highlighted probability of success and collision in the table). If a collision occurs in proximity of the goal in a certain environment state, that state will be then characterized by a much lower estimated value with respect to its neighbors in Q and so the agent will never select the action that makes it to reach that state anymore, but it would prioritize the greedy actions with higher estimated values. This lower estimated value is due mainly by the sparse bonus reward that has been added to the algorithm, i.e. -100 of penalty when a state closed to the goal is reached. If just the reward function was adopted, a state closed to the goal would be identified by a quite high reward even if it would correspond to a collision state.

The other reward function that provides acceptable outcomes is the Gaussian Euclidean reward function. As already specified, this reward function has a trend similar to the exponential one, except the fact that its increasing trend decreases in proximity of its mean (zero in the selected Gaussian distribution in equation 4.6). Consequently, it becomes less efficient in the range closer to the goal. This drawback does not influence too much the results of discretized Q-learning and discretized SARSA, because both approaches present a convergence rate as well as success/collision probability which are comparable to the ones of exponential Euclidean distance reward. On the contrary, Deep Q-learning and Deep SARSA need much more episodes to converge if Gaussian Euclidean distance is employed. Deep Q-learning and Deep SARSA are both characterized by a continuous state-space, consequently the agent is allowed to reach any environment state in the state-space according to the correspondent action vector. Thus, if the reward does not allow a good prioritization of the environment states close to the goal, it makes continuous state-space agents require more episodes to understand which states are actually closer to the goal and which bring the agent colliding with obstacles in proximity of

the goal. Moreover, both this last algorithms adopt experience replay with mini-batches (2.1.8) and, consequently, the relative Q feed-forward neural network is just trained according to 75 random samples (mini-batches) from the learned experience. As a result, if many collisions occur in proximity of the goal, their agent will tend to exploit that knowledge colliding even more times in states close to the ones where it has already collided with.

The same situation happens applying the simple Euclidean distance reward, which is the least performing for all the algorithms, both in terms of convergence episode and in success/collision probability. In particular, it is completely inefficient in continuous state space cases, i.e. Deep Q-learning and Deep SARSA, because, as already mentioned in the previous paragraph, these approaches present a much larger state-space and they are more correlated to the experience they made while exploiting and, consequently, if the reward does not provide a good prioritization of the states near the goal, they tend to fail more times than succeeding.

Due to all the performed considerations, exponential reward function is adopted for all the successive experiments.

5.1.3 First experiments on real setup - Algorithms assessment

After defining all the necessary RL parameters both in simulation and on the real setup, it is now time to identify which RL algorithm actually presents the best performance. Thus, the results introduced in section 5.1.2 are now assessed from different points of view:

- **Convergence episode**: required episodes before reaching the target with the minimum number of actions and consequently maximizing the cumulative reward.
- **Actions at convergence**: number of actions required on average to reach the goal when convergence is achieved.
- **Success/collision probability**: probability to collide with obstacles versus probability of successfully reaching the goal without collisions.
- **Training period**: time required to train the agent for 1000 episodes with 250 iterations each.

Considering the cited evaluation parameters and exponential Euclidean distance as reward function, the distribution in figure 5.10 is obtained.

As possible to notice from figure 5.10 and table 5.3, each algorithm presents some pros and cons. Deep Q-learning is the one that guarantees faster convergence to the (sub)optimal policy; as a matter of fact, it takes 260 episodes (4746 seconds, i.e. approximately 1 hour and 20 minutes) to converge to the maximum average cumulative reward which corresponds to 78 agent actions on average to reach the goal. The same number of actions are required by Q-learning with discretized state-space, but this last approach necessitates at least 385 episodes (6308 seconds, i.e. approximately one hour and 45 minutes) to achieve convergence. Thanks to the possibility of using the complete non-discretized state-space, at each step the Deep Q-learning agent obtains more detailed state of the environment with respect to discretized Q-learning, allowing a more accurate approximation of the value function and, consequently, faster convergence. Even though, larger state-space makes also the agent to collide with obstacles more times (29.3% with respect to 28.4% of discretized Q-lerning), because larger state-space corresponds to a wider range of collision states as well.

The same comparative analysis can be performed across discretized SARSA and Deep SARSA: the former presents a collision probability of 28.3% with respect to 36.2% of Deep SARSA, but the latter achieves convergence in 372 episodes (6059 seconds, i.e. approximately one hour and 40 minutes) with respect to 398 episodes (6618 seconds, i.e. approximately one hour and 50

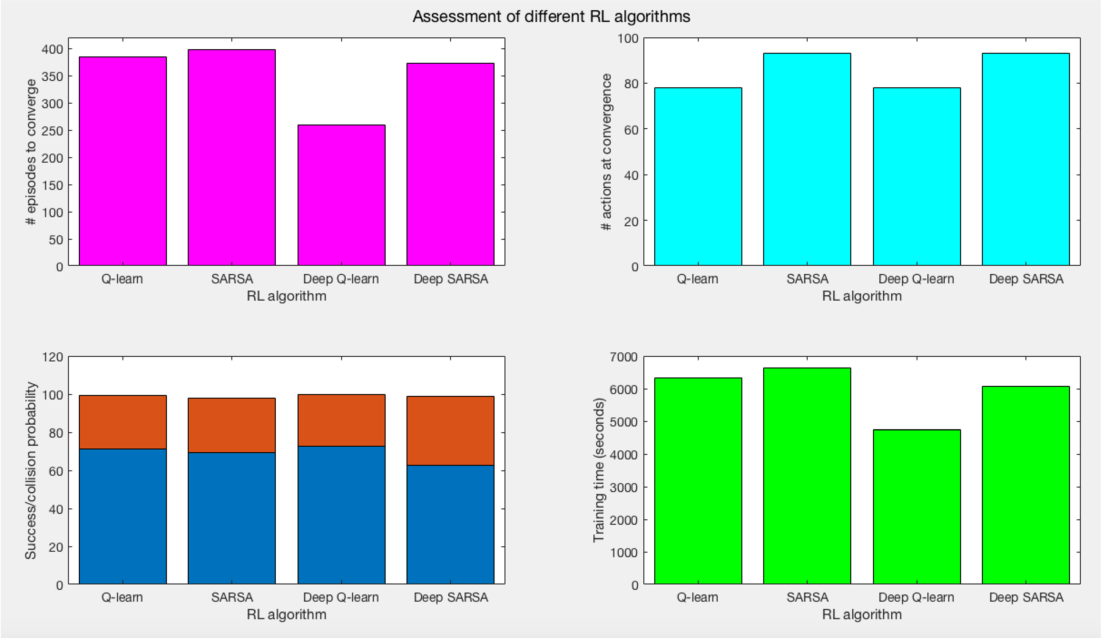

Figure 5.10: Assessment of RL algorithms according to convergence episode (magenta, upper left-hand corner), number of actions required at convergence (cyan, upper right-hand corner), success/collision probability (blue/red respectively, lower left-hand corner) and necessary training time (green, lower right-hand corner). Exponential Euclidean distance is considered as a reference reward function.

minutes) of discretized SARSA. Therefore, both resulting agents perform 93 actions on average to reach the goal while convergence is achieved.

In order to make a final assessment about the considered RL algorithms, it is now necessary to highlight the differences between off-policy learning (Q-learning) and on-policy learning (SARSA) approaches. As already mentioned in section 2.1.7, on-policy learning algorithms are more "conservative" with respect to off-policy algorithms, because they update the current $Q(S_t, A_t)$ value-function estimate according to next action that can be taken from the next state $Q(S_{t+1}, A_{t+1})$, assuming that the current policy is followed (see equation 2.12). Thus, this update strategy makes these algorithms to care more about their performance during the learning phase and, consequently, to avoid states that are considered to be dangerous, like states closed to obstacles, because a random action that provokes collision can be selected from those states (ϵ-greedy algorithm). On the other hand, off-policy algorithms always update the current $Q(S_t, A_t)$ value-function estimate according to the greedy action that can be taken from the next state $max_a(Q(S_{t+1}, a))$, assuming that the optimal policy is followed (see equation 2.13). In other words, off-policy algorithms always privileges the shortest trajectory, while on-policy algorithms can prefer longer but safer trajectories to reach the goal. This conditions has actually been met during the performed experiments; as a matter of fact, both the implemented on-policy RL algorithms (i.e. discretized SARSA and Deep SARSA) converge to a non-optimal policy which corresponds to 93 actions required to reach the goal with respect to the 78 actions of the off-policy RL algorithms (i.e. discretized Q-learning and Deep Q-learning). Thus, on-policy algorithms do not ensure convergence to an optimal short policy, but, at same time, they most often present a lower probability of colliding with obstacles, as possible to notice from table 5.3. Nevertheless, the difference between the two collision probabilities is not so significant as expected, i.e. 28.3% versus 28.4% for discretized SARSA versus discretized Q-learning when exponential Euclidean reward is applied. In continuous state-space, Deep SARSA collides even more than Deep Q-learning with exponential Euclidean reward, 36.2% versus 29.3% for Deep SARSA versus Deep Q-learning, because of the fact that the considered environment is over-

constrained and, consequently, avoiding the collision from one side can make the robotic arm to collide on the other side. As a result, off-policy approaches are preferable for this kind of applications, because they guarantee a better convergence to a sub-optimal and shorter trajectory without colliding with obstacles much more frequently than when on-policy algorithms are employed.

In line with the considerations made so far, discretized Q-learning and Deep Q-learning algorithms have been taken as a reference during the last experimental phases of the project.

5.2 Later experiments and results

This section describes the last experimental phase of the project. As mentioned in the previous paragraph, discretized Q-learning and Deep Q-learning algorithms with exponential Euclidean distance reward have been employed in this stage in order to perform further analysis and modifications to the existing architecture, making it more efficient and goal-oriented. In particular, both algorithms have first been tested on the cross configuration of the pipe-like environment with goal on the left, on the right of the pipe and straight (see figures 4.14a, 4.14b and 4.14c). The cross configuration is the most generic configuration that the robotic arm can be asked to learn, because in that situation it is allowed to go left, right or continue straight in the pipe. Thus, after learning all these possible goal locations in the cross environment (i.e. left, right, straight), the same knowledge can be employed to learn similar configurations (see figures 4.14d and 4.14e). In this perspective, the most performing RL approach has also been employed for investigating its adaptability to similar environment layouts by applying transfer learning strategies (see section 5.2.3).

5.2.1 A goal-oriented planning strategy

In the first experiments it has been noticed that each RL agent needs to explore many states of the environment before evaluating which state presents a higher Q-value and so can actually bring it closer to the goal. In order to make the agent to explore more areas that are in the neighborhood of the goal and not to let it waste time in exploring further states, a smart and goal-oriented planning strategy has been implemented.

In RL context, planning is defined as any process that employs a model of the environment to foretell future events and, consequently, to create or improve a policy (4). The adopted value-function based RL algorithms are model-free, so they do not require a model of the environment to achieve an objective, but an initial planning can be beneficial in the moment in which convergence has to be prioritized and large exploration of the considered environment is not necessary.

In the investigated pipe-like environment, the third link of the robotic manipulator (i.e. end-effector link) is the one that presents more freedom of movement: while the first and second links are constrained by the lower part of the pipe, the end-effector link can freely move clockwise or counter-clockwise depending on the goal location and on the curvature of the pipe. Moreover, the pipe curve is actually explored by the end-effector link, because it is the only one able to reach those regions of the environment; consequently, its correspondent actions should be prioritized, since it is actually the link that should move more often to reach the goal faster.

With these ideas in mind, the following planning strategy has been developed:

where x_{goal} corresponds to the x pixel coordinate of the goal, while x_{ee} corresponds to the x pixel coordinate of the end-effector at its initial configuration.

The algorithm in 13 changes the already specified initialization of the action-value function Q, according to the selected action vector (see equation 5.1). Depending on the employed algorithm, that initialization is performed differently:

Algorithm 13: Goal-oriented planning: smart initialization of the end-effector link motion

Move the robotic arm to its initial configuration;
Check the goal and end-effector pixel location with camera;
Initialize Q;
if $x_{goal} < x_{ee}$ **then**
    Increase Q value correspondent to counter-clockwise action of the end-effector of 10 units;
else
    Increase Q value correspondent to clockwise action of the end-effector of 10 units;
end

- **Discretized Q-learning**: as specified in section 4.2.1, the value-function table is initialized according to the adopted reward function (equation 4.3), so that states closer to the goal are initialized with higher value-estimates than states further away from it. However, all the actions for each state are initialized with the same value because, at that point, it has not yet been experienced which action brings the agent in the right direction. Applying the planning strategy, the end-effector action which move the agent towards the goal is prioritized and, consequently, the correspondent state-action pair is incremented by a factor of 10.

- **Deep Q-learning**: as mentioned in section 4.2.2, the value-function neural network weights are initialized to zero. If planning strategy is applied, the target weights correspondent to the end-effector action which move the agent towards the goal will be instead initialized at 10.

Increasing the goal-directed end-effector action makes the ϵ-greedy algorithm (i.e. algorithms 11 and 12) to privilege that action while exploiting the greedy policy. In this way, the agent will tend more likely to move the end-effector link in the preferable direction and, consequently, exploring states closer to it for the remaining exploration period.

An incremental factor of 10 has been selected because the planning should just affect the initial phase of the learning time and, consequently, the increment should vanish after some episodes. During the training, the state-action pairs are updated according to the experience performed by the agent and the relative obtained reward; after some iterations, the agent should be able to autonomously realize which action to take based on the acquired knowledge and not just going on selecting the cited end-effector action.

5.2.2 Learning cross-environment

As already mentioned, the cross-environment is the most general layout of the pipe network: it consists of one entrance and three exits, i.e. left pipe, right pipe and pipe on the front of the entrance. Thus, if the agent learns how to reach all the possible exits with the minimum amount of actions, it should acquire the necessary knowledge to face any other configuration of the pipe (see figure 4.15).

In the next paragraphs the results obtained while learning the environment shown in figures 4.14a, 4.14b are 4.14c are reported.

