UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Tècnica Superior d'Enginyeria
de Telecomunicació de Barcelona

# DEEP REINFORCEMENT LEARNING BASED APPROACHES FOR CAPACITY SHARING IN RADIO ACCESS NETWORK SLICING

**A Master's Thesis**

**Submitted to the Faculty of the**

**Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona**

**Universitat Politècnica de Catalunya**

**by**

**Víctor Garcia Cruz**

**In partial fulfilment**

**of the requirements for the degree of**

**MASTER IN TELECOMMUNICATIONS ENGINEERING**

**Advisor: Oriol Sallent Roig**

**Barcelona, July 2020**


**Title of the thesis:** Deep Reinforcement Learning based Approaches for Capacity Sharing in Radio Access Network Slicing

**Author:** Víctor Garcia Cruz

**Advisor:** Oriol Sallent Roig

## Abstract

Network slicing has become a fundamental capability for 5G networks to support the expected high variety of service requirements over a common physical network infrastructure. Each network slice can be customized for a specific application, making that the radio resources have to be accordingly managed by the Radio Access Network (RAN) part of the slice. In this thesis, three different Deep Reinforcement Learning (DRL) based approaches are presented to optimize the resource allocation among slices. A RAN slicing simulator scenario is developed, where the DRL mechanisms build knowledge about the network and learn how to optimize the capacity allocation for each tenant at every moment of time. The performance of each approach is studied based on simulation results, and before the comparison between the algorithms, the set of hyperparameters of each approach is tuned to optimize the learning process.

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Tècnica Superior d'Enginyeria
de Telecomunicació de Barcelona

telecos
BCN

## Acknowledgements

First of all, I would like to express my gratitude to Oriol Sallent, the tutor of the thesis, for giving me the opportunity to develop this project and for all the guidance during the project. In addition, I would like to thank Irene Vilà, who has been involved during all the thesis giving me technical support and helping me to achieve a correct development of the thesis.

I also give thanks to Sergio Garcia for the informatics support allowing me to complete the simulations much faster.

Finally, I thank my family and friends for their unconditional support, without them the way would have been much more difficult.

## Revision history and approval record

| Revision | Date | Purpose |
|---|---|---|
| 0 | 11/05/2020 | Document creation |
| 1 | 06/07/2020 | Document revision |
| 2 | 20/07/2020 | Document revision |
| | | |

| Written by: | | Reviewed and approved by: | |
|---|---|---|---|
| Date | 20/07/2020 | Date | 21/07/2020 |
| Name | Víctor Garcia Cruz | Name | Oriol Sallent Roig |
| Position | Project Author | Position | Project Supervisor |

# Table of contents

# List of Figures

# List of Tables

# 1. Introduction

The first commercial mobile communication network was launched in the 80s allowing only analogue voice communication. Since then, mobile communications have evolved a lot achieving unimagined things, but the arrival of 5G is one of the most important technological developments of our time, connecting everyone and everything.

5G networks are intended to simultaneously support a wide range of applications with a high variety of requirements; an expected versatility that cannot be met through a common network setting. In this respect, network slicing offers a flexible and scalable solution for accommodating this diversity in a single physical network, becoming a fundamental capability for 5G. However, the realization of radio access network (RAN) slices is particularly challenging because it requires to handle the radio resource management (RRM) so that the resource allocation is optimized accordingly.

The conventional approaches for service and resource management require complete and perfect knowledge of the system, becoming inefficient. In this context, deep reinforcement learning (DRL) based mechanisms are used to acquire knowledge about the network and learn how to make these optimal decisions locally and independently. DRL combines the contributions of both deep learning and reinforcement learning (RL), being able to handle large-scale and dynamic systems as 5G networks will be.

In this thesis, different DRL off-policy mechanisms are applied in a RAN slicing deployment scenario to optimize the capacity sharing algorithm among slices. The simulator consists of different python classes, each of them representing the different elements of the environment. The DRL based approaches are implemented with the *TensorFlow* library from Python, which is an open-source library to help the development and training of the ML models. Before analysing and comparing the results from the algorithms, the model hyperparameters have been studied carefully to set them with the most appropriate values to improve the learning process.

## 1.1. Objectives

The main objectives of the project to be achieved are:

- To get familiar with *TensorFlow* library and design a code that can be easily upgrade in the future.
- To develop an algorithm capable of manage capacity sharing among slices through DRL methods.
- To tune the model hyperparameters to optimize the learning process.
- To be able to obtain measurable results from the different algorithms and study the performance of each of them.
- Make a comparative analysis between a method based on value function and an actor-critic approach.

## 1.2. Thesis Outline

This project is divided in the following chapters:

- **Chapter 1** presents a short introduction about the problem statement, including also the objectives of the project and the work plan followed.
- **Chapter 2** gives a theoretical background of the Thesis. It starts introducing the concepts of Network Slicing and RAN Slicing for 5G. Then, the fundamental principles of RL are explained to end up presenting some DRL-based approaches.
- **Chapter 3** presents the agent-environment model that has been defined for the problem and how the simulator works. The algorithm of each DRL approach is detailed in this chapter.
- **Chapter 4** is composed of the different studies that have been carried out in the project.
- **Chapter 5** explains the cost assessment of the project.
- **Chapter 6** ends with the conclusions and future steps for the project.

## 1.3. Work Plan and deviations

The project is divided in different phases. The first phase of the project includes the revision of the state of the art of both the 5G and DRL approaches, introducing the main concepts needed for the development of the project. After that, the scenario is created and the common classes of the simulator are implemented. With this created, the different studies are carried out, one for each approach (DQN, DDQN and DDPG). Finally, all the information collected is analysed to compare the performance of each approach and be able to write the final document.

The following image presents the Gantt diagram of the project.

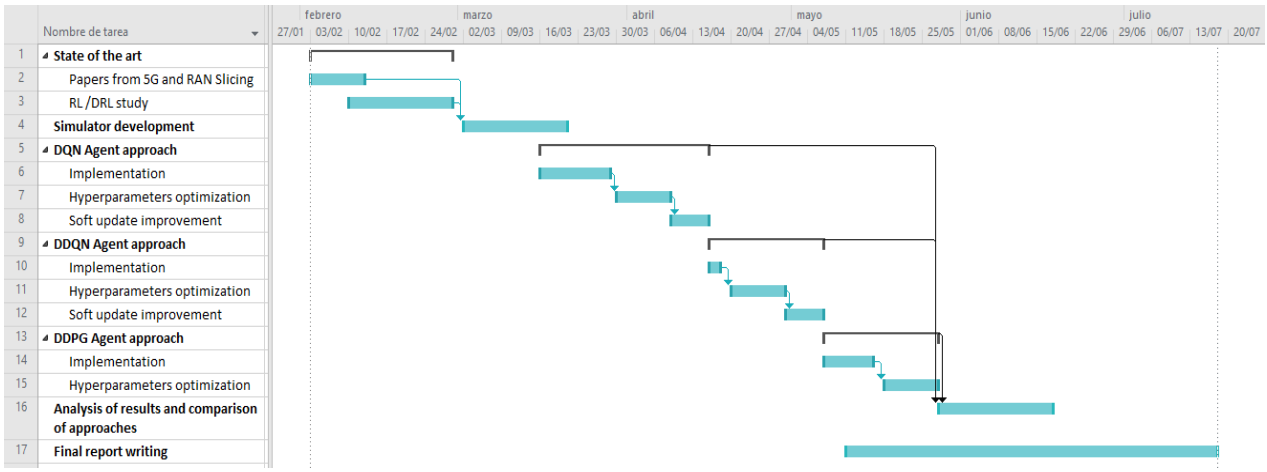


Figure 1. Gantt diagram

# 2. State of the art

This theoretical chapter presents the context in which the project is carried out. First, an introduction to 5G is done, explaining the new concept of network slicing. Then, the realization of Radio Access Network (RAN) slices is explained, focusing on the radio protocol layers. After that, we overview the fundamental concepts of reinforcement learning (RL), and how it can be combined with deep learning techniques to obtain deep reinforcement learning (DRL). Finally, some DRL-based approaches are explained, which are the basis of the algorithms implemented in the thesis.

## 2.1. Introduction to 5G

The tremendous increase of wireless data services, driven by the popularity of smart mobile devices and communication technologies, provokes the necessity of the fifth generation mobile communication networks, 5G.

5G is developed by the Third Generation Partnership Project (3GPP) entity, and its commercial deployment has already started. It will connect billions of elements that have not been connected before, simultaneously supporting a wide range of application scenarios and business models (e.g. automotive, smart cities, high-tech manufacturing). These new applications are mainly divided into three major use cases [1] (Figure 2):

- **Enhanced Mobile Broadband (eMBB):** this family represents the growing scenarios of a fully connected society, requiring a high data rate.
- **Ultra-reliable and Low Latency Communications (URLLC):** this scenario has strong requirements for capabilities, such as latency or throughput, to allow real-time interaction, as could be a remote medical surgery.
- **Massive Machine Type Communications (mMTC):** this family covers a very large number of connected devices typically transmitting relative low volume of data.

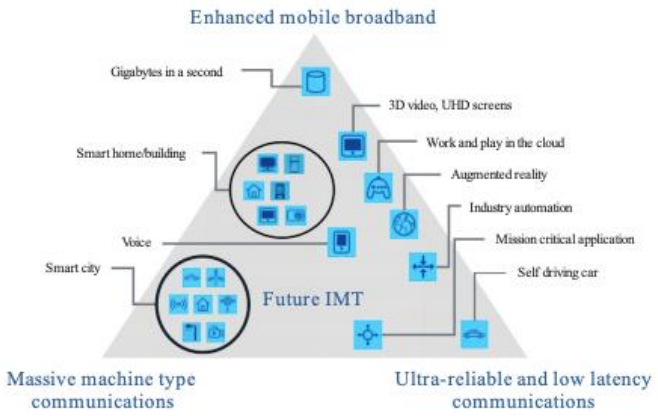


**Figure 2. 5G use cases [1]**

5G will be able to support both the evolution of these use cases and the emergence of new ones.

This extensive range of applications will result in a wide variety of requirements (i.e. high data rates, high mobility, latencies below 1ms), which cannot be met through a common

network setting; the current *one-size-fits-all* network architecture requires more flexibility and scalability. For these reasons, network slicing is considered one of the pillars of 5G systems, allowing the cost-effective operation of multiple logical networks in a common physical infrastructure [2].

## 2.2. Network Slicing

Network slicing refers to the partitioning of a physical network into several virtual networks, each customized and optimized for a specific type of application. A 5G network slice is composed of a collection of network functions and specific radio access technology settings that are combined for a specific use case [3]. In addition to providing a particular system behaviour, network slicing should be able to provide a particular tenant with a given level of guaranteed network resources and isolation with regard to other concurrent slices [5].

The requirements to fulfil will be determined between the network operator and tenants in the form of Service Level Agreements (SLA) in which key performance indicators (KPIs) such as throughput, latency or connectivity will be specified [6]. In order to meet with these SLAs, network slicing encompasses both 5G Core Network (5GC), referred to as CN slice, and New Generation radio Access Network (NG-RAN), referred to as Radio Access Network (RAN) slice. Both parts need to be sliced into several instances serving different types of users, devices and use cases.

A network slice is uniquely identified by a *Slice ID* within a 5G network, which is identified by a Public Land Mobile Network (PLMN) identity. The CN slices are responsible of deploying the Control Plane (CP) and User Plane (UP) functions, including network functions (NFs). These NFs could be common for several CN slices or only serve a specific one, and are implemented as virtualized network functions (VNFs).

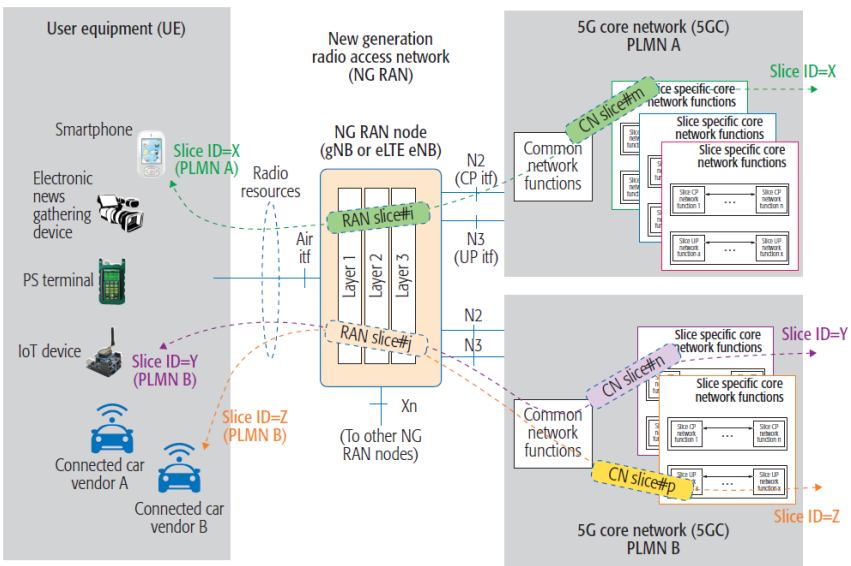


**Figure 3. Network slicing architecture [4]**

Regarding the NG-RAN, it consists of new generation nodes (gNBs) and/or evolved nodes of Long Term Evolution (eLTE eNBs), which are single NFs that provide the UP/CP

protocols. As illustrated in Figure 3, there could be a diverse range of User Equipment (UEs) connected to the same NG-RAN node through different slices [4].

However, the realization of RAN slices is particularly challenging because it requires addressing how the pool of radio resources available to one NG-RAN node can be configured to simultaneously deliver multiple and diverse RAN behaviours. In the next chapter are explained the fundamental design challenges of RAN slicing, focusing on the physical layer descriptors.

## 2.3. RAN Slicing

A first step to understand the impact of network slicing to the 5G RAN design has been given by identifying RAN-specific requirements [7] needed to fulfil the network slicing vision. The set of requirements is the following:

- Utilization of RAN resources should be maximized.
- RAN should be slice-aware, making it possible to prioritize different service and signalling in a different way.
- RAN should support mechanisms for traffic differentiation in order to be able to treat different slices or services differently within the multi service slices.
- RAN should support protection mechanisms for slice isolation so that events within one slice do not have a negative impact on another slice.
- RAN should support efficient management mechanisms to efficiently set up new slices or operate new services.

With these requirements as a starting point, a set of new blocks of information, configuration description and protocol features has to be introduced across the protocol layers of the radio interface. Figure 4 illustrates the proposed overall framework from RAN slicing support within a NG-RAN node, whose different blocks and descriptors are explained below [4].

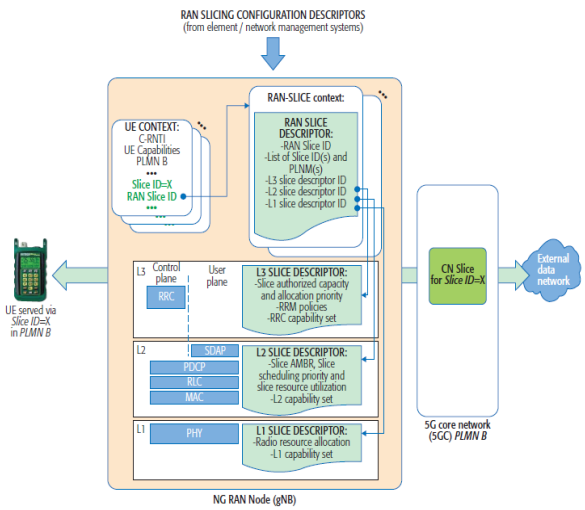


**Figure 4. Framework of the realization of RAN slices in a NG-RAN node**

UE Context: it is a block of information that contains all the required data to maintain the RAN services towards the UE. It is instantiated within the RAN when the UE becomes active and it establishes a logical control connection. A RAN slice identifier (*RAN Slice ID*) is added to this block to use it as a pointer to a new block of information called RAN Slice Context.

RAN Slice Context: it contains all the data necessary to support the operation of a particular RAN slice along with the *Slice ID(s)* that are served through that RAN slice. This block of information also contains the configuration descriptors of the radio protocol layers 3, 2 and 1 (L3, L2 and L1) for the realization of the RAN slice.

These layer descriptors are detailed in the following subchapters, based on the information taken from [4].

### 2.3.1. L3 Configuration

A *L3 Slice Descriptor* is necessary to specify the capacity allocation for the RAN slice. This layer comprises the radio resource management (RRM) functions, which govern the operation of the slice, and the capability set of the Radio Resource Control (RRC) protocol in use.

When multiple RAN slices are realized over share radio resources, the RRM functions such as radio bearer control (RBC), radio admission control (RAC) and connection mobility control (CMC) have to assure that each slice gets the expected amount of resources and handles any conflict that might appear across slices. In order to dictate the operation of these RRM functions, a set of parameters per RAN slice are proposed:

- **Slice Authorized Capacity:** this parameter is used by the RAC for the admission or rejection of radio bearers (RBs), which are the data transfer services delivered by the radio protocol stack. Its purpose is to limit the number and characteristics of the RBs established for the slice, and it is done with a combination of two types of parameters: resource-oriented parameters (e.g. maximum number of physical resource blocks (PRBs) to be served through a slice or limit of UEs connected) and rate-oriented parameters (e.g. maximum guaranteed bit rate (GBR) per slice).
- **Slice Allocation Priority:** this parameter allows for conflict resolution among UE/RB resource requirements across slices that cannot be solved based only on the *Slice Authorized Capacity* parameters. The priority and pre-emption policies at UE/RB levels are solved through the allocation and retention priority (ARP) parameter included in the QoS profile.

Regarding the RRC protocol, when multiple RAN slices are configured to share the same set of common logical channels, some features have to be incorporated within this protocol:

- Protocol fields to allow UEs to discriminate among signalling from different slices.
- System information block (SIB) messages to advertise the slice that can be reached from the cell.
- SIB messages to support cell re-selection, including information of the neighbour cell.
- Allow paging cycles to be organized considering the specific needs to each slice.

### 2.3.2. L2 Configuration

Layer 2 comprises the Medium Access Control (MAC) sub-layer for multiplexing and scheduling the packet transmissions over a set of transport channels exposed by L1. Considering that the current MAC operation is based on individual UE and DRB-specific QoS profile, it is necessary to define the Packet Scheduling (PS) behaviours and specify the capability set of the L2 sub-layers processing functions.

Therefore, the proposed *L2 Slice Descriptor* includes the following parameters to dictate the operation of the MAC scheduler and yield isolation:

- **Slice – AMBR:** to limit the aggregate bit rate of all the services that do not have a guaranteed bit rate (non-GBR services).
- **Slice Scheduling Priority:** to manage short-term traffic congestion conflicts between RBs with the same QoS profile.
- **Slice Resource Utilization:** used to establish constraints on the amount of physical layer resources consumed by the slice. It can be formulated as a percentage of the overall L1 resources managed by the MAC scheduler.

### 2.3.3. L1 Configuration

The new physical layer for 5G NR is being defined to provide high flexibility for the use of different waveforms and adaptable time-frequency frame structures. L1 provides L2 with transfer services in the form of transport channels, which define how the data is transferred.

The L1 optimal settings can differ per slice type, so it is needed to establish a way to divide the L1 radio resource structure allowing that different settings can be simultaneously applied. The mixing of L1 slices could be achieved through the use of the different OFDM numerologies.

Numerology is a new concept introduced in the 5G NR, which represents the subcarrier separation, $\Delta f$, which can take values of 15, 30, 60 or 120 kHz. Depending on the numerology ($\Delta f$) and the channel bandwidth, the number of available PRBs will be different, as shown in Table 1.

| Δf (kHz) | 5 MHz | 10 MHz | 15 MHz | 20 MHz | 25 MHz | 30 MHz | 40 MHz | 50 MHz | 60 MHz | 70 MHz | 80 MHz | 90 MHz | 100 MHz |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 25 | 52 | 79 | 106 | 133 | 160 | 216 | 270 | N.A | N.A | N.A | N.A | N.A |
| 30 | 11 | 24 | 38 | 51 | 65 | 78 | 106 | 133 | 162 | 189 | 217 | 245 | 273 |
| 60 | N.A | 11 | 18 | 24 | 31 | 38 | 51 | 65 | 79 | 93 | 107 | 121 | 135 |

**Table 1. Number of PRBs for each bandwidth/numerology [8]**

It is important to say that not all the numerologies are used in every Frequency Range (FR). The table above is used for FR 1 (channel bandwidth goes from 5 to 100 MHz) and for FR2 (channel bandwidth from 50 to 400 MHz) is considered sub-carrier separations of 60 or 120 kHz.

## 2.4. Reinforcement Learning (RL)

As it is said before, 5G networks are expected to support a variety of applications with diverse requirements, becoming increasingly heterogeneous and decentralized. In this context, network softwarization has become an important aspect. However, considering the dynamics and uncertainty of the wireless network environments of 5G, conventional approaches for resource management that require complete and perfect knowledge of the systems are inefficient or even inapplicable.

In such unpredictable network environments, reinforcement learning (RL) has proved to be a viable tool when addressing real-time dynamic-decision-making-problems [11]. In this chapter, the field of RL with its fundamental principles is presented.

### 2.4.1. Agent-Environment interaction

RL is an area of machine learning that is focused on learning through interaction. An agent takes actions at a certain system state and observes the responses from the environment, within which the agent can learn to optimize its behaviour. Anything that the agent cannot change arbitrarily is considered to be part of the environment, and the actions can be any decision the agent wants to learn. The agent adopts a trial-and-error search.

The agent-environment interaction model is illustrated in Figure 5.

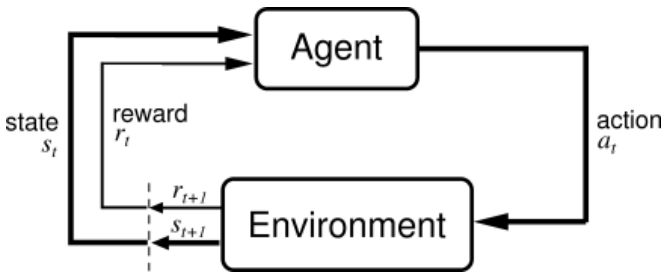


**Figure 5. Agent-Environment interaction model [12]**

According to this model, the learning process is done in few steps. At each time step t, the agent observes a state $s_t$ and choose an action $a_t$ to perform. When the action is taken, the environment and the agent transition to a new state, $s_{t+1}$, based on the current state and the chosen action. Moreover, every time the environment moves to a new state, it also provides a scalar reward $r_{t+1}$ to the agent as feedback, indicating how good or bad has been the action selected by the agent over the environment. This process is repeated until the agent approaches to an optimal behaviour.

### 2.4.2. Markov Decision Process (MDP)

Formally, RL environments can be described as a Markov decision process (MDP) [9]. The Markov property says that "The future is independent of the past given the present", which means that the state is a sufficient statistic of the future; once it is known, the history can be thrown away [13]. Therefore, starting only from knowledge of the first state, it is possible to make a prediction of all the expected future states and rewards iterating the model.

The description of the MDP is done with a tuple of five elements $< S, A, P, R, \gamma >$, where [13]:

- $S$ is a finite set of states that the environment can reach.
- $A$ is the finite set of the possible actions that the agent can choose.
- $P$ is the state transition probability function:

$$P(s', a, s) = \mathbb{P}[s_{t+1} = s' | s_t = s, a_t = a] \quad \textbf{(1)}$$

- $R$ is the reward function:

$$R(s', a, s) = \mathbb{E}[r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'] \quad \textbf{(2)}$$

- $\gamma$ is the discount factor, $\gamma \in [0,1]$. It determines how much importance is to be given to the immediate rewards and future rewards. Lower values place more emphasis on immediate rewards, while values close to 1 give more importance to future rewards.

The objective is to maximize the cumulative reward received from the environment [16], which is called return function, $G_t$.

$$G_t = r_{t+1} + r_{t+2} + r_{t+3} + \cdots + r_T \quad \textbf{(3)}$$

The equation above works well with episodic tasks that have a terminal state. However, if the task is continuous and does not have any terminal state, the expected return will go to infinite. So, it is necessary to use the discount factor presented before to avoid the infinite return discounting the cumulative rewards. Consequently, the expected return will take the form of:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \quad \textbf{(4)}$$

In order to maximize (4), the agent will learn a policy $\pi$ that describes the way of acting to maximize the expected return in every state. In general, it defines a probability distribution over actions for each state:

$$\pi(a|s) = P(a_t = a \mid s_t = s) \quad \textbf{(5)}$$

The goal of RL is to find an optimal policy $\pi^*$ that achieves the maximum expected return from all states [9]:

$$\pi^* = \underset{\pi}{\operatorname{argmax}}\, \mathrm{E}[G_t | \pi] \quad \textbf{(6)}$$

### 2.4.3. Value functions

There are two main approaches to solving RL problems: methods based on value functions and methods based on policy search [9]. Moreover, there is also a hybrid actor-critic approach that employs both value functions and policy search. This chapter is focused on value functions methods.

Two types of value functions can be differentiated: state-value functions, $V_\pi(s)$, and state-action value functions, $Q_\pi(s,a)$. The state value function describes the value of a state when following a policy. It is the expected return when starting from state $s$ acting according to our policy $\pi$ [18]:

$$V_\pi(s) = \mathbb{E}[G_t \,|s_t = s, \pi] \tag{7}$$

The optimal policy, $\pi^*$, has also a corresponding optimal state-value function, $V^*(s)$, which can be defined as:

$$V^*(s) = \max_\pi V_\pi(s) \quad \forall s \in S \tag{8}$$

Similarly, the state-action value function tells the value of taking an action in some state when following a certain policy. It is the expected return starting from state $s$, taking action $a$, and then following policy $\pi$. This function is usually called *Q-function* and the corresponding values, *Q-values.*

$$Q_\pi(s,a) = \mathbb{E}[G_t \,|s_t = s, a_t = a, \pi] \tag{9}$$

The optimal Q-function is the maximum over all policies:

$$Q^*(s,a) = \max_\pi Q_\pi(s,a) \tag{10}$$

#### 2.4.3.1. Bellman Equations

In order to find the optimal policies and value functions, Bellman Equations are used, which allow to start solving these MDPs. Bellman Equation states that value functions can be decomposed into two parts [16]:

- Immediate reward, $r_{t+1}$.
- Discounted value of successor states,

Mathematically, the Bellman Equation can be defined as [13]:

$$V(s) = \mathbb{E}\left[r_{t+1} + \gamma V(s_{t+1}) \mid s_t = s\right] \tag{11}$$

This equation tells how good it is to be in a particular state $s$. It is computed as the immediate reward of moving to state $s'$ plus the discounted value of the successor state. Following image shows a backup diagram to fully understand the Bellman Equation.

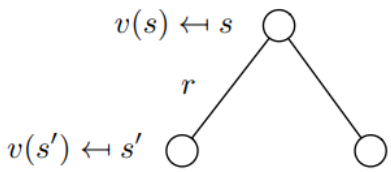


**Figure 6. Backup diagram for Bellman Equation [13]**

However, to solve the MDPs is used specifically the Bellman Expectation Equation. This equation is the same as Bellman Equation, but now it tells the value of being in a particular state subjected to some policy. There is a specific equation for each type of value function.

On the one hand, the Bellman Expectation Equation for state-value functions is defined mathematically as:

$$V_\pi(s) = \mathbb{E}_\pi[r_{t+1} + \gamma V_\pi(s_{t+1}) \mid s_t = s] \quad \textbf{(12)}$$

To understand better the meaning of this equation, a backup diagram is made [17].

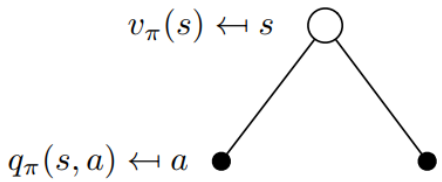


**Figure 7. Backup diagram for state-value function**

This backup diagram describes the value of being in a particular state subjected to some policy. From the state $s$ there is some probability that each action is taken, and there is a Q-value for each of the actions. Then, the Q-values are averaged telling how good it is to be in a particular state. Thus, the state-values can be expressed as a function of the Q-values.

$$V_\pi(s) = \sum_{a \in A} \pi(a|s)\, Q_\pi(s,a) \quad \textbf{(13)}$$

On the other, the same process can be done with the Q-function. From Bellman Expectation Equation, Q-functions can be expressed as follows:

$$Q_\pi(s,a) = \mathbb{E}_\pi[r_{t+1} + \gamma Q_\pi(s_{t+1}, a_{t+1}) \mid s_t = s, a_t = a] \quad \textbf{(14)}$$

The backup diagram for this function is also done:

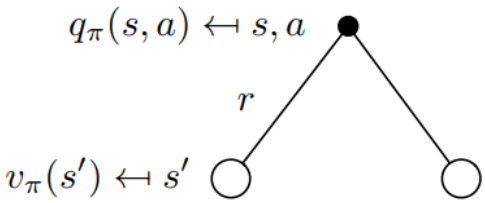


**Figure 8. Backup diagram for state-action value function**

Now, this diagram describes how good it is to take a particular action in a state. Depending on the action, the agent may be moved to any of the states of the environment. The state-values of both states are averaged again, adding an immediate reward that tells how good it is to take this particular action.

Mathematically, it can be defined as:

$$Q_\pi(s,a) = R(s',a,s) + \gamma \sum_{s' \in S} P(s',a,s)\, V_\pi(s') \tag{15}$$

Next step is to join both backup diagrams. Depending on how it is done, two different diagrams are obtained:

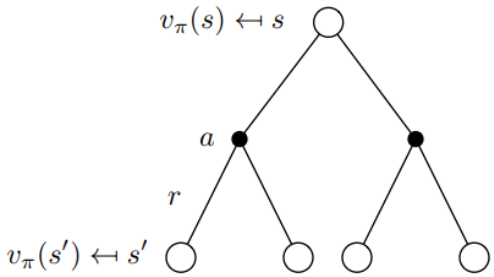


Figure 9. Backup diagram for state-value function

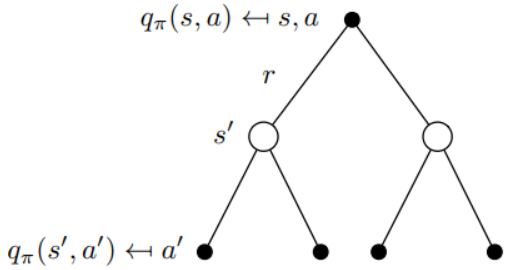


Figure 10. Backup diagram for state-action value function

On the one hand, from Figure 9 it can be obtained how good it is to be in state $s$ after taking some action $a$ and landing on another state $s'$, following a policy $\pi$.

$$V_\pi(s) = \sum_{a \in A} \pi(a|s) \left( R(s',a,s) + \gamma \sum_{s' \in S} P(s',a,s)\, V_\pi(s') \right) \tag{16}$$

On the other hand, from Figure 10 it is obtained how good it is to take some action $a$, which makes the agent transitions to a state $s'$, and from which it is taken another action $a'$, following a particular policy all along.

$$Q_\pi(s,a) = R(s',a,s) + \gamma \sum_{s' \in S} P(s',a,s) \sum_{a \in A} \pi(a'|s')\, Q_\pi(s',a') \tag{17}$$

#### 2.4.3.2. Bellman Optimality Equations

At this point, optimal policy can be defined. One policy $\pi$ is better than another policy $\pi'$ if the value functions with $\pi$ for all states is greater than the value function with $\pi'$ for all states. This concept can be intuitively expressed as:

$$\pi \geq \pi' \; if \; V_\pi(s) \geq V_{\pi'}(s), \forall s \tag{18}$$

The optimal policy is found by maximizing over the optimal state-action value function presented in equation (10), $Q^*(s,a)$. In other words, the best policy can be found by choosing the action greedily at every state.

$$\pi^*(a|s) = \begin{cases} 1 \; if \; a = \max\limits_{a \in A} Q^*(s,a) \\ 0 \; otherwise \end{cases} \quad \textbf{(19)}$$

Knowing $Q^*(s,a)$, the optimal policy is immediately obtained. Considering this and the equation (12), the optimal state-value function of equation (8) can be rewritten as:

$$V^*(s) = \max_a Q^*(s,a) \quad \textbf{(20)}$$

After that, Bellman Optimality Equations can be easily defined. These equations are the same as Bellman Expectation Equations but the only difference is that instead of taking the average of the actions, the agent can take the action with the maximum value [17]. Therefore, equations (15) and (16) can be redefined as:

$$V^*(s) = \max_a R(s',a,s) + \gamma \sum_{s' \in S} P(s',a,s)\, V^*(s') \quad \textbf{(21)}$$

$$Q^*(s,a) = R(s',a,s) + \gamma \sum_{s' \in S} P(s',a,s) \max_{a'} Q^*(s',a') \quad \textbf{(22)}$$

All these equations are used to formalise a RL problem in a MDP, but it is necessary to explain how these equations can be solved. There are two different types of methods depending on the knowledge that the agent has of the environment.

If the agent has plenty knowledge about the environment (known MDP), the system can be solved. These systems are solved by using model-based methods like Dynamic programming [14].

In contrast, if the agent has no knowledge about the environment (unknown MDP), an estimation of the value function of the system is done. The methods that perform this estimation are known model-free methods, divided into two main classes of algorithms: Monte-Carlo and Temporal-Difference (TD).

Following paragraphs explain the TD algorithms, highlighting the Q-learning algorithm, which is one of the interesting use cases for this thesis.

#### 2.4.3.3. Temporal-Difference learning: Q-learning algorithm

TD methods are based on the bootstrapping operation, which is the capability of using the estimate of the values of future states to update the estimate of the current state. These methods perform a sample backup, within which the new estimate is obtained from a sample of the possible options for the next. Mathematically, it can be expressed as [9]:

$$Q_\pi(s_t,a_t) \leftarrow Q_\pi(s_t,a_t) + \alpha\delta \quad \textbf{(23)}$$

where $\alpha$ is the learning rate and $\delta = Y - Q_\pi(s_t, a_t)$ the TD error. Here, $Y$ is the target that will vary depending on the algorithm used.

TD methods can be classified into two main groups: on-policy and off-policy learning algorithms. On-policy means that the policy for evaluation is the same policy that is used to generate the experience. This policy often has some element of randomness (exploration) to guarantee that the system converges. SARSA is an on-policy learning algorithm, which results in setting:

$$Y = r_{t+1} + \gamma Q_\pi(s_{t+1}, a_{t+1}) \quad \textbf{(24)}$$

On the other hand, off-policy means that the policy used in the process of learning has no correlation with the policy being evaluated. Q-learning is the most well-known off-policy model-free RL algorithm. In this case, the learned action-value function directly approximates the optimal Q-function independently of the policy being followed [12].

$$Y = r_{t+1} + \gamma \max_a Q_\pi(s_{t+1}, a) \quad \textbf{(25)}$$

Thus, the updated rule of Q-values with Q-learning algorithm is

$$Q_\pi(s_t, a_t) \leftarrow Q_\pi(s_t, a_t) + \alpha\left(r_{t+1} + \gamma \max_a Q_\pi(s_{t+1}, a) - Q_\pi(s_t, a_t)\right) \quad \textbf{(26)}$$

The algorithm of this process is shown below, taken from [12].

---

**$Algorithm\ 1$: $Q-learning$**

---

$Initialize\ Q(s,a), \forall s \in S, a \in A, arbitrarly, and\ Q(terminal\ state, \cdot) = 0$

**$Repeat$** $(for\ each\ episode)$:

$Initialize\ S$

**$Repeat$** $(for\ each\ step\ of\ episode)$:

$Choose\ a_t\ from\ s_t\ using\ policy\ derived\ from\ Q(e.g. \varepsilon - greedy)$

$Take\ action\ a_t\ and\ observe\ r_{t+1}, s_{t+1}$

$$Q_\pi(s_t, a_t) \leftarrow Q_\pi(s_t, a_t) + \alpha\left(r_{t+1} + \gamma \max_a Q_\pi(s_{t+1}, a) - Q_\pi(s_t, a_t)\right)$$

$s_t \leftarrow s_{t+1}$

**$Until$** $s_t\ is\ terminal$

---

The agent calculates and updates the Q-function for the optimal actions until convergence with a properly set learning rate. The Q-function updating process is defined as training.

As Q-learning is an off-policy TD method, it allows both behaviour and target policies to improve. The target policy $\pi$ is greedy (always looking for the best action), whereas the behaviour policy $\mu$ that is used for the learning process is $\varepsilon$-greedy.

The $\varepsilon$-greedy policy is one solution for the exploration vs exploitation dilemma of TD learning algorithms. Exploration means choosing random actions during some steps to get new information that could be useful to lead to better policies. Exploitation is the process of choosing the actions which performs better with the acquired information. It is necessary to balance exploration and exploitation to reach convergence of the system, and $\varepsilon$-greedy policy achieves it. It selects a randomly action with probability $\varepsilon$, and with probability $1-\varepsilon$ selects the best action ($a = arg\max_{a \in A} Q(s,a)$).

### 2.4.4. Policy search

The methods mentioned above indirectly learn the policy based on the estimate of the value functions. These value-based approaches are effective in handling problems in a discrete action field, but they cannot be applied when dealing with a problem with a continuous action space. In continuous spaces, finding the greedy policy requires an optimization of the action at every time step, which can be done with an actor-critic approach based on the DPG (deterministic policy gradient) algorithm. To understand this algorithm, first it is necessary to explain the policy search and policy gradient concepts.

Policy search methods, instead of maintaining a value function model they directly search for an optimal policy $\pi^*$. Typically, a parametrized policy $\pi_\theta$ is chosen, whose parameters are updated to maximize the expected return, $\mathbb{E}[G_t|\pi]$, using either gradient-based or gradient-free optimization [9]. Gradient-free optimization can effectively cover low dimensional parameter spaces, but for a large number of parameters gradient-based training remains the most used method.

#### 2.4.4.1. Policy gradient

Policy gradient algorithms are the most popular class of continuous action reinforcement learning algorithms. The basic idea behind policy gradient algorithms is to adjust the parameters $\theta$ of the policy in the direction of the performance gradient $\nabla_\theta J(\pi_\theta)$ [23]. $J(\pi_\theta)$ is the objective function, which is defined as $J(\pi) = \mathbb{E}[G_t|\pi]$. A stochastic policy can be defined as:

$$\pi_\theta = P[a|s;\theta] \tag{27}$$

Then, the corresponding gradient is [23]:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s\sim p^\pi, a\sim\pi_\theta}[\nabla_\theta log\pi_\theta(a|s) Q^\pi(s,a)] \tag{28}$$

This shows that the gradient is an expectation of possible states and actions. In particular, despite the fact that the state distribution $p^{\pi}(s)$ depends on the policy parameters, the policy gradient does not depend on the gradient of the state distribution. The unknown part, $Q^{\pi}(s,a)$, is normally estimated by using the actual returns $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$ as an approximation for each $Q^{\pi}(s_t, a_t)$ [24].

### 2.4.5. Actor-critic methods

The actor-critic is a widely used architecture based on the policy gradient theorem. It is the combination of value functions with an explicit representation of the policy. The "actor" (policy) learns by using feedback from the "critic" (value function), as shown in Figure 11 [12]. In doing so, these methods trade-off variance reduction of policy gradients with bias introduction from value function methods [9].

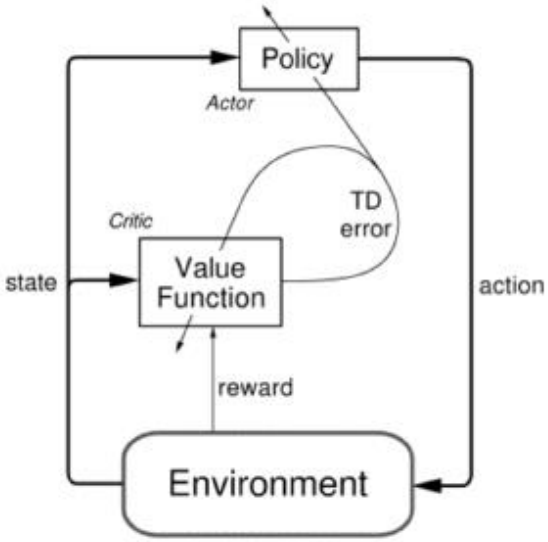


**Figure 11. Actor-critic architecture**

The function of each "component" is the following [23]:

- An actor adjusts the parameters $\theta$ of the stochastic policy $\pi_\theta$ by stochastic gradient ascent of equation (27). Instead of the unknown true action-value function $Q^{\pi}(s_t, a_t)$, an action-value function $Q^{w}(s_t, a_t)$ is used, with parameter vector $w$.
- A critic estimates the action-value function $Q^{w}(s_t, a_t) \approx Q^{\pi}(s_t, a_t)$ using an appropriate policy evaluation such as temporal-difference learning.

Therefore, the process can be described as following. The actor receives a state from the environment and chooses an action to perform. At the same time, the critic receives the state and reward resulting from the previous interaction. The critic uses the TD error calculated from this information to update itself and the actor.

#### 2.4.5.1. Deterministic Policy Gradient (DPG)

One development in the context of actor-critic algorithms is deterministic policy gradients (DPGs), which extend the standard policy gradient theorems for stochastic policies to deterministic policies. One of the major advantages of DPG is that, while stochastic policy gradients integrate over both state and action spaces, DPGs only integrate over the state space, requiring fewer samples in problems with large action spaces.

The DPG algorithm maintains a parametrized actor function $\mu(s;\theta^{\mu})$ which specifies the current policy by deterministically mapping states to a specific action [24]:

$$\mu_{\theta} : S \rightarrow A \tag{29}$$

with parameter vector $\theta \in \mathbb{R}^n$. Thus, the deterministic analogue to the policy gradient, equation (27), is [23]:

$$\nabla_{\theta} J(\mu_{\theta}) = \mathbb{E}_{s\sim p^{\mu}}\left[\nabla_{\theta}\mu_{\theta}(s)\, \nabla_{a} Q^{\mu}(s,a)|_{a=\mu_{\theta}(s)}\right] \tag{30}$$

The critic can be updated either with an on-policy or off-policy algorithm. The off-policy methods, which is the interesting use case for this thesis (Q-learning), learn a deterministic policy $\mu_{\theta}$ from trajectories generated by an arbitrary stochastic behaviour policy $\beta(a|s)$. The off-policy deterministic policy gradient is then:

$$\nabla_{\theta} J_{\beta}(\mu_{\theta}) = \mathbb{E}_{s\sim p^{\beta}}\left[\nabla_{\theta}\mu_{\theta}(s)\, \nabla_{a} Q^{\mu}(s,a)|_{a=\mu_{\theta}(s)}\right] \tag{31}$$

Notice that the formula dos not have importance sampling factor. The reason is that the importance of sampling is to approximate a complex distribution with a simple one, but the output of the policy here is a value instead of a distribution. The Q-value in the formula corresponds to the critics and is updated by Q-learning.

## 2.5. Deep Reinforcement Learning (DRL)

Although RL can be an option for solving service management problems of 5G networks, the conventional RL algorithms suffer from slow convergence speed, leading to poor performance. Therefore, a combination of RL and deep learning, deep reinforcement learning (DRL), has been proposed to overcome these limitations as it has great potential for handling large-scale and dynamics systems.

Now, the combination of RL and deep learning techniques, deep reinforcement learning (DRL), is presented. In this section, DRL-based approaches are explained, as they are the basis of the algorithms used in this thesis.

### 2.5.1. Deep Q Network (DQN)

Q-learning requires the agent to maintain and update a set of Q-values for all the state-action pairs. However, 5G networks are expected to be highly heterogeneous and large in scale, having a number of possible system state-values intractably large. In this case, the

cost of calculating and maintaining all Q-values become practically infeasible, which is called the curse of dimensionality [10].

To address this issue, Deep Q Network (DQN) has been proposed. DQN agent is typical of DRL models that apply a deep neural network (DNN) to approximate the Q-function (Figure 12) as $Q(s, a;\ \theta)$, where $\theta$ is a vector containing all the weights and biases in the neural network.

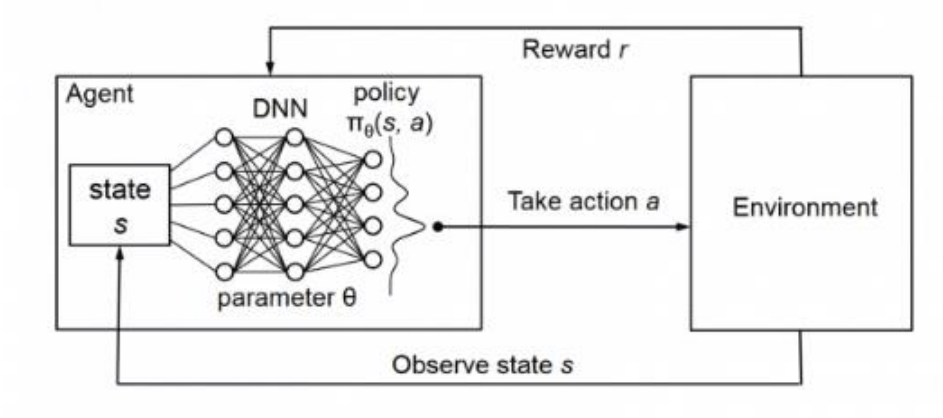


**Figure 12. Illustration of a DQN agent [19]**

As illustrated in Figure 12, the DQN takes the current observed system states as the input. All inputs states are transferred to different neural network layers with some particular weight factors, $\theta$, which is the parameter that is updated during the training. The weights in each layer begin with random values, and these are iteratively improved over time to make the network more accurate. A loss function is used to quantify how inaccurate the network is, and a gradient descent is performed to minimize this loss function. The DQN outputs a set of Q-values with respect to all the possible actions; there are as many output nodes as possible actions the agent can take.

The next action is determined by the maximum output of the Q-values. The goal of DQN is to find the most feasible weight factors from historical data.

The major differences between RL and DRL are the use of two techniques, which addressed the fundamental instability problem of function approximation.

1. **Experience Replay:** each of the experiences between the agent and the environment can be summarized in tuples $< s_t, a_t, s_{t+1}, r_{t+1} >$, which are stored in a replay buffer $D$. In RL, the arriving samples were used to train the parameters, whereas DRL randomly selects a mini-batch of samples from that buffer to train the DQN's parameters [10]. As the samples are being randomly selected, the correlation between consecutive samples from the environment is avoided.
2. **Target Network:** two neural networks are created to avoid instabilities during training [20]. One network (Q network) is updated every training step $Q(s, a;\ \theta)$ and the other, the target network $Q(s, a; \theta^-)$, is updated every C steps copying the weights from the actual Q network.

After sampling the mini-batch of transitions from the replay buffer, the weights of the neural network ($\theta_i$) are updated via the gradient descent of the loss function, which is shown in the following formula [20]:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim D_i}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a;\ \theta_i)\right)^2\right] \quad \textbf{(32)}$$

The equation above represents the loss, which is just the squared difference between target Q and the predicted Q. The target used by DQN is then [22]:

$$Y^{DQN} = r_{t+1} + \gamma \max_{a} Q(s_{t+1}, a; \theta_t^-) \quad \textbf{(33)}$$

### 2.5.2. Double Deep Q Network (DDQN)

This subchapter presents another learning algorithm called Double DQN (DDQN), but before examining its contribution, an overview of Double Q-learning algorithm is done.

The max operator in standard Q-learning and DQN uses the same values both to select and to evaluate the action. This can lead to a problem of overestimation, in which the agent always chooses the non-optimal action in any given state only because it has the maximum Q-value. To prevent this, the action selection and action evaluation can be decoupled, which is the idea behind Double Q-learning [21].

In the Double Q-learning algorithm, two value functions are learned by assigning each experience randomly to update one of the two value functions, such that there are two sets of weights, $\theta$ and $\theta'$. For each update, one set of weights is used to determine the greedy policy and the other to determine its value. The Double Q-learning target can then be written as [21]:

$$Y^{DoubleQ} = r_{t+1} + \gamma\, Q\left(s_{t+1}, \operatorname*{argmax}_{a} Q(s_{t+1}, a; \theta_t); \theta_t'\right) \quad \textbf{(34)}$$

Notice that the selection of the action is still due to the online weights $\theta$, but the second set of weights $\theta'$ is used to evaluate the value of the policy.

At this point, the idea is to join the contributions of both Double Q-learning and DQN, resulting in Double DQN algorithm. DQN architecture provides a natural candidate for the second value function without introducing additional networks. Therefore, in DDQN it is proposed to [22]:

- Use the DQN network (predicted network) to select what is the best action to take for the next state (the action with the highest Q-value).
- Use the target network to compute the target Q-value of taking that action at the next state.

$$Y^{DoubleDQN} = r_{t+1} + \gamma\, Q\left(s_{t+1}, \operatorname*{argmax}_{a} Q(s_{t+1}, a; \theta_t); \theta_t^-\right) \quad \textbf{(35)}$$

In comparison to Double Q-learning (28), the weights of the second network $\theta_t'$ are replaced with the weights of the target network $\theta_t^-$. The update to the target network is the same as in DQN, copying the weights from the predicted network every C steps. This version of DDQN is perhaps the minimal possible change to DQN towards Double Q-learning. The goal is to get most benefit of Double Q-learning, while keeping rest of the DQN algorithm intact.

The loss formula is defined as:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim D_i}\left[\left(r + \gamma\ Q\left(s', \underset{a'}{\operatorname{argmax}}\, Q(s', a'; \theta_i)\,; \theta_i^-\right) - Q(s, a;\ \theta_i)\right)^2\right] \quad \textbf{(36)}$$

### 2.5.3. Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradient (DDPG) is an algorithm able to solve continuous problems with DRL. The idea is to provide modifications to DPG inspired by the success of DQN, which allow to use deep neural network to learn in large state and action spaces.

The whole model of DDPG is composed of an actor network and a critic network as illustrated in Figure 13, where the difference between DQN and DDPG is also highlighted. The actor network $\mu(s; \theta^\mu)$ serves as the policy and will output the action to perform. The critic network $Q(s, a; \theta^Q)$ serves as the Q-function and will take the action and the state as inputs. The output of the critic will be the estimation reward for that particular action in that state, in contrast to DQN that outputs a set of Q-values with respect to all the possible actions.

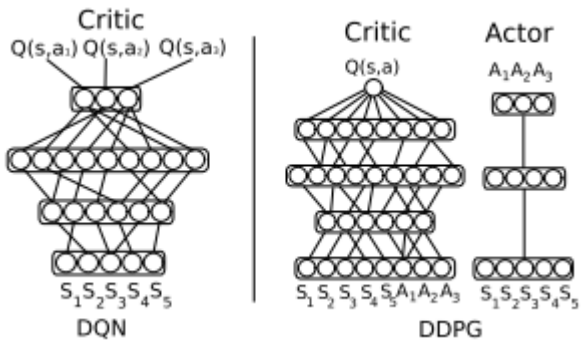


**Figure 13. DQN and DDPG architecture, from [26]**

DDPG uses the two techniques introduced by DQN, replay buffer and target network. The replay buffer is used to sample experience to update the neural network weights. Transitions are sampled from the environment according to the exploration policy and the tuple $< s_t, a_t, s_{t+1}, r_{t+1} >$ is stored in the replay buffer.

Regarding the target networks, two more neural networks are created:

- A target Q network or target critic network, with weights $\theta^{Q'}$
- A target policy network or target actor network, with weights $\theta^{\mu'}$

These targets networks are updated every $C$ steps via "soft updates", by having them slowly track the learned networks [26]:

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'} \quad \textbf{(37)}$$

By doing this, the target values are constrained to change slowly, greatly improving the stability of learning.

When training the network, a random mini-batch is sampled from the buffer and then the target Q-value is computed.

$$y_i = r_i + \gamma Q'\left(s_{i+1}, \mu'\left(s_{i+1};\, \theta^{\mu'}\right);\, \theta^{Q'}\right) \quad \textbf{(38)}$$

After that, both critic and actor network are updated. On the one hand, the critic is updated by minimizing the mean-squared error loss between the updated Q-value and the original Q-value.

$$L = \frac{1}{N}\sum_{i}\left(y_i - Q(s_i, a_i; \theta^{Q})\right)^2 \quad \textbf{(39)}$$

On the other hand, the actor policy is updated using the sampled policy gradient [26]:

$$\nabla_{\theta^{\mu}} J \approx \frac{1}{N}\sum_{i} \nabla_a Q(s, a; \theta^{Q})|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^{\mu}} \mu(s; \theta^{\mu})|_{s_i} \quad \textbf{(40)}$$

A major challenge of learning in continuous action spaces is exploration. In DDPG, an exploration policy $\mu'$ is created by adding noise sampled from a noise process $N$ to our actor policy [25]:

$$a_t = \mu(s_t|\theta^{\mu}) + N_t \quad \textbf{(41)}$$

$N$ can be chosen to suit the environment. In the DDPG paper [25], the authors proposed an Ornstein-Uhlenbeck process to generate temporally correlated exploration, i.e. it generates noise that is correlated with the previous noise.

# 3. System model and simulator development

In this chapter, the different tools used in the project to reach the results are described. Initially, the thesis motivation is presented, explaining the problem of the project and how it will be solved. Then, the system model is described with the different elements that are part of it. Finally, the simulator description is done explaining the classes used and how the algorithms are implemented to achieve our goal.

## 3.1. Thesis motivation

In 5G scenarios, the radio resources have to be shared dynamically among the different tenants of a cell, achieving an efficient distribution of the resources. This thesis is focused on how the capacity allocation for the RAN slices can be managed. The total capacity available in a cell is shared between the different tenants, so each tenant will have a capacity from the total. Since each tenant generates an aggregate traffic, which can vary over the time, the capacity assigned to each tenant must be adapted to the traffic demand in order to satisfy the SLA and make an efficient use of the resources. Therefore, the aim of this thesis is to develop an algorithm able to configure the capacity allocation for each tenant at every moment of time, which is called capacity sharing algorithm.

In order to achieve a correct operation of the above-mentioned algorithm, DRL-based approaches are proposed to address the resource optimization problem and improve the performance of the network slicing. These techniques will learn knowledge about the network through interaction and will be able to make optimal decisions. In this thesis, the three techniques used are DQN, DDQN and DDPG algorithms. All three are off-policy algorithms that combine the training process and the evaluations; the model is being trained and every certain number of steps the evaluation of the model is performed.

In order to create the agents, some hyperparameters have to be defined, whose impact on the system is analysed. For each of the studies, it is observed how the variation of these parameters affects the performance of the system, in terms of convergence time, total reward and throughput. Convergence time is the time, measured in steps, the system needs to reach an optimal policy of the capacity allocation.

## 3.2. Agent-Environment model

The first step is to define the different elements that will be part of the environment, necessary to allow the agent to solve the problem. This project considers a deployment of a NG-RAN comprised of a single cell with a total capacity $C_t$ and two different tenants, each of them provided with a RAN Slice.

For each tenant $k$, a SLA is established with two control parameters:

- **Scenario Aggregated Guaranteed Bit Rate ($SAGBR_k$):** the capacity that must be guaranteed to each tenant whenever the offered traffic is lower than the total capacity available.
- **Maximum Aggregated Bit Rate ($MABR_k$):** maximum capacity that can be provided to a tenant, it cannot be beaten in any circumstance.

Next figure shows a scheme of the solution:

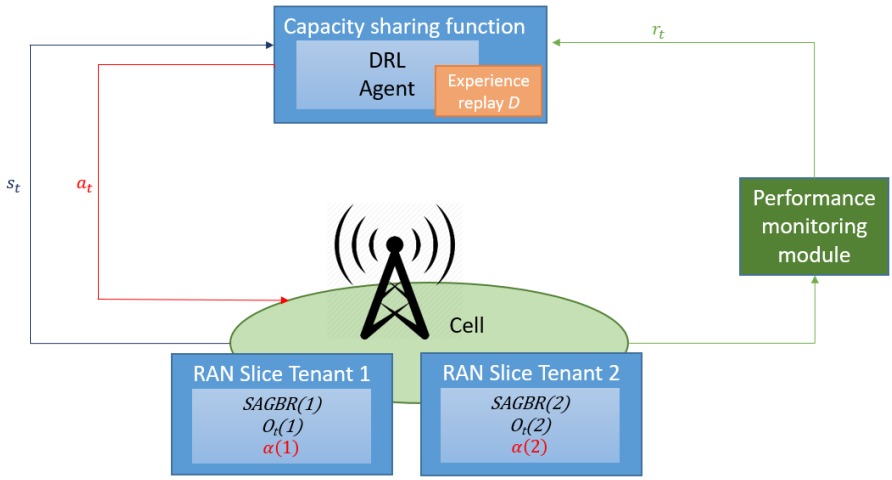


**Figure 14. Overall system architecture**

As it is seen in the literature, from the environment it is necessary to define the state observed by the agent and how the reward is computed. In addition, it is necessary to describe which actions the agent can perform. These three features are detailed below:

State: It is defined as a tuple of $< \rho_t(1), \alpha_t(1), \rho_t(2), \alpha_t(2), \alpha_{ava} >$, where each parameter represents:

- $\boldsymbol{\rho_t(k)}$: It is the percentage of occupation of the resources used by the tenant $k$ at the time $t$.
- $\boldsymbol{\alpha_t(k)}$: It is the percentage of the capacity allocated to the tenant $k$ at the time $t$.
- $\boldsymbol{\alpha_{ava}}$: It is the percentage of the capacity that is not allocated to any tenant, i.e. capacity available:

$$\alpha_{ava} = 1 - \sum_{k=1}^{K} \alpha_t(k) \quad \textbf{(42)}$$

Reward: Each time the agent performs an action, which will be explained next, the environment will return a reward that is defined as:

$$r_t = \varphi_1 \cdot \frac{1}{2}\left(\gamma_{SLA}(1) + \gamma_{SLA}(2)\right) + \varphi_2 \cdot \gamma_{ut} \quad \textbf{(43)}$$

To compute this equation, it is necessary first to define two rates:

- **SLA satisfaction rate, $\boldsymbol{\gamma_{SLA}(k)}$:** this rate is used to know if the SLA established for each tenant is being satisfied. It will be equal to 1 whenever the offered traffic and the throughput are equal, or when the SLA is fulfilled. It is defined as:

$$\gamma_{SLA}(k) = min\left(\frac{Thr_t(k)}{min\left(min(O_t(k), MABR(k)), SAGBR(k)\right)}, 1\right) \quad \textbf{(44)}$$

where $Thr_t(k)$ is the throughput, which is defined as $Thr_t(k) = C_T \cdot \rho_t(k)$, and $O_t(k)$ the offered traffic.