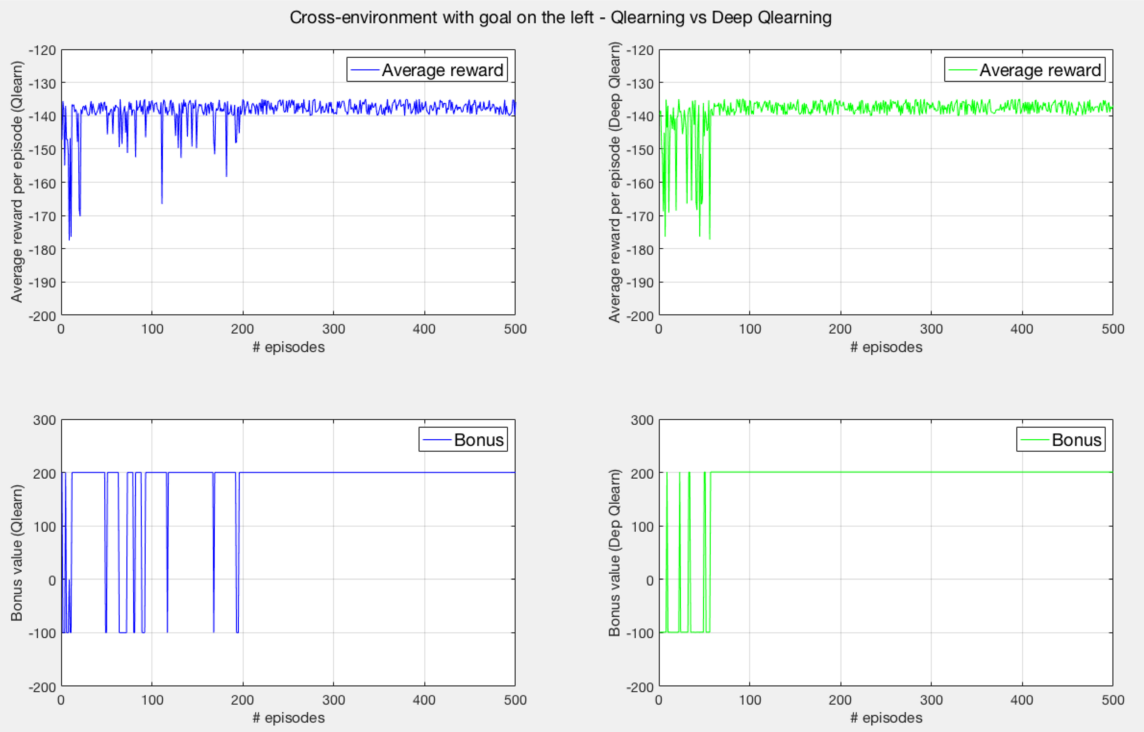

Figure 5.11: Results of learning cross-environment with goal on the left (see figure 4.14a), applying Q-learning and Deep Q-learning with exponential Euclidean distance as reward function (equation 4.5). In the upper plots, the y-axis of each plot represents the average cumulative reward that the agent gains at the end of each episode. A training time of 1000 episodes has been chosen, but just the first 350 episodes are shown to highlight the trend before convergence is reached. In the lower plots, the behavior of the bonus is shown (y-axis) with respect to the number of available episodes.

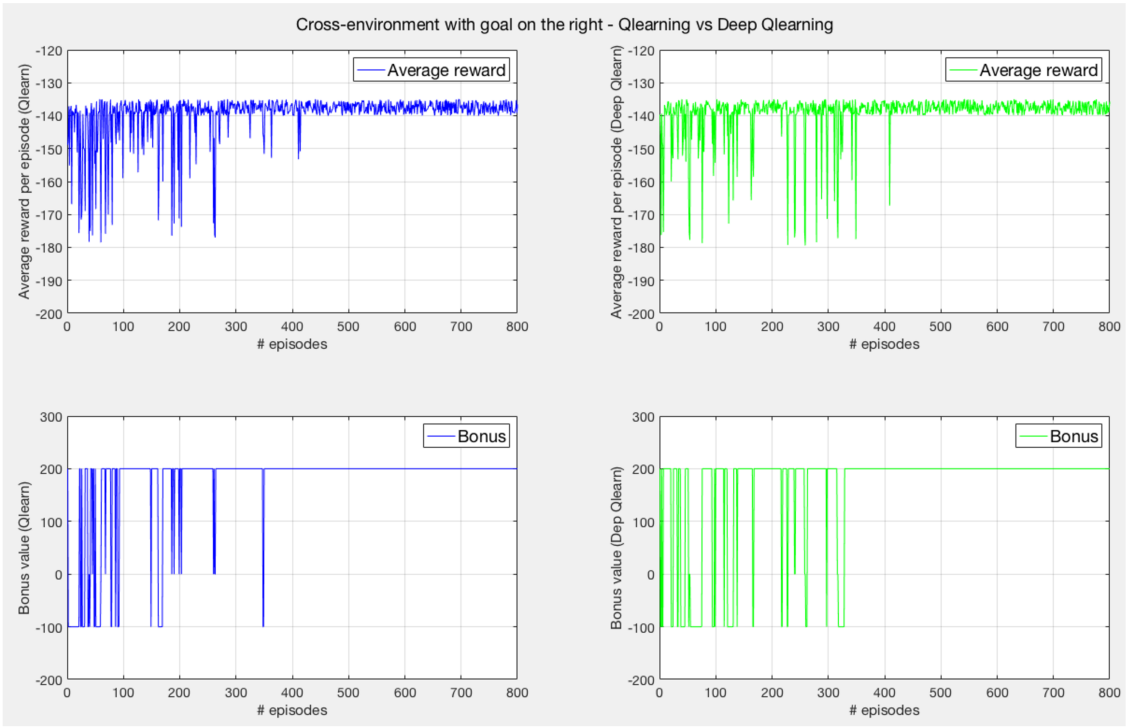

Figure 5.12: Results of learning cross-environment with goal on the right (see figure 4.14b), applying Q-learning and Deep Q-learning with exponential Euclidean distance as reward function (equation 4.5). In the upper plots, the y-axis of each plot represents the average cumulative reward that the agent gains at the end of each episode. A training time of 1000 episodes has been chosen, but just the first 600 episodes are shown to highlight the trend before convergence is reached. In the lower plots, the behavior of the bonus is shown (y-axis) with respect to the number of available episodes.

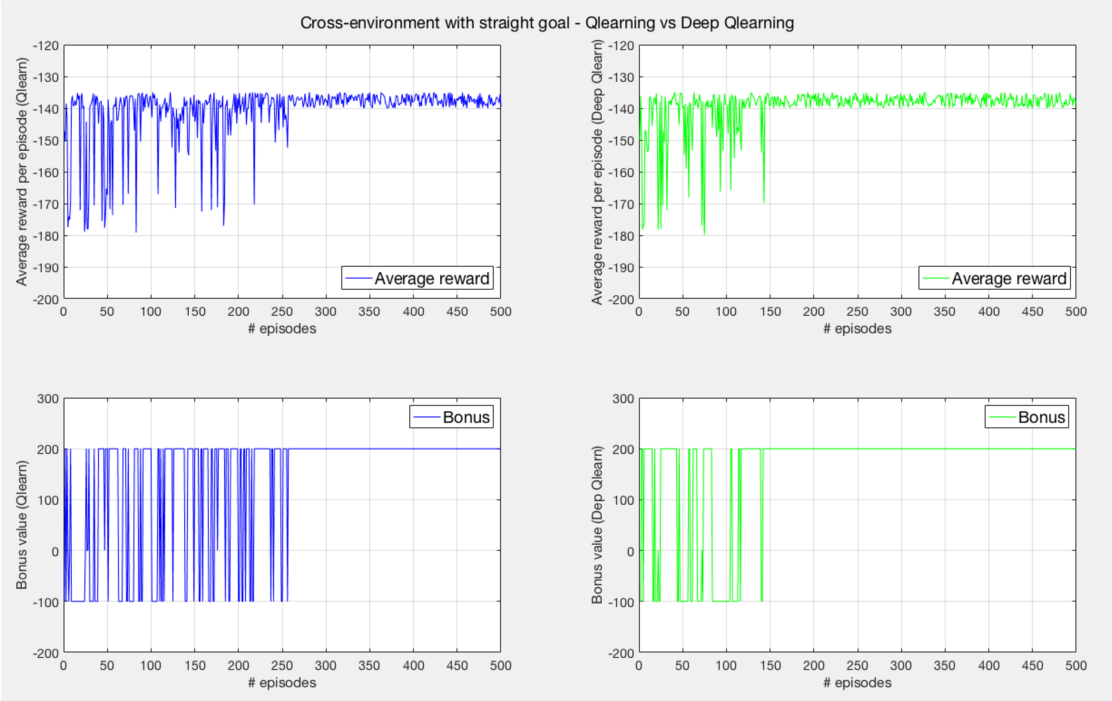

Figure 5.13: Results of learning cross-environment with straight goal (see figure 4.14c), applying Q-learning and Deep Q-learning with exponential Euclidean distance as reward function (equation 4.5). In the upper plots, the y-axis of each plot represents the average cumulative reward that the agent gains at the end of each episode. A training time of 1000 episodes has been chosen, but just the first 500 episodes are shown to highlight the trend before convergence is reached. In the lower plots, the behavior of the bonus is shown (y-axis) with respect to the number of available episodes.

In order to assess each algorithm in each cross-environment, convergence episode and probability of success and collision are also reported (see table 5.4).

Cross-environment with goal on the left (graph 5.11)			
	Convergence episode	*Probability of success*	*Probability of collision*
Discretized Q-learning	214	85.5%	14.2%
Deep Q-learning	57	12.3%	87.7%
Cross-environment with goal on the right (graph 5.12)			
	Convergence episode	*Probability of success*	*Probability of collision*
Discretized Q-learning	416	73.1%	23.9%
Deep Q-learning	410	68.8%	29.3%
Cross-environment with straight goal (graph 5.13)			
	Convergence episode	*Probability of success*	*Probability of collision*
Discretized Q-learning	257	53.4%	43.8%
Deep Q-learning	145	54.8%	44.8%

Table 5.4: Comparison among convergence episode, probability of success and probability of collision for each cross-environment for each algorithm.

As possible to notice from both table 5.4 and figures 5.11, 5.12 and 5.13, Deep Q-learning converges to the (sub)optimal trajectory faster than discretized Q-learning in all the environments. In particular, it takes just 57 episodes to learn the cross-environment with goal on the left, even

if in the phase before convergence the agent collides with obstacles most of its time (i.e. 87.7%). This situation demonstrates how much combining planning with experience replay can be effective: the planning strategy helps the agent to faster learn to move its end-effector in the right direction, while the experience replay with mini-batches makes use of that knowledge about the goal location to quickly improve the behavior of the agent and make it even more goal-oriented. Nevertheless, if the goal is located on the right, the Deep Q-learning agent takes 410 episodes to reach convergence with respect to the 416 episodes of discretized Q-learning, even though the right-goal environment is almost symmetric with respect to the left-goal one. This slower convergence rate is mainly due to the initial configuration of the robotic arm and to the selected action vector 5.1. As already specified, the second link is the one that is allowed to perform longer movements and, in the initial configuration, it is already rotated clockwise in the direction of the goal (see figure 4.14b); thus, even if it performs few clockwise actions, it tends to frequently collide with the obstacle on its right and, consequently, reset again to its initial configuration. As a result, the agent takes some more episodes to understand that it should not select too many clockwise actions for the second link, even if those actions make the end-effector to be closer to the goal.

On the other hand, the planning strategy has not been applied in learning the straight-goal cross-environment in figure 4.14c. Under these conditions, the goal marker is almost in line with the end-effector marker and, consequently, the agent should freely choose both clockwise and counter-clockwise end-effector actions depending on the motion of the first two links. Even though, Deep Q-learning continues to be faster than discretized Q-learning as expected (145 episodes to converge with respect to 257 episodes of discretized Q-learning). Therefore, the larger state-space makes the Deep Q-learning agent to present higher collision probability than the discretized Q-learning case, as happened in all the performed experiments (for further details see section 5.1.2).

Despite all these considerations, all the implemented cross-environment experiments have been carried out successfully, learning accurate sub-optimal policies. Thus, at this point, the acquired knowledge represented by the Q action-value function can be adopted to make the robotic arm to be able to quickly learn mostly all the possible configurations of a pipe network (see figure 4.15), applying transfer learning approaches.

5.2.3 Testing RL algorithm adaptability: a transfer learning approach

One of the requirements of this project is to verify whether the chosen RL algorithm is environment-independent, i.e. if it does not require too many information about the environment in which it operates, so that its acquired knowledge becomes easily adaptable to similar environments, initializing the Q value-function in the same way (see section 3.3.1). This flexibility is of fundamental importance when the robot is actually asked to inspect a complete pipe-network: it could learn a simple configuration of the pipe like the cross-environment in figures 4.14a, 4.14b and 4.14c and then apply the obtained knowledge to inspect the whole pipe network efficiently and effortlessly.

In section 5.2.2, Deep Q-learning has been demonstrated to be faster in reaching convergence with respect to discretized Q-learning, even if a larger and more complete state-space is considered. Consequently, this RL approach is adopted to test the transfer learning strategy. In machine learning, transfer learning is a technique in which knowledge acquired while solving one problem is stored, so that it can be re-used for similar applications afterwards. In this case, the acquired knowledge corresponds to the policy that characterized the Q neural network, which has already been trained for each of the cross-environment situations (figures 4.14a, 4.14b and 4.14c) and then stored in the MATLAB folder.

As already mentioned at the beginning of the chapter, the environments in figures 4.14d and 4.14e have been considered to test a transfer learning strategy from the cross-environment in

figure 4.14a. These acute and obtuse curves correspond to the remaining possible layouts of a pipe-network (see figure 4.15a and 4.15c) and so, learning both of them will make the agent able to face any kind of pipe network. In this case, only left-goal environments have been considered to avoid wasting time in repetitive and non-representative experiments for right-goal environments, because if the transfer learning approach is proved to work for the left-goal situation, it can be also easily adaptable to the right-goal one.

To get a better visualization of the objective of transfer learning approach, the figure below is provided:

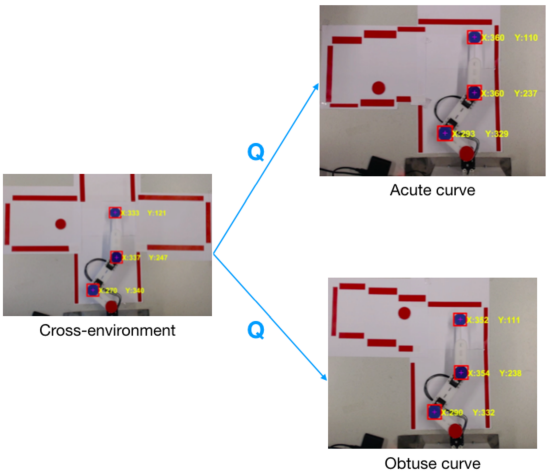

Figure 5.14: Transfer learning approach to make use of gained knowledge (Q neural network) of the cross-environment to quickly learn acute curve and obtuse curve layouts of the pipe.