- **Utilization rate, $\gamma_{ut}$:** this rate is used to compute how much capacity that is being allocated to a tenant is not being used.

$$\gamma_{ut} = \frac{\sum_{k=1}^{K} Thr_t(k)}{\sum_{k=1}^{K} C_T \cdot \alpha(k)} = \frac{\sum_{k=1}^{K} \rho_t(k)}{\sum_{k=1}^{K} \alpha_t(k)} \quad \textbf{(45)}$$

To govern the importance of each rate, two weights ($\varphi_1$, $\varphi_2$) are defined. In this thesis, both weights are set to 0.5, giving equal importance to both.

Action: At each time step, the agent will consider the joint action of the two tenants of the cell. The action consists of modifying the assigned capacity to each tenant by a positive/negative percentage value (e.g. ±10%) or doing nothing. Therefore, the action is defined as a vector, $\underline{a_t} = [a_t(1), a_t(2)]$, where $a_t(k)$ is the modification of each tenant. The allocated capacity of the tenant $k$ is then modified as:

$$\alpha_t(k) = \alpha_{t-1} + a_t(k) \quad \textbf{(46)}$$

The actions can be defined either in a discrete or continuous action space, depending on the approach used. Changing the action step means changing how the capacity is adjusted, so it will have a big impact in terms of convergence of the system and reward obtained.

There are three particular cases that have to be described:

1- When the system is initiated ($t = 0$), the allocated capacity is established by the SAGBR of each tenant:

$$\alpha_{t=0}(k) = \frac{SAGBR(k)}{\sum_{k=1}^{K} SAGBR(k)} \quad \textbf{(47)}$$

2- If a tenant does not have assigned capacity and the action to perform is a negative value (e.g. -10%), the assigned capacity of that tenant will still being 0. Mathematically, it can be expressed as:

$$if\ a_t(k) = -a \quad when \quad \alpha_{t-1}(k) = 0\% \rightarrow \alpha_t(k) = 0\% \quad \textbf{(48)}$$

3- If a tenant is being provided with the MABR capacity and the action to perform is a positive value (e.g. +10%), the assigned capacity of that tenant will still being MABR. Mathematically, it can be expressed as:

$$if\ a_t(k) = +a \quad when \quad \alpha_{t-1}(k) = \frac{MABR(k)}{C_T} \rightarrow \alpha_t(k) = \frac{MABR(k)}{C_T} \quad \textbf{(49)}$$

### 3.3. Simulator description

The implementation of the model described, and the analysis of the different algorithms proposed is done with a simulator that has been programmed using Python programming language. The main reason for using Python is the utilization of the *TensorFlow* library,

which is an open source library to help the development and training of the ML models. Some key features that the simulator has are:

- With the *TensorFlow* library, the implementation of the DRL approaches can be done easier and it facilitates the possibility to upgrade the code in the future.
- The simulator is easy to configure as different scenarios might be analysed.
- The simulator is able to analyse the different algorithms proposed, so changing the agent solution can be done easily.
- Once the simulator is executed, it exports all the relevant information needed to analyse the results and make a comparison between algorithms.

### 3.3.1. Simulator classes

As Python is an object-oriented language, the simulator can be created by implementing different classes, each of them representing an element of the model. The classes used are a combination between new classes created and classes imported from the *TensorFlow* library. The three classes created in this project are:

- **Base Station:** this class represents an individual base station. In order to initialize a base station, the different configuration parameters have to be set up (number of tenants, number of PRBs, total capacity, etc.). It is used to execute the actions of the agent and obtain the corresponding state of performing that action.
- **BS Controller:** this class is the controller of the different base stations of the scenario. It is responsible for communicating to the base station the action to perform and it computes the reward of taking that action. As the proposed scenario we will use only considers one base station, the controller will only manage one.
- **Environment:** this class calls the above classes to interact with the agent. In order to be compatible with the agents and the TensorFlow libraries, the environment has to be written in a particular form, as it has to be converted from Python to TensorFlow. In order to achieve it, this class must have some methods:
    - *Step():* this method applies an action to the environment calling the methods from the two classes above. It returns a tuple containing the next observation[1] of the environment and the reward for the action.
    - *Reset():* it starts a new sequence and provides an initial TimeStep. This method is not called explicitly, as it is assumed that environments reset automatically, either when they finish the episode or when *step()* is called the first time.
    - *Observation_spec():* it returns a nest describing the name, shape, datatypes and ranges of the observations.
    - *Action_step():* it returns a nest describing the name, shape, datatypes and ranges of the possible actions.

Regarding the classes used from the library, the following can be highlighted:

[1] In the TensorFlow library, the state of the environment is called observation.

- **DQN Agent / DDQN Agent / DDPG Agent:** they contain methods that facilitate the implementation of the different approaches.
- **Q Network:** this class represents the neural network model that can learn to predict Q-values for all actions, given an observation from the environment. To create a QNetwork object, it is necessary to pass the *observation_spec, action_spec* and a tuple describing the number and size of the model's hidden layers.
- **Replay Buffer:** this class is used to create the replay buffer, who is responsible for keeping track of data collected from the environment.

### 3.3.2. Hyperparameters

It is important to distinguish between the model parameters and model hyperparameters. The parameters are all the configuration variables that are internal to the model and whose value can be estimated from data (i.e. the weights of the neural networks).

The model hyperparameters are the ones that help with the learning process. They are set manually before starting the training and they have an important impact on the performance of the model, so it is important to choose appropriate values.

The hyperparameters used in the model are:

- **Learning rate:** it controls how much the model changes in response to the estimated error each time the weights are updated during the training, i.e. the learning rate controls the speed at which the model learns. A too small value may result in a very long training and if the learning rate is too large the model may not converge.
- **Batch size:** this hyperparameter indicates how many tuples are chosen randomly when training the model. It is used a mini-batch gradient descent configuration (batch size is set to more than one but less than the total number of tuples).
- **Replay buffer max length:** this hyperparameter tells the maximum capacity of samples that can be stored in the replay buffer.
- **Discount factor:** this hyperparameter is used when doing the transition to the next step. It determines how much importance is to be given to the immediate rewards and future rewards. Lower values place more emphasis on immediate rewards, while values close to 1 give more importance to future rewards.
- **Hidden layers:** they define the size and number of layers of the neural network. They represent the capacity to learn the optimal behaviour, so as more complex the problem, the more nodes and layers will be needed. However, if they neural network is much larger than needed, it will lead to a problem of overfitting.
- **Initial collect steps:** it tells the number of steps done before starting the training with a random policy. It is done to collect some data that can be useful to get information about states that may not be reached during the training.
- **Maximum training steps:** it tells the maximum number of iterations that the simulation will last. This hyperparameter loses importance when studying the convergence time of the algorithm, as the simulation will usually stop before reaching this value.

There are other hyperparameters that impact less on the performance of the model (e.g. the interval of how often the evaluation is done) that are not studied in this thesis.

### 3.3.3. Script

All these classes are combined in a script, where it is implemented the code to run the simulator. The script operation is the following.

It starts initializing the hyperparameter values that will be used during the simulation and the base station configuration parameters. After that, the Base Station Controller is instantiated, who at the same time will instantiate the different base stations of the scenario (one in this case). Once these two objects are created, the environment is instantiated and converted to TensorFlow. To validate the environment, a random policy generates actions and iterates over some episodes to make sure things are working as intended. Two objects environments have to be created, one for training and another for evaluation.

Then, the agent is instantiated, who will be the responsible for managing the capacity sharing algorithm. The agents contain two policies: the main policy used for evaluation (*agent.policy)* and the policy used for data collection (*agent.collect_policy)*. In addition, a third policy (*random_policy)* is created independently of agents that randomly selects an action for each time step. The replay buffer is instantiated and the random policy is executed for a few steps to collect data that is recorded in the replay buffer.

After all of that, the process of learning can start. The agent's network is being trained and every certain number of steps the evaluation of the model is performed. The evaluation is done computing the return, which is the sum of rewards obtained while running the policy in the environment for an episode. After evaluating the model, it is checked if the training is over, either because of the system converges or the total number of iterations is reached. When this happens, the results with all the information are exported.

Next diagram (Figure 15) is a representation of all this process.

The process of training will vary depending on the approach used for the agent. Next subchapters explain the process for the three DRL-based approaches used in this thesis: DQN, DDQN and DDPG.

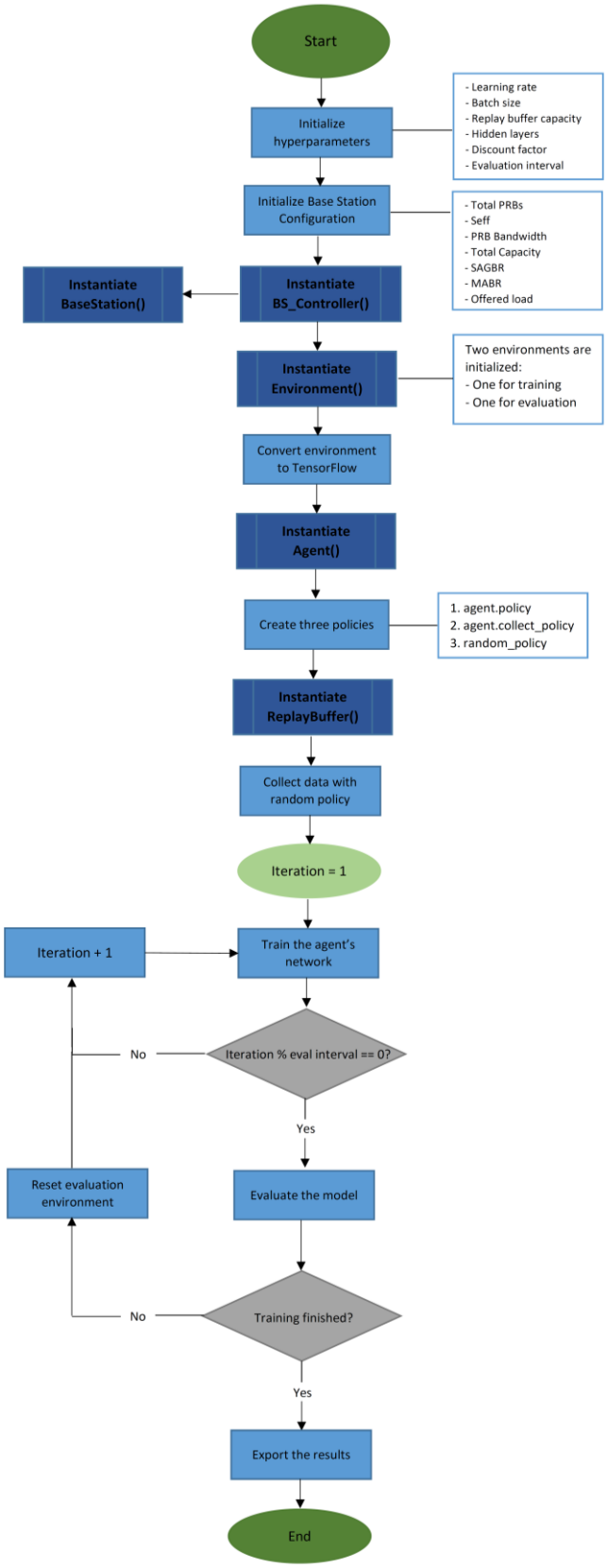


**Figure 15. Simulation algorithm diagram**

### 3.3.4. DQN

The first approach studied is DQN, which can be used in any environment that has a discrete action space. We consider that the possible actions that the agent can take at each step are $a_t(k) =$ [-10%, 0%, 10%], so there will be eight possible combinations between the two tenants.

At the heart of a DQN agent is a Q-network, which is instantiated describing the dimension of the DNN. We have to specify the number and size of the model's hidden layer. In order to instantiate a *DQN Agent* object, there are some arguments that have to be initiated, highlighting the following two:

- *Optimizer:* it is the gradient descent optimization algorithm used for updating the model in response to the output of the loss function. We use Adam optimizer, which stands for adaptive moment estimation, and is a way of using past gradients to compute current gradient. It is here where the learning rate hyperparameter is set.
- *Loss function:* it is a function for computing the TD error loss. The mean squared error (MSE) is used.

Then, the agent can be initialized with random weights, which are copied to the target network. Once the agent and all the elements of the model are initialized, the training process can start. We consider that the target network is updated at every step, so the training process is as following:

**Algorithm 2: DQN training**

$Initialize\ Q-network\ with\ random\ weights\ \theta$

$Initialize\ target\ network\ with\ weights\ \theta^{-} \leftarrow \theta$

**For** $t = 1$ **to** $t_{max}$ **do**

$With\ probability\ \varepsilon\ select\ a\ random\ action\ a_t$

$otherwise\ select\ a_t = \underset{a \in A}{\operatorname{argmax}}\, Q(s_t, a; \theta_t)$

$Execute\ a_t\ to\ the\ environment$

$Observe\ r_{t+1}, s_{t+1}\ from\ the\ environment$

$Store\ transition < s_t, a_t, r_{t+1}, s_{t+1} > in\ D$

$Sample\ a\ random\ mini-batch\ of\ tuples\ from\ D$

$Compute\ target\ Q\ values\ with$

$$Y^{DQN} = r_{t+1} + \gamma \max_{a} Q(s_{t+1}, a; \theta_t^{-})$$

$Perform\ gradient\ descent\ on$

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim D_i}\left[ \left( r + \gamma \max_{a'} Q(s', a'; \theta_i^{-}) - Q(s, a;\ \theta_i) \right)^2 \right]$$

$Train\ the\ network\ \ \theta_t \leftarrow \theta_{t+1}$

$Update\ target\ network\ parameters$

**End**

The process of updating the network (from compute target Q-value onwards) in the algorithm above is done with methods from the *DqnAgent* class, which at the same time interact with methods of other classes. Some methods of the DqnAgent class that can be highlighted are:

- *Loss():* it computes the loss for DQN training.
- *Compute next Q-values():* it computes the Q-value of the next state for TD error computation.
- *Train():* it is the method we call to train the network.

### 3.3.5. DDQN

The second approach implemented is DDQN, which uses the DQN network to select the action to perform and the target network to compute the Q-value. The possible actions are the same than with DQN, $a_t(k) =$ [-10%, 0%, 10%].

The *DdqnAgent* class inherits all the methods from *DqnAgent* except the *compute_next_Q_values()* method. Therefore, to instantiate the object we have to do the same process as before, initializing the same variables (*optimizer, loss function, etc.)*.

Thus, the process of learning will be the same as with DQN, but it will change how the target Q-value is computed, which is the reason why that method has changed. The algorithm then, is the following:

**Algorithm 3: DDQN training**

$Initialize\ Q-network\ with\ random\ weights\ \theta$

$Initialize\ target\ network\ with\ weights\ \theta^- \leftarrow \theta$

**For** $t = 1$ **to** $t_{max}$ **do**

$With\ probability\ \varepsilon\ select\ a\ random\ action\ a_t$

$otherwise\ select\ a_t = \underset{a \in A}{\text{argmax}}\, Q(s_t, a; \theta_t)$

$Execute\ a_t\ to\ the\ environment$

$Observe\ r_{t+1}, s_{t+1}\ from\ the\ environment$

$Store\ transition < s_t, a_t, r_{t+1}, s_{t+1} > in\ D$

$Sample\ a\ random\ mini-batch\ of\ tuples\ from\ D$

$Compute\ target\ Q\ value\ with$

$$Y^{DoubleDQN} = r_{t+1} + \gamma\, Q\left(s_{t+1}, \underset{a}{\text{argmax}}\, Q(s_{t+1}, a; \theta_t); \theta_t^-\right)$$

$Perform\ gradient\ descent\ on$

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim D_i}\left[\left(r + \gamma\ Q\left(s', \underset{a'}{\text{argmax}}\, Q(s', a'; \theta_i); \theta_i^-\right) - Q(s, a;\ \theta_i)\right)^2\right]$$

$Update\ target\ network\ parameters$

**End**

### 3.3.6. DDPG

The third and last approach is DDPG, where the action space changes from discrete to continuous. Instead of having three possible action values in each tenant, the agent can take any value between -10 and 10, so the capacity can be better adjusted.

Now, there are two networks (actor and critic) and their corresponding target networks. We consider that both networks are dimensioned equally, i.e. the two neural networks have the same number of hidden layers and nodes.

In order to instantiate the *DDPGAgent* object, first it is necessary to create the two networks. Both networks need the state and action specifications to know the shape and type of these elements. The actor has the state as the input and the action as output, and the critic has both the state and the action (output of actor) as inputs.

It is necessary to initialize also two optimizers, one for each network, which will be of the same type as before, Adam optimizers. At each of these optimizers, a learning rate is specified, so now we can control the learning of each network separately.

Once we have the critic network, actor network and the optimizer of each network, the *DDPGAgent* object can be instantiated. The loss function used remains the same as before (MSE).

After all this process, the DDPG agent can be initialized. The training process is detailed in Algorithm 4.

Now, the main classes used from the TensorFlow library are *CriticNetwork, ActorNetwork* and *DdpgAgent.* From *DdpgAgent,* some of the methods used to update the network weights at every step are:

- *Critic loss():* it computes the critic loss for DDPG agent.
- *Actor loss():* it computes the actor loss for DDPG agent. The total loss is the sum of both losses.
- *Get target update():* it performs a soft update of the target network parameters.
- *Apply gradients():* it apply the gradients of both actor and critic losses.
- *Train():* it is the equivalent of the DQN method, it is the method we call at to train the network.

**Algorithm 4: DDPG training**

*Randomly initialize critic network* $Q(s,a;\theta^Q)$ *with weights* $\theta^Q$

*Randomly initialize actor network* $\mu(s;\theta^\mu)$ *with weights* $\theta^\mu$

*Initialize target network* $Q'$ *and* $\mu'$ *with weights* $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$

**For** $t = 1$ **to** $t_{max}$ **do**

*Select* $a_t = \mu(s_t|\theta^\mu) + N_t$ *according to the current policy and exploration noise*

*Execute* $a_t$ *to the environment*

*Observe* $r_{t+1}, s_{t+1}$ *from the environment*

*Store transition* $< s_t, a_t, r_{t+1}, s_{t+1} >$ *in* $D$

*Sample a random mini − batch of tuples from* $D$

*Compute target* $Q$ *value with*

$$Y^{DDPG} = r_i + \gamma Q'\left(s_{i+1}, \mu'\left(s_{i+1};\ \theta^{\mu'}\right);\ \theta^{Q'}\right)$$

*Update critic by minimizing the loss*:

$$L = \frac{1}{N}\sum_i \left(y_i - Q(s_i, a_i; \theta^Q)\right)^2$$

*Update the actor using the sampled policy gradient*:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a; \theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s;\theta^\mu)|_{s_i}$$

*Update target netwok parameters*

**End**

# 4. Studies

In this chapter, the different studies that have been carried out in the project are presented. First, the parameters of the cell considered to make the simulation are described. Then, we study each DRL-based approach individually, looking for the optimum configuration and presenting the results. Finally, a comparison between the three mechanisms is done.

## 4.1. System architecture

As explained in 3.2, the scenario proposed consists of a deployment of NG-RAN comprised of a single cell with two different tenants, each of them provided with a RAN Slice. Table 2 presents the considered simulation parameters of the cell, which are common for both tenants.

| Parameter | Value |
|---|---|
| Number of cells | 1 |
| Number of tenants | 2 |
| Cell radius | 115m |
| Path loss and shadowing model | Not considered |
| Number of PRBs | 273 |
| PRB Bandwidth | 360 kHz |
| Channel bandwidth | 100 MHz |
| Average spectral efficiency | 5 b/s/Hz |
| Total capacity | $Ct = Nt * Seff * PRB_{BW} = 491{,}4\ Mb/s$ |

**Table 2. Cell configuration**

The deployment therefore assumes a gNB with a single cell configured with a channel of 100 MHz organized in 273 PRBs composed by 12 subcarriers with a separation $\Delta f$=30kHz, corresponding to one of the numerologies defined in Table 1.

For each tenant $k$, a SLA is established with the two control parameters explained in 3.2, SAGBR and MABR. Two different configurations of these control parameters have been defined to get a better understanding of the system's behaviour.

| | Configuration #0 | | Configuration #1 | |
|---|---|---|---|---|
| **Control parameters** | **RAN Slice ID=1** | **RAN Slice ID=2** | **RAN Slice ID=1** | **RAN Slice ID=2** |
| SAGBR | 60% | 40% | 40% | 60% |
| MABR | 80% | | 80% | |

**Table 3. SLA configurations**

It is also necessary to define the traffic offered to each tenant at each time step, as it will be the reference to adjust the capacity. We distinguish between two different loads:

- The offered load used for training the model. It is a large file that contains the offered load obtained every three minutes.
- The offered load used for the evaluation of the model. It is a pattern of the load for three days at three minute intervals. It has a total of 1440 steps and is expressed on a per-unit basis of the total capacity. Therefore, the maximum total reward that can be obtained in one evaluation is 1440. The pattern for each tenant is the following:

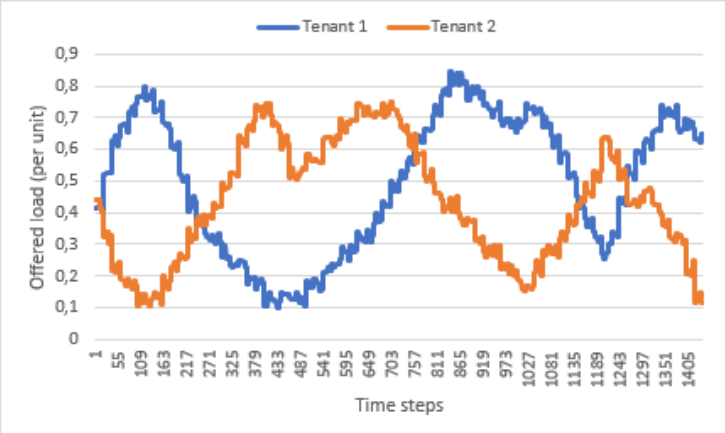


Figure 16. Offered load per tenant

## 4.2. Algorithm behaviour

The first study is to understand how the algorithm works and its evolution during the simulation. It is done with DQN approach, which is the first implemented. The simulation will be running during 50.000 steps of training and every 250 steps an evolution of the model will be performed. Then, all the information of the model is exported. We mainly focus on 5 different metrics, which are: evolution of the total reward obtained from each evaluation, capacity assigned to each tenant, occupation of each tenant, SLA satisfaction rate and utilization rate.

The set of hyperparameters are set to reference values:

| Hyperparameter | Value |
|---|---|
| Initial collect steps | 500 |
| Replay buffer max length | 200.000 |
| Exploration rate | 0,1 |
| Learning rate | 1e-4 |
| Batch size | 64 |
| Discount factor | 0,9 |
| Hidden layers (L1,L2) | (100,) |

Table 4. Hyperparameters configuration

### 4.2.1. Configuration #0

The first simulation is done with the Configuration #0 of Table 3, SAGBR = [60%, 40%]. We start studying the evolution of the total reward, which is the following:

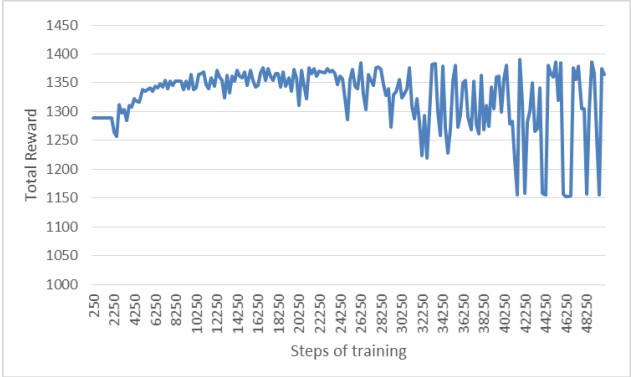


Figure 17. Total Reward evolution

At the beginning, we see that the total reward is increasing little by little while the policy is learning the good behaviour, but then it starts to diverge because of an overfitting. The first evaluations have a total reward of 1288, which corresponds to a behaviour of not doing any action different from $\underline{a_t}$= [0%, 0%].

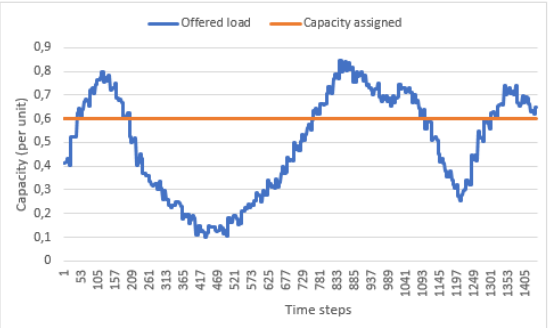


Figure 18. Capacity assigned to tenant 1

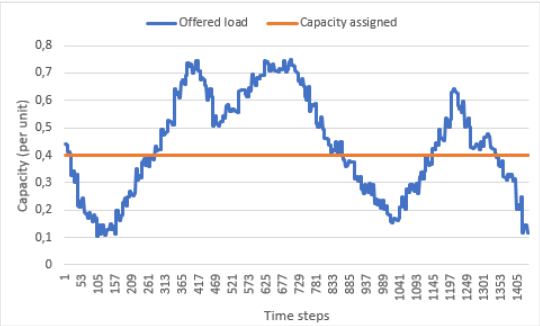


Figure 19. Capacity assigned to tenant 2

The resources are not efficiently distributed, sometimes more capacity is allocated to a tenant than is needed, leaving the other tenant with less capacity than the offered load. This is the main reason why we need the DRL agent. In Figure 20 are illustrated the two rates used to compute the reward, SLA rate is always 1 as the SAGBR is always meet but in the utilization rate there is a lot of room for improvement.

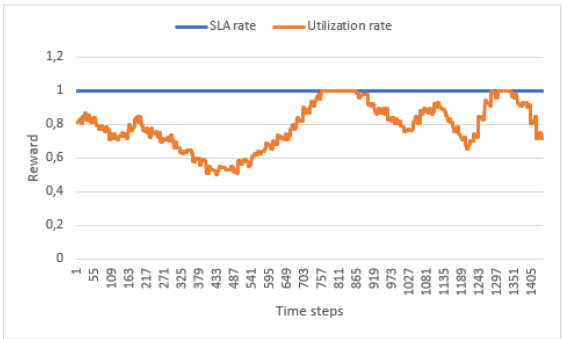


Figure 20. Utilization and SLA satisfaction rate

The idea is, then, to optimize the utilization of the resources but interfering as little as possible with the SAGBR compliance.

After some few evaluations without changes in the policy, the agent starts to learn and to modify the policy. The following images show the performance on some evaluations to see the evolution.

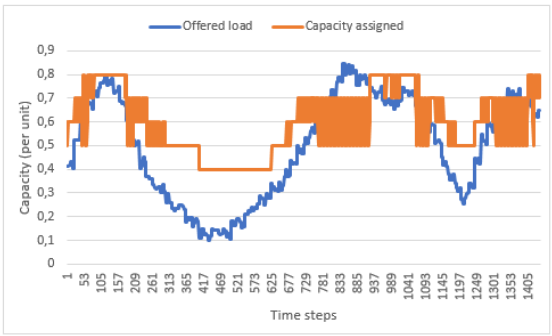


Figure 21. Capacity assigned to tenant 1. Evaluation 15

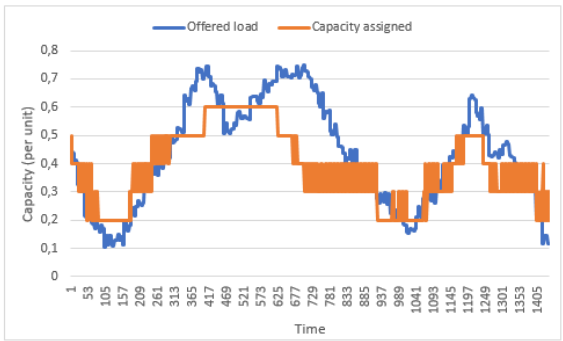


Figure 22. Capacity assigned to tenant 2. Evaluation 15

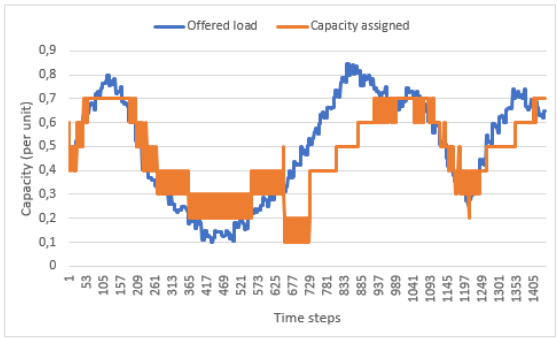


Figure 23. Capacity assigned to tenant 1. Evaluation 25

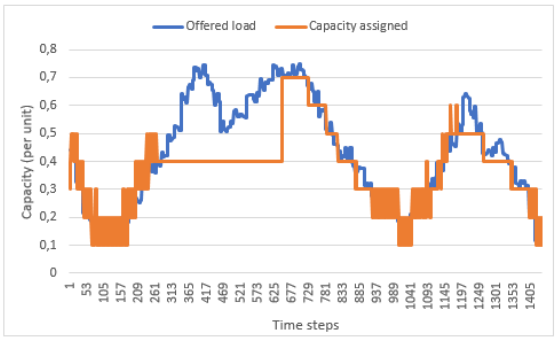


Figure 24. Capacity assigned to tenant 2. Evaluation 25

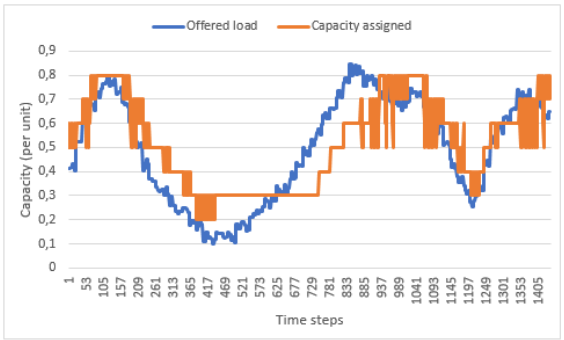


Figure 25. Capacity assigned to tenant 2. Evaluation 40

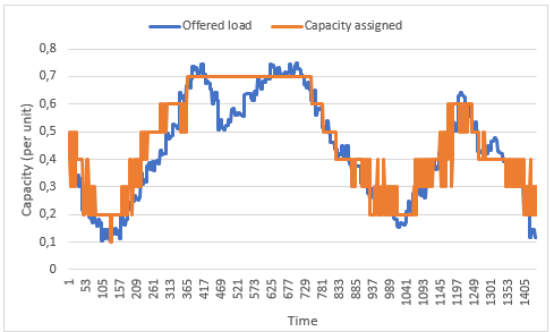


Figure 26. Capacity assigned to tenant 2. Evaluation 40

Little by little the capacity is allocated according to the offered traffic. The respective total rewards of these three evaluations are 1311, 1344 and 1366. After some steps more, the agent reaches a good policy:

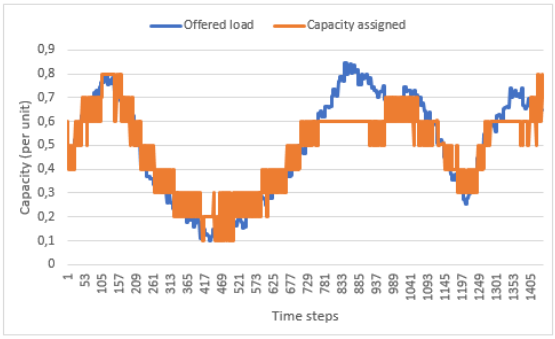


Figure 27. Capacity assigned to tenant 1. Optimal policy

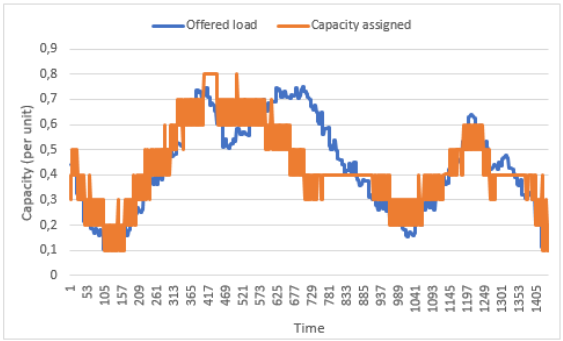


Figure 28. Capacity assigned to tenant 2. Optimal policy

The total reward obtained in this evaluation is 1378. It is not so much bigger than those obtained before, but the behaviour improves quite a lot. The reward gives information about the performance of the model, but it is necessary to look at the other metrics to know if the behaviour is the desired. We see that the capacity is assigned according to the offered load of both tenants. When the total traffic is lower than the total capacity available, the resources are efficiently distributed, allocating to each tenant what they need. However, in the middle of the simulation there are some steps where the total traffic is higher than the total capacity available (this situation occurs between the time steps 650-950 more or less, which we will call the "critic steps"), and the resource assignment follows the offered load of tenant 1. This happens because in the steps just before these critic steps, the capacity in the tenant 1 is below its SAGBR and of tenant 2 is above. Therefore, as the offered load of tenant 1 goes up, its allocated capacity will increase until the SAGBR established is fulfilled. To fully understand what it happens in those steps, the two previous figures are added together to show the total offered load versus the total capacity assigned.

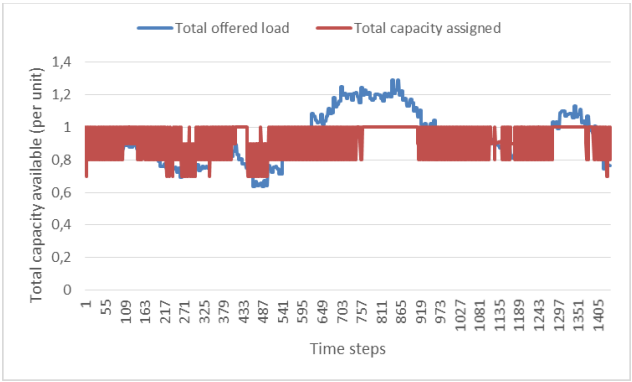


Figure 29. Total offered load vs total capacity assigned

In the figure above the y-axis represents the total capacity available expressed in a per unit basis, and is appreciated that in some steps the traffic exceeds the total capacity available requiring the 120% of it. It is in these steps where the SAGBR parameter gains importance and the behaviour of the model is as explained above. In this evaluation the performance is quite good, but it would be better if in all the steps that the traffic exceeds the total capacity, the allocated capacity would be the 100% of the total.

Finally, the SLA satisfaction of the two tenants together and the utilization rates are plotted to see the improvement respect Figure 20. As the two rates overlap, they are plotted individually. The SLA satisfaction rate (Figure 30) is 1 except in some steps that the throughput is below the SAGBR but the traffic is higher. The utilization rate (Figure 31) improves but still there are some steps that more capacity is assigned than needed. This happens because of a limitation of how we have defined the possible actions, as the allocation is in steps of 10%, it is difficult to adjust it perfectly. The MABR only defines the maximum capacity that can be allocated, so in this case it does not exceed the 80% of the total at any time.

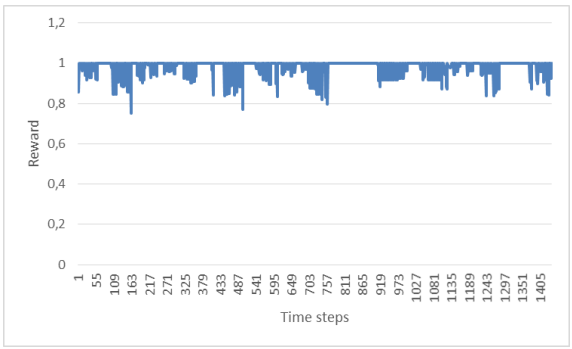


Figure 30. SLA satisfaction rate

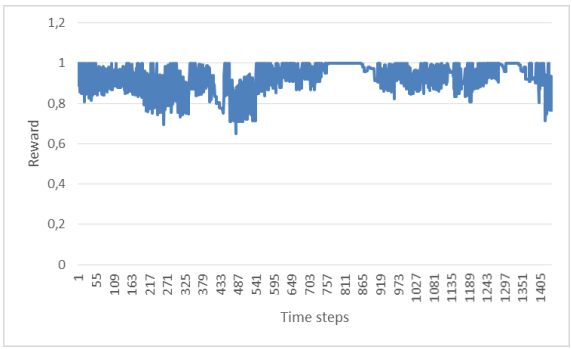


Figure 31. Utilization rate

### 4.2.2. Configuration #1

To further study the impact of the SAGBR on the model, we have done another simulation with the Configuration #1 of Table 3 (SAGBR = [40%, 60%]), which is the opposite of before. The values of the total reward will be more or less the same, we are interested in how the resources are now allocated. Therefore, we focus on the optimum behaviour of the model, to see what changes from one configuration to another.

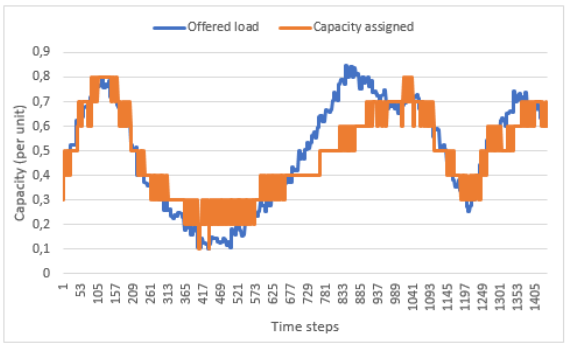


Figure 32. Capacity assigned to tenant 1. Optimal policy

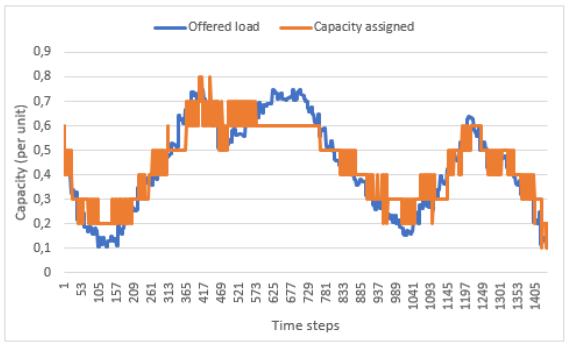


Figure 33. Capacity assigned to tenant 2. Optimal policy

Now, the roles of both tenants have changed. The main difference is appreciated in the critic steps. When both tenants require more capacity than available, tenant 2 receives the 60% and tenant 1 the 40%, and until the offered load of tenant 2 does not drop, tenant 1 will not receive more capacity. Thus, the SAGBR parameter tells the capacity that will be allocated to each tenant when the offered load of each exceeds their SAGBR established. We can say that in these situations the SAGBR with the biggest SAGBR is the dominant.

In the steps where only one tenant has an offered load higher than its SAGBR (e.g. at the beginning of the simulation, steps 80-200, where tenant 1 has a load around 80% of capacity and tenant 2 around 20%), more capacity will be allocated to the tenant that needs it as the other does not need to use its SAGBR.

Regarding the rates of the reward, illustrated below, the evolution is similar as before. The agent will always try to maximize both rates, but in some steps it will not be possible.

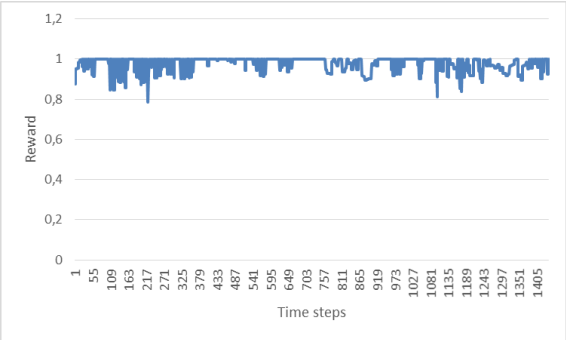


Figure 34. SLA satisfaction rate

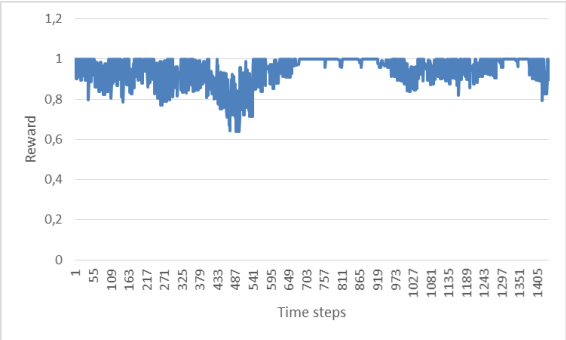


Figure 35. Utilization rate

We have seen how the algorithm works and the behaviour of the model. The agent needs of some steps of training to learn a good policy but then the training has to be stopped to avoid overfitting. In the following chapters, the three DRL-based approaches are studied in order to optimize their learning process and stop the training when the optimal policy is achieved.

## 4.3. DQN Agent

In this section the results of the first approach studied, DQN, are analysed. The objective is that the agent achieves a behaviour such that the capacity is assigned to each tenant according with the offered load of Figure 16. All the results below are done with the configuration #0 of Table 3.

### 4.3.1. Hyperparameters optimization

Once the agent is implemented, the first step to do is to tune the model hyperparameters to optimize the learning process. The study of each hyperparameter individually is presented in Annex 1, from which an initial value of each hyperparameter can be deducted. In order to obtain the best hyperparameter set configuration, we have implemented three different methodologies. In these methodologies, we only focus on the hyperparameters that have a bigger impact on the model performance, i.e. learning rate, batch size, discount factor, and hidden layers; the others remain the same for all the studies. The fixed hyperparameters with their values are described in the following table:

| Hyperparameter | Value |
|---|---|
| Maximum number of iterations | 100.000 |
| Evaluation interval | 250 |
| Initial collect steps | 5.000 |
| Replay buffer max length | 200.000 |
| Exploration rate | 0,1 |

**Table 5. Fixed hyperparameters**

At the beginning of the methodologies, all the hyperparameters we are going to optimize are initialized to the reference value, see Table 6, and then they are studied looking for the best value among a set of them. These values have been obtained from the study of Annex 1, which gives an initial approach to the optimization process.

| Hyperparameter | Value |
|---|---|
| Learning rate | 1e-4 |
| Batch size | 128 |
| Discount factor | 0,9 |
| Hidden layers (L1,L2) | (100,) |

**Table 6. Initial values of analysed hyperparameters**

Each methodology does the study in a different order, so the results can vary. It is also important to say that there is a factor of randomness, so it is possible that for the same configuration different results are obtained. The three methodologies are:

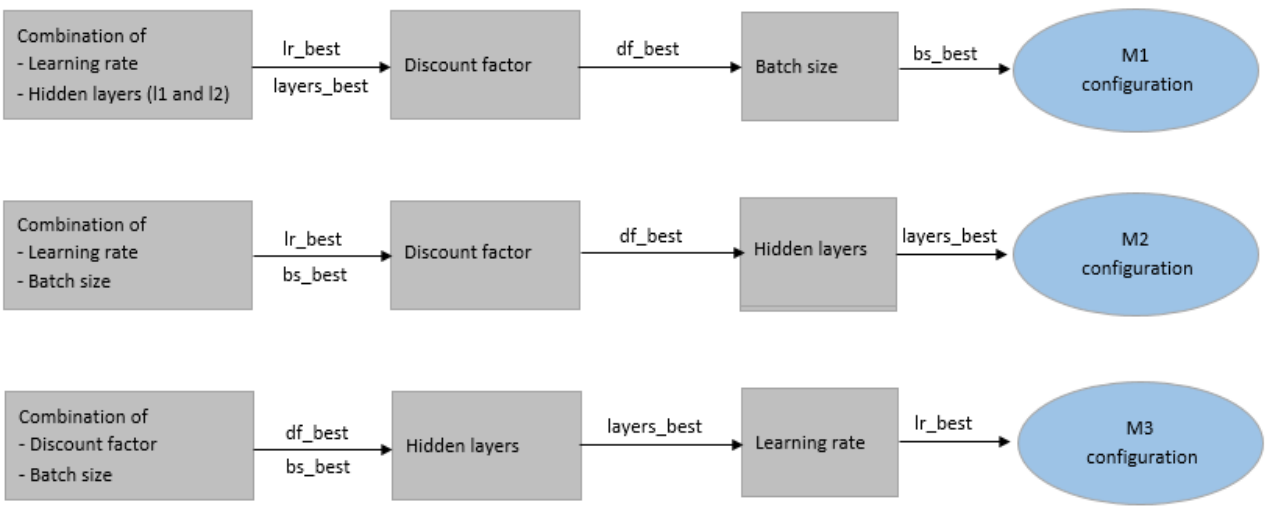


**Figure 36. Methodologies for DQN hyperparameters tuning**

To decide if one configuration is better than another, we look at the convergence of the algorithm. It is important to determine when the training can be finished to avoid overfitting.

As we have explained, the model will be training and every 250 steps an evaluation will be performed. The criteria to determine the convergence is the following: when all the evaluations in an interval of 7.500 steps of training have a total reward above 1375, and the variation in the total reward between the last evaluation and each of the others within the interval is less than 1%, the training will be over. The study of the convergence interval step is in Annex 2. If no convergence is reached, the process will stop after 100.000 iterations.

For each of the simulations, the following metrics are exported to get more information of each configuration: the average total reward of the evaluations, the deviation between them, and the simulation time (in seconds). Then, from the best configuration considered, all the information of the model simulation is exported (throughput, capacity assigned, reward evolution, SLA satisfaction rate, etc.).

The results obtained with each methodology are presented below. We only show the configurations that have achieved convergence before the end of the simulation. The first column of the tables tells in which step of the methodology the respective configuration is obtained, so the same configuration could be study in different steps.

| Step | Learning rate | Batch size | Discount factor | L1 | L2 | Reward | Dev. | Iterations | Time (s) |
|---|---|---|---|---|---|---|---|---|---|
| **1** | 0,0002 | 64 | 0,9 | 80 | 0 | 1366,371 | 16,06 | 36250 | 1169,30 |
| **3** | 0,0002 | 256 | 0,9 | 80 | 0 | 1357,382 | 29,68 | 72500 | 2592,30 |
| **M1** | 0,0002 | 64 | 0,9 | 80 | 0 | 1366,371 | 16,06 | 36250 | 1169,30 |

Table 7. DQN Configurations with Methodology 1

| Step | Learning rate | Batch size | Discount factor | L1 | L2 | Reward | Dev. | Iterations | Time (s) |
|---|---|---|---|---|---|---|---|---|---|
| **1** | 0,0002 | 256 | 0,9 | 100 | 0 | 1371,029 | 11,83 | 59000 | 2063,43 |
| **1** | 0,0001 | 256 | 0,9 | 100 | 0 | 1366,864 | 11,86 | 64000 | 2269,45 |
| **1** | 0,00009 | 128 | 0,9 | 100 | 0 | 1370,926 | 17,10 | 93500 | 3186,22 |
| **1** | 0,00009 | 256 | 0,9 | 100 | 0 | 1371,683 | 15,30 | 95000 | 3419,88 |
| **2** | 0,0002 | 256 | 0,8 | 100 | 0 | 1368,539 | 10,44 | 26000 | 946,51 |
| **M2** | 0,0002 | 256 | 0,8 | 100 | 0 | 1368,539 | 10,44 | 26000 | 946,51 |

Table 8. DQN Configurations with Methodology 2

| Step | Learning rate | Batch size | Discount factor | L1 | L2 | Reward | Dev. | Iterations | Time (s) |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 0,0001 | 256 | 0,9 | 100 | 0 | 1372,986 | 15,26 | 70750 | 2493,26 |
| **1** | 0,00008 | 256 | 0,9 | 100 | 0 | 1355,508 | 17,82 | 80750 | 2953,57 |
| **M3** | 0,0001 | 256 | 0,9 | 100 | 0 | 1372,986 | 15,26 | 70750 | 2493,26 |

**Table 9. DQN Configurations with Methodology 3**

The three methodologies output different configurations that converge before the learning process finishes. Looking at the results, we can see that there are some values for each hyperparameter that usually works well, differing from the reference value in some cases. The most obvious case is the batch size, it has been initialized to 128 but after the analysis, we see that 256 is the best value. Because of this, if the batch size is not studied at the first steps of the methodology, that methodology will obtain a few good configurations, which is what happens in methodology 1. Another hyperparameter that changes from the reference value is the learning rate, whose best value seems to be 2e-4 instead of 1e-4. These two hyperparameters are considered the most important, so when they are well configured the impact of the other two hyperparameters is less significant as long as they are in a good range. In the case of hidden layers it seems that the initial value (100,) is also the optimum although with the methodology 1 we have obtained a different value. Therefore, taking into account that it is preferable to optimize the learning rate and batch size values at the beginning, the best methodology is M2.

Regarding the performance of each configuration, the results obtained do not differ a lot from each other. The average of the total reward is around 1370 and the deviation is usually between 10 and 15.

Once we know that the best methodology is M2, we execute it 3 times to study the randomness factor and obtain a more confident result. The best configuration obtained in each of these simulations is shown in the next table.

| | Learning rate | Batch size | Discount factor | L1 | L2 | Reward | Dev. | Iterations | Time (s) |
|---|---|---|---|---|---|---|---|---|---|
| **1** | 0,0003 | 256 | 0,85 | 100 | 0 | 1365,40 | 11,45 | 64500 | 2222,53 |
| **2** | 0,0002 | 256 | 0,9 | 100 | 0 | 1360,54 | 14,06 | 23500 | 744,69 |
| **3** | 0,0003 | 256 | 0,85 | 100 | 0 | 1359,92 | 17,27 | 21250 | 754,75 |

**Table 10. Best Configurations for DQN Agent**

The three configurations do not match with any of the configurations obtained before, but they do no differ a lot. In addition, from these three configurations there are two that are equal, so we can say that the randomness factor affects a little but not too much when

looking for the best configuration. However, it can affect a bit more whether the configuration converges sooner or later, as the update is done choosing samples randomly and on this will depend how fast the good policy is learned.

Finally, we can say that an optimum configuration for the DQN Agent is:

| | Learning rate | Batch size | Discount factor | L1 | L2 |
|---|---|---|---|---|---|
| **Optimum** | 0,0003 | 256 | 0,85 | 100 | 0 |

**Table 11. Optimum Configuration for DQN Agent**

### 4.3.2. Soft update improvement

After the previous study, the soft update technique is added to try to improve the results. Thus, each time the target network is updated, it does it in the following way:

$$\theta^- \leftarrow \tau\theta + (1-\tau)\theta^- \quad \textbf{(50)}$$

Now, there is a new hyperparameter to optimize that is $\tau$. As we do not know if this parameter can influence the others, it is necessary to define a new methodology. It is based on the M2 of before, but in place of the batch size we put the soft update hyperparameter. The batch size is the only hyperparameter that we assume that 256 is the optimum value, and now it will not be studied. The hyperparameters of Table 5 remain the same. The methodology is then:

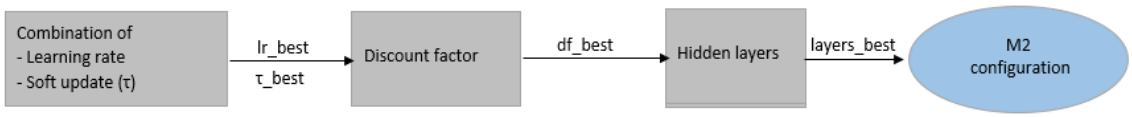


**Figure 37. Methodology with soft update**

As before, we only show the configurations that have achieved convergence before the end of the simulation.

| Step | Learning rate | Batch size | Discount factor | L1 | L2 | τ | Reward | Dev. | Iterations | Time (s) |
|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 0,0003 | 256 | 0,9 | 100 | 0 | 0,1 | 1365,89 | 15,24 | 36750 | 1245,96 |
| **1** | 0,0003 | 256 | 0,9 | 100 | 0 | 0,01 | 1371,61 | 6,66 | 20500 | 710,46 |
| **1** | 0,0003 | 256 | 0,9 | 100 | 0 | 0,001 | 1368,52 | 9,07 | 33250 | 1147,57 |
| **1** | 0,0002 | 256 | 0,9 | 100 | 0 | 0,1 | 1371,53 | 12,44 | 64000 | 2222,10 |
| **1** | 0,0002 | 256 | 0,9 | 100 | 0 | 0,001 | 1360,83 | 11,68 | 21000 | 737,82 |
| **1** | 0,0001 | 256 | 0,9 | 100 | 0 | 1 | 1364,84 | 18,36 | 47750 | 1667,91 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 0,0001 | 256 | 0,9 | 100 | 0 | 0,1 | 1355,44 | 17,57 | 37000 | 1309,15 |
| **1** | 0,0001 | 256 | 0,9 | 100 | 0 | 0,01 | 1371,06 | 14,89 | 60250 | 2137,20 |
| **1** | 0,00009 | 256 | 0,9 | 100 | 0 | 1 | 1356,40 | 19,33 | 47500 | 1687,35 |
| **1** | 0,00009 | 256 | 0,9 | 100 | 0 | 0,1 | 1336,89 | 25,19 | 33750 | 1204,72 |
| **1** | 0,00008 | 256 | 0,9 | 100 | 0 | 0,01 | 1356,23 | 18,19 | 42000 | 1535,29 |
| **1** | 0,00008 | 256 | 0,9 | 100 | 0 | 0,001 | 1355,93 | 14,92 | 58000 | 2127,49 |
| **2** | 0,0003 | 256 | 0,85 | 100 | 0 | 0,01 | 1362,39 | 13,41 | 14750 | 543,37 |
| **2** | 0,0003 | 256 | 0,9 | 100 | 0 | 0,01 | 1364,92 | 10,63 | 27000 | 999,59 |
| **3** | 0,0003 | 256 | 0,85 | 80 | 0 | 0,01 | 1360,16 | 20,71 | 37250 | 1376,37 |
| **3** | 0,0003 | 256 | 0,85 | 100 | 0 | 0,01 | 1362,15 | 10,49 | 20000 | 747,87 |
| **M** | 0,0003 | 256 | 0,85 | 100 | 0 | 0,01 | 1362,39 | 13,41 | 14750 | 543,37 |

**Table 12. DQN Configurations obtained with Methodology with soft update**

The improvement is evident, we have obtained so much configurations that converge and they need of less iterations than before. The fact of updating the target network very little instead of at once, makes that the agent chooses better actions to perform and improves the stability of the training. The good policy is learned faster.

To remove the randomness factor and obtain a confident configuration, we execute the methodology three times as in 4.2.1. The best configurations are:

| | Learning rate | Batch size | Discount factor | L1 | L2 | τ | Reward | Dev. | Iterations | Time (s) |
|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 0,0003 | 256 | 0,9 | 100 | 0 | 0,001 | 1370,32 | 7,12 | 18750 | 660,56 |
| **2** | 0,0002 | 256 | 0,9 | 100 | 0 | 0,001 | 1369,35 | 10,68 | 16750 | 585,77 |
| **3** | 0,0003 | 256 | 0,9 | 100 | 0 | 0,001 | 1361,25 | 15,64 | 24250 | 788,28 |

**Table 13. Best Configurations with DQN Agent using soft update**

We obtain two different configurations with different learning rate. To decide the best configuration, we study the policy obtained when finishing the training. We will use the

information exported from the first and second tries. Even so, first we plot the total reward evolution to see if there are big differences.

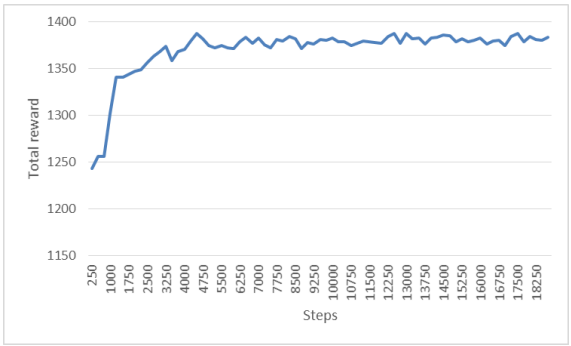


Figure 38. Total reward evolution of Config. 1

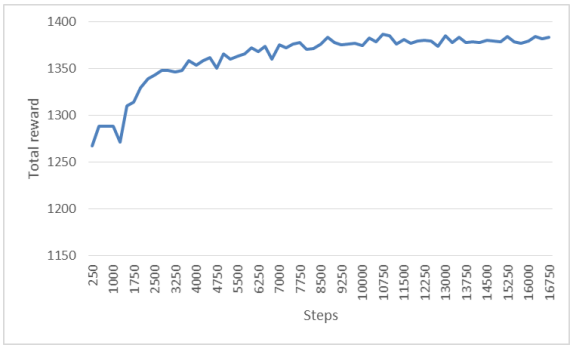


Figure 39. Total reward evolution of Config. 2

Both graphs are very similar, the learning is done a bit faster with the configuration 1 because of the bigger learning rate, but then it needs 2000 steps more of training to reach the convergence. It is necessary to look at the allocation of the capacity of the last evaluation to get more information about each process.