In order to make use of that gained experience, a goal-oriented transfer learning approach has to be implemented. In particular, the cited transfer learning strategy should be as much generic as possible, so that it can be easily employed in different layouts of the environment. As possible to notice from figure 5.14, the knowledge that can be beneficially transferred to each of the curves is the initial trajectory up to when the curve is entered by the end-effector. From that moment onward, the agent should start exploring in order to autonomously localize the new goal and constraints in each different curve and, consequently, learn the new (sub)optimal policy. To understand when the end-effector has entered the curve, it is sufficient to place the end-effector at that location and use the camera to evaluate the Euclidean distance between the end-effector marker and the marker of the lower obstacle of the curvature. If this distance is smaller than 100 pixels (almost 9 centimeters), it means that the end-effector has reached the curve. With these ideas in mind, the Deep Q-learning with transfer-learning algorithm shown in 14 has been developed.

As possible to notice from algorithm 14, just the second and the third links are allowed to move in the moment in which the end-effector enters the curve, applying an ϵ-greedy strategy with $\epsilon = 0.1$. These choices have been made for mainly three reasons:

- The first link does not actually contribute to the motion. Since the first link is the shortest, its motion does not facilitate the learning of the environment, once the end-effector has reached the curve. Just the second and third link degrees of freedom are sufficient in that situation.
- Facilitate the learning process: moving just two links make the learning phase faster since less actions correspond to less possible trajectories for the robotic arm.

Algorithm 14: Deep Q-learning with transfer-learning: from left-goal cross-environment to acute or obtuse curve

Init Q network as the already trained cross-environment Q;
while *episode != final episode* **do**
  Init and observe S;
  while *step != final step* **do**
    if *distance(ee-obs) < 100* **then**
      Choose only 2^{nd} and 3^{rd} link A from S using ϵ-greedy π derived from Q;
    else
      Follow the greedy π derived from Q;
    end
    Take action A, observe R and S';
    Store experience (S, A, R, S') in D;
    Sample best 75 transitions (S_s, A_s, R_s, S'_s) from D looking at the correspondent R;
    Calculate target T_s;
    if *condition* **then**
      S' terminal state then $T_s = R_s$;
    else
      $T_s = R_s + \gamma max_a Q(S'_s, a)$;
    end
    Train the Q-network using $(T_s - Q(S_s, A_s))^2$ as loss function;
    $S = S'$;
  end
end

- Reducing the exploration period to just 10% of the available episodes encourages the agent to exploit the learned experience.

In the moment in which the action have been taken, the experience replay with mini-batches section is performed. While in the previous version of the algorithm (algorithm 3) the training of the Q network was performed on randomly selected 75 transitions from the Replay Memory, in this case the best 75 experiences are instead selected. To identify these best samples, it is sufficient to look at the reward value of each experience and select the 75 experiences which present the highest rewards. In this way, the agent will tend to exploit the best policies it has learned so far, instead of exploiting non-optimal trajectories which bring it away from the goal. With this approach, the robotic arm is no more allowed to freely explore new areas of the environment, because it would tend to explore just environment states which are closer to the ones with highest optimal values. This strategy speeds up the convergence to the learned policy, but, at the same time, the drawback of not experience the actual optimal policy is present. Since in this case the robotic arm is not allowed to explore a lot but just inside the curve, the mentioned disadvantage of this strategy is not so restrictive, because the end-effector is already quite closed to the goal, therefore, a sub-optimal policy can be still discovered and it should be a good approximation of the optimal one.

At this point, the results obtained for the acute and obtuse curve environments (figures 4.14d and 4.14e) applying the proposed transfer learning approach are reported.

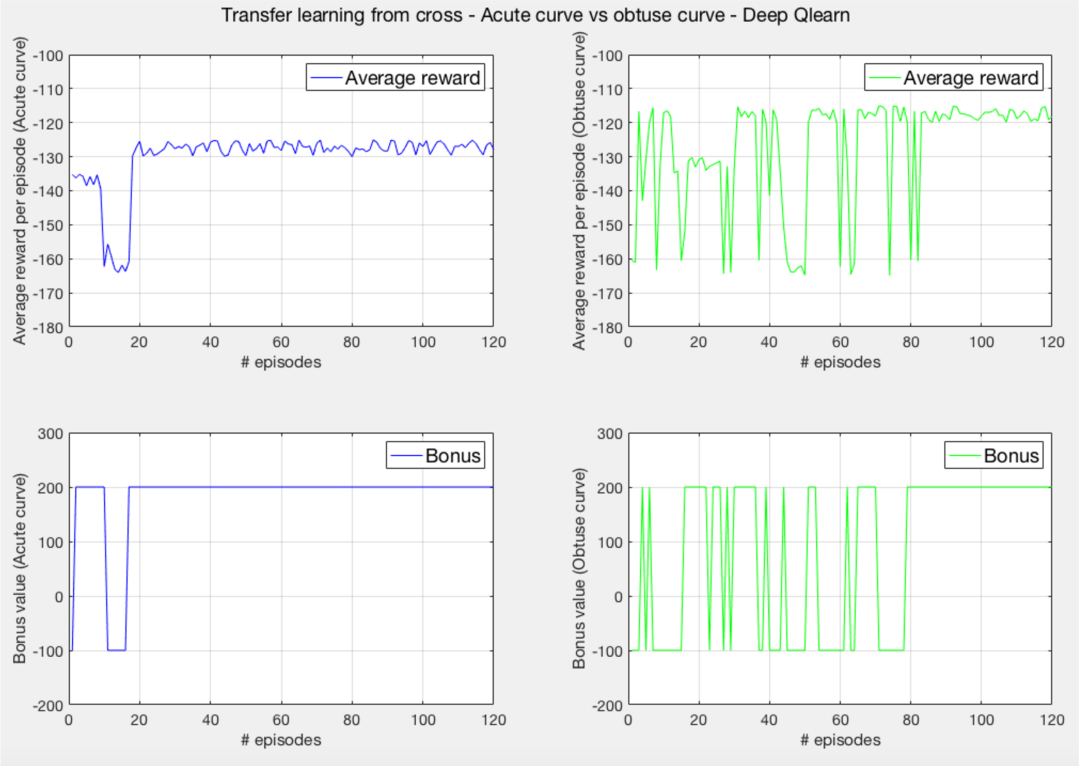

Figure 5.15: Results of transfer learning from cross-environment to acute and obtuse curve. In the upper plots, the y-axis of each plot represents the average cumulative reward that the agent gains at the end of each episode. A training time of 120 episodes has been chosen, but just the first 120 episodes are shown to highlight the trend before convergence is reached. In the lower plots, the behavior of the bonus is shown (y-axis) with respect to the number of available episodes.

As possible to notice from the plots in figure 5.15, transfer learning approach allows a really fast convergence to the sub-optimal policy in both the cases, i.e. 17 episodes for the acute curve layout, with respect to 266 episodes without transfer learning, and 83 episodes for the obtuse curve layout, with respect to 387 episodes without transfer learning. Nevertheless, the acute curve results to be much easier to be learned with respect to the obtuse curve. As a matter of fact, while learning the acute curve, the Deep Q-learning agent collides with obstacles for just five times with respect to the 41 collisions while learning the obtuse curve. This difference is mainly due to the fact that the acute curve goal location is not so far with respect to the learned left-goal cross-environment (look at figures 4.14a and 4.14d). Consequently, the agent can still make use of the acquired experience also while learning the new configuration of the pipe and the exploration period can just help it to experience the small state-space between the previous goal location and the new one. On the other hand, in the obtuse curve configuration, the agent tends to reach the lower side of the new curve, because it is part of the trajectory learned in the left-goal cross-environment; however, exploiting that learned policy makes the robotic arm to collide very often in the cited area from different state configurations. As a result, the agent requires more episodes and more exploration to figure out the sub-optimal path that allows it to reach the goal without colliding.

After this analysis, it is possible to state that transfer learning approach is beneficial and efficient for both the new environments, because it permits to really accelerate the training period. Even though, it is able to achieve ideal results only in the moment in which the new goal location is not so far from the already learned trajectory. If this is not the case, the implemented agent still requires some exploration steps to figure out the new goal location and learn an accurate optimal policy.

At this point, a final experiment has been performed to test transfer learning approach without reset the robotic arm once a collision occurs, but allowing it to move back to the state before the collision took place, continuing the iterations. As mentioned in section 5.1.1, this collision avoidance approach was not so efficient in simulation environment, because of the fact that the agent was often making use of all the available 250 iterations per episode before reaching the goal, since it was colliding with the present obstacles from different consecutive states before being able to avoid them completely. Consequently, this approach was discovered to be much slower than the "reset" one. Nevertheless, since applying a transfer learning strategy makes the algorithm to converge much faster as just demonstrated, this "no-reset" collision avoidance approach has been tested on the environment in figure 4.14d and compared with the "reset" case (see figure 5.16).

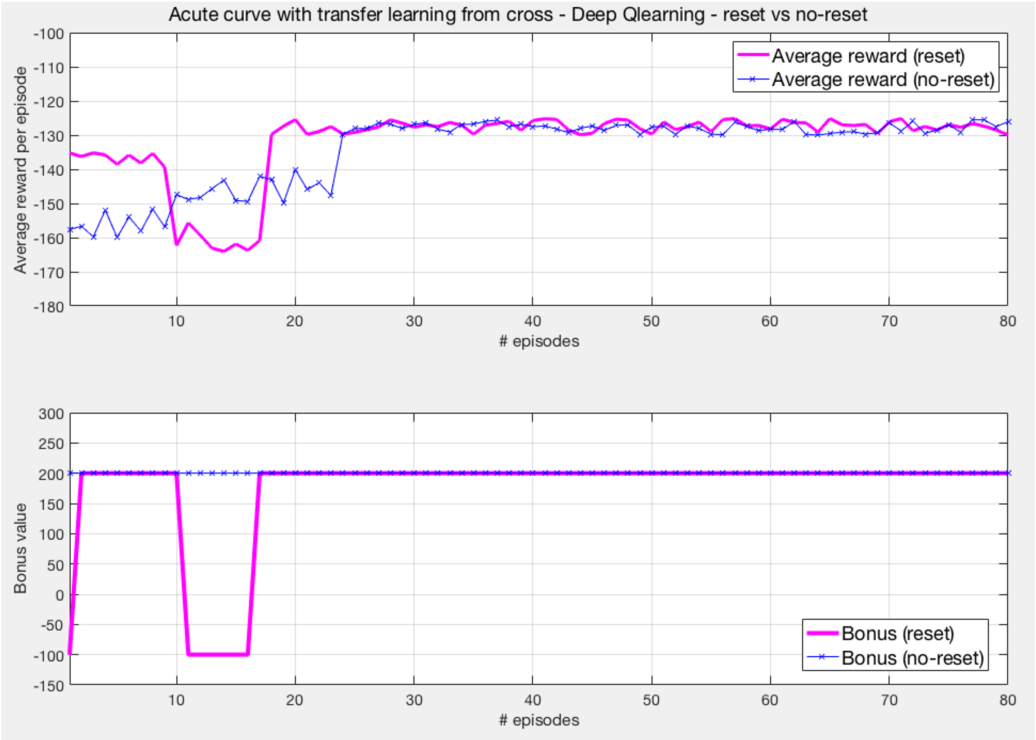

Figure 5.16: Transfer learning to acute curve with goal at the bottom (see figure 4.14d), applying Deep Q-learning with exponential Euclidean distance as reward function (equation 4.5). In the upper plots, the y-axis of each plot represents the average cumulative reward that the agent gains at the end of each episode for both reset and no-reset collision avoidance approach. A training time of 1000 episodes has been chosen, but just the first 80 episodes are shown to highlight the trend before convergence is reached. In the lower plots, the behavior of the bonus is shown (y-axis) with respect to the number of available episodes for both reset and no-reset case.

As possible to notice from the lower bonus plot of figure 5.16, the "no-reset" case always reach the goal at the end of each episode with respect to the "reset" approach (see the lower plot in figure 5.16). Moreover, its average cumulative reward is better than the one of "reset" case, because, even if it makes use of more actions for each iteration, it does not end an episode colliding with an obstacle which is not so closed to the goal as in the episodes range 11-18 of the "reset" approach. Even though, the "no-reset" strategy reaches convergence in 23 episodes with respect to the 17 episodes required when the manipulator is reset after a collision.

According to these considerations, the "no-reset" approach takes more episodes to learn the sub-optimal trajectory, even if its average cumulative reward is better, and, at the same time, it is slower than the other approach: it requires an average of 21 seconds per episode, with respect to 15 seconds of the "reset" case.

5.3 Final evaluation of the proposed algorithms

In this section, a final evaluation of the proposed reinforcement learning algorithms is performed, analyzing how the project requirements described in section 3.3 have been satisfied.

Value-function based RL algorithms have been adopted in this project, with a particular interest towards on-policy and off-policy approaches. Before applying them on the real-setup, RL and NN parameters have been investigated both in simulation (see section 5.1.1) and on the real-setup (see section 5.1.2). In RL perspective, the ϵ exploration-exploitation trade-off has been noticed to really affect the convergence rate of the algorithms (see figure 5.4): too high exploration (i.e. $\epsilon = 0.6$) makes the convergence slower, while too high exploitation (i.e. $\epsilon = 0.1$) does not guarantee convergence to the optimal policy. On the other hand, when Deep RL algorithms are adopted, it is necessary to appropriately tune the number of hidden neurons in the feed-forward neural network structure (see figure 5.5): a low amount of hidden neurons (i.e. $HN = 50$) does not ensure a sufficient combination of the training examples, while increasing the amount of hidden neurons (i.e. $HN = 300$) slow down the training time but ensures convergence to the optimal solution.

Another RL factor that has been deeply investigated is the reward function. As a matter of fact three reward functions have been considered: standard Euclidean distance, exponential Euclidean distance and Gaussian Euclidean distance (equations 4.4, 4.5 and 4.6), all three based on the Euclidean distance between the end-effector and the goal. The most efficient reward function has been proven to be the exponential Euclidean reward, because it is the only one that actually guarantees a always increasing trend and, consequently, a goal-oriented prioritization of the value estimates of the state-action pairs (see table 5.3).