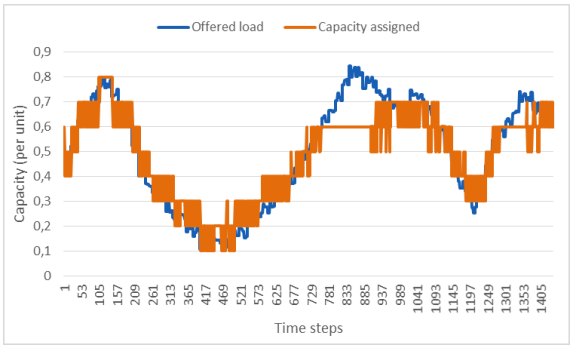


Figure 40. Behaviour of the policy obtained with Config. 1 in Tenant 1

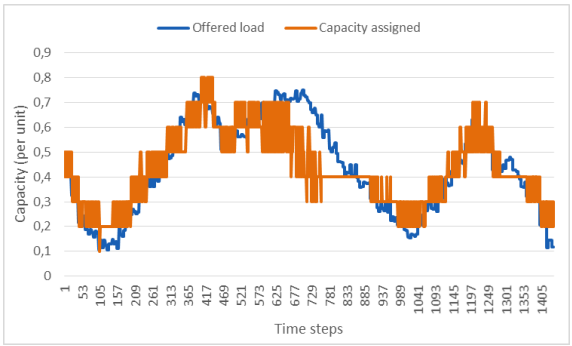


Figure 41. Behaviour of the policy obtained with Config. 1 in Tenant2

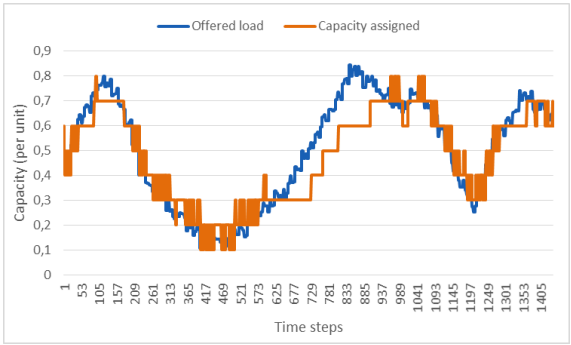


Figure 42. Behaviour of the policy obtained with Config. 2 in Tenant 1

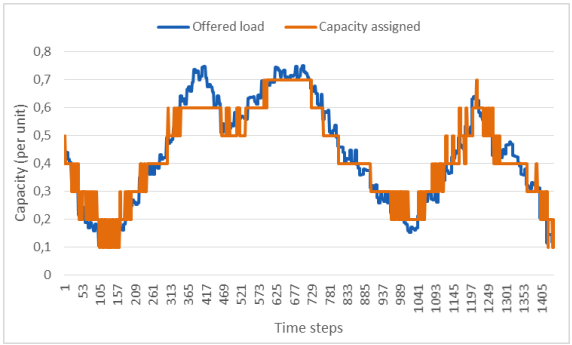


Figure 43. Behaviour of the policy obtained with Config. 2 in Tenant 2

When the offered load is below the total capacity, the behaviour is more or less the same, but when it exceeds the total capacity available, the policies obtained are very different from each other. With Config. 1 the capacity is allocated according to the SAGBR, so in the critic steps, the tenant 1 receives more resources until it arrives at the 60% of the capacity. On the other hand, with Config. 2 the capacity is adjusted really well to the tenant 2 but it does not respect the SLA established, so the policy is not the desired. In both cases the total reward of the last evaluation is high, Config. 1 obtains 1385 and Config 2. 1388. Therefore, obtaining a high reward does not mean to arrive at the desired policy. To understand why Config. 2 obtains a bigger reward although the policy is not what we were looking for, we analyse the two reward rates.

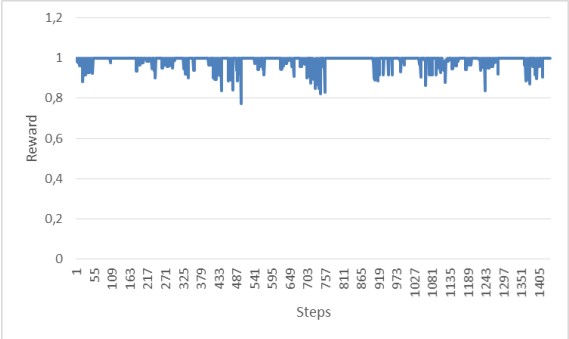


Figure 44. SLA satisfaction rate with Config. 1

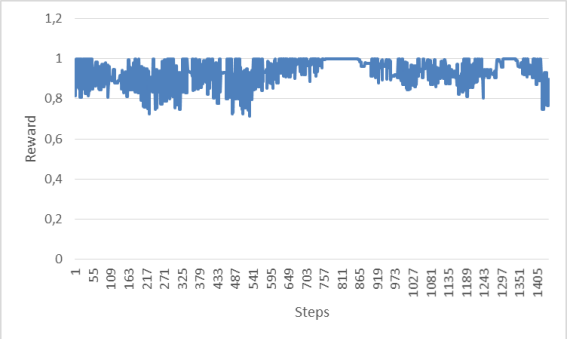


Figure 45. Utilization rate with Config. 1

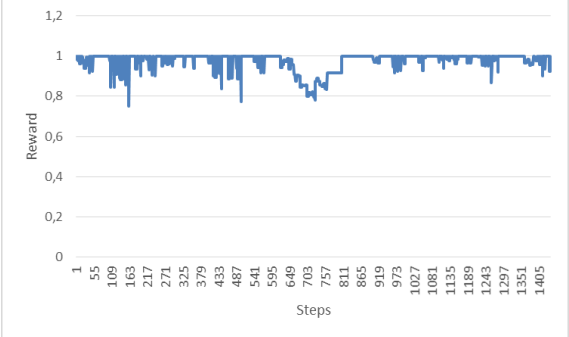


Figure 46. SLA satisfaction rate with Config. 2

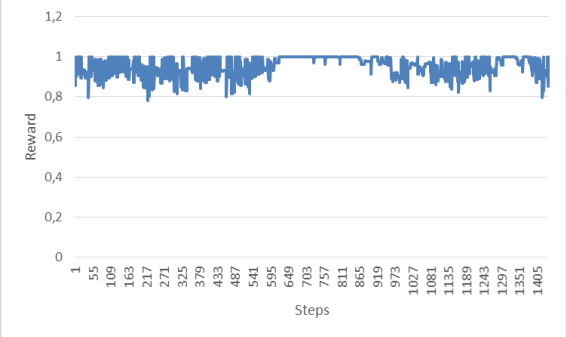


Figure 47. Utilization rate with Config. 2

In the figures above we can clearly see what we have explained. With Config. 2 the SLA is not respected when the offered load exceeds the total capacity available, so in these steps the SLA rate is being affected. However, the utilization rate has better performance than with Config. 1. The reason is that the capacity is assigned below the traffic, so the throughput is limited but there is not assigned capacity that is not being used. With Config. 1 the SLA rate is almost always 1 except on some steps.

These figures are transformed into numbers to fully understand it. In the next table is presented the total reward, the total reward of each rate (SLA satisfaction of both tenants and utilization), and the amount of time that SAGBR is fulfilled in both tenants, no matter if by a big difference or just a little.

| | Total Reward | SLA rate | Utilization rate | SAGBR compliance |
|---|---|---|---|---|
| **Config. 1** | 1385,01 | 1424,52 | 1345,51 | 76,9% |
| **Config. 2** | 1388,59 | 1407,16 | 1370,02 | 69,72% |

**Table 14. Total reward decomposition**

From the table we see that the SLA is always bigger than the utilization rate by how we have defined the actions, the way to increase the utilization rate would be allocating less capacity than the offered load as Config. 2 does, but it will probably affect the SLA. The SAGBR is satisfied more time with Config. 1 than with 2 as expected.

To sum up, with the two configurations we obtain a policy with a high total reward, so the agent learns how to solve the problem well. However, depending on whether it learns a policy respecting the SAGBR or prioritizing the utilization rate, the final policy will be the desired or not.

Therefore, as we want that the SAGBR is fulfilled, we can say that the optimum configuration for the DQN Agent with soft update is:

| | Learning rate | Batch size | Discount factor | L1 | L2 | τ |
|---|---|---|---|---|---|---|
| **Optimum** | 0,0003 | 256 | 0,9 | 100 | 0 | 0,001 |

**Table 15. Optimum configuration for DQN Agent with soft update**

### 4.3.3. Final results

In this chapter is compared the performance of the DQN algorithm with soft update and without. To do it, a new simulation for each case is executed, but adding new conditions to the criteria that decides the convergence. To determine that the algorithm has converged it is necessary that the policy of the last evaluation also meets the SAGBR of tenant 1, to avoid obtaining a policy with a big reward but that it is not the desired.

Several simulations have been executed with this new criteria, but the results below show the best performance that we have obtained with each of these configurations.

| Approach | Learning rate | Batch size | Discount factor | L1 | L2 | τ |
|---|---|---|---|---|---|---|
| **DQN (full update)** | 0,0003 | 256 | 0,85 | 100 | 0 | 1 |
| **DQN (soft update)** | 0,0003 | 256 | 0,9 | 100 | 0 | 0,001 |

**Table 16. DQN Optimum configurations**

The two configurations are almost equal, the only hyperparameter that differs apart from the soft update hyperparameters is the discount factor. The comparison is presented below. The simulation with DQN with full update has converged in 27.750 steps and the simulation with soft update in 17.000 steps.

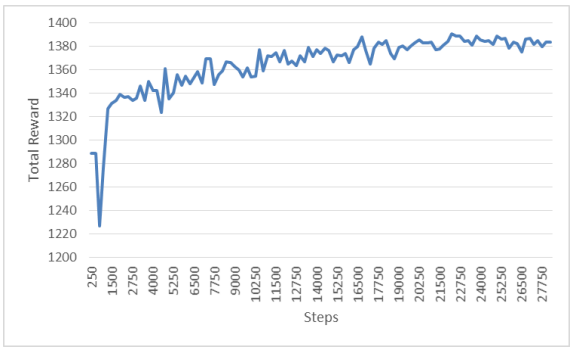


Figure 48. Total reward evolution of DQN with full update

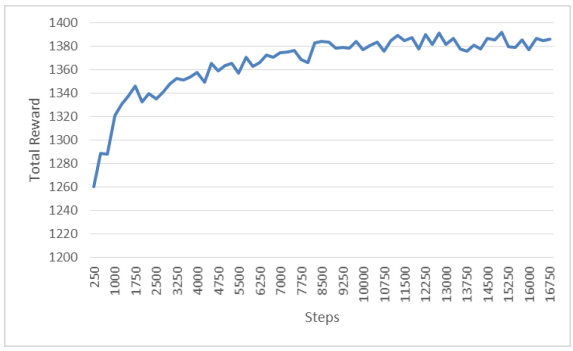


Figure 49. Total reward evolution of DQN with soft update

The implementation of the soft update technique improves the performance of the algorithm, the model learns how to solve correctly the problem faster than with full update. Now, we focus on the behavior of the policy obtained in each case, which should be similar as we have ensured the SAGBR compliance of tenant 1.

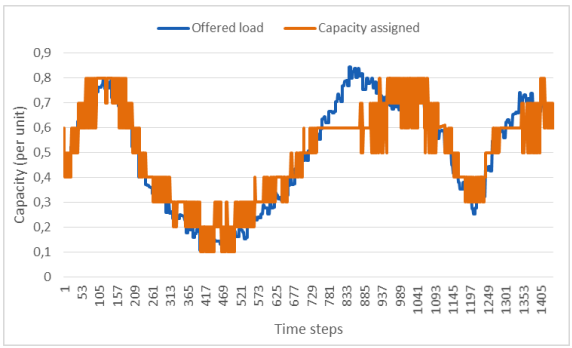


Figure 50. Policy behaviour of tenant 1. DQN with full update

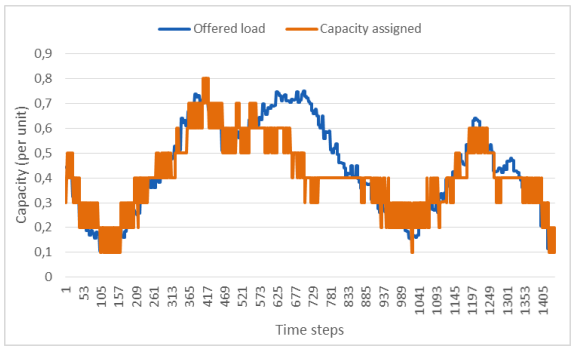


Figure 51. Policy behaviour of tenant 2. DQN with full update

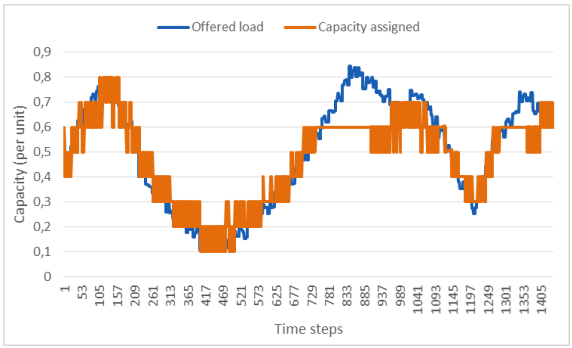


Figure 52. Policy behaviour of tenant 1. DQN with soft update

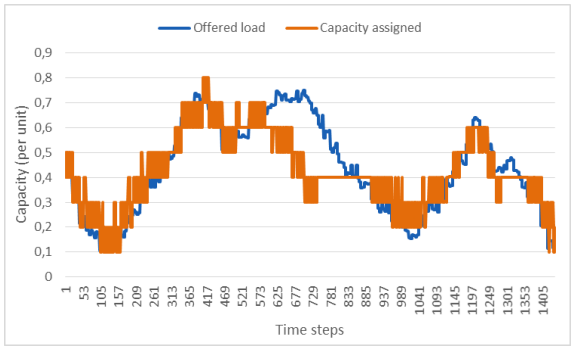


Figure 53. Policy behaviour of tenant 2. DQN with soft update

As we expected, the two policies behave similarly. When the offered load is lower than the total capacity available, the resources are allocated to the tenant that needs them, and when the simulation enters on the critic steps the model respects the SAGBR established.

Therefore, the process has succeeded in both cases as we have got the desired policy. However, there are few differences between the two policies and to get more information about them we look at the two reward rates.

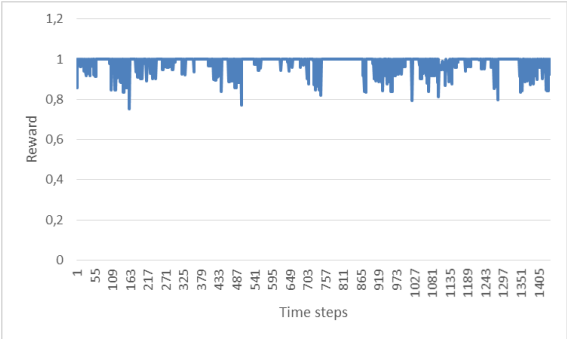


Figure 54. SLA satisfaction rate. DQN with full update

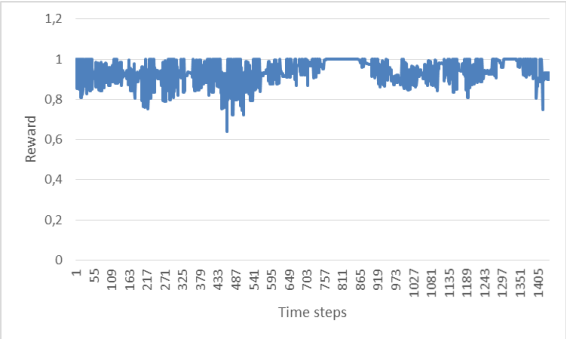


Figure 55. Utilization rate. DQN with full update

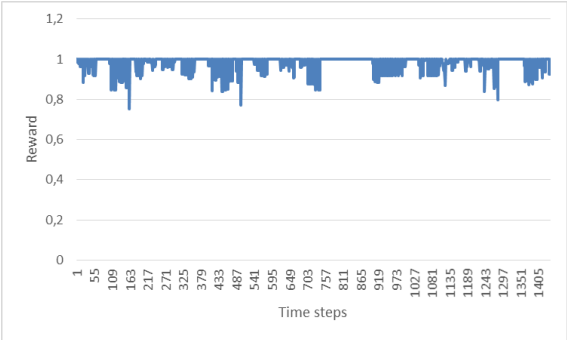


Figure 56. SLA satisfaction rate. DQN with soft update

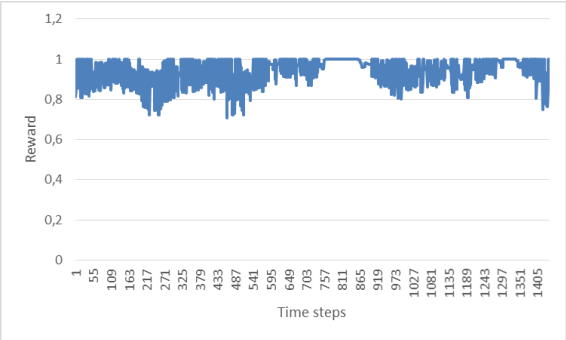


Figure 57. Utilization rate. DQN with full update

From the figures, it is very difficult to draw a conclusion as they are almost equal. It seems that the SLA satisfaction rate with soft update is a bit better, but by a negligible difference. The graphs are transformed into numbers to have something numerical that allows us to compare the two policies objectively.

In the following table different metrics are presented. It is worth mentioning the meaning of some of them.

- **SLA rate:** it is the accumulative reward of the SLA satisfaction rate of both tenants.
- **Utilization rate:** it is the accumulative reward of the utilization rate.
- **Total reward:** it is the average between the SLA rate and the utilization rate.
- **SAGBR compliance:** it computes the amount of time that the SAGBR is fulfilled in both tenants.
- **Capacity allocated:** it computes the amount of time that we are assigning either equal or more capacity than required. It distinguishes between two cases:
    - The capacity allocated should be the 100% of the capacity available when the offered load exceeds the total capacity available.
    - The total capacity allocated is either equal or bigger than the total offered load when this is lower than the total capacity available.

| Approach | Total Reward | SLA rate | Utilization rate | SAGBR compliance | Capacity allocated | Iterations | Time |
|---|---|---|---|---|---|---|---|
| **DQN (full update)** | 1383,91 | 1414,01 | 1353,82 | 71,8% | 67,8% | 27750 | 989,23 |
| **DQN (soft update)** | 1386,14 | 1417,69 | 1354,59 | 74,1% | 65% | 17000 | 592,46 |

**Table 17. Numerical metrics of the policy obtained**

In the table, we can appreciate the few differences between the two algorithms. The soft update simulation has finished with a slightly better policy than the full update. Both utilization and SLA satisfaction rates have bigger rewards with the soft update algorithm. As in 4.3.2, the SLA rate has a reward quite bigger than the utilization rate; if we have defined the action steps smaller, the utilization rate would be bigger. The SAGBR compliance is also favourable to the soft update, with three out of four steps meeting the SLA of both tenants. The only metric that is favourable to the full update algorithm is the capacity allocated, as there are fewer steps with the traffic above the capacity but below 1. Utilization rate and capacity allocated could be inversely proportional in some steps, as the agent has to decide if it allocates more capacity than needed affecting the utilization rate, or it gives less capacity than needed affecting the capacity allocated. Thus, allocating always more capacity than required could lead to a problem of over-provisioning, so the capacity should be adjusted efficiently. As the time the simulation is running is proportional to the number of training steps, with full update the algorithm lasts more than with soft update.

To sum up, implementing the soft update technique when updating the target network improves the stability of the training, and in turn the performance of the algorithm. The most important improvement comes from the convergence iterations, as the algorithm needs considerably fewer steps of training to achieve the same policy, even better.

## 4.4. DDQN Agent

DDQN is the second approach studied in this thesis. The objective is the same as with DQN, to adjust the capacity according to the offered load respecting the SLA established. First, it will be studied with the full update, and then the soft update technique will be also implemented to see the impact on this approach.

### 4.4.1. Hyperparameters optimization

It is necessary to also tune the hyperparameters configuration to optimize the learning process. As DDQN is an evolution of DQN, the starting point will be the same as with DQN (Table 6), and we only study the methodology that performs best, which is the second.


Figure 58. DDQN Methodology with full update

The criteria to decide the convergence has not changed, the interval is still 7.500 steps and if the model does not converge the simulation will stop after 100.000 iterations. The configurations that have achieved convergence before ending the simulation are:

| Step | Learning rate | Batch size | Discount factor | L1 | L2 | Reward | Dev. | Iterations | Time (s) |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0,0002 | 256 | 0,9 | 100 | 0 | 1363,40 | 17,12 | 35500 | 1238,26 |
| 1 | 0,0001 | 128 | 0,9 | 100 | 0 | 1364,71 | 18,07 | 60250 | 1995,83 |
| 1 | 0,00009 | 128 | 0,9 | 100 | 0 | 1356,14 | 19,31 | 64750 | 2106,6 |
| 2 | 0,0002 | 256 | 0,85 | 100 | 0 | 1364,11 | 15,00 | 27500 | 946,90 |
| 2 | 0,0002 | 256 | 0,85 | 90 | 0 | 1361,55 | 21,25 | 31250 | 1002,44 |
| 3 | 0,0002 | 256 | 0,85 | 100 | 0 | 1368,73 | 17,33 | 40750 | 1318,78 |
| M | 0,0002 | 256 | 0,85 | 100 | 0 | 1364,11 | 15,00 | 27500 | 946,90 |

Table 18. DDQN Configurations with methodology with full update

The results obtained are similar to the DQN results, in fact, the configuration is the same except for the learning rate, which now decreases from 3e-4 to 2e-4. The number that the algorithm needs to converge is also more or less similar, which does not meet the expectations, as we expected an improvement with DDQN.

### 4.4.2. Soft update improvement

In order to try to improve the performance and to see the enhancement regards DQN, the soft update technique is implemented to update the target network at each step. Instead of copying the weights of the DDQN network exactly, now the weights are copied following the next equation:

$$\theta^- \leftarrow \tau\theta + (1-\tau)\theta^- \quad \textbf{(51)}$$

In order to deepen on its impact, two more methodologies have been created in addition to the one used with DQN. The three methodologies used are:

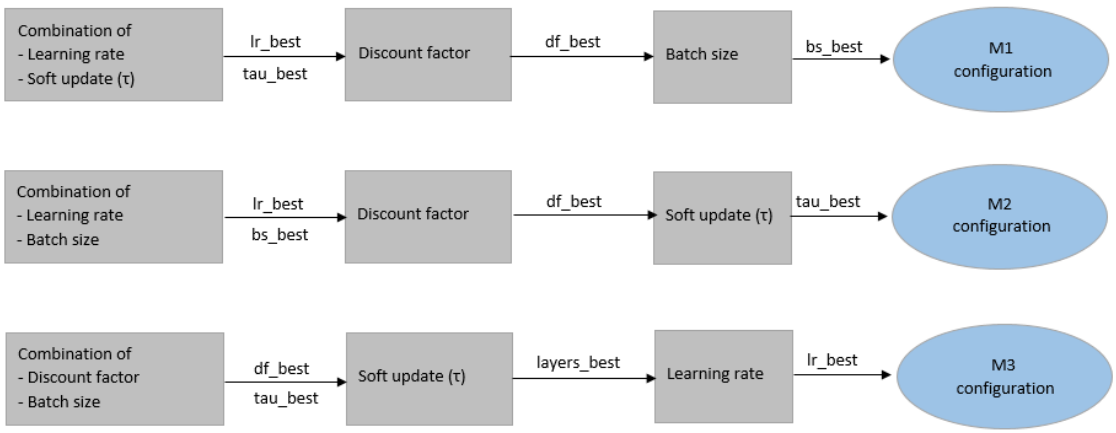


Figure 59. DDQN Methodologies with soft update

Now, the hidden layers do not enter on the study and they are set to (100,), which seems to be a good value for them. In their place, we put the soft update parameter. The results are presented below:

| Step | Learning rate | Batch size | Discount factor | L1 | L2 | τ | Reward | Dev. | Iterations | Time (s) |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0,0003 | 256 | 0,9 | 100 | 0 | 0,001 | 1372,59 | 10,56 | 32000 | 1113,53 |
| 1 | 0,0002 | 256 | 0,9 | 100 | 0 | 0,001 | 1368,57 | 11,14 | 27750 | 975,63 |
| 1 | 0,0001 | 256 | 0,9 | 100 | 0 | 0,1 | 1365,43 | 16,16 | 48750 | 1721,19 |
| 1 | 0,0001 | 256 | 0,9 | 100 | 0 | 0,01 | 1371,37 | 10,23 | 94500 | 3338,43 |
| 1 | 0,00009 | 256 | 0,9 | 100 | 0 | 0,1 | 1357,59 | 15,77 | 64500 | 2298,21 |
| 1 | 0,00009 | 256 | 0,9 | 100 | 0 | 0,001 | 1362,76 | 14,50 | 46750 | 1683,50 |
| 1 | 0,00008 | 256 | 0,9 | 100 | 0 | 0,001 | 1365,45 | 16,90 | 90500 | 3292,82 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **2** | 0,0002 | 256 | 0,9 | 100 | 0 | 0,001 | 1371,3 | 9,28 | 32250 | 1184,49 |
| **M** | 0,0002 | 256 | 0,9 | 100 | 0 | 0,001 | 1368,57 | 11,14 | 27750 | 975,63 |

Table 19. DDQN Configurations with Methodology 1 with soft update

| Step | Learning rate | Batch size | Discount factor | L1 | L2 | τ | Reward | Dev. | Iterations | Time (s) |
|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 0,0003 | 128 | 0,9 | 100 | 0 | 0,01 | 1372,60 | 6,59 | 57000 | 1884,71 |
| **1** | 0,0003 | 256 | 0,9 | 100 | 0 | 0,01 | 1368,19 | 10,11 | 33750 | 1183,46 |
| **1** | 0,0002 | 128 | 0,9 | 100 | 0 | 0,01 | 1344,14 | 26,34 | 99750 | 3292,63 |
| **1** | 0,0002 | 256 | 0,9 | 100 | 0 | 0,01 | 1372,55 | 8,65 | 23000 | 820,07 |
| **1** | 0,0001 | 256 | 0,9 | 100 | 0 | 0,01 | 1348,50 | 14,00 | 25000 | 840,89 |
| **1** | 0,00009 | 256 | 0,9 | 100 | 0 | 0,01 | 1360,28 | 16,21 | 41500 | 1490,49 |
| **1** | 0,00008 | 256 | 0,9 | 100 | 0 | 0,01 | 1353,65 | 22,94 | 88750 | 3232,42 |
| **2** | 0,0002 | 256 | 0,85 | 100 | 0 | 0,01 | 1364,45 | 14,22 | 34500 | 1248,20 |
| **3** | 0,0002 | 256 | 0,9 | 100 | 0 | 0,1 | 1367,77 | 11,39 | 42500 | 1544,21 |
| **3** | 0,0002 | 256 | 0,9 | 100 | 0 | 0,01 | 1367,56 | 12,34 | 32750 | 1193,29 |
| **3** | 0,0002 | 256 | 0,9 | 100 | 0 | 0,001 | 1365,16 | 12,48 | 20500 | 711,29 |
| **M** | 0,0002 | 256 | 0,9 | 100 | 0 | 0,001 | 1365,16 | 12,48 | 20500 | 711,29 |

Table 20. DDQN Configurations with Methodology 2 with soft update

| Step | Learning rate | Batch size | Discount factor | L1 | L2 | τ | Reward | Dev. | Iterations | Time (s) |
|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 0,0001 | 256 | 0,8 | 100 | 0 | 0,01 | 1365,22 | 16,33 | 32000 | 986,42 |
| **1** | 0,0001 | 256 | 0,9 | 100 | 0 | 0,01 | 1359,36 | 16,31 | 39000 | 1218,94 |
| **1** | 0,0001 | 256 | 0,8 | 100 | 0 | 0,1 | 1368,9 | 21,07 | 84500 | 2646,96 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 0,0001 | 256 | 0,8 | 100 | 0 | 0,001 | 1367,51 | 13,46 | 53000 | 1839,01 |
| **1** | 0,00009 | 256 | 0,8 | 100 | 0 | 0,01 | 1358,65 | 43,24 | 73500 | 2369,08 |
| **M** | 0,0001 | 256 | 0,8 | 100 | 0 | 0,01 | 1365,22 | 16,33 | 32000 | 986,42 |

**Table 21. DDQN Configurations with Methodology 3 with soft update**

Looking at the results obtained, there is not a significant improvement with the soft update technique such as there is with DQN. There is not a big difference between methodologies, as they output more or less the same number of configurations, but it seems that methodology 2 is still the best. The best configuration from the tables above, which is the repeated configuration for both methodologies 1 and 2, is:

| | Learning rate | Batch size | Discount factor | L1 | L2 | τ |
|---|---|---|---|---|---|---|
| **Optimum** | 0,0002 | 256 | 0,9 | 100 | 0 | 0,001 |

**Table 22. Optimum configuration from methodologies**

However, the reason of not improving so much the results could be because we are assuming that (100,) it is a good configuration for the hidden layers, but it is possible that DDQN needs another value. To check it, we take the optimum configuration from the tables above and we analyse the behaviour for another layer values. The results obtained are the following:

| Step | Learning rate | Batch size | Discount factor | L1 | L2 | τ | Reward | Dev. | Iterations | Time (s) |
|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 0,0002 | 256 | 0,9 | 100 | 0 | 0,001 | 1369,901 | 7,908472 | 25500 | 854,6207 |
| **1** | 0,0002 | 256 | 0,9 | 200 | 0 | 0,001 | 1375,329 | 8,353404 | 35000 | 1112,838 |
| **1** | 0,0002 | 256 | 0,9 | 300 | 0 | 0,001 | 1369,352 | 7,700596 | 17000 | 554,7427 |
| **2** | 0,0002 | 256 | 0,9 | 200 | 0 | 0,001 | 1363,901 | 10,28879 | 12750 | 440,7607 |
| **3** | 0,0002 | 256 | 0,9 | 100 | 0 | 0,001 | 1375,144 | 6,558289 | 22250 | 1021,076 |
| **3** | 0,0002 | 256 | 0,9 | 200 | 0 | 0,001 | 1370,411 | 5,618274 | 15000 | 486,6663 |
| **M** | 0,0002 | 256 | 0,9 | 200 | 0 | 0,001 | 1363,901 | 10,28879 | 12750 | 440,7607 |

**Table 23. DDQN Configurations with other layer values**

Changing the hidden layers hyperparameter the results has improved a lot. The assumption that (100,) was a good value was wrong, DDQN needs more nodes to solve quickly the problem. Specifically, the optimum configuration is with (200,), where the model converges in only 12.750 steps. Now, the results obtained with this configuration are analysed.

First, we plot the total reward evolution, which is illustrated in the figure below. The learning has been done really quickly, achieving high rewards with few steps of training.

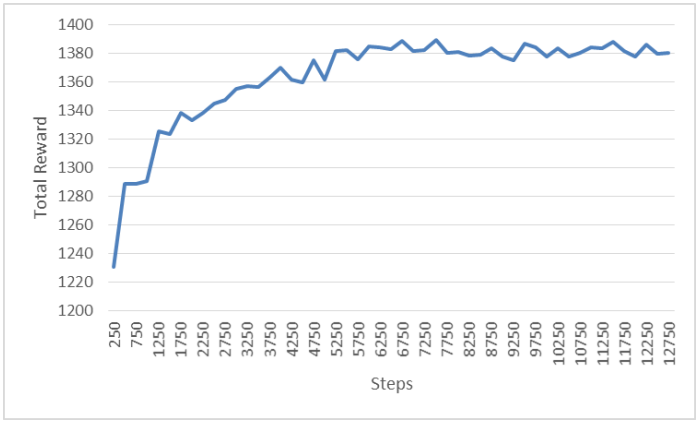


Figure 60. Total Reward evolution

After that, the policy obtained from this simulation is analysed.

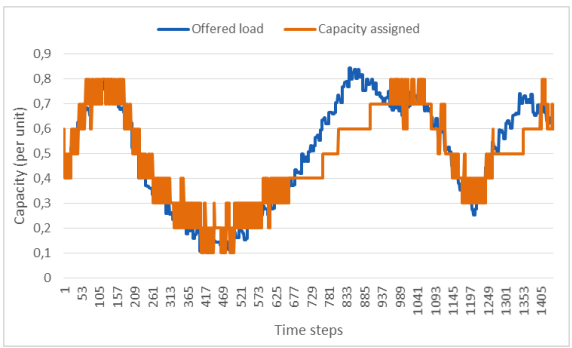


Figure 61. Policy behaviour of tenant 1

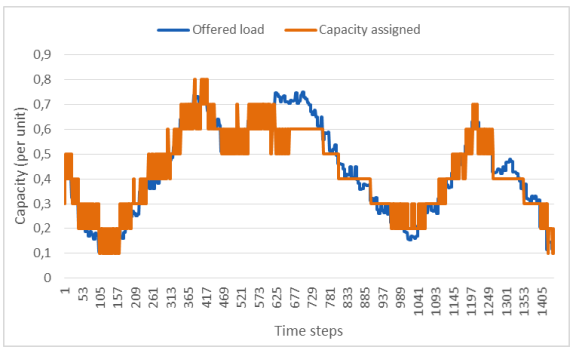


Figure 62. Policy behaviour of tenant 2

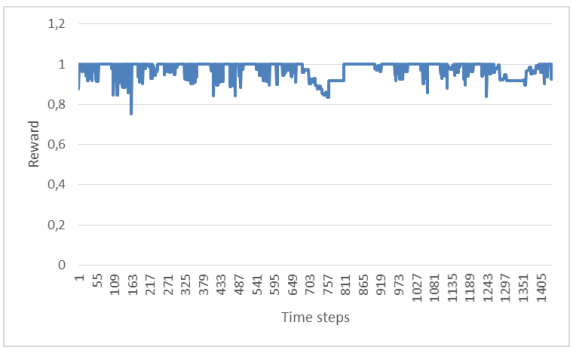


Figure 63. SLA satisfaction rate

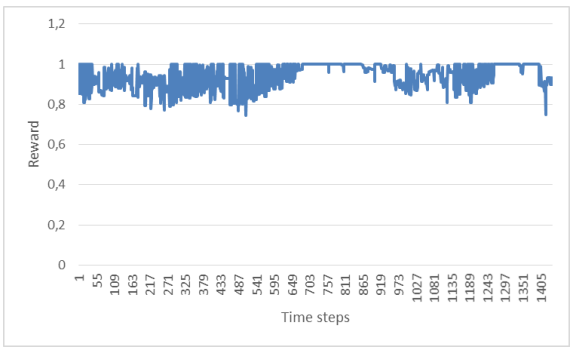


Figure 64. Utilization rate

The last evaluation obtains a reward of 1380,25. The policy obtained is not the desired, as in the critic steps the SAGBR of tenant 1 is not meet while the tenant 2 is receiving more capacity than its SAGBR. However, although in those steps the SAGBR is not accomplished, the utilization rate achieves a good behaviour as the assigned capacity is below the offered load.

Finally, we can conclude that an optimum configuration for the DDQN algorithm with soft update is:

| | Learning rate | Batch size | Discount factor | L1 | L2 | τ |
|---|---|---|---|---|---|---|
| **Optimum** | 0,0002 | 256 | 0,9 | 200 | 0 | 0,001 |

Table 24. DDQN Optimum configuration with soft update

### 4.4.3. Final results

In this chapter is compared the performance of DDQN with soft update and without. A new simulation is done for each case, but the criteria to determine the convergence is not the same. To avoid what happened in the last chapter, where the total reward was big but the policy was not the desired, when deciding whether the algorithm has converged or not the policy of the last evaluation has to meet the SAGBR of tenant 1.

Several simulations have been executed with this new criteria, but the results below show the best performance that we have obtained with each of these configurations.

After the configuration obtained from the previous chapter, the hidden layers hyperparameter of the configuration obtained with full update is changed to (200,), so the two optimum configurations for DDQN are:

| Approach | Learning rate | Batch size | Discount factor | L1 | L2 | τ |
|---|---|---|---|---|---|---|
| **DDQN (full update)** | 0,0002 | 256 | 0,85 | 200 | 0 | 1 |
| **DDQN (soft update)** | 0,0002 | 256 | 0,9 | 200 | 0 | 0,001 |

Table 25. DDQN Optimum configurations

The total reward evolution of each case is presented in the figures below. The simulation done with full update lasts 23.750 iterations to converge, and with soft update it converges in 12.500. The two configurations learn quite quickly how to solve the problem efficiently achieving high rewards. On the one hand, with full update there is a bit of oscillation while learning the good behaviour, as the evaluations not always improve the previous one. On the other hand, with soft update the evaluations usually improve the performance of the previous one until the model enters on the convergence interval. In both cases, the total rewards obtained on the convergence interval are above 1380.

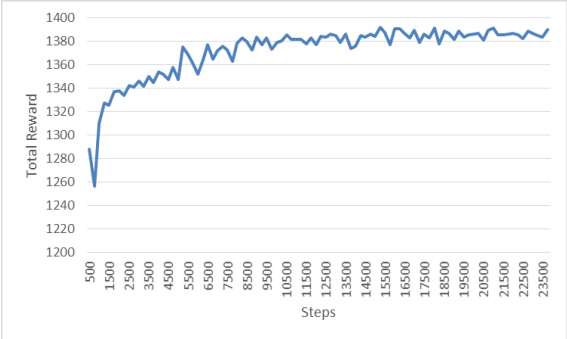


Figure 65. Total reward evolution of DDQN with full update

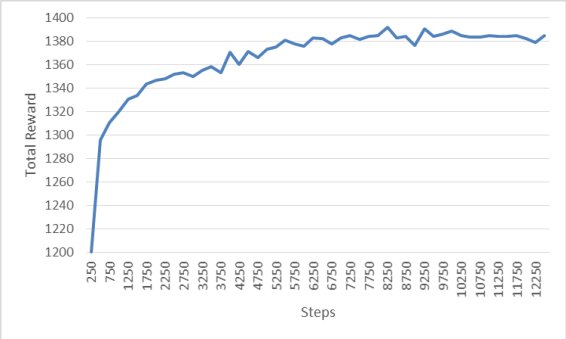


Figure 66. Total reward evolution of DDQN with soft update

After studying the learning process, it is necessary to study the policy obtained with each configuration. As we add the condition of fulfilling the SLA established in the critic steps, the two policies should be optimum.

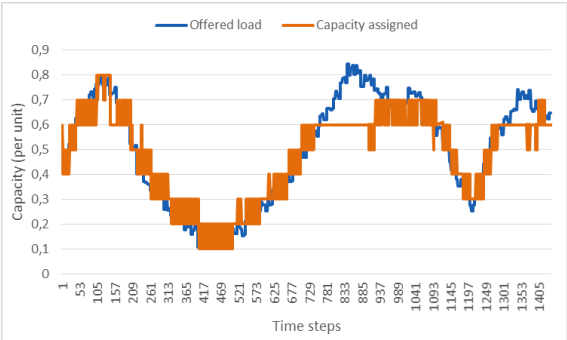


Figure 67. Policy behaviour of tenant 1. DDQN with full update

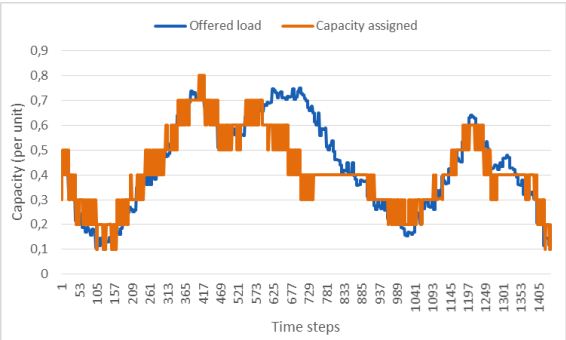


Figure 68. Policy behaviour of tenant 2. DDQN with full update

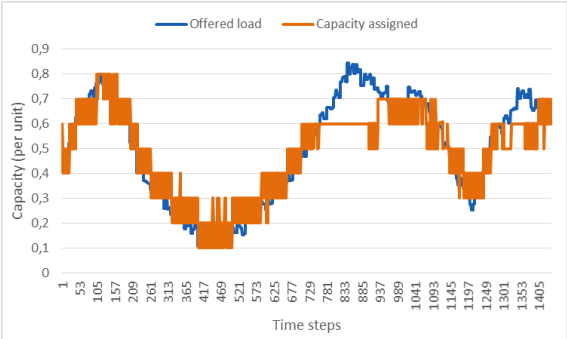


Figure 69. Policy behaviour of tenant 1. DDQN with soft update

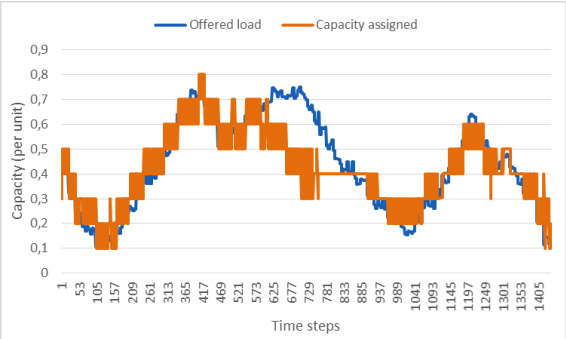


Figure 70. Policy behaviour of tenant 2. DDQN with soft update

The policies obtained are almost equal, both of them do an efficient allocation of the resources and when there could be some problems of capacity the SAGBR is met. After that, the two reward rates are plotted to see its performance.

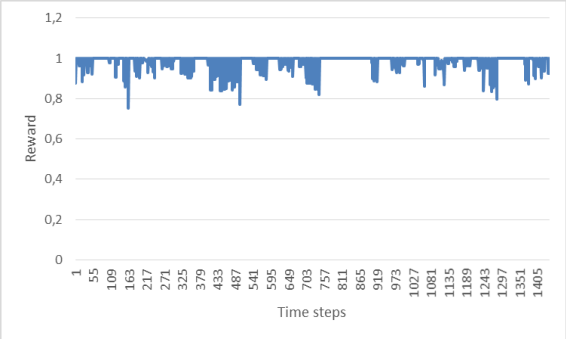


Figure 71. SLA satisfaction rate. DDQN with full update

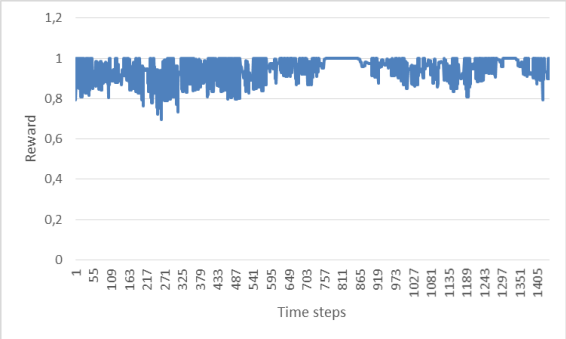


Figure 72. Utilization rate. DDQN with full update

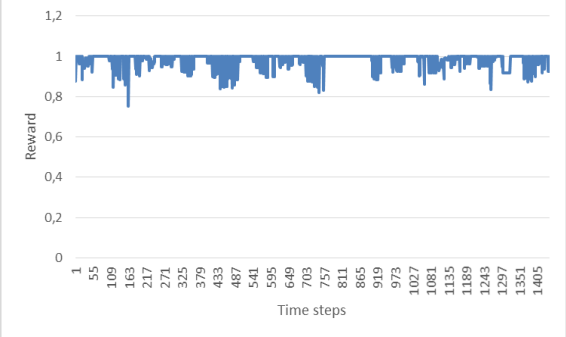


Figure 73. SLA satisfaction rate. DQN with soft update

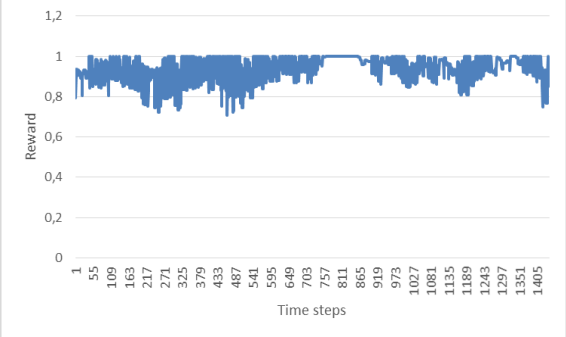


Figure 74. Utilization rate. DDQN with soft update

The two rates have got the same performance with the two configurations, from the figures we cannot say anything clear. It is necessary to transform these figures into numbers to have some comparable metrics. The metrics presented are the same as with DQN.

| Approach | Total reward | SLA rate | Utilization rate | SAGBR compliance | Capacity allocated | Iterations | Time |
|---|---|---|---|---|---|---|---|
| DDQN (full update) | 1390,11 | 1418,25 | 1361,97 | 72,64% | 64,30% | 23750 | 762,23 |
| DDQN (soft update) | 1384,95 | 1416,61 | 1353,28 | 70,5% | 68,89% | 12250 | 429,64 |

Table 26. Numerical metrics of the policy obtained

In this case, the policy obtained with full update improves the policy with soft. The main difference is in the utilization rate, where there is a difference of 8 points. This happens because with full update there are more steps that, instead of giving more capacity than the offered load, the capacity does not support all the traffic of the tenant, so the throughput is limited to the capacity but the utilization rate performs better. With the capacity allocated parameter, this phenomenon is also verified, as with soft update the agent is squeezing the available capacity more than with full update.

Although the total reward obtained with full update is better than with soft update, the required iterations to converge have improved so much that the soft update configuration is preferable to the full update. It is worth mentioning that each time a simulation is executed the results can vary, so it is possible that the policy obtained sometimes could be better

with full and others with soft update. What it is clear is that with soft update the model will converge sooner most of the time.

## 4.5. DDPG Agent

The third and last approach studied is DDPG. Now, the action space is continuous so the problem it is not exactly the same as with DQN or DDQN. The total reward will be higher as the capacity can be better adjusted. However, the complexity of the problem has increased since the capacity can be adjusted much more to the required traffic, so it will be necessary to increase the dimension of the neural network. As in the other two approaches has been proved that the soft update improves the behaviour of the agent, with DDPG we include this technique from the beginning.

### 4.5.1. Hyperparameters optimization

The process is the same as before, we create three different methodologies to find the optimum configuration that learns faster. Now, as there are two networks, actor and critic, there will be more hyperparameters to tune. In Annex 3, it is explained the first simulations done to obtain an initial good configuration that will be the reference value for each hyperparameter, see Table below.

| Hyperparameter | Value |
|---|---|
| Actor learning rate | 1e-4 |
| Critic learning rate | 1e-3 |
| Batch size | 256 |
| Discount factor | 0,9 |
| Hidden layers (L1,L2) | (200,500) |
| Soft update ($\tau$) | 0,01 |

**Table 27. Initial values of hyperparameters**

The three methodologies are:

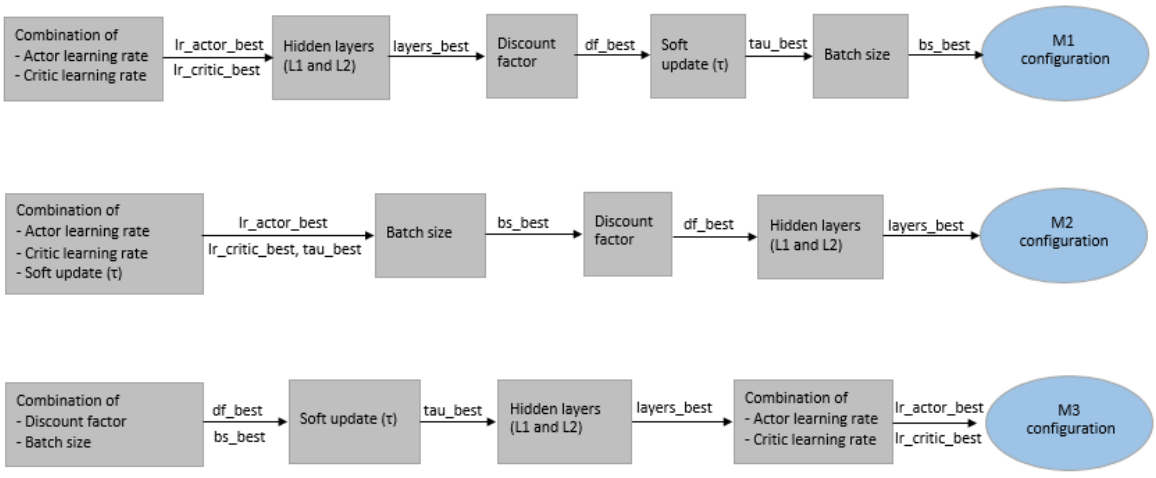


**Figure 75. Methodologies for DDPG hyperparameters tuning**

The criteria for studying the convergence has changed. As the total reward is higher, we will assume that the training is over when the last evaluation is above 1405 and the variation between this evaluation and all the others evaluations in an interval of 12.500 steps is less than 0.75%. If no convergence is reached, the process will stop after 100.000 iterations. The metrics exported are the same as before.

The results obtained with each methodology are the following. We only show the configurations that have achieved convergence before the end of the simulation.

| Step | Actor learning rate | Critic learning rate | Batch size | Discount factor | L1 | L2 | τ | Reward | Dev. | Iterations | Time (s) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 0,0001 | 0,01 | 256 | 0,9 | 200 | 500 | 0,01 | 1324,43 | 109,95 | 29250 | 2099,30 |
| **1** | 0,0001 | 0,005 | 256 | 0,9 | 200 | 500 | 0,01 | 1335,01 | 100,59 | 29750 | 2158,28 |
| **1** | 0,0001 | 0,001 | 256 | 0,9 | 200 | 500 | 0,01 | 1361,79 | 82,717 | 26250 | 1915,19 |
| **1** | 0,00001 | 0,01 | 256 | 0,9 | 200 | 500 | 0,01 | 1356,53 | 47,729 | 50500 | 3683,13 |
| **1** | 0,00001 | 0,005 | 256 | 0,9 | 200 | 500 | 0,01 | 1363,13 | 52,309 | 43000 | 3138,66 |
| **1** | 0,00001 | 0,001 | 256 | 0,9 | 200 | 500 | 0,01 | 1371,24 | 45,505 | 28500 | 2087,83 |
| **2** | 0,0001 | 0,001 | 256 | 0,9 | 200 | 500 | 0,01 | 1380,49 | 9,1878 | 18750 | 1381,94 |
| **2** | 0,0001 | 0,001 | 256 | 0,9 | 200 | 700 | 0,01 | 1372,07 | 50,697 | 26750 | 2346,93 |
| **2** | 0,0001 | 0,001 | 256 | 0,9 | 300 | 300 | 0,01 | 1363,71 | 86,728 | 48000 | 3217,24 |
| **2** | 0,0001 | 0,001 | 256 | 0,9 | 300 | 500 | 0,01 | 1342,24 | 103,85 | 98250 | 8915,72 |
| **2** | 0,0001 | 0,001 | 256 | 0,9 | 300 | 700 | 0,01 | 1239,48 | 113,72 | 74750 | 8504,25 |
| **2** | 0,0001 | 0,001 | 256 | 0,9 | 400 | 700 | 0,01 | 1288,94 | 140,55 | 39000 | 5458,87 |
| **3** | 0,0001 | 0,001 | 256 | 0,85 | 200 | 500 | 0,01 | 1371,20 | 63,016 | 70000 | 5221,50 |
| **3** | 0,0001 | 0,001 | 256 | 0,95 | 200 | 500 | 0,01 | 1360,33 | 3,0992 | 15500 | 1163,81 |
| **4** | 0,0001 | 0,001 | 256 | 0,95 | 200 | 500 | 0,001 | 1283,08 | 112,21 | 46500 | 3501,49 |
| **5** | 0,0001 | 0,001 | 256 | 0,95 | 200 | 500 | 0,01 | 1299,69 | 110,53 | 58250 | 4253,06 |
| **M** | 0,0001 | 0,001 | 256 | 0,95 | 200 | 500 | 0,01 | 1360,33 | 3,0992 | 15500 | 1163,81 |

**Table 28. DDPG Configurations with Methodology 1**

| Step | Actor learning rate | Critic learning rate | Batch size | Discount factor | L1 | L2 | τ | Reward | Dev. | Iterations | Time (s) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0,0001 | 0,01 | 256 | 0,9 | 200 | 500 | 0,01 | 1344,58 | 80,54 | 33250 | 2424,45 |
| 1 | 0,0001 | 0,005 | 256 | 0,9 | 200 | 500 | 0,005 | 1397,35 | 9,68 | 34000 | 2496,66 |
| 1 | 0,0001 | 0,001 | 256 | 0,9 | 200 | 500 | 0,01 | 1346,57 | 92,26 | 22500 | 1652,32 |
| 1 | 0,00001 | 0,01 | 256 | 0,9 | 200 | 500 | 0,01 | 1311,24 | 123,49 | 34000 | 2522,27 |
| 1 | 0,00001 | 0,01 | 256 | 0,9 | 200 | 500 | 0,005 | 1384,81 | 28,22 | 24250 | 1800,12 |
| 1 | 0,00001 | 0,005 | 256 | 0,9 | 200 | 500 | 0,01 | 1396,73 | 6,60 | 18750 | 1391,68 |
| 1 | 0,00001 | 0,005 | 256 | 0,9 | 200 | 500 | 0,005 | 1316,37 | 115,8 | 80500 | 5997,83 |
| 1 | 0,00001 | 0,001 | 256 | 0,9 | 200 | 500 | 0,01 | 1340,37 | 84,72 | 40500 | 3028,83 |
| 1 | 0,00001 | 0,001 | 256 | 0,9 | 200 | 500 | 0,005 | 1329,36 | 64,65 | 51500 | 3854,07 |
| 2 | 0,00001 | 0,005 | 128 | 0,9 | 200 | 500 | 0,01 | 1394,87 | 7,22 | 19250 | 1114,96 |
| 2 | 0,00001 | 0,005 | 256 | 0,9 | 200 | 500 | 0,01 | 1369,72 | 46,76 | 19750 | 1500,12 |
| 3 | 0,00001 | 0,005 | 256 | 0,95 | 200 | 500 | 0,01 | 1354,28 | 96,92 | 44500 | 3380,58 |
| 4 | 0,00001 | 0,005 | 256 | 0,9 | 200 | 700 | 0,01 | 1395,14 | 8,97 | 19250 | 1736,06 |
| 4 | 0,00001 | 0,005 | 256 | 0,9 | 300 | 700 | 0,01 | 1348,84 | 78,29 | 21500 | 2477,97 |
| M | 0,00001 | 0,005 | 256 | 0,9 | 200 | 500 | 0,01 | 1396,73 | 6,60 | 18750 | 1391,68 |

Table 29. DDPG Configurations with Methodology 2

| Step | Actor learning rate | Critic learning rate | Batch size | Discount factor | L1 | L2 | τ | Reward | Dev. | Iterations | Time (s) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0,0001 | 0,001 | 128 | 0,85 | 200 | 500 | 0,01 | 1310,74 | 75,45 | 46500 | 2546,56 |
| 0 | 0,0001 | 0,001 | 256 | 0,85 | 200 | 500 | 0,01 | 1273,08 | 120,04 | 40250 | 2945,44 |
| 0 | 0,0001 | 0,001 | 256 | 0,9 | 200 | 500 | 0,01 | 1275,57 | 112,27 | 40250 | 2957,80 |
| 0 | 0,0001 | 0,001 | 128 | 0,95 | 200 | 500 | 0,01 | 1385,72 | 24,81 | 17500 | 976,041 |
| 0 | 0,0001 | 0,001 | 256 | 0,95 | 200 | 500 | 0,01 | 1347,83 | 68,32 | 47000 | 3478,79 |
| 1 | 0,0001 | 0,001 | 128 | 0,95 | 200 | 500 | 0,01 | 1304,44 | 71,37 | 64500 | 3610,50 |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 0,0001 | 0,001 | 128 | 0,95 | 200 | 500 | 0,005 | 1373,11 | 42,41 | 18750 | 1048,69 |
| **2** | 0,0001 | 0,001 | 128 | 0,95 | 300 | 700 | 0,01 | 1318,11 | 66,30 | 50000 | 3942,53 |
| **3** | 0,00001 | 0,01 | 128 | 0,95 | 200 | 500 | 0,01 | 1358,86 | 47,81 | 43500 | 2289,08 |
| **3** | 0,00001 | 0,005 | 128 | 0,95 | 200 | 500 | 0,01 | 1371,08 | 40,45 | 50250 | 2633,28 |
| **3** | 0,00001 | 0,001 | 128 | 0,95 | 200 | 500 | 0,01 | 1372,93 | 44,09 | 44250 | 2326,02 |
| **M** | 0,0001 | 0,001 | 128 | 0,95 | 200 | 500 | 0,01 | 1385,72 | 24,81 | 17500 | 976,04 |

**Table 30. DDPG Configurations with Methodology 3**

The three methodologies output a lot of configurations that converge early, so the order of the hyperparameters study impact less than with the other approaches. This is mainly for one reason, the best configurations obtained are almost equal to the reference configuration, so when studying a hyperparameter for other values it impacts less on the performance of the model as all the others are well configured. However, the reward and deviation columns obtain results that vary a lot one from the other. This happens because at the beginning of some processes the evaluations get a poor performance since the model needs more training to start working well.