Once all the parameters have been appropriately determined, the four implemented agents (i.e. two on-policy algorithms and two off-policy algorithms) have been assessed on the real-setup by observing the results reported in sections 5.1.2 and 5.1.3. Comparing the performance of the implemented four agents (see figure 5.10 and table 5.3), it is easy to notice that the off-policy Q-Learning-based agents are able to obtain better results than the SARSA-based agents, both in terms of success and convergence rate as well as optimality of the learned policy. Consequently, Q-learning-based algorithms are able to reach the goal with a less amount of actions with respect to SARSA-based algorithms (see histogram 5.10), mainly because SARSA usually prefers a longer but safer trajectory with respect to the shortest path learned by the Q-learning approach.

These algorithms have been also evaluated according to the way they deal with large state-space. If on one hand, discretized Q-learning and SARSA apply a discretization on the available environment state-space in order to be able to store all the possible state-action pairs in a Q-table, on the other hand Deep RL approaches like Deep Q-learning and Deep-SARSA make use of a function approximation strategy (i.e. feed-forward neural network) to manage the full continuous state-space. Employing continuous state-space allows to the Deep RL agent to obtain more detailed states of the environment and so a more accurate approximation of the Q value-function, converging faster to the sub-optimal policy. Nevertheless, a more detailed state-space makes these agents collide with obstacles even more, because of the fact that they can collide from a larger range of environment states.

For all the aforementioned reasons, Q-learning based approaches have been taken as a reference for the final experimental phase of the project (see section 5.2). In this perspective, both the algorithms have been tested in the cross-environment in figures 4.14a, 4.14b and 4.14c (see section 5.2.2), by applying an initial planning strategy to encourage the robotic arm to move in the direction of the goal. Satisfying results have been obtained under these conditions in particular with Deep Q-learning algorithm (see table 5.4) thanks to the combination of planning and experience replay with mini-batches. Consequently, the latter has been further em-

ployed to investigate RL adaptability to similar configurations of the environment (see figures 4.14d and 4.14e and section 5.2.3) adopting transfer learning strategies. In this context, the transfer learning approach has been proven to be an effective tool when the considered Deep Q-learning agent has to learn similar but not equal configuration of the pipe (see plots in figure 5.15 and 5.16). As a matter of fact, transfer learning allows to speed up the convergence rate of 60% in case of obtuse curve and 90% in case of acute curve, completely satisfying environment independence requirement. In this way, the Deep Q-learning algorithm can be adopted to learn any possible configuration of a pipe network (see figure 4.15).

5.3.1 Critical appraisal

In this section, a critical appraisal is reported focusing on the points of strength and weakness of the proposed navigation approach.

Strengths

- **Adaptability**.

  The implemented RL-based navigation system is easily adaptable to different kinds of constrained environments and, if transfer learning approach is employed, it allows a robotic arm to learn similar environment layouts quickly.

- **No accurate environment model required**.

  All the implemented RL algorithms operate without the need of an accurate model of the environment; as a matter of fact, since value-function based approaches are applied, they are all model-free and the model is learned through interaction between the agent and the environment. This interaction is guaranteed by the presence of camera and encoders signals.

- **Reduction of training time**.

  New and modified versions of ϵ-greedy algorithms have been provided to guarantee a more uniform and faster exploration of the environment. With the introduction of the goal-oriented planning (section 5.2), the RL agent has become able to directly orientate the end-effector towards the goal, avoiding wasting time in non-of-interest environment regions. Moreover, the employment of Euclidean distance based reward functions in combination with sparse rewards (positive for the successes and negative for collisions) encouraged the agent to progress to more rewarded states in a shorter period of time.

- **Fast and reliable physical setup**.

  The developed setup has been discovered to be fast-responding and reliable in all the situations. It can be easily employed to test different kinds of algorithms, since its software architecture is well documented and easily modifiable on MATLAB by non-expert users. The communication with the camera is also straightforward but good light conditions should be guaranteed to be able to detect markers without any troubles.

- **Tests in both simulation and real world**.

  Tests in both simulation and real-world have been performed. All the tested algorithms have been brought success when learning policies and, intrinsically, the algorithms have been demonstrated to be able to extend to different kinds of virtual and physical environments.

- **Numerous tests have been performed.**

  Many tests have been performed for most of the algorithms in particular to appropriately tune and assess RL parameters. Consequently, the resulting parameters selection is quite optimized for the current navigation problem.

Weaknesses

- **Low repeatability** .

  Single experiments are not repeatable even if the environment conditions do not change. This situation is due to the fact that the ϵ-greedy algorithm is characterized by a random action selection component which is unpredictable and different in each new experiment.

- **Unpredictable learning time**.

  As previously mentioned, experiments are not repeatable and, consequently, the learning period before reaching convergence cannot be predicted even if the environment conditions are the same. Nevertheless, each experiment with each algorithm usually starts converging at an episode which is in the same order of magnitude of other experiments under the same conditions with the same algorithm.

- **Collisions avoidance**.

  Most of the experiments have been performed with the reset obstacles avoidance approach (see section 5.1.1). Consequently, when a collision occurred, the manipulator was reset to a predefined initial configuration. This solution cannot be applied in a physical environment with actual obstacles (e.g. pipe environment) because the robotic arm could not be able to reach its initial position due to the obstacles location.

- **Non-fluent motion of the manipulator**.

  Since the action vector is considered to be discrete, the manipulator cannot freely move with just one movement to the desired location, but each joint can just rotate of some predefined degrees. Consequently, the overall observable motion is not fluent as expected.

- **Camera-based algorithm**.

  The proposed algorithm is camera-based if the physical setup is considered. As already mentioned, camera signals are sufficiently reliable but they cannot be obtained in low-light conditions. Thus, if an actual pipe is employed, the camera could never be adopted and alternative solutions based on other types of sensors should be figured out.

5.3.2 Comparison with existing RL-based approaches

As already specified in section 3.1, RL has been increasingly adopted in arm planning applications because, with respect to optimal control, it does not require a specific model of the system under consideration, but it operates through interaction between the RL agent and the environment. This characteristic is of great benefit when the available model is not accurate and the use of that imprecise knowledge would make the objective hard to be reached.

Despite these considerations, in the literature there are not too many examples of applications of RL in over-constrained environment navigation tasks. In (12) and (13), two robotic arms are trained to learn to reach a goal location inside constrained environment avoiding obstacles applying Q-learning algorithm. In both the cases, a sparse reward function is considered, which is positive when the target is reached and really negative once a collision occurs. This sparse reward function has also been applied into the adopted RL architecture (i.e. +200 when the goal is reached and -100 when the manipulator structure hits an obstacle), but it has been then integrated with another trivial reward function, i.e. the Euclidean distance reward (standard, exponential and Gaussian). The exponential Euclidean distance reward function has been already applied in (14) to make a six degrees of freedom robot arm with two grippers able to pick

up an object. Even though, no comparison among different reward functions in robotics/navigation applications is provided in (14) to justify the efficiency of the selected reward.

Concerning the curse of dimensionality issue, most of the examples found in the literature prefer to discretize the state and action space under consideration not to over-complicate the algorithm design (15), (16), (17), (10), (9). Nevertheless, good results have been obtained adopting neural networks as function approximators for the Q-table (4), (9), (10), (14), (8). Consequently, this approach has been applied in the research in order to compare discretized RL algorithms together with deep RL algorithms. To guarantee convergence to sub-optimal solutions and avoid convergence to local minima or even divergence, the basic deep RL algorithm has been modified to include experience replay, as already proposed in (8), (14). Experience replay eliminates the problem of correlation between consecutive transitions and reduces variance between different updates. As a consequence of this latter modification, really satisfying results have been obtained with Deep Q-learning algorithm.

6 Conclusions and recommendations

6.1 Conclusions

Employment of inspection robots in pipe network is really beneficial: their small size allow them to navigate inside most of the hydraulic and gas pipes, eliminating the need to dig or open a wall for further inspections or pipe replacements. These kinds of robots are usually equipped with cameras and sensors for collecting different types of data, such as travelled route, diameter of the inspected pipe, geographical position of possible leaks or damaged sections, internal pressure and temperature and so on. However, they are usually remotely driven by an external human operator, that, thanks to the gathered data, decides the navigation pattern. In this perspective, an autonomous navigation system would be of great help to make the cited robots able to independently explore the pipe network without the need of an external supervision.

Reinforcement Learning approaches have been demonstrated to be effective tools in implementing automatic navigation in unknown environments. Reinforcement learning can be considered as an optimization problem, whose goal is to establish a control policy that obtains a maximum reward, starting from the state in which the robot is located in the environment in which it operates. These reinforcement learning techniques are particularly suitable for those robotics applications where robotic errors are not immediately critical and for which a function can be defined to evaluate the performance of the robot. In fact, reinforcement learning uses a global assessment measure, i.e. the reward function, which controls and drives the learning process. In this context, it differs from supervised learning schemes, which use specific target values for the individual units. In robotics field, this property can be particularly advantageous in those situations in which only the desired overall behavior of the robot is known; however, at the same time, this can be a problem, as it is difficult to determine which parameter should be modified within the controller to increase the cumulative reward. It is precisely through the feedback of the performances that the agent learns the correspondence between the state (the representation of a particular situation) and the action to be taken.

In this project, value-function based reinforcement learning algorithms have been investigated to make a three degrees of freedom planar robotic arm able to autonomously navigate inside a constrained pipe-like environment, as the ones reported in figure 4.14. In order to implement significant experiments of the algorithms, a real robotic arm has been designed and then manufactured with RaM laser cut technology. Three servo-motors have been employed to actuate the joints of the cited manipulator, that has been then placed inside an environment with markers representing virtual obstacles and a goal that should be reached by its end-effector. The interaction between the robotic manipulator and the mentioned markers is recorded by an external un-calibrated monocular camera, which is able to detect and localize the features of interest in the scene (i.e. obstacles, goal, joints and end-effector of the manipulator). In this way, no model of the robotic arm is required to drive its motion, since a vision-guided state estimation approach has been employed, and, at the same time, no further hardware is necessary to detect distances or collisions (e.g. position and torque sensors). A significant MATLAB-based simulation environment has been also developed to facilitate and speed up the first experimental stage of the project.

These two environments, i.e. simulation and real-setup, has been taken as a reference to test two reinforcement learning algorithms: Q-learning and SARSA, with both discretized state-action pairs and continuous state-space with discrete actions. In the discretized case, both the approaches make use of a Q-table to store all the values estimates of each state-action pair, making the exploration more imprecise but less likely affected by collisions with the present obstacles. On the other hand, the continuous state-space implementation is able to carry out a

more detailed approximation of the value-function Q through a deep neural-network, improving the convergence, but, at the same time, making the agent able to collide from a larger range of environment states.

In order to facilitate the learning phase both in terms of exploration-exploitation trade-off and convergence rate, algorithms parameters have been accurately tuned to figure out the correlation between their values and the performance of the navigation. In particular, different kinds of Euclidean distance-based reward functions have been analyzed to make the navigation quicker and more goal-oriented: simple Euclidean distance reward function, exponential Euclidean distance reward function and Gaussian Euclidean distance reward function. The second has been found to be the most efficient (look at the table in 5.3), because of the fact that it presents an always increasing trend also in proximity of the goal, which is not the case of Gaussian Euclidean distance that starts decreasing when the goal is closer. With the performed tuning choices, all the approaches have been demonstrated to be able to learn a (sub)optimal trajectory from the initial configuration of the robotic arm to the goal. SARSA on-policy algorithms have been found to be more "conservative" with respect to off-policy Q-learning: they prefer to learn longer and most likely safer trajectories, avoiding dangerous situations for the agent. As reported in table 5.3, discretized SARSA algorithm present on average smaller probability of collision with respect to discretized Q-learning, i.e. 31.2% with respect to 31.5%. On the contrary, Q-learning based algorithms are more goal-oriented, shortening the required path as much as possible. As a matter of fact, as noticeable from figure 5.10, Q-learning-based algorithms require on average 78 actions to reach the goal with respect to 93 necessary actions for SARSA-based algorithms. Nevertheless, on-policy methodologies have been noticed not to be so much efficient in over-constrained environments like the considered ones, because avoiding obstacles on one side of the pipe can result into a collision on the other side of the pipe itself. Indeed Deep SARSA collides even more than Deep Q-learning with exponential Euclidean reward, 36.2% versus 29.3%. Consequently, Q-learning based algorithms have been identified as the most competent techniques in learning different configurations of the pipe network with minimum effort.

Due to the latter consideration, Q-learning-based algorithms have been applied to test algorithm adaptability to similar configurations of the environment, applying a transfer learning approach (see section 5.2 and figure 4.14). Transfer learning is based on the idea of adopting the knowledge acquired for a certain configuration of the environment to learn similar but not equal layouts, initializing the new Q with the already trained one. With this idea in mind, it has been demonstrated that transfer learning allows to speed up the convergence rate of the considered Deep Q-learning algorithm of at least 60% with respect to a "new-to-be-learned" environment. This finding is of fundamental importance in navigation perspective, because it allows a non-modelled robotic system to easily navigate inside different pipe configurations with the minimum amount of training. As a matter of fact, the correspondent RL agent could learn a basic configuration of the environment like the cross shown in figure 4.14 and then utilize this knowledge to freely navigate inside any possible configuration of a pipe network, e.g. characterized with obtuse and acute curves.

This research demonstrated that RL algorithms (in this case value-function based) can be adopted to make a three degrees of freedom planar robotic arm able to autonomously navigate inside constrained pipe-like environments as the considered ones. These approaches present many pros in terms of adaptability to different non-modelled environment layouts and quite feasible training periods for autonomous applications, usually in the range of 2-3 hours depending on the number of training episodes. Even though, RL is characterized by two main cons:

- Low repeatability, i.e. each experiment is not repeatable since the exploration of the environment is based on random action selection.

- Unpredictable learning time: since each experiment is not repeatable, convergence can be reached after a different amount of episodes, even if the experiment is performed under the same conditions.

To circumvent these two related issues, it is possible to occasionally save in the workspace the Q, so that, if the experiment has to be repeated another time, the already acquired knowledge about the cited environment can be used to facilitate and speed up the learning. Eventually, it is possible to state that the implemented RL navigation system can be used to learn movement policies for tasks that involve a goal to be reached and many obstacles to be avoided.

6.2 Recommendations for future researches

Even if satisfying results have already been obtained in particular applying a Deep Q-learning approach with experience replay and mini-batches, further analysis can still be performed to improve convergence and to guarantee higher flexibility of the proposed approaches. The integration of continuous actions into the continuous state-space architecture could be of great interest. The robotic arm would be able to reach the goal also applying just one wide movement of the end-effector link. According to the literature, policy-search methodologies are more suitable for allowing the employment of continuous state-action, since they are no more related to the estimation of the proper value-function as in the considered case. Even though, to make use of the already implemented architecture, it is possible to extend it to an actor-critic approach. In this perspective, the policy becomes completely independent from the value-function: the actor, i.e. the policy, chooses the action to be performed, while the critic, i.e. the value-function Q, criticizes that action selection (4). With this approach in mind, a deterministic policy gradient algorithm can be applied to deal with the continuous action space and then integrated with the standard Deep Q-learning algorithm to learn the required optimal policy. If this objective is achieved, the action-space can be extended to the velocity signals, so that the robotic arm can move towards different environment states with different speeds and, consequently, determine how to reach the goal faster.

Moreover, further analysis should be made in the collision avoidance field. As already mentioned, in this implementation the robotic arm can be reset when a collision occurs or it can move back to the previous state before the collision. The former case is faster, but could make the manipulator colliding with obstacles in its way back to its initial configuration. The latter approach has been noticed to be too slow and, consequently, applied only when transfer learning approaches are tested. Thus, an optimal obstacle avoidance strategy has not been figured out yet; a good suggestion could be to design a mapping of the obstacles location and, consequently, make the robotic arm to completely avoid those states which are closed to the obstacles. In this way, the manipulator would tend to remain further away from the obstacles, reducing the probability of collision.

In the proposed setup only virtual obstacles symbolized by markers are employed. In order to make the environment even more representative of an actual pipe, physical obstacles can be added. Adopting physical obstacles would require the implementation of an interaction strategy between them and the structure of the manipulator. In this perspective, the integration of motors equipped with a torque control architecture would be advantageous. Thus, the torque acting on the joints of the manipulator could be easily estimated and, consequently, based on its value, a collision could be identified.

From a more practical point of view, the adopted setup can just operate in limited work-spaces, because of its small dimensions. Its structure could have not been further elongated due to its planar configuration: the first joint of the robot has to support the whole weight of the manipulator and counterbalance the momentum of the gravity force. Consequently, it was not possible to increase its size even more, because the structure would have been affected by too

many vibrations, making the motion not accurate. Thus, two solutions have been figured out to obtain larger workspace:

- Make use of smaller and lighter servo-motors.
- Put the manipulator in a standing position and use a vertical panel to represent the pipe environment instead of the horizontal one. Consequently, the camera can be placed at a certain horizontal distance to record the scene.

At this point, a gripper can be employed instead of the actual end-effector, in order to make the considered robot able to grasp an object. This additional degree of freedom could be beneficial for collecting and moving objects which are present in the workspace.

A Appendix 1

In this appendix, all the necessary information about the adopted setup are provided, both in terms of mechanical implementation and software architecture.

A.1 DYNAMIXEL AX12A from Robotis datasheet

Description	Specification
Weight	54.6g
Dimension	32mm x 50mm x 40mm
Gear Ratio	254 : 1
Operation Voltage (V)	12
Stall Torque (N.m)	**1.5 (12V)**
Stall Current (A)	1.5
No Load Speed (RPM)	**59 (12V)**
Motor	Cored Motor
Minimum Control Angle	**about 0.29 degrees x 1,024**
Operating Range	Actuator Mode : 300 degrees Wheel Mode : Endless turn
Operating Voltage	**9~12V (Recommended voltage : 11.1V)**
Max. Current	900mA
Standby Current	50mA
Operating Temperature	-5°C ~ 70°C
Command Signal	Digital Packet
Protocol	Half duplex Asynchronous Serial Communication (8bit,1stop,No Parity)
Link (physical)	TTL Level Multi Drop (daisy chain type Connector)
ID	254 ID (0~253)
Baud Rate	7843bps ~ 1 Mbps
Feedback Functions	Position, Temperature, Load, Input Voltage, etc.
Material	Case : Engineering Plastic Gear : Engineering Plastic
Position Sensor	**Potentiometer**
Default Setting	**ID #1 (1 Mbps)**

Figure A.1: DYNAMIXEL AX12A (20)

A.2 Power supply of the servo-motors

A power supply is used to deliver the required voltage (12V) to the Dynamixel AX12A servo-motors.

Figure A.2: Power supply that provides the operating voltage to DYNAMIXEL AX12A motors (12 V as specified in A.1)

A.3 DYNAMIXEL AX12A features addresses

Area	Address (Hexadecimal)	Name	Description	Access	Initial Value (Hexadecimal)
EEPROM	0 (0X00)	Model Number(L)	Lowest byte of model number	R	12 (0X0C)
	1 (0X01)	Model Number(H)	Highest byte of model number	R	0 (0X00)
	2 (0X02)	Version of Firmware	Information on the version of firmware	R	-
	3 (0X03)	ID	ID of Dynamixel	RW	1 (0X01)
	4 (0X04)	Baud Rate	Baud Rate of Dynamixel	RW	1 (0X01)
	5 (0X05)	Return Delay Time	Return Delay Time	RW	250 (0XFA)
	6 (0X06)	CW Angle Limit(L)	Lowest byte of clockwise Angle Limit	RW	0 (0X00)
	7 (0X07)	CW Angle Limit(H)	Highest byte of clockwise Angle Limit	RW	0 (0X00)
	8 (0X08)	CCW Angle Limit(L)	Lowest byte of counterclockwise Angle Limit	RW	255 (0XFF)
	9 (0X09)	CCW Angle Limit(H)	Highest byte of counterclockwise Angle Limit	RW	3 (0X03)
	11 (0X0B)	the Highest Limit Temperature	Internal Limit Temperature	RW	70 (0X46)
	12 (0X0C)	the Lowest Limit Voltage	Lowest Limit Voltage	RW	60 (0X3C)
	13 (0X0D)	the Highest Limit Voltage	Highest Limit Voltage	RW	140 (0XBE)
	14 (0X0E)	Max Torque(L)	Lowest byte of Max. Torque	RW	255 (0XFF)
	15 (0X0F)	Max Torque(H)	Highest byte of Max. Torque	RW	3 (0X03)
	16 (0X10)	Status Return Level	Status Return Level	RW	2 (0X02)
	17 (0X11)	Alarm LED	LED for Alarm	RW	36(0x24)
	18 (0X12)	Alarm Shutdown	Shutdown for Alarm	RW	36(0x24)
RAM	24 (0X18)	Torque Enable	Torque On/Off	RW	0 (0X00)
	25 (0X19)	LED	LED On/Off	RW	0 (0X00)
	26 (0X1A)	CW Compliance Margin	CW Compliance margin	RW	1 (0X01)
	27 (0X1B)	CCW Compliance Margin	CCW Compliance margin	RW	1 (0X01)
	28 (0X1C)	CW Compliance Slope	CW Compliance slope	RW	32 (0X20)
	29 (0X1D)	CCW Compliance Slope	CCW Compliance slope	RW	32 (0X20)
	30 (0X1E)	Goal Position(L)	Lowest byte of Goal Position	RW	-
	31 (0X1F)	Goal Position(H)	Highest byte of Goal Position	RW	-
	32 (0X20)	Moving Speed(L)	Lowest byte of Moving Speed (Moving Velocity)	RW	-
	33 (0X21)	Moving Speed(H)	Highest byte of Moving Speed (Moving Velocity)	RW	-
	34 (0X22)	Torque Limit(L)	Lowest byte of Torque Limit (Goal Torque)	RW	ADD14
	35 (0X23)	Torque Limit(H)	Highest byte of Torque Limit (Goal Torque)	RW	ADD15
	36 (0X24)	Present Position(L)	Lowest byte of Current Position (Present Velocity)	R	-
	37 (0X25)	Present Position(H)	Highest byte of Current Position (Present Velocity)	R	-
	38 (0X26)	Present Speed(L)	Lowest byte of Current Speed	R	-
	39 (0X27)	Present Speed(H)	Highest byte of Current Speed	R	-
	40 (0X28)	Present Load(L)	Lowest byte of Current Load	R	-
	41 (0X29)	Present Load(H)	Highest byte of Current Load	R	-
	42 (0X2A)	Present Voltage	Current Voltage	R	-
	43 (0X2B)	Present Temperature	Current Temperature	R	-
	44 (0X2C)	Registered	Means if Instruction is registered	R	0 (0X00)
	46 (0X2E)	Moving	Means if there is any movement	R	0 (0X00)
	47 (0X2F)	Lock	Locking EEPROM	RW	0 (0X00)
	48 (0X30)	Punch(L)	Lowest byte of Punch	RW	32 (0X20)
	49 (0X31)	Punch(H)	Highest byte of Punch	RW	0 (0X00)

Figure A.3: Features addresses DYNAMIXEL AX12A (20)

A.4 READ-WRITE code

This code shows how to send position and velocity values to a DYNAMIXEL AX12A servomotors and read those values from the encoders to get sure that those settings have been reached.

```
1
2  % Copyright 2017 ROBOTIS CO., LTD.
3
4  %
5  % *********     Read and Write Example      *********
6  %
7  %
8  % Available DXL model on this example : All models using Protocol
       1.0
9
10 clc;
11 clear all;
12
13 lib_name = '';
14
15 if strcmp(computer, 'PCWIN')
16   lib_name = 'dxl_x86_c';
17 elseif strcmp(computer, 'PCWIN64')
18   lib_name = 'dxl_x64_c';
19 elseif strcmp(computer, 'GLNX86')
20   lib_name = 'libdxl_x86_c';
21 elseif strcmp(computer, 'GLNXA64')
22   lib_name = 'libdxl_x64_c';
23 elseif strcmp(computer, 'MACI64')
24   lib_name = 'libdxl_mac_c';
25 end
26
27 % Load Libraries
28 if ~libisloaded(lib_name)
29     [notfound, warnings] = loadlibrary(lib_name, 'dynamixel_sdk.h',
          'addheader', 'port_handler.h', 'addheader', '
         packet_handler.h');
30 end
31
32 % Control table address
33 ADDR_AX_TORQUE_ENABLE       = 24;           % Control table address
       is different in Dynamixel model
34 ADDR_AX_GOAL_POSITION       = 30;
35 ADDR_AX_PRESENT_POSITION    = 36;
36 ADDR_AX_GOAL_VELOCITY       = 32;
37 ADDR_AX_PRESENT_VELOCITY    = 38;
38
39 % Protocol version
40 PROTOCOL_VERSION            = 1.0;          % See which protocol
       version is used in the Dynamixel
41
```

```
% Default setting
DXL_ID                          = 1;                % Dynamixel ID: 1
BAUDRATE                        = 57600;
DEVICENAME                      = 'COM4';           % Check which port is
    being used on your controller
                                                    % ex) Windows: 'COM1'
                                                          Linux: '/dev/
                                                        ttyUSB0' Mac: '/dev
                                                        /tty.usbserial-*'

TORQUE_ENABLE                   = 1;                % Value for enabling
    the torque
TORQUE_DISABLE                  = 0;                % Value for disabling
    the torque
DXL_MINIMUM_POSITION_VALUE      = 400;              % Dynamixel will rotate
     between this value
DXL_MAXIMUM_POSITION_VALUE      = 500;              % and this value (note
    that the Dynamixel would not move when the position value is
    out of movable range. Check e-manual about the range of the
    Dynamixel you use.)
DXL_MOVING_STATUS_THRESHOLD = 10;                   % Dynamixel moving
    status threshold
DXL_MINIMUM_VELOCITY_VALUE = 100;                   % Dynamixel will have a
     speed between this value
DXL_MAXIMUM_VELOCITY_VALUE = 800;                   % and this value

ESC_CHARACTER                   = 'e';              % Key for escaping loop

COMM_SUCCESS                    = 0;                % Communication Success
     result value
COMM_TX_FAIL                    = -1001;            % Communication Tx
    Failed

% Initialize PortHandler Structs
% Set the port path
% Get methods and members of PortHandlerLinux or PortHandlerWindows
port_num = portHandler(DEVICENAME);

% Initialize PacketHandler Structs
packetHandler();

index = 1;
dxl_comm_result = COMM_TX_FAIL;                     % Communication result
dxl_goal_position = [DXL_MINIMUM_POSITION_VALUE
    DXL_MAXIMUM_POSITION_VALUE];            % Goal position
dxl_goal_velocity = [DXL_MINIMUM_VELOCITY_VALUE
    DXL_MAXIMUM_VELOCITY_VALUE];

dxl_error = 0;                                      % Dynamixel error
dxl_present_velocity = 0;                           % Present position
dxl_present_velocity = 0;
```

```
77
78
79 % Open port
80 if (openPort(port_num))
81     fprintf('Succeeded to open the port!\n');
82 else
83     unloadlibrary(lib_name);
84     fprintf('Failed to open the port!\n');
85     input('Press any key to terminate...\n');
86     return;
87 end
88
89
90 % Set port baudrate
91 if (setBaudRate(port_num, BAUDRATE))
92     fprintf('Succeeded to change the baudrate!\n');
93 else
94     unloadlibrary(lib_name);
95     fprintf('Failed to change the baudrate!\n');
96     input('Press any key to terminate...\n');
97     return;
98 end
99
100
101 % Enable Dynamixel Torque
102 write1ByteTxRx(port_num, PROTOCOL_VERSION, DXL_ID,
        ADDR_AX_TORQUE_ENABLE, TORQUE_ENABLE);
103 dxl_comm_result = getLastTxRxResult(port_num, PROTOCOL_VERSION);
104 dxl_error = getLastRxPacketError(port_num, PROTOCOL_VERSION);
105 if dxl_comm_result ~= COMM_SUCCESS
106     fprintf('%s\n', getTxRxResult(PROTOCOL_VERSION, dxl_comm_result
            ));
107 elseif dxl_error ~= 0
108     fprintf('%s\n', getRxPacketError(PROTOCOL_VERSION, dxl_error));
109 else
110     fprintf('Dynamixel has been successfully connected \n');
111 end
112
113
114 while 1
115     if input('Press any key to continue! (or input e to quit!)\n',
            's') == ESC_CHARACTER
116         break;
117     end
118
119     % Write goal position
120     write2ByteTxRx(port_num, PROTOCOL_VERSION, DXL_ID,
            ADDR_AX_GOAL_POSITION, dxl_goal_position(index));
121     dxl_comm_result = getLastTxRxResult(port_num, PROTOCOL_VERSION)
            ;
122     dxl_error = getLastRxPacketError(port_num, PROTOCOL_VERSION);
```

```
123     if dxl_comm_result ~= COMM_SUCCESS
124         fprintf('%s\n', getTxRxResult(PROTOCOL_VERSION,
                dxl_comm_result));
125     elseif dxl_error ~= 0
126         fprintf('%s\n', getRxPacketError(PROTOCOL_VERSION,
                dxl_error));
127     end
128
129     % Write goal velocity
130     write2ByteTxRx(port_num, PROTOCOL_VERSION, DXL_ID,
            ADDR_AX_GOAL_VELOCITY, dxl_goal_velocity(index));
131     dxl_comm_result = getLastTxRxResult(port_num, PROTOCOL_VERSION)
            ;
132     dxl_error = getLastRxPacketError(port_num, PROTOCOL_VERSION);
133     if dxl_comm_result ~= COMM_SUCCESS
134         fprintf('%s\n', getTxRxResult(PROTOCOL_VERSION,
                dxl_comm_result));
135     elseif dxl_error ~= 0
136         fprintf('%s\n', getRxPacketError(PROTOCOL_VERSION,
                dxl_error));
137     end
138
139     while 1
140         % Read present position
141         dxl_present_position = read2ByteTxRx(port_num,
                PROTOCOL_VERSION, DXL_ID, ADDR_AX_PRESENT_POSITION);
142         dxl_comm_result = getLastTxRxResult(port_num,
                PROTOCOL_VERSION);
143         dxl_error = getLastRxPacketError(port_num, PROTOCOL_VERSION
                );
144         if dxl_comm_result ~= COMM_SUCCESS
145             fprintf('%s\n', getTxRxResult(PROTOCOL_VERSION,
                    dxl_comm_result));
146         elseif dxl_error ~= 0
147             fprintf('%s\n', getRxPacketError(PROTOCOL_VERSION,
                    dxl_error));
148         end
149
150         fprintf('[ID:%03d] GoalPos:%03d  PresPos:%03d\n', DXL_ID,
                dxl_goal_position(index), dxl_present_position);
151
152         % Read present velocity
153         dxl_present_velocity = read2ByteTxRx(port_num,
                PROTOCOL_VERSION, DXL_ID, ADDR_AX_PRESENT_VELOCITY);
154         dxl_comm_result = getLastTxRxResult(port_num,
                PROTOCOL_VERSION);
155         dxl_error = getLastRxPacketError(port_num, PROTOCOL_VERSION
                );
156         if dxl_comm_result ~= COMM_SUCCESS
157             fprintf('%s\n', getTxRxResult(PROTOCOL_VERSION,
                    dxl_comm_result));
```

```
158         elseif dxl_error ~= 0
159             fprintf('%s\n', getRxPacketError(PROTOCOL_VERSION,
                    dxl_error));
160         end
161
162         fprintf('[ID:%03d] GoalVel:%03d  PresVel:%03d\n', DXL_ID,
                dxl_goal_velocity(index), dxl_present_velocity);
163
164         if ~(abs(dxl_goal_position(index) - dxl_present_position) >
                DXL_MOVING_STATUS_THRESHOLD)
165             break;
166         end
167     end
168
169     % Change goal position and velocity
170     if index == 1
171         index = 2;
172     else
173         index = 1;
174     end
175 end
176
177
178 % Disable Dynamixel Torque
179 write1ByteTxRx(port_num, PROTOCOL_VERSION, DXL_ID,
        ADDR_AX_TORQUE_ENABLE, TORQUE_DISABLE);
180 dxl_comm_result = getLastTxRxResult(port_num, PROTOCOL_VERSION);
181 dxl_error = getLastRxPacketError(port_num, PROTOCOL_VERSION);
182 if dxl_comm_result ~= COMM_SUCCESS
183     fprintf('%s\n', getTxRxResult(PROTOCOL_VERSION, dxl_comm_result
        ));
184 elseif dxl_error ~= 0
185     fprintf('%s\n', getRxPacketError(PROTOCOL_VERSION, dxl_error));
186 end
187
188 % Close port
189 closePort(port_num);
190
191 % Unload Library
192 unloadlibrary(lib_name);
193
194 close all;
195 clear all;
```