Now, from the best configuration of each methodology is exported all the information of the model (capacity assigned, reward evolution, reward rates, etc.), so a comparison between these three methodologies is done to decide with which we get a better policy. What we will exactly do is to study the behaviour of the model in the last evaluation, without considering the previous ones.

The three configurations that will be studied are:

| | Actor learning rate | Critic learning rate | Batch size | Discount factor | L1 | L2 | τ | Reward | Dev. | Iterations | Time (s) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **M1** | 0,0001 | 0,001 | 256 | 0,95 | 200 | 500 | 0,01 | 1360,33 | 3,0992 | 15500 | 1163,81 |
| **M2** | 0,00001 | 0,005 | 256 | 0,9 | 200 | 500 | 0,01 | 1396,73 | 6,60 | 18750 | 1391,68 |
| **M3** | 0,0001 | 0,001 | 128 | 0,95 | 200 | 500 | 0,01 | 1385,72 | 24,81 | 17500 | 976,04 |

**Table 31. DDPG best configurations**

We start analysing the total reward evolution of each configuration, see figures below. The learning process is quite curious, M2 is the only evolution that seems "normal" and the policy is learned little by little. With the other two configurations, M1 and M3, from one evaluation to another the total reward increases a lot and then it remains stable. It is important to say that not always a high total reward means that the good policy is achieved, this is the reason we need that it remains stable during some steps to consider that is reached.

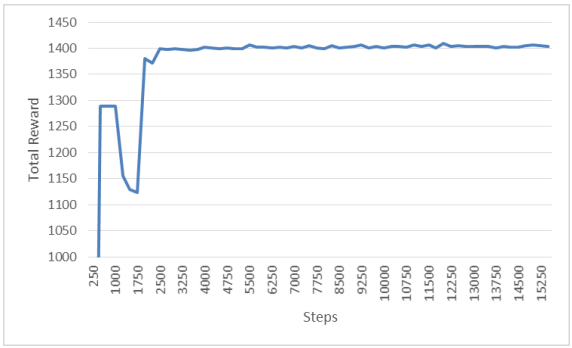


Figure 76. Total reward evolution of M1

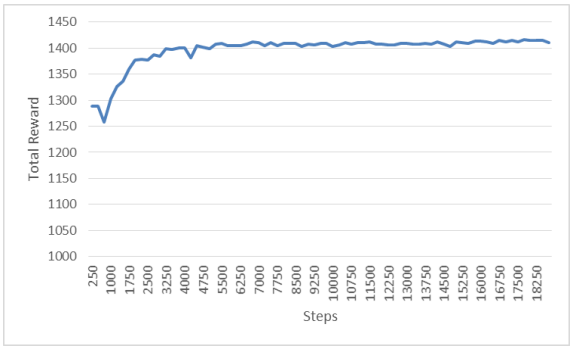


Figure 77. Total reward evolution of M2

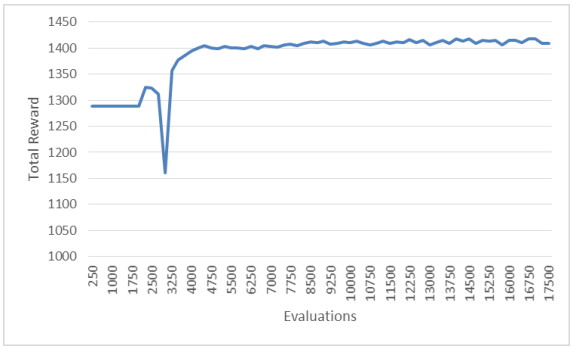


Figure 78. Total reward evolution of M3

As the three charts are very similar, no conclusion can be deducted from them. Therefore, it is necessary to look at the policy behaviour obtained.

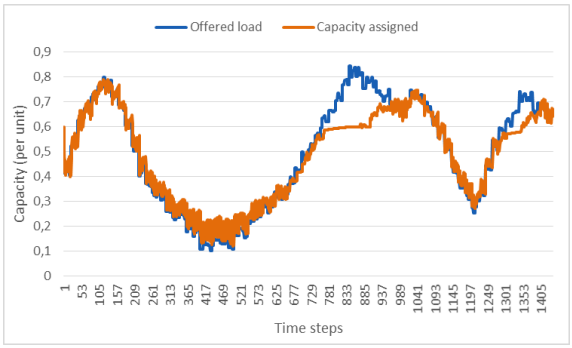


Figure 79. Behaviour of the policy obtained with M1. Tenant 1

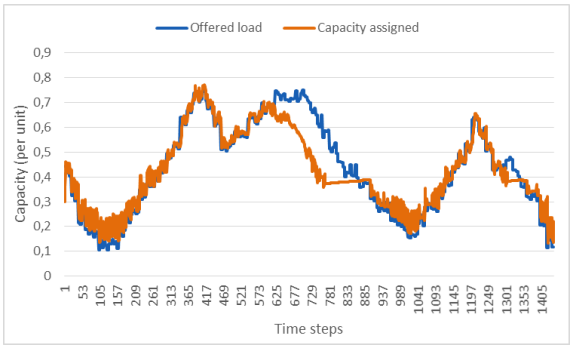


Figure 80. Behaviour of the policy obtained with M1. Tenant 2

We can say that the policy obtained with M1 is optimal. The capacity is always adjusted to the offered load of both tenants, and when it enters on the critic steps, the policy respects the SAGBR.

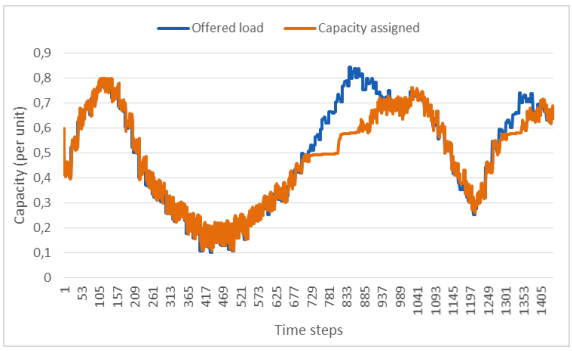


Figure 81. Behaviour of the policy obtained with M2. Tenant 1

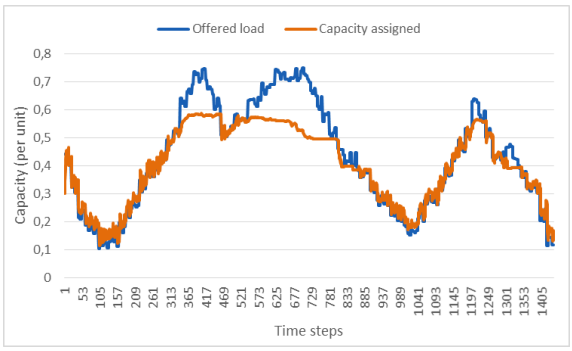


Figure 82. Behaviour of the policy obtained with M2. Tenant 2

In this case the policy obtained is not good. When the offered load is below the total capacity available the resources are efficiently distributed, but when the simulation enters on the critic steps, the model does not do a good assignment of the resources. The SAGBR of tenant 1 is not accomplished, the agent is allocating the 50% of the capacity when the traffic is higher instead of giving the capacity needed until arrive to the 60%.

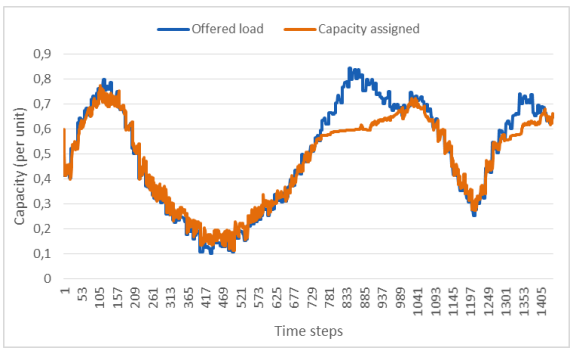


Figure 83. Behaviour of the policy obtained with M3. Tenant 1

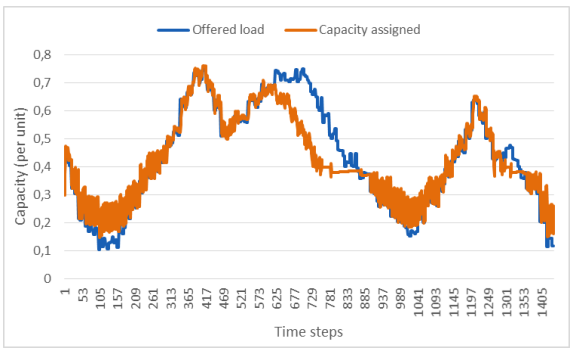


Figure 84. Behaviour of the policy obtained with M3. Tenant 2

The policy with M3 is similar to the policy of M1, so it has a good behaviour. However, it seems that in some points the capacity could be better adjusted, e.g. between the steps 105 and 200 of tenant 2. M1 and M3 only differ in the batch size configuration that M1 sets it to 256 and M3 to 128. From what we know from experience, 256 gives always better performance but to finally decide which policy is better between these two, we look at the SLA satisfaction rate of the last evaluation.

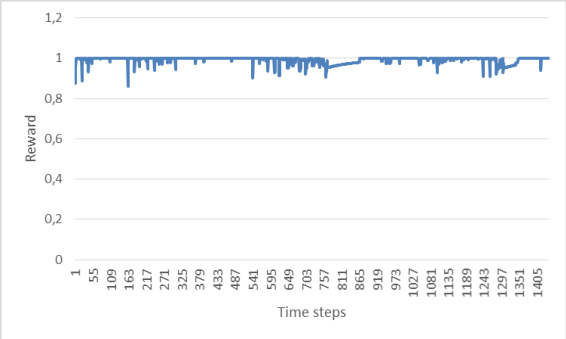


Figure 85. SLA satisfaction rate with M1

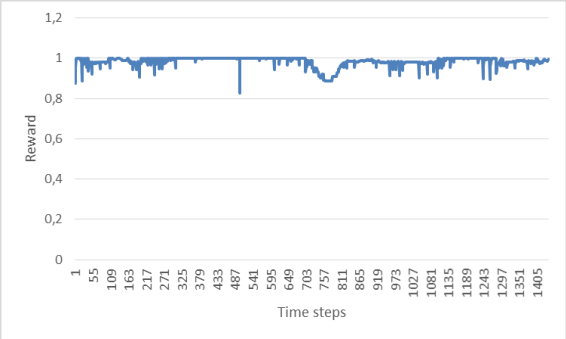


Figure 86. SLA satisfaction rate with M3

The results are what we expected, the two figures are really similar being the biggest difference on the critic steps, where with M3 the rate has like a little gap. In addition, we compute the amount of time that the SLA is fulfilled in both tenants. The results are also favourable to M1: M1 satisfy the SLA the 81,18% of the time and M3 the 77,57%.

After all of that, we can say that the best hyperparameters configuration for the DDPG approach is the obtained with the methodology 1.

| | Actor learning rate | Critic learning rate | Batch size | Discount factor | L1 | L2 | τ |
|---|---|---|---|---|---|---|---|
| **Optimum** | 0,0001 | 0,001 | 256 | 0,95 | 200 | 500 | 0,01 |

Table 32. DDPG optimum configuration

## 4.6. Comparative of the three approaches

In this chapter, the comparative between the three approaches is done. First, DQN and DDQN are compared between each other as they are the two value functions approaches. After that, both of them are compared with the DDPG approach, which is an actor-critic method. The comparison is done with the best results obtained from each algorithm.

### 4.6.1. DQN vs DDQN

In order to compare these two approaches, we take the simulations done in the chapters 4.3.3 and 4.4.3. In these chapters was proved that the policy obtained was optimum, so here we will focus on the learning process and the numerical metrics of each approach. It is worth mentioning that the results below are the best performance we have obtained from each of the approaches.

We distinguish between the use of the soft update technique or not for the target network. The optimum hyperparameters configuration for each approach is the following:

| Approach | Learning rate | Batch size | Discount factor | L1 | L2 | τ |
|---|---|---|---|---|---|---|
| **DQN (full update)** | 0,0003 | 256 | 0,85 | 100 | 0 | 1 |
| **DQN (soft update)** | 0,0003 | 256 | 0,9 | 100 | 0 | 0,001 |
| **DDQN (full update)** | 0,0002 | 256 | 0,85 | 200 | 0 | 1 |
| **DDQN (soft update)** | 0,0002 | 256 | 0,9 | 200 | 0 | 0,001 |

**Table 33. DQN and DDQN optimum configurations**

The configurations used are similar. The difference between DQN and DDQN is the learning rate, which is a bit smaller with DDQN. In both approaches, the discount factor increases when using the soft update. To compare the learning process, the total reward evolution is plotted. The comparison distinguishes between the usage of the soft update technique or not.

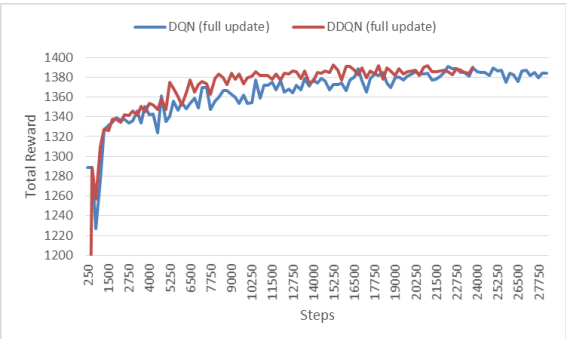


**Figure 87. DQN vs DDQN total reward evolution. Full update**

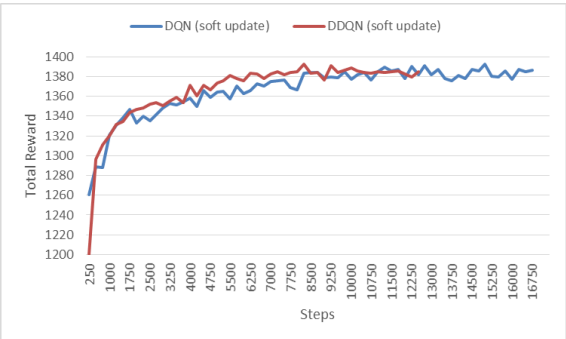


**Figure 88. DQN vs DDQN total reward evolution. Soft update**

In both cases, the DDQN method improves DQN performance. The first evaluation done is worse with DDQN, but then the learning is faster achieving the convergence easily. With

DQN the learning is going slowly, evaluation after evaluation the behaviour improves, but little by little until it arrives at the range of total rewards that get good policies. This movement to the good range of values is what DDQN does it faster. Both algorithms follow a good learning process, one faster than the other, but there are not big oscillations between evaluations and the model always improves its behaviour until the convergence interval.

Now, we compare the policy obtained with each method. It is compared numerically, to see easier the possible differences.

| Approach | Total Reward | SLA rate | Utilization rate | SAGBR compliance | Capacity allocated | Iterations | Time |
|---|---|---|---|---|---|---|---|
| **DQN (full update)** | 1383,91 | 1414,01 | 1353,82 | 71,8% | 67,8% | 27750 | 989,23 |
| **DQN (soft update)** | 1386,14 | 1417,69 | 1354,59 | 74,1% | 65% | 17000 | 592,46 |
| **DDQN (full update)** | 1390,11 | 1418,25 | 1361,97 | 72,64% | 64,30% | 23750 | 762,23 |
| **DDQN (soft update)** | 1384,95 | 1416,61 | 1353,28 | 70,5% | 68,89% | 12250 | 429,64 |

**Table 34. DQN and DDQN policies**

Looking at the table above, there are not big differences between approaches except for the iterations. The four policies obtained are very similar, the SLA rate is around 1415/1418 and the utilization rate does not exceed the 1365, so the total reward is between 1383 and 1390. Taking into account the limitations of the action steps, the four approaches have achieved an optimum solution.

The main objective of the project is to find the approach that solve faster and better the problem presented, and in view of the little differences between the policies, the best approach is DDQN with soft update, which only needs 12.250 steps of training to solve it.

The results obtained match with what we know from the theory. The DDQN is an evolution of the DQN, whose objective is to handle the problem of the overestimation of Q-values. At the beginning of the training, where the agent does not have enough information of which is the best action to take, if the agent takes the maximum Q value (which can be noisy) as the best action can lead to false positives. Therefore, when decoupling the action selection from the target Q value computation, the overestimation of Q values is reduced, and the learning is faster.

### 4.6.2. DQN / DDQN vs DDPG

The second comparison is done between the three algorithms implemented. DDPG has been only studied with soft update, so the comparison will be limited to this type of approaches. The DQN and DDQN parameters taken are the same as before, and with DDPG we take the results of the best simulation of 4.5.1, whose good behaviour has been proved.

The configuration for DQN and DDQN was presented in the Table 32 and for DDPG is the following:

| Approach | Actor learning rate | Critic learning rate | Batch size | Discount factor | L1 | L2 | τ |
|---|---|---|---|---|---|---|---|
| DDPG (soft update) | 0,0001 | 0,001 | 256 | 0,95 | 200 | 500 | 0,01 |

Table 35. DDPG optimum configuration

Apart from the two learning rates that DDPG adds, the biggest difference with the other approaches is the increment of the hidden layers. The main difference between these three algorithms is how the action step is defined, with DDPG the agent can take any value between -10 and 10, so the capacity will be better adjusted but the neural network needs more capacity. The soft update hyperparameter has also changed, now it is increased by a factor of 10, so the update will be a bit bigger in each step.

The learning process of each method is:

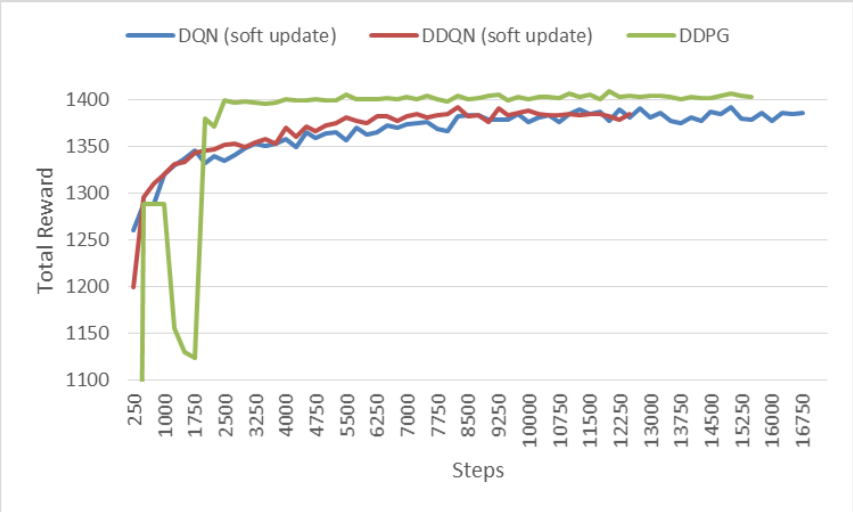


Figure 89. DQN vs DDQN vs DDPG total reward evolution

The evolution of the DDPG method is quite different from the DQN/DDQN methods. In the first steps of the simulation, the DDPG performs worse than DQN/DDQN because of the more difficult problem and more nodes to configure. However, in an interval of 1000 steps of training, the DDPG agent starts acting really well and keeps on high rewards to finally converge. The training steps needed for DDPG is between the iterations for DQN and for DDQN. From the figure, is appreciated the increment in total reward that produces the change of the action step. To deepen in its performance, the metrics of the policy obtained with DDPG is compared with the metrics of DQN and DDQN with soft update.

| Approach | Total Reward | SLA rate | Utilization rate | SAGBR compliance | Capacity allocated | Iterations | Time |
|---|---|---|---|---|---|---|---|
| DQN (soft update) | 1386,14 | 1417,69 | 1354,59 | 74,1% | 65% | 17000 | 592,46 |
| DDQN (soft update) | 1384,95 | 1416,61 | 1353,28 | 70,5% | 68,89% | 12250 | 429,64 |
| DDPG | 1407,14 | 1430,12 | 1384,17 | 81,18% | 61,7% | 15500 | 1163,81 |

Table 36. DQN, DDQN and DDPG policy metrics

The metrics meet the expectations. The total reward obtained has increased 20 points, and the biggest improvement comes with the utilization rate. As the actions could be better adjusted to the offered load, the capacity does not exceed the load by a big difference, so the utilization rate is benefited. By the same token, in the critic steps, where the DQN/DDQN agent had to decide how much capacity allocate to each tenant in steps of 10%, decreasing this percentage allows the agent to meet the SLA in steps that was not possible. This can also be seen with the SAGBR compliance metric, since the SAGBR is met the 81,18% of the time, an increment of 10% with respect the other methods.

However, not everything is positive with DDPG. The increment in the number of nodes and layers leads to an increment of the computational time that the simulation lasts. Specifically, this difference comes from each time the target networks are updated. In addition, the capacity allocated metric has got worse too. This metric computes the amount of time that the capacity is either equal, above the offered load, or 100% of the capacity available when the load exceeds the total capacity. The reason this metric has performed worse comes from the critic steps, where with the DDPG algorithm there are a lot of steps with a total capacity allocated of 99,x%, instead of the 100% used with DQN/DDQN. This happens because it is a bit more difficult to adjust the capacity to the 100% with the continuous action space. If we change the criteria to compute this metric: at every step that the total capacity allocated is bigger than the 98% of the available when the total offered load exceeds the capacity available, we assume that the metric is fulfilled. Then, the metric improves a 10% obtaining a 71,23%, which improves the performance of both DQN and DDQN.

After all of that, we can conclude that the DDPG algorithm improves the performance of DQN and DDQN but paying some consequences. However, it is worth mentioning that the problem is not exactly the same because the action space is different, so the comparison is not entirely fair.

The three approaches have achieved ending the simulation with an optimum policy. If we want the capacity to match perfectly the traffic, DDPG is the appropriate approach. However, if we want a solution where the resources are well managed and the training is more dynamic, DDQN is the one.

# 5. Budget

In this chapter is presented the cost estimation of the project. The costs are mainly divided into two different categories: the labour cost and the cost of the tools used in the project.

The project was done by a person with a degree in telecommunications engineering. Therefore, the approximate value of cost per hour is equivalent to the expenses of a junior engineer, 10€ per hour. To this value it has to be added the 30% of fees, so the cost per hour will be 13€/hour. The time of dedication for implementing all the tasks required for the project and their cost are summarized below.

| Task | Hours | Cost |
|---|---|---|
| Bibliographic study | 130 | 1690 € |
| Algorithm implementation | 500 | 6500 € |
| Thesis documentation | 180 | 2340 € |
| **Total** | **810** | **10530 €** |

**Table 37. Labour cost**

The total hours correspond to a dedication of 30 hours per week for a total of 27 weeks.

The project do not require any hardware element except for the computers. When a computer is used it wears out, so it is necessary to compute the amortization cost of it. In this project, because of the amount of simulations done, two computers have been used with an initial cost of 1.100€ for computer. The amortization cost is considered of a 33% per year, in this case 366,67€. Counting that the project has lasted 7 months more or less, the total cost of the computer is 214€.

The rest of tools used in the project are licensing free, so there is no cost associated to them.

Thus, considering both the labour and computer cost, the total cost of the project is 10.744€.

# 6. Conclusions and future development

In this thesis, it has been studied the performance evaluation of a capacity sharing algorithm through DRL based approaches, specifically DQN, DDQN and DDPG. To do it, a NG-RAN simulator scenario has been developed following the agent-environment model introduced by RL. The simulator has been implemented with Python because it has allowed us to use the TensorFlow library, which is an open- source ML library that has helped us to implement the algorithms and facilitates the possibility to upgrade the code in the future.

The three algorithms implemented are off-policy algorithms, which means that the policy for training is not the same as the policy being evaluated. The training policy combines the exploration and the exploitation whereas the evaluation policy is always looking for the best action. The main difference between the three algorithms is how we have defined the action space. DQN and DDQN have been designed with a discrete action space, where the agent can decide to modify the capacity of each tenant by a 10%, -10% or doing nothing. DDPG, however, has been defined with a continuous action space between -10 and 10, allowing the agent to better adjust the capacity to the traffic.

A key point of the project has been tuning the hyperparameters of each method. It was a necessary study to achieve the optimum behaviour of the agent, and it is the analysis that has possibly taken more time to get it. In order to achieve it, different methodologies have been created alternating the order in which each hyperparameter is studied to optimize its value. It has been seen that the order is extremely important, as the hyperparameters that impact more on the systems should be prioritized. In addition, the initial values of the set of hyperparameters are also really important, because if we do not initialize all of them to a good stating value, the first steps of the methodology will not perform well. From this study, we have been able to verify that batch size and learning rate are the two most important hyperparameters, with a good configuration of them the rest of hyperparameters lose importance. The hidden layers, which define how the neural network is, gives an idea of the difficulty of the problem, bigger they are, more capacity they have and more tricky the problem is.

Once the hyperparameters were optimized, the soft update technique has been implemented to improve the performance of the algorithms. This technique introduces a new hyperparameter, so the optimization has done another time as the soft update parameter could affect the others. The improvement of this technique has been studied for DQN and DDQN approaches, with DDPG it was implemented from the beginning. This technique updates the target network a little instead of copying the weights of the online network at once, which avoids instabilities during training and improves the performance. In this case, the improvement is measured in convergence time, as with soft update the agent learns the optimal policy much faster.

After analysing and comparing the results obtained with each approach, we can say that with the three approaches we have achieved solving the problem successfully, but with different performances. It has been proved that DDQN approach improves the DQN, as it is an evolution that handles the problem of the overestimation of the Q-values. The improvement, but, comes from the necessary iterations to converge, since the policy

obtained is more or less the same. From the total reward, we have distinguished between the SLA satisfaction rate of both tenants together and the utilization rate. The second has got a worse performance because of the action step, the ±10% of variation has not allowed a better adjustment of the capacity to the offered load.

DDPG approach, as the action space is continuous, has improved the performance of DQN and DDQN in terms of the policy obtained but not in the convergence iterations. Now, the utilization rate has been benefited from this change in the action step and there is not so much over-provisioning of capacity. However, the DDPG algorithm needs an increment in the number of nodes of the neural network, which leads to an increment in the real time the simulation lasts.

In conclusion, thanks to the simulator, it has been possible to analyse the different algorithms implemented and verify their correct execution. All the algorithms have managed to optimize the capacity sharing algorithm among slices, highlighting DDQN and DDPG. On the one hand, with DDQN we have achieved a dynamic behaviour of the network allocating the resources optimally but without a very high precision. On the other hand, DDPG approach has achieved the best performance that perfectly fits the capacity to the offered load but at the expense of a slower process.

### 6.1. Future development

In this thesis, we have seen a first approach of using DRL-based approaches for capacity sharing in RAN slicing of 5G networks, but there are some studies that can be done to gain more knowledge and upgrade the simulator:

- Change the action step of DQN/DDQN approaches to study their performance when the actions to take are smaller. With this study, a closer comparison could be done with DDPG algorithm.
- Study the impact of the exploration vs exploitation dilemma. This aspect is also really important in the RL problems and we have not taken into consideration.
- Include new traffic shapes in the simulator to study new cases that can happen in real life.
- Expand the simulator to a multi-agent model, where different cells have to be managed and each agent will be in charge of a cell.

# Bibliography

[1] NGMN Alliance, "NGMN 5G White Paper", Feb. 2015.

[2] D.Marabissi and R.Fantacci, "Highly Flexible RAN Slicing Approach to Manage Isolation, Priority, Efficiency", *IEEE Access,* vol. 7, pp. 97130-97142, 2019. DOI: 10.1109/ACCESS.2019.2929732

[3] S. Zhang, "An Overview of Network Slicing for 5G,", *IEEE Wireless Communications*, vol. 26, no. 3, pp. 111-117, June 2019. DOI: 10.1109/MWC.2019.1800234

[4] R. Ferrús, O. Sallent, J. Pérez-Romero, R. Agustí, "On 5G Radio Access Network Slicing: Radio Interface Protocol Features and Configuration", IEEE Communications Magazine, September, 2017.

[5] J. Pérez-Romero, O. Sallent, R. Ferrús, R. Agustí, "On the Configuration of Radio Resource management in a Sliced RAN", 2018

[6] H. Hirayama, Y. Tsukamoto, S. Nanba and K. Nishimura, "RAN Slicing in Multi-CU/DU Architecture for 5G Services," 2019 *IEEE 90th Vehicular Technology Conference (VTC2019-Fall)*, Honolulu, HI, USA, 2019, pp. 1-5.

[7] I. Leonardo da Silva, G. Mildh, A. Trogolo, E. Buracchini, P. Spapis, A. Kaloxylos, G. Zimmemann, N. Bayer, "On the impact of network slicing on 5G radio Access networks", *IEEE Communications Magazines*, September, 2016.