B Appendix 2

In this appendix, all the main codes related to RL algorithms implementation are reported.

B.1 RGB markers detection

This code shows how to detect RGB markers in a scene and get their pixel coordinates.

```
%% Take the input image
data = imread('tags.png');
%%
tic;
% Convert the input image into gray-scale image
data_gray = rgb2gray(data);
% Now to track RGB objects in real time the RGB component should be
% subtracted from the grayscale image
diff_im_red = imsubtract(data(:,:,1), data_gray);
diff_im_green = imsubtract(data(:,:,2), data_gray);
diff_im_blue = imsubtract(data(:,:,3), data_gray);
% Sum the three obtained images together to get a final image with
    all the
% RGB components
diff_im = imadd(diff_im_red, diff_im_green);
diff_im = imadd(diff_im, diff_im_blue);

%Use a median filter to filter out noise
diff_im = medfilt2(diff_im, [3 3]);
% Convert the resulting grayscale image into a binary image.
diff_im = im2bw(diff_im,0.18);

% Perform a blob analysis, removing all those pixels less than 300
    [px]
diff_im = bwareaopen(diff_im,300);

% Label all the connected components in the image.
bw = bwlabel(diff_im, 8);

% Get a set of properties for each labeled region.
properties = regionprops(bw, 'BoundingBox', 'Centroid');
boundaries = zeros(length(properties), 4);
centroids = zeros(length(properties), 2);
% Display the image
imshow(data)

hold on

    %This is a loop to bound the RGB markers in a rectangular box.
    for object = 1:length(properties)
        bb = properties(object).BoundingBox;
        bc = properties(object).Centroid;
        boundaries(object, :) = bb;
```

```
42          centroids(object, :) = bc;
43          rectangle('Position',bb,'EdgeColor','r','LineWidth',2)
44          plot(bc(1),bc(2), '-m+')
45          a=text(bc(1)+15,bc(2), strcat('X: ', num2str(round(bc(1))),
                '    Y: ', num2str(round(bc(2)))));
46          set(a, 'FontName', 'Arial', 'FontWeight', 'bold', 'FontSize
                ', 12, 'Color', 'black');
47      end
48
49      hold off
50
51  toc;
```

B.2 Q-table initialization

Q-table is initialized as the reward function applying the following code:

```
1  % Initialized action-value function based on the reward function R
2
3  Q = repmat(R,[1, length(actions)]);
```

Listing B.1: Initialization of the Q-table based on the selected reward function

B.3 SARSA with discretized state-space

In this section, the main code of the implemented SARSA algorithm with discretized state-space is provided.

```
1  %% SARSA algorithm
2  tic;
3  for episode = 1:max_episode
4
5      disp(['episodes left: ', num2str(max_episode-episode)]);
6
7      if episode > 1
8          % Reset the robot at the beginning of each new episode
9          [dxl1_present_position, dxl2_present_position,
               dxl3_present_position] = move_robot_3DOF(
               dxl_comm_result, dxl_error, DXL_INITIAL_POSITION_VALUE1
               , DXL_INITIAL_POSITION_VALUE2,
               DXL_INITIAL_POSITION_VALUE3);
10         [centroids, boundaries] = take_image(cam, 1);
11     end
12
13     theta = [dxl1_present_position, dxl2_present_position,
           dxl3_present_position];
14
15     % Interpolate the state within our discretization
16     [~,state_Idx] = min(sum((states - repmat(theta,[size(states,1)
           ,1])).^2,2));
17
18     cumReward = 0;
19
```

```
20      %% PICK AN ACTION
21          % Choose an action:
22          % EITHER 1) pick the best action according the Q matrix (
                EXPLOITATION). OR
23          % 2) Pick a random action (EXPLORATION)
24          if (rand()>epsilon || episode == max_episode) && episode > 1
25              if all(Q(state_Idx,:) == Q(state_Idx, 1))
26                  action_Idx = action_selection(count_tot, count_max,
                        count_min, num_actions);
27              else
28                  [~,action_Idx] = max(Q(state_Idx,:)); % Pick the
                        action the Q matrix thinks is best!
29              end
30          elseif episode == 1
31                  action_Idx = randi(length(actions),1);
32          else
33               action_Idx = action_selection(count_tot, count_max,
                    count_min, num_actions);
34          end
35
36      for step = 1:max_steps
37
38          if action_Idx < 3
39              DXL1_NEW_POSITION_VALUE = actions(action_Idx)+
                    dxl1_present_position;
40              if DXL1_NEW_POSITION_VALUE < theta1_min ||
                    DXL1_NEW_POSITION_VALUE > theta1_max % if the
                    new_angle is out of robot angle-range, do not
                    update it
41                  DXL1_NEW_POSITION_VALUE = dxl1_present_position;
42              end
43              [dxl1_present_position] = move_robot_3DOF_ax(
                    dxl_comm_result, dxl_error, DXL1_NEW_POSITION_VALUE
                    , 1);
44
45          elseif action_Idx < 5
46              DXL2_NEW_POSITION_VALUE = actions(action_Idx)+
                    dxl2_present_position;
47              if DXL2_NEW_POSITION_VALUE < theta2_min ||
                    DXL2_NEW_POSITION_VALUE > theta2_max
48                  DXL2_NEW_POSITION_VALUE = dxl2_present_position;
49              end
50              [dxl2_present_position] = move_robot_3DOF_ax(
                    dxl_comm_result, dxl_error, DXL2_NEW_POSITION_VALUE
                    , 2);
51          else
52              DXL3_NEW_POSITION_VALUE = actions(action_Idx)+
                    dxl3_present_position;
53              if DXL3_NEW_POSITION_VALUE < theta3_min ||
                    DXL3_NEW_POSITION_VALUE > theta3_max
54                  DXL3_NEW_POSITION_VALUE = dxl3_present_position;
```

```
55              end
56              [dxl3_present_position] = move_robot_3DOF_ax(
                    dxl_comm_result, dxl_error, DXL3_NEW_POSITION_VALUE
                    , 3);
57          end
58
59          theta_new = [dxl1_present_position, dxl2_present_position,
                dxl3_present_position];
60
61          [centroids, boundaries] = take_image(cam, 1);
62
63         if length(centroids(:,1)) < 3
64              disp('Adjust light conditions and then press a key to
                    continue ');
65             pause;
66          else
67              [x1, y1, x2, y2, xEE, yEE, link_x, link_y] = kinematics
                    (centroids, boundaries, Origin);
68
69          end
70          xEE_saved(:, step) = xEE;
71          yEE_saved(:, step) = yEE;
72
73
74          [~,state_new_Idx] = min(sum((states - repmat(theta_new,[
                size(states,1),1])).^2,2)); % Interpolate again to find
                 the new state the system is closest to.
75
76          reward = reward_evaluation(theta);
77
78           %% PICK AN ACTION
79          % Choose an action:
80          % EITHER 1) pick the best action according the Q matrix (
                EXPLOITATION). OR
81          % 2) Pick a random action (EXPLORATION)
82          if (rand()>epsilon || episode == max_episode) && episode >
                1
83              if all(Q(state_Idx,:) == Q(state_Idx, 1))
84                  action_Idx_2 = action_selection(count_tot,
                        count_max, count_min, num_actions);
85              else
86                  [~,action_Idx_2] = max(Q(state_new_Idx,:)); % Pick
                        the action the Q matrix thinks is best!
87              end
88          elseif episode == 1
89                  action_Idx_2 = randi(length(actions),1);
90          else
91               action_Idx_2 = action_selection(count_tot, count_max,
                    count_min, num_actions);
92          end
93
```

```
94            collision_total = collision_detection (link_x, link_y,
                  boundaries_obs);
95            if ismember(1, collision_total)
96               bonus = -100;
97               count_hit = count_hit + 1;
98               disp('Obstacle hit!');
99               success = 0;
100              collision = 1;
101              count_reach_goal(:,episode) = 0;
102           elseif  reward > tradeoff_DEMO
103               bonus = 200;
104               count_success = count_success + 1;
105               count_step(:,episode) = 1;
106               required_actions(:,episode) = step;
107               sprintf('Goal reached in %d actions', step)
108               count_reach_goal(:,episode) = 1;
109               success = 1;
110               collision = 0;
111           else
112               bonus = 0;
113               success = 0;
114               collision = 0;
115               count_reach_goal(:,episode) = 0;
116           end
117
118           cumReward = cum_reward_evaluation(reward, cumReward, step);
119           cumReward_ep(:,episode) = cumReward;
120
121            % Check collision state
122           if collision == 1
123               [~, col] = find(collision_total == 1);
124               for index = 1:length(col)
125                  collision_loc = cat(1,collision_loc,[link_x(col(
                        index)) link_y(col(index))]);
126               end
127           end
128
129           % Update Q-table
130           Q(state_Idx,action_Idx) = Q(state_Idx,action_Idx) + alpha *
                 ( reward + gamma*Q(state_new_Idx,action_Idx_2) - Q(
                state_Idx,action_Idx) + bonus);
131           Q_saved(:,episode) = Q(state_Idx,action_Idx);
132           R_saved(:,episode) =  reward;
133           bonus_saved(:,episode) = bonus;
134
135           % Count the selected actions
136           [count_new, count_max, count_min] = action_probability(
                action_Idx, count_tot, num_actions);
137           count_tot = count_new;
138
139           state_Idx = state_new_Idx;
```

```
140             action_Idx = action_Idx_2;
141             theta = theta_new;
142 
143 
144             if success == 1 || collision == 1
145                 break
146             end
147         end
148 
149         epsilon = epsilon*epsilon_decay;
150 
151         if episode > 50
152             if count_reach_goal((episode-49):episode) == ones(1, 50)
153                 epsilon = 0;
154                 disp('JUST EXPLOITING!!');
155             elseif episode > 85*max_episode/100
156                 epsilon = 0;
157                 disp('JUST EXPLOITING!!');
158             end
159         end
160 
161     end
162     toc;
163     time = toc;
```

Listing B.2: Initialization of the Q-table based on the selected reward function

B.4 Q-learning with discretized state-space

In this section, the main code of the implemented Q-learning algorithm with discretized state-space is provided.

```
1  %% Q-learning algorithm
2  tic;
3  for episode = 1:max_episode
4  
5      disp(['episodes left: ', num2str(max_episode-episode)]);
6  
7      if episode > 1
8          % Reset the robot at the beginning of each new episode
9          [dxl1_present_position, dxl2_present_position,
               dxl3_present_position] = move_robot_3DOF(
               dxl_comm_result, dxl_error, DXL_INITIAL_POSITION_VALUE1
               , DXL_INITIAL_POSITION_VALUE2,
               DXL_INITIAL_POSITION_VALUE3);
10         [centroids, boundaries] = take_image(cam, 1);
11         [x1, y1, x2, y2, xEE, yEE, link_x, link_y] = kinematics(
               centroids, boundaries, Origin);
12     end
13 
14     theta = [dxl1_present_position, dxl2_present_position,
           dxl3_present_position];
15 
```

```
16      % Interpolate the state within our discretization
17      [~,state_Idx] = min(sum((states - repmat(theta,[size(states,1)
            ,1])).^2,2));
18
19      cumReward = 0;
20
21      for step = 1:max_steps
22
23          penalty = 0;
24
25           %% PICK AN ACTION
26          % Choose an action:
27          % EITHER 1) pick the best action according the Q matrix (
                EXPLOITATION). OR
28          % 2) Pick a random action (EXPLORATION)
29          if (rand()>epsilon || episode == max_episode) && episode >
                1
30              if all(Q(state_Idx,:) == Q(state_Idx, 1))
31                  action_Idx = action_selection(count_tot, count_max,
                        count_min, num_actions);
32              else
33                  [~,action_Idx] = max(Q(state_Idx,:)); % Pick the
                        action the Q matrix thinks is best!
34              end
35          elseif episode == 1
36                  action_Idx = randi(length(actions),1);
37          else
38               action_Idx = action_selection(count_tot, count_max,
                     count_min, num_actions);
39          end
40
41          if action_Idx < 3
42              DXL1_NEW_POSITION_VALUE = actions(action_Idx)+theta(1);
43              if DXL1_NEW_POSITION_VALUE < theta1_min ||
                    DXL1_NEW_POSITION_VALUE > theta1_max % if the
                    new_angle is out of robot angle-range, do not
                    update it
44                  DXL1_NEW_POSITION_VALUE = theta(1);
45              end
46              [dxl1_present_position] = move_robot_3DOF_ax(
                    dxl_comm_result, dxl_error, DXL1_NEW_POSITION_VALUE
                    , 1);
47
48          elseif action_Idx < 5
49              DXL2_NEW_POSITION_VALUE = actions(action_Idx)+theta(2);
50              if DXL2_NEW_POSITION_VALUE < theta2_min ||
                    DXL2_NEW_POSITION_VALUE > theta2_max
51                  DXL2_NEW_POSITION_VALUE = theta(2);
52              end
```

```
53              [dxl2_present_position] = move_robot_3DOF_ax(
                    dxl_comm_result, dxl_error, DXL2_NEW_POSITION_VALUE
                    , 2);
54          else
55              DXL3_NEW_POSITION_VALUE = actions(action_Idx)+theta(3);
56              if DXL3_NEW_POSITION_VALUE < theta3_min ||
                    DXL3_NEW_POSITION_VALUE > theta3_max
57                  DXL3_NEW_POSITION_VALUE = theta(3);
58              end
59              [dxl3_present_position] = move_robot_3DOF_ax(
                    dxl_comm_result, dxl_error, DXL3_NEW_POSITION_VALUE
                    , 3);
60          end
61
62          theta_new = [dxl1_present_position, dxl2_present_position,
                dxl3_present_position];
63
64          [centroids, boundaries] = take_image(cam, 1);
65          if length(centroids(:,1)) < 3
66             disp('Adjust light conditions and then press a key to
                   continue ');
67             pause;
68          else
69              [x1, y1, x2, y2, xEE, yEE, link_x, link_y] = kinematics
                    (centroids, boundaries, Origin);
70          end
71
72          xEE_saved(:, step) = xEE;
73          yEE_saved(:, step) = yEE;
74
75          %% UPDATE Q-MATRIX
76
77          [~,state_new_Idx] = min(sum((states - repmat(theta_new,[
                size(states,1),1])).^2,2)); % Interpolate again to find
                 the new state the system is closest to.
78
79          reward = reward_evaluation(theta);
80
81
82           collision_total = collision_detection (link_x, link_y,
                boundaries_obs);
83           if ismember(1, collision_total)
84              bonus = -100;
85              count_hit = count_hit + 1;
86              disp('Obstacle hit!');
87              success = 0;
88              collision = 1;
89              count_reach_goal(:,episode) = 0;
90           elseif  reward > tradeoff_DEMO
91               bonus = 200;
92               count_success = count_success + 1;
```

```
93              count_step(:,episode) = 1;
94              required_actions(:,episode) = step;
95              sprintf('Goal reached in %d actions', step)
96              count_reach_goal(:,episode) = 1;
97              success = 1;
98              collision = 0;
99          else
100             bonus = 0;
101             success = 0;
102             collision = 0;
103             count_reach_goal(:,episode) = 0;
104         end
105
106         cumReward = cum_reward_evaluation(reward, cumReward, step);
107         cumReward_ep(:,episode) = cumReward;
108
109         % Update Q-table
110         Q(state_Idx,action_Idx) = Q(state_Idx,action_Idx) + alpha *
                ( reward + gamma*max(Q(state_new_Idx,:)) - Q(state_Idx
                ,action_Idx) + bonus);
111         Q_saved(:,episode) = Q(state_Idx,action_Idx);
112         R_saved(:,episode) =  reward;
113         bonus_saved(:,episode) = bonus;
114
115         % Count the selected action
116         [count_new, count_max, count_min] = action_probability(
                action_Idx, count_tot, num_actions);
117         count_tot = count_new;
118
119         % Check collision state
120         if collision == 1
121             [~, col] = find(collision_total == 1);
122             for index = 1:length(col)
123                 collision_loc = cat(1,collision_loc,[link_x(col(
                        index)) link_y(col(index))]);
124             end
125         end
126
127         theta = theta_new;
128         state_Idx = state_new_Idx;
129
130         if success == 1 || collision == 1
131             break
132         end
133     end
134
135     epsilon = epsilon*epsilon_decay;
136     cumReward_ep(:,episode) = -cumReward;
137
138     if episode > 50
139         if count_reach_goal((episode-49):episode) == ones(1, 50)
```

```
            epsilon = 0;
            disp('JUST EXPLOITING!!');
        elseif episode > 85*max_episode/100
            epsilon = 0;
            disp('JUST EXPLOITING!!');
        end
    end

end
toc;
time = toc;
```

Listing B.3: Initialization of the Q-table based on the selected reward function

B.5 Deep RL - store new experience in the replay memory

This code shows how to store the new experienced tuple in the Replay Memory when Deep Q-learning is applied. The same code is adopted also to store the experience in SARSA, but the only difference is that also the new action selected from the new state is stored.

```
function store_transition_Qlearn(obj,s,a,r,s_new)
        % Add transition to the Replay Memory
        obj.S = cat(1,obj.S,s);
        obj.A = cat(1,obj.A,a);
        obj.R = cat(1,obj.R,r);
        obj.S_new = cat(1,obj.S_new,s_new);

        % Update num of elements in the memory
        obj.elemsInBuffer = obj.elemsInBuffer+1;

        % Check if the memory is full, and if so - delete from the
            beginning (FIFO)
        if (obj.elemsInBuffer >obj.memory_size)
            obj.S(1,:)=[];
            obj.A(1,:)=[];
            obj.R(1,:)=[];
            obj.S_new(1,:)=[];

            % Update num of elements in the memory
            obj.elemsInBuffer = obj.elemsInBuffer-1;
        end

end
```

B.6 SARSA with continuous state-space - Deep SARSA

In this section, the main code of the implemented SARSA algorithm with continuous state-space is provided.

```
%% SARSA algorithm
tic;
for episode = 1:max_episode

```

```
5       disp(['episodes left: ', num2str(max_episode-episode)]);
6
7       if episode > 1
8           % Reset the robot at the beginning of each new episode
9           [dxl1_present_position, dxl2_present_position,
                dxl3_present_position] = move_robot_3DOF(
                dxl_comm_result, dxl_error, DXL_INITIAL_POSITION_VALUE1
                , DXL_INITIAL_POSITION_VALUE2,
                DXL_INITIAL_POSITION_VALUE3);
10          [centroids, boundaries] = take_image(cam, 1);
11      end
12
13      theta = [dxl1_present_position, dxl2_present_position,
            dxl3_present_position];
14
15      %% PICK AN ACTION
16          % Choose an action:
17          % EITHER 1) pick the best action according the Q matrix (
                EXPLOITATION). OR
18          % 2) Pick a random action (EXPLORATION)
19          if (rand()>epsilon || episode == max_episode) && episode >
                1
20              [~,action_Idx] = max(Q(theta')); % Pick the action the
                    Q matrix thinks is best!
21          elseif episode == 1
22                  action_Idx = randi(length(actions),1);
23          else
24               action_Idx = action_selection(count_tot, count_max,
                     count_min, num_actions);
25          end
26          cumReward = 0;
27
28      for step = 1:max_steps
29
30          if action_Idx < 3
31              DXL1_NEW_POSITION_VALUE = actions(action_Idx)+
                    dxl1_present_position;
32              if DXL1_NEW_POSITION_VALUE < theta1_min ||
                    DXL1_NEW_POSITION_VALUE > theta1_max % if the
                    new_angle is out of robot angle-range, do not
                    update it
33                  DXL1_NEW_POSITION_VALUE = dxl1_present_position;
34              end
35              [dxl1_present_position] = move_robot_3DOF_ax(
                    dxl_comm_result, dxl_error, DXL1_NEW_POSITION_VALUE
                    , 1);
36
37          elseif action_Idx < 5
38              DXL2_NEW_POSITION_VALUE = actions(action_Idx)+
                    dxl2_present_position;
```

```
39              if DXL2_NEW_POSITION_VALUE < theta2_min ||
                    DXL2_NEW_POSITION_VALUE > theta2_max
40                  DXL2_NEW_POSITION_VALUE = dxl2_present_position;
41              end
42              [dxl2_present_position] = move_robot_3DOF_ax(
                    dxl_comm_result, dxl_error, DXL2_NEW_POSITION_VALUE
                    , 2);
43          else
44              DXL3_NEW_POSITION_VALUE = actions(action_Idx)+
                    dxl3_present_position;
45              if DXL3_NEW_POSITION_VALUE < theta3_min ||
                    DXL3_NEW_POSITION_VALUE > theta3_max
46                  DXL3_NEW_POSITION_VALUE = dxl3_present_position;
47              end
48              [dxl3_present_position] = move_robot_3DOF_ax(
                    dxl_comm_result, dxl_error, DXL3_NEW_POSITION_VALUE
                    , 3);
49          end
50
51          theta_new = [dxl1_present_position, dxl2_present_position,
                dxl3_present_position];
52
53          [centroids, boundaries] = take_image(cam, 1);
54
55          if length(centroids(:,1)) < 3
56              disp('Adjust light conditions and then press a key to
                    continue ');
57             pause;
58          else
59              [x1, y1, x2, y2, xEE, yEE, link_x, link_y] = kinematics
                    (centroids, boundaries, Origin);
60          end
61
62          xEE_saved(:, step) = xEE;
63          yEE_saved(:, step) = yEE;
64
65          reward = reward_evaluation(theta);
66
67          %% PICK AN ACTION
68          % Choose an action:
69          % EITHER 1) pick the best action according the Q matrix (
                EXPLOITATION). OR
70          % 2) Pick a random action (EXPLORATION)
71          if (rand()>epsilon || episode == max_episode) && episode >
                1
72              [~,action_Idx_2] = max(Q(theta_new')); % Pick the
                    action the Q matrix thinks is best!
73          elseif episode == 1
74                  action_Idx_2 = randi(length(actions),1);
75          else
```

```
76              action_Idx_2 = action_selection(count_tot, count_max,
                    count_min, num_actions);
77          end
78
79           collision_total = collision_detection (link_x, link_y,
                boundaries_obs);
80           if ismember(1, collision_total)
81              bonus = -100;
82              count_hit = count_hit + 1;
83              disp('Obstacle hit!');
84              success = 0;
85              collision = 1;
86              count_reach_goal(:,episode) = 0;
87           elseif  reward > tradeoff_DEMO
88               bonus = 200;
89               count_success = count_success + 1;
90               count_step(:,episode) = 1;
91               required_actions(:,episode) = step;
92               sprintf('Goal reached in %d actions', step)
93               count_reach_goal(:,episode) = 1;
94               success = 1;
95               collision = 0;
96           else
97               bonus = 0;
98               success = 0;
99               collision = 0;
100              count_reach_goal(:,episode) = 0;
101          end
102
103         % Count the selected actions
104         [count_new, count_max, count_min] = action_probability(
                action_Idx, count_tot, num_actions);
105         count_tot = count_new;
106
107     %% TRAIN Q-NETWORK
108
109         reward_tot = reward + bonus;
110         R_saved(:,episode) = reward;
111
112         cumReward = cum_reward_evaluation(reward, cumReward, step);
113         cumReward_ep(:,episode) = cumReward;
114
115
116         %% Store (S,A,R,S') in the ReplayMemory
117
118          memory_buffer.store_transition(theta, action_Idx,
                reward_tot, theta_new, action_Idx_2);
119
120         theta = theta_new;
121         action_Idx = action_Idx_2;
122
```

```
123         %%
124
125         for mini_batch = 1:batches
126
127             %% Sample batch for training transitions (s,a,r,s')
                    from memory_size
128             [temp_s,temp_a,temp_r,temp_s_new,temp_a_new] =
                    memory_buffer.sample_mini_batch(batch_size);
129
130             %% Gradient descent Q
131              Y = Q(temp_s');     %s is in rows and the NN accepts s
                     as a column vector
132              Q_next = Q_target(temp_s_new');
133
134             for i=1:batch_size
135
136                 Y(temp_a(i),i) = temp_r(i) + gamma*Q_next(
                        temp_a_new(i),i);
137
138             end
139
140             % train on estimated Q_next and rewards
141             Q = train(Q,temp_s',Y);
142         end
143
144         if success == 1 || collision == 1
145             break
146         end
147
148     end
149
150     if ( mod(episode,C)==0)
151        Q_target=Q;
152        disp ('updated target network');
153     end
154
155     epsilon = epsilon*epsilon_decay;
156
157     if episode > 50
158        if count_reach_goal((episode-49):episode) == ones(1, 50)
159            epsilon = 0;
160            disp('JUST EXPLOITING!!');
161        elseif episode > 85*max_episode/100
162            epsilon = 0;
163            disp('JUST EXPLOITING!!');
164        end
165     end
166
167 end
168 toc;
169 time = toc;
```

Listing B.4: Initialization of the Q-table based on the selected reward function

B.7 Q-learning with continuous state-space - Deep Q-learning

In this section, the main code of the implemented Q-learning algorithm with continuous state-space is provided.

```
%% Q-learning algorithm
tic;
for episode = 1:max_episode

    disp(['episodes left: ', num2str(max_episode-episode)]);

    if episode > 1
        % Reset the robot at the beginning of each new episode
        [dxl1_present_position, dxl2_present_position,
            dxl3_present_position] = move_robot_3DOF(
            dxl_comm_result, dxl_error, DXL_INITIAL_POSITION_VALUE1
            , DXL_INITIAL_POSITION_VALUE2,
            DXL_INITIAL_POSITION_VALUE3);
        [centroids, boundaries] = take_image(cam, 1);
    end

    theta = [dxl1_present_position, dxl2_present_position,
        dxl3_present_position];

    cumReward = 0;

    for step = 1:max_steps

        %% PICK AN ACTION
        % Choose an action:
        % EITHER 1) pick the best action according the Q matrix (
            EXPLOITATION). OR
        % 2) Pick a random action (EXPLORATION)
        if (rand()>epsilon || episode == max_episode) && episode >
            1
            [~,action_Idx] = max(Q(theta')); % Pick the action the
                Q matrix thinks is best!
        elseif episode == 1
                action_Idx = randi(length(actions),1);
        else
             action_Idx = action_selection(count_tot, count_max,
                count_min, num_actions);
        end


        if action_Idx < 3
            DXL1_NEW_POSITION_VALUE = actions(action_Idx)+
                dxl1_present_position;
            if DXL1_NEW_POSITION_VALUE < theta1_min ||
                DXL1_NEW_POSITION_VALUE > theta1_max % if the
```

```
            new_angle is out of robot angle-range, do not
            update it
35              DXL1_NEW_POSITION_VALUE = dxl1_present_position;
36          end
37          [dxl1_present_position] = move_robot_3DOF_ax(
                dxl_comm_result, dxl_error, DXL1_NEW_POSITION_VALUE
                , 1);
38
39      elseif action_Idx < 5
40          DXL2_NEW_POSITION_VALUE = actions(action_Idx)+
                dxl2_present_position;
41          if DXL2_NEW_POSITION_VALUE < theta2_min ||
                DXL2_NEW_POSITION_VALUE > theta2_max
42              DXL2_NEW_POSITION_VALUE = dxl2_present_position;
43          end
44          [dxl2_present_position] = move_robot_3DOF_ax(
                dxl_comm_result, dxl_error, DXL2_NEW_POSITION_VALUE
                , 2);
45      else
46          DXL3_NEW_POSITION_VALUE = actions(action_Idx)+
                dxl3_present_position;
47          if DXL3_NEW_POSITION_VALUE < theta3_min ||
                DXL3_NEW_POSITION_VALUE > theta3_max
48              DXL3_NEW_POSITION_VALUE = dxl3_present_position;
49          end
50          [dxl3_present_position] = move_robot_3DOF_ax(
                dxl_comm_result, dxl_error, DXL3_NEW_POSITION_VALUE
                , 3);
51      end
52
53      theta_new = [dxl1_present_position, dxl2_present_position,
            dxl3_present_position];
54
55      [centroids, boundaries] = take_image(cam, 1);
56
57      if length(centroids(:,1)) < 3
58          disp('Adjust light conditions and then press a key to
                continue ');
59          pause;
60      else
61          [x1, y1, x2, y2, xEE, yEE, link_x, link_y] = kinematics
                (centroids, boundaries, Origin);
62
63      end
64
65      xEE_saved(:, step) = xEE;
66      yEE_saved(:, step) = yEE;
67
68      reward = reward_evaluation(theta);
69
```

```
70            collision_total = collision_detection (link_x, link_y,
                  boundaries_obs);
71            if ismember(1, collision_total)
72               bonus = -100;
73               count_hit = count_hit + 1;
74               disp('Obstacle hit!');
75               success = 0;
76               collision = 1;
77               count_reach_goal(:,episode) = 0;
78            elseif  reward > tradeoff_DEMO
79                bonus = 200;
80                count_success = count_success + 1;
81                count_step(:,episode) = 1;
82                required_actions(:,episode) = step;
83                sprintf('Goal reached in %d actions', step)
84                count_reach_goal(:,episode) = 1;
85                success = 1;
86                collision = 0;
87            else
88                bonus = 0;
89                success = 0;
90                collision = 0;
91                count_reach_goal(:,episode) = 0;
92            end
93
94           cumReward = cum_reward_evaluation(reward, cumReward, step);
95           cumReward_ep(:,episode) = cumReward;
96
97           if collision == 1
98               [~, col] = find(collision_total == 1);
99               for index = 1:length(col)
100                 collision_loc = cat(1,collision_loc,[link_x(col(
                        index)) link_y(col(index))]);
101              end
102          end
103
104          % Count the selected actions
105          [count_new, count_max, count_min] = action_probability(
                 action_Idx, count_tot, num_actions);
106          count_tot = count_new;
107
108   %% TRAIN Q-NETWORK
109
110          reward_tot = reward + bonus;
111          R_saved(:,episode) = reward;
112
113          %% Store (S,A,R,S') in the ReplayMemory
114
115          memory_buffer.store_transition(theta, action_Idx,
                 reward_tot, theta_new);
116
```

```
117         theta = theta_new;
118
119         %%
120         for mini_batch = 1:batches
121
122             %% Sample batch for training transitions (s,a,r,s')
                    from memory_size
123             [temp_s,temp_a,temp_r,temp_s_new] = memory_buffer.
                    sample_mini_batch(batch_size);
124
125             %% Gradient descent Q
126              Y = Q(temp_s');     %s is in rows and the nn accepts s
                     as a column vector
127              Q_next = Q_target(temp_s_new');
128              Q_next_max = max(Q_next);
129
130             for i=1:batch_size
131
132                 Y(temp_a(i),i) = temp_r(i) + gamma*Q_next_max(i);
133
134             end
135
136             % train on estimated Q_next and rewards
137             Q = train(Q,temp_s',Y);
138         end
139
140
141
142 %% If number of loop singles=c: Q_target=Q
143
144         if success == 1 || collision == 1
145             break
146         end
147
148     end
149
150     epsilon = epsilon*epsilon_decay;
151     cumReward_ep(:,episode) = -cumReward;
152
153    if ( mod(episode,C)==0)
154        Q_target=Q;
155        disp ('updated target network');
156    end
157
158     if episode > 50
159         if count_reach_goal((episode-49):episode) == ones(1, 50)
160             epsilon = 0;
161             disp('JUST EXPLOITING!!');
162         elseif episode > 85*max_episode/100
163             epsilon = 0;
164             disp('JUST EXPLOITING!!');
```

```
165             end
166         end
167
168 end
169 toc;
170 time = toc;
```

Listing B.5: Initialization of the Q-table based on the selected reward function

C Acronyms

DP	Dynamic Programming
GPI	Generalized Policy Iteration
GUI	Graphical User Interface
HN	Hidden Neurons
MC	Monte-Carlo
MDP	Markov Desicion Process
NN	Neural Network
RaM	Robotics and Mechatronics
RL	Reinforcement Learning
TD	Temporal Difference

Bibliography

[1] B. Wang, H. Luo, Y. Jin and M. He, *Path planning for detection robot climbing on robot blade surfaces of wind turbine based on neural networks*. Advances in Mechanical Engineering, p. 1-10, 2013.

[2] F.B.W. Berndsen, *Development of a robotic arm suitable for practicing control algorithms taught in the course Modern Robotics*. Bachelor's thesis, University of Twente, 2016.

[3] E.L. Thorndike, *Animal intelligence: an experimental study of the associative processes in animals*.

[4] R.S. Sutton and A.G. Barto, *Reinforcement Learning: An Introduction*. A Bradford Book, The MIT Press Cambridge, Massachusetts London, England, 2018.

[5] D.E. Moriarty, A.C. Schultz and J.J. Grefenstette, *Evolutionary algorithms for reinforcement learning*. Journal of Artificial Intelligence Research, p. 241-276, 1999.

[6] J.C.H.C. Watkins and P. Dayan, *Technical Note: Q-learning*. Machine Learning, Kluwer Academic Publishers, Boston, 1992.

[7] S.D.R.D.C.W.A. Darrel Whitley, *Genetic Reinforcement Learning for Neurocontrol Problems*. Springer, vol. 3, p. 259-284, 1993.

[8] Y. Li, *Deep Reinforcement Learning: An Overview*. 2017.

[9] J. Kober, J.A. Bagnell, and J. Peters, *Reinforcement Learning in Robotics: A Survey*. The International Journal of Robotics Research. Vol. 32, pp. 1238-1274, 2013.

[10] J. Kober, and J. Peters, *Reinforcement Learning in Robotics: A Survey*. Wiering M., van Otterlo M. (eds) Reinforcement Learning. Adaptation, Learning, and Optimization, vol 12. Springer, 2012.

[11] C.V. Hu, *Cliff Walking: A Case Study to Compare Sarsa and Q-Learning*. https://github.com/cvhu/CliffWalking.

[12] C.K. Tham and R.W. Prager, *A Modular Q-Learning Architecture for Manipulator Task Decomposition*. Proceedings of the Eleventh International Conference on Machine Learning, 1994.

[13] M. Duguleana, F.G. Barbuceanu, A. Teirelbar and G. Moghan, *Obstacle avoidance of redundant manipulators using neural networks based reinforcement learning*. Robotics and Computer-Integrated Manufacturing, vol. 28, pp. 132-146, 2012.

[14] S. James, *3D Simulated Robot Manipulation Using Deep Reinforcement Learning*. Individual Project MEng, Imperial College London, 2016.

[15] L. Busoniu, B. De Schutter, and R. Babuška, *Decentralized Reinforcement Learning Control of a Robotic Manipulator*. 9th International Conference on Control, Automation, Robotics and Vision, 2006.

[16] W. Kim, T. Kim, H.J. Kim and S. Kim, *Three-link Planar Arm Control Using Reinforcement Learning*. 14th International Conference on Ubiquitous Robots and Ambient Intelligence, 2017.

[17] B. Nemec, M. Zorko and L. Zlajpha, *Learning of a ball-in-a-cup playing robot*. Proc. of 19th International Workshop on Roboticsin Alpe-Adria-Danube Region RAAD, Budapest, Hungary, 2010.

[18] *Markov Decision Process*. Wikipedia: The Free Encyclopedia; available from https://en.wikipedia.org/wiki/Markov_decision_process, retrieved: May, 2018.

[19] T. Matiisen, *Guest Post (Part I): Demystifying Deep Reinforcement Learning*. available in https://ai.intel.com/demystifying-deep-reinforcement-learning/

[20] *Robotis E-manual AX12A* http://support.robotis.com/en/product/actuator/dynamixel/ax_series/dxl_ax_actuator.htm

[21] I.P. Vieira, A.A. Neto and L.A. Mozelli, *Fiducial markers applied for pose tracking of a robotic manipulator: application in visual servoing control*. XIII Latin American Robotics Symposium and IV Brazilian Robotics Symposium, 2016.

[22] J.Sturm, C. Plagemann and W. Burgard, *Body schema learning for robotic manipulators from visual self-perception*. Journal of Physiology, pp. 220-231, 2009.

[23] K. Mattenberger, *Controlling Nao's arms with color markers*. Bachelor Project Report, Universität Freiburg, 2011.

[24] Y.L. Kuo, B.H. Liu and C.Y. Wu, *Pose Determination of a Robot Manipulator Based on Monocular Vision*. IEEE Access, 2016.

[25] T.B. Moeslund, *Introduction to video and image processing: building real systems and applications*. Part of the Undergraduate Topics in Computer Science book series (UTICS), Springer, 2012.

[26] E. Olson, *AprilTag: A robust and flexible visual fiducial system*. IEEE International Conference on Robotics and Automation, 2011.

[27] V. Ortenzi, N. Marturi, R. Stolkin, J.A. Kuo and M. Mistry, *Vision-guided state estimation and control of robotic manipulators which lack proprioceptive sensors*. International Conference on Intelligent Robots and Systems (IROS), Daejeon, Korea, 2016.

[28] J.R. Sanchez-Lopez, A. Marin-Hernandez and E.R. Palacios-Hernandez, *Visual Detection, Tracking and Pose Estimation of a Robotic Arm End Effector*. 2018.

[29] Robotis, *USB2DYNAMIXEL e-Manual*, in http://support.robotis.com/en/product/auxdevice/interface/usb2dxl_manual.htm.

[30] MATLAB website, https://nl.mathworks.com.

[31] DYNAMIXEL SDK library installation procedure, http://emanual.robotis.com/docs/en/software/dynamixel/dynamixel_sdk/overview/#license

[32] P. Corke, *Machine Vision Toolbox*, http://petercorke.com/wordpress/toolboxes/machine-vision-toolbox, 2011.

[33] MathWorks, *Measuring Planar Objects with a Calibrated Camera*. https://nl.mathworks.com/help/vision/examples/measuring-planar-objects-with-a-calibrated-camera.html, 2018.

[34] J. Heaton, *Introduction to Neural Networks with Java*. Heaton Research, Second edition, 2008.

[35] Mathworks, *traingdm: Gradient descent with momentum backpropagation*. Introduced in R2006a; available from https://it.mathworks.com/help/nnet/ref/traingdm.html, 2006.

[36] M.A. Omar, A Che Soh, M.K. Hassan and Z. Kadir, *Lightning severity classification utilizing the meteorological parameters: A neural network approach*. IEEE International Conference on Control System, Computing and Engineering (ICCSCE), 2013.