[8] J. Pérez-Romero, "5G Mobile Communication Systems, Chapter 5: 5G New Radio (5G NR)", ETSETB, UPC, 2019.

[9] K. Arulkumaran, M. P. Deisenroth, M. Bundage and A. A. Bharath, "Deep Reinforcement Learning: A brief survey", *IEEE Signal Processing Magazine,* vol. 34, no. 6, pp. 26-38, Nov. 2017, DOI: 10.1109/MSP.2017.2743240

[10] Z. Xiong, Y. Zhang, D. Niyato, R. Deng, P. Wang and L. Wang, "Deep Reinforcement Learning for Mobile 5G and Beyond: Fundamentals, Applications, and Challenges," in *IEEE Vehicular Technology Magazine*, vol. 14, no. 2, pp. 44-52, June 2019, DOI: 10.1109/MVT.2019.2903655.

[11] L.P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *J. Artificial Intell. Res*., vol. 4, pp. 237–285, 1996. DOI: 10.1613/jair.301.

[12] R. S. Sutton and A.G. Barto, "Reinforcement Learning: An Introduction", Cambridge, MA: MIT Press, 1998.

[13] D.Silver, "RL Course, Lecture 2: Markov Decision Processes", UCL, [Online] Available: https://www.davidsilver.uk/wp-content/uploads/2020/03/MDP.pdf

[14] D. Silver, "RL Course, Lecture 3: Planning by Dynamic Programming", UCL, [Online], Avaiable: https://www.davidsilver.uk/wp-content/uploads/2020/03/DP.pdf

[15] D. Silver, "RL Course, Lecture 4: Model-Free Prediction", UCL, [Online] Available: https://www.davidsilver.uk/wp-content/uploads/2020/03/MC-TD.pdf

[16] A. Singh, "Reinforcement Learning: Markov Decision Process (Part 1)", *Towards data science*, July 2018, [Online] Available: https://towardsdatascience.com/introduction-to-reinforcement-learning-markov-decision-process-44c533ebf8da

[17] A. Singh, "Reinforcement Learning: Bellman Equation and Optimality (Part 2)", *Towards data science,* August 2019, [Online] Available: https://towardsdatascience.com/reinforcement-learning-markov-decision-process-part-2-96837c936ec3

[18] J. Greaves, "Understanding RL: The Bellman Equations", November, 2017, [Online] Available: https://joshgreaves.com/reinforcement-learning/understanding-rl-the-bellman-equations/

[19] H. Mao, M. Alizadeh, I. Menache, S. Kandula, "Resource management with Deep Reinforcement Learning", *HotNets*, 2016, DOI: 10.1145/3005745.3005750

[20] V. Mnih et al., "Human-level control through deep reinforcement learning", *Nature Publishing Group, a division of Macmillan Publishers Limited.,* 2015, DOI: 10.1038/nature14236

[21] H. van Hasselt, "Double Q-learning", *Proc. Neural Information Processing Systems,* 2010, pp.2613-2621

[22] H. van Hasselt, A. Guez and D. Silver, "Deep Reinforcement Learning with Double Q-learning", *Proc. Neural Information Processing System,* 2016, pp.2094-2100

[23] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra and M. Riedmiller, "*Deterministic policy gradient algorithms"*, In *ICML*, 2014.

[24] R.S. Sutton, D. Mcallester, S. Singh, Y. Mansour, "*Policy Gradient Methods for Reinforcement Learning with Function Approximation*". In *Proceedings of the NIPS*, Denver, CO, USA, 1 January 2000; pp. 1057–1063, doi:10.1.1.37.9714.

[25] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "*Continuous control with deep reinforcement learning*", in *CoRR,* abs/1509.02971 (2015).

[26] O. Sigaud, "*Reinforcement Learning: Deep Deterministic Policy Gradient",* in *Sorbonne Université*, [Online] Available: http://pages.isir.upmc.fr/~sigaud/teach/ddpg.pdf

[27] R. Liessner, J. Schmitt, A. Dietermann and B. Bake, "*Hyperparameter Optimization for Deep Reinforcement Learning in Vehicle Energy Management"*, in *Proceedings of the 11th International Conference on Agents and Artificial Intelligence - Volume 2: ICAART,* 2019, DOI: 10.5220/0007364701340144

# Annex 1. Hyperparameters study

In this annex is presented the study of each hyperparameter individually. The objective is to see how each hyperparameter affects the learning process and to get an acceptable value as a starting point for the following studies. At the beginning all the hyperparameters are initialized to a default configuration:

| Hyperparameter | Value |
|---|---|
| Learning rate | 1e-4 |
| Batch size | 64 |
| Discount factor | 0.9 |
| Hidden layers | (100,0) |
| Initial collect step | 500 |
| Replay buffer max length | 100.000 |

**Table 38. Initial hyperparameters values**

Then, the training starts and an evaluation is performed every 500 steps, the total number of iterations of the process is 50.000 steps. After the simulation, the information regarding each evaluation is exported. For each hyperparameter, three studies have usually been made: with a very small value, with an intermediate value and a very large value.

The results are mainly compared with the reward evolution of the evaluations. As the evaluation has 1440 steps, the maximum would be a reward of 1440, but from how we have defined the model, the good rewards would be between 1370 and 1390.

### 1. Learning rate

Learning rate is the hyperparameter that controls how much the weights of the neural network are adjusted in response to the estimated error. As it is probably the most important hyperparameter, the analyses have been done for 5 values: 0.1, 0.01, 1e-3, 1e-4 and 1e-5. The reward evolution for each value is the following:

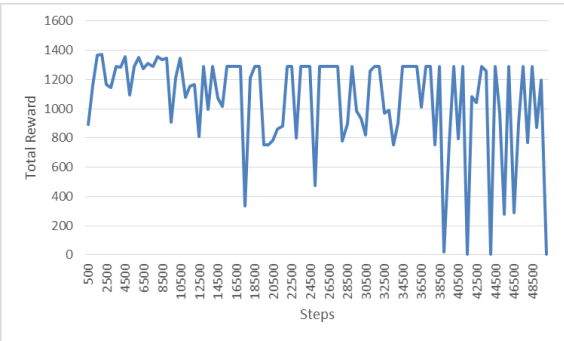


**Figure 90. Reward evolution with learning rate = 0.1**

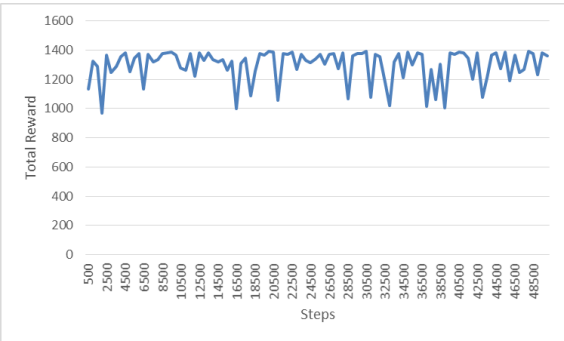


**Figure 91. Reward evolution with learning rate = 0.01**

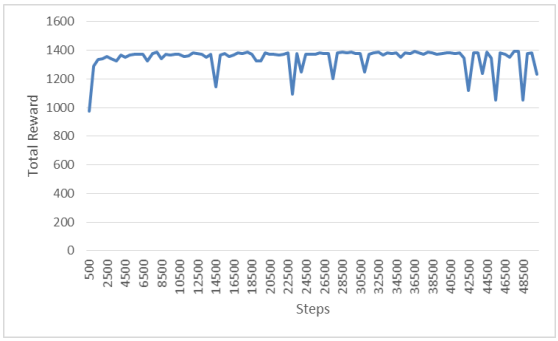


Figure 92. Reward evolution with learning rate = 1e-3

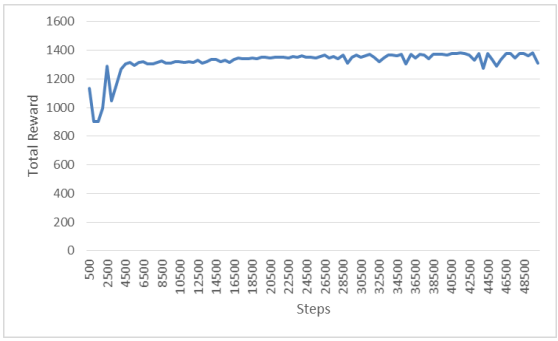


Figure 93. Reward evolution with learning rate = 1e-4

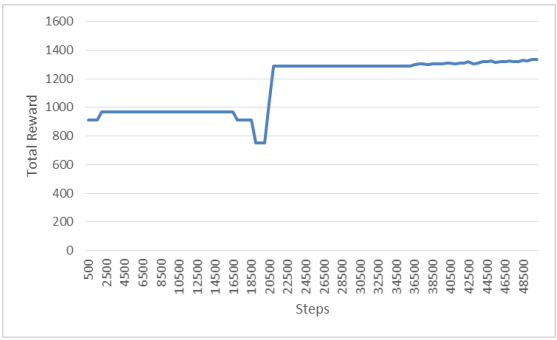


Figure 94. Reward evolution with learning rate = 1e-5

From these results, it is obvious that the extreme values (0.1 and 1e-5) do not achieve a good performance. With the biggest value the model diverge, and with the smallest the learning is so slow that the behaviour is almost always the same. The two values with good results are 1e-3 and 1e-4, being the second better. Although with 1e-4 the learning is a bit slower than with 1e-3, the reward then is more stable in good results. Therefore, from these graphs we can say that a value near 1e-4 could be a good value for our problem.

## 2. Batch size

Batch size indicates how many tuples are chosen randomly when training the model. To do the analysis we have done three simulations with 4, 128 and 1024. The reward evolution for each case is:

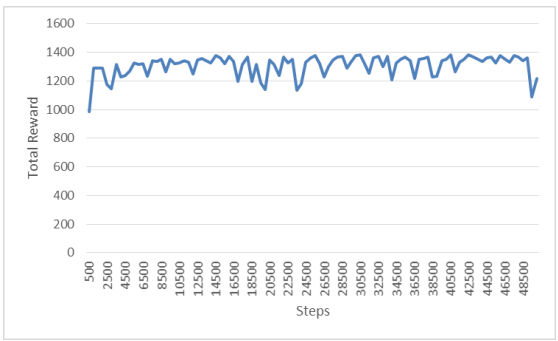


Figure 95. Reward evolution with batch size = 4

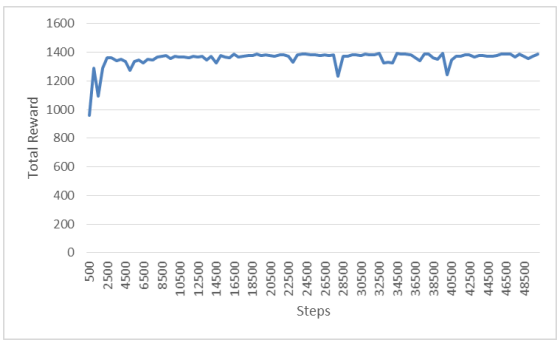


Figure 96. Reward evolution with batch size = 128

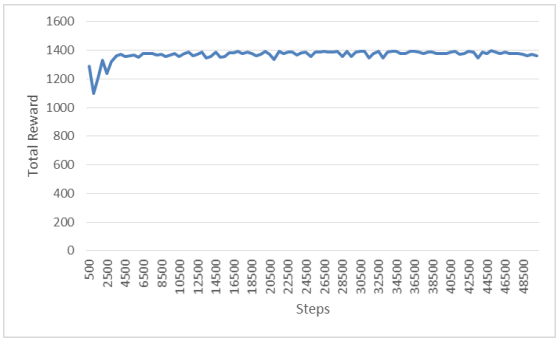


Figure 97. Reward evolution with batch size = 1024

Now, the impact on the reward is less significant than with learning rate. Between 128 and 1024 the difference is very small and with 4 we can appreciate more scattered values. Therefore, as only with this metric it is difficult to draw a conclusion we look at the training loss. At every step of training, the value of the loss is exported to see its evolution during the process. As it is a very large file, we plot the first 5000 steps, which give enough information of its behaviour.

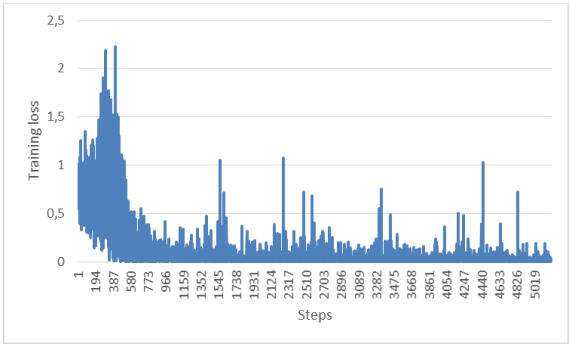


Figure 98. Training loss with batch size = 4

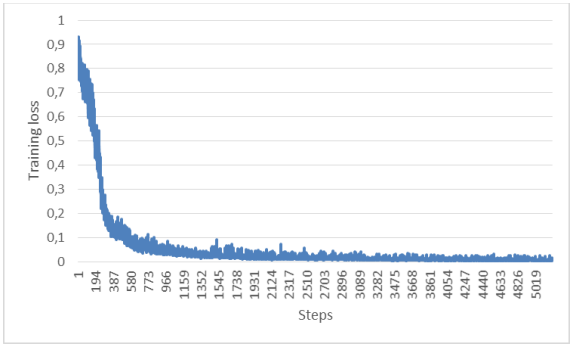


Figure 99. Training loss with batch size = 128

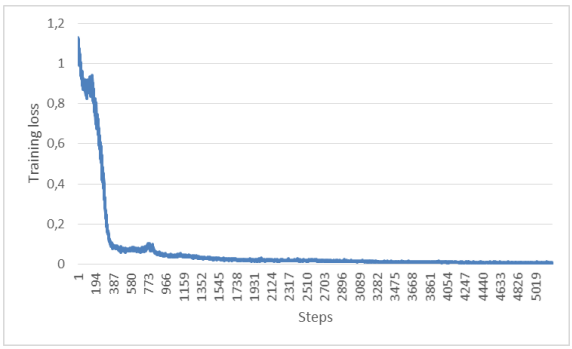


Figure 100. Training loss with batch size = 1024

We see how as bigger the mini-batch, smaller the loss and lower the fluctuation. The reason is that when taking few samples to update the weights, it is more probable that they are correlated so the update will not be done right and the loss will increase. When increasing the value, the correlation will almost disappear and also the fluctuations.

The last metric we study to make a decision is the time, in seconds, that the simulation lasts. As bigger the batch size, more samples are taken at each step so the process will be longer. In our case, each simulation lasted:

- Batch size = 4 → 1082s
- Batch size = 128 → 1234s
- Batch size = 1024 → 2044s

With all this, we can conclude that the best value is 128. It achieves almost the same results than 1024 and the process is more dynamic.

### 3. Discount factor

Discount factor parameter is used when doing the transition to the next state, it represents the importance we are giving to the future rewards. When it is close to 1, future rewards have more importance, and when it is close to 0 we give more emphasis on immediate rewards. We have done the study for 0.1, 0.5, 0.9 and 0.99.

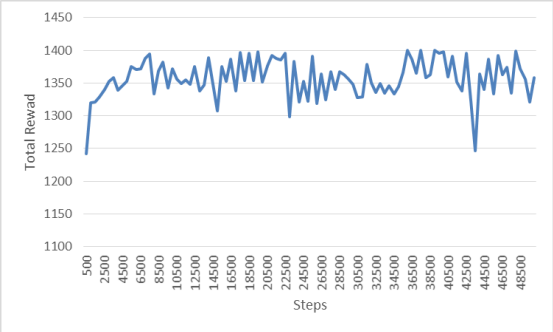


Figure 101. Reward evolution with discount factor = 0.1

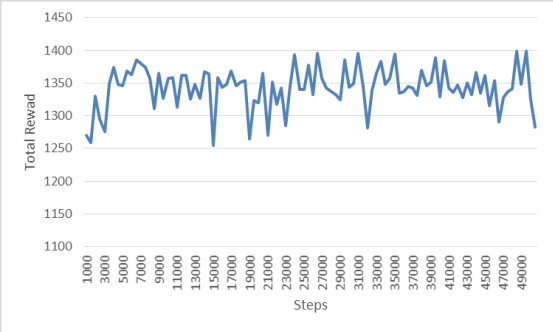


Figure 102. Reward evolution with discount factor = 0.5

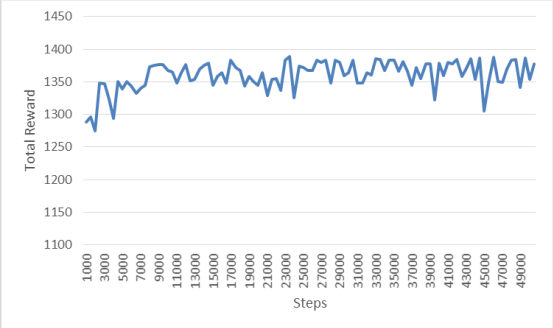


Figure 103. Reward evolution with discount factor = 0.9

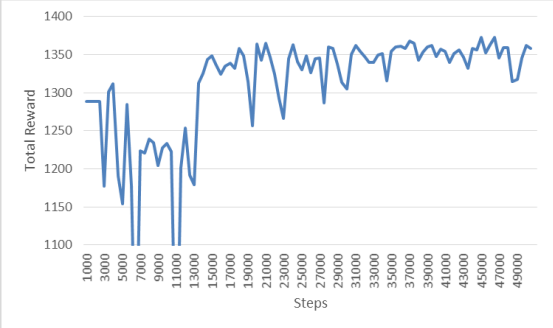


Figure 104. Reward evolution with discount factor = 0.99

In order to see better the results, the y-axis boundaries have been changed. From the figures, we can say that the best results are obtained with 0.9. It is the value from which the fluctuations between evaluations are lower and it seems to be more stable. With the other three values, there are more fluctuations and they do not achieve a good behaviour of the model during some consecutive evaluations.

### 4. Hidden layers

This hyperparameter is used to define the size and number of layers of the neural network. In DQN, the input is the state $< \rho_t(1), \alpha_t(1), \rho_t(2), \alpha_t(2), \alpha_{ava} >$, and the output is a set of Q-values, one for each possible action. After that, the action with the maximum Q-value is chosen.

As a first step, we study 4 different configurations with only one layer to see the results. They are done with 16, 100, 400 and 1000.

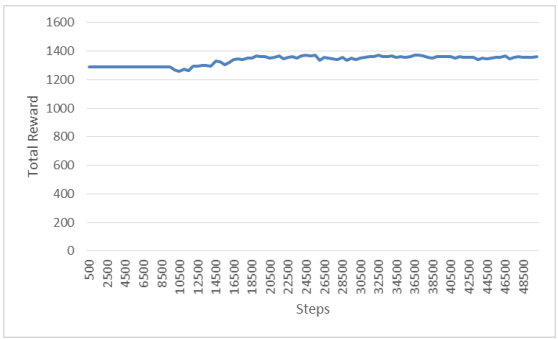


Figure 105. Reward evaluation with layers = (16,)

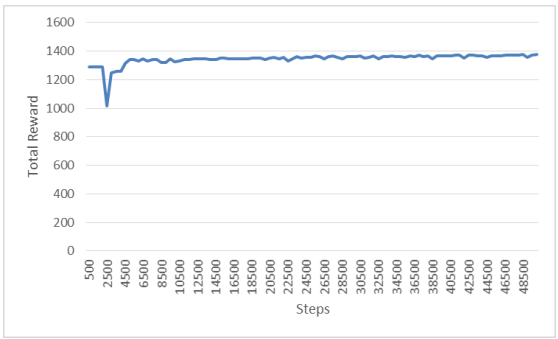


Figure 106. Reward evolution with layers = (100,)

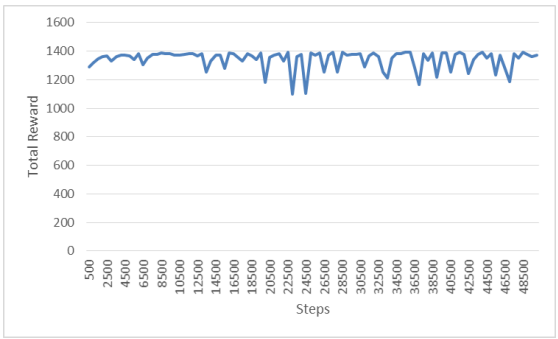


Figure 107. Reward evolution with layers = (400,)

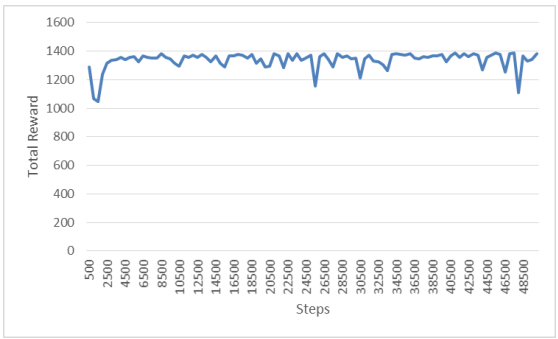


Figure 108. Reward evolution with layers = (1000,)

When the layer is small, the agent does not do any action at the first evaluations (1288 is the reward obtained when all the actions performed are (0,0)), it needs of more steps of training to start performing actions. However, when the policy changes the results obtained are better than with bigger layers, which suffer from oscillations between evaluations. Thus, the two configurations we consider good are (16,) and (100,).

After that, we add a second layer to see if the results improve. The layers studied are (16,16), (32, 32) and (100,100).

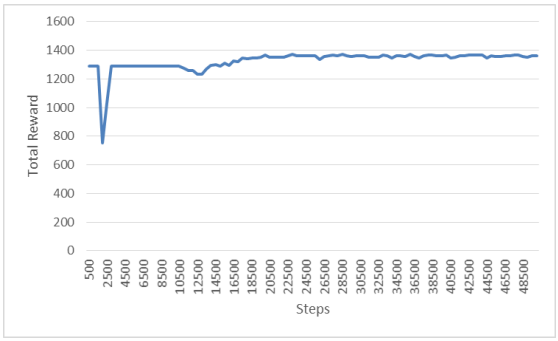


Figure 109. Reward evolution with layers = (16,16)

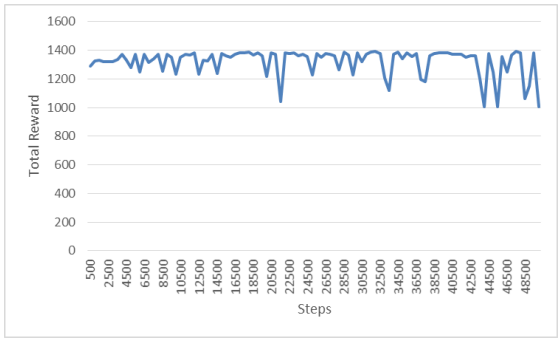


Figure 110. Reward evolution with layers = (32,32)

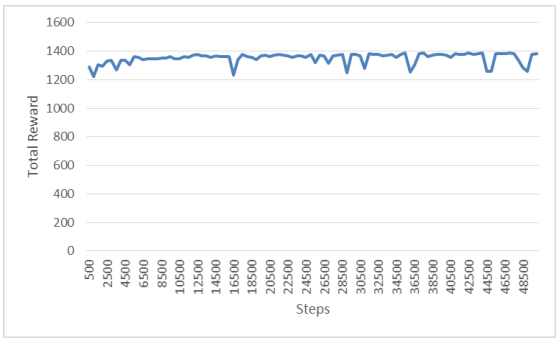


Figure 111. Reward evolution with layers = (100,100)

With (32, 32) the results are not good, so we compare the results with two layers with those of one.

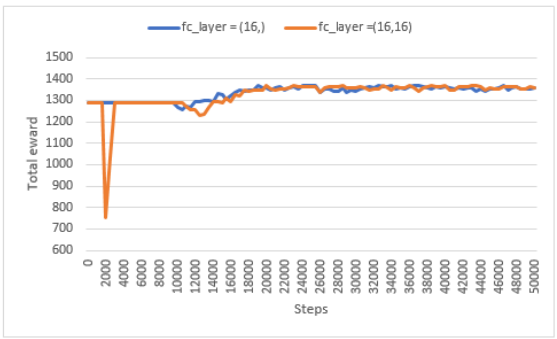


Figure 112. Reward evolution with layers = (16,) vs layers = (16,16)


Figure 113. Reward evolution with layers = (100,) vs layers (100,100)

From this comparison, we see that there is not a big improvement adding a second layer. With (100,100) it seems that the agent obtain a good policy a bit earlier but then in some evaluations the behaviour is not good, with (100,) the policy is always good although it needs more training. Therefore, considering that the more nodes there is, the slower the process, one layer it seems to be more appropriate for our problem.

Finally, we compare the results with (16,) and (100,) to see which is better.

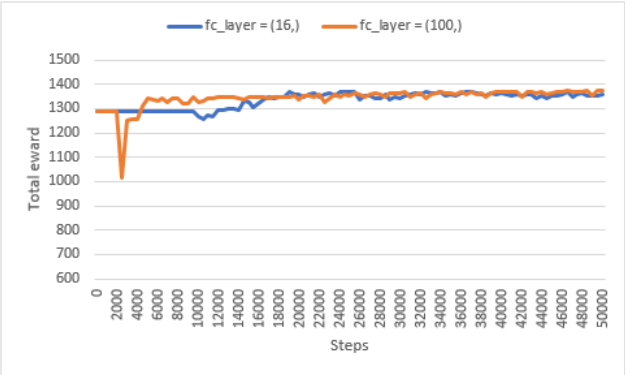


Figure 114. Reward evolution with layers = (16,) vs layers (100,)

We have arrived at the conclusion that (100,) is a good configuration for the DQN agent.

### 5. Initial collect steps

This hyperparameter tells the number of steps done with a random policy before the training starts. To see the impact of this hyperparameter, we look at how the capacity is assigned to each tenant at the evaluation number 25. We have done the simulation with three values: 1, 2.500 and 25.000.

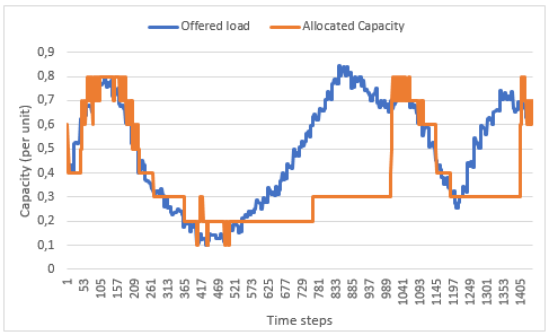


Figure 115. Allocated Capacity of tenant 1. Evaluation 25 with collect steps = 1

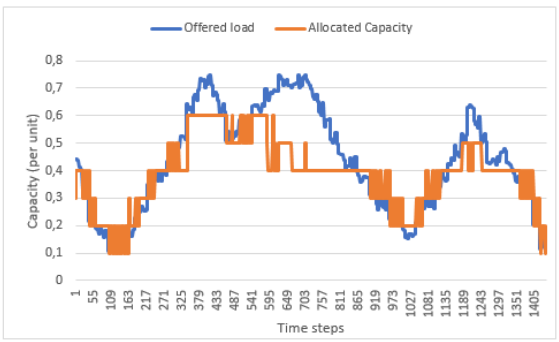


Figure 116. Allocated Capacity of tenant 2. Evaluation 25 with collect steps = 1

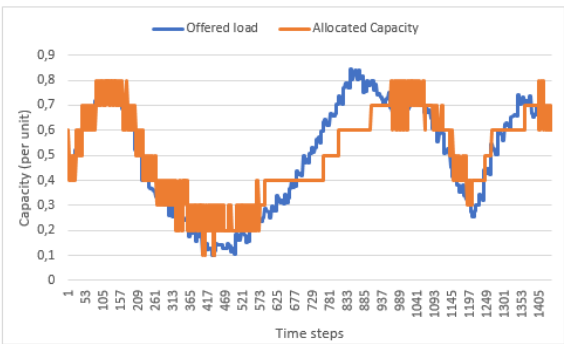


Figure 117. Allocated Capacity of tenant 1. Evaluation 25 with collect steps = 2500

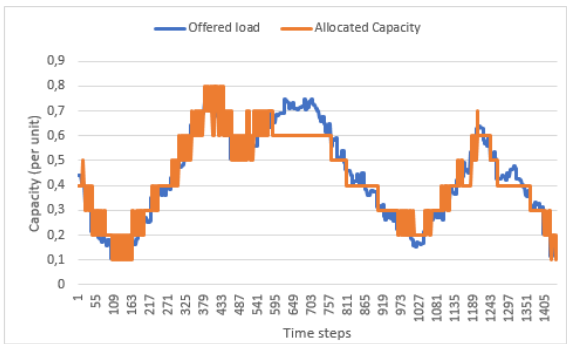


Figure 118. Allocated Capacity of tenant 2. Evaluation 25 with collect steps = 2500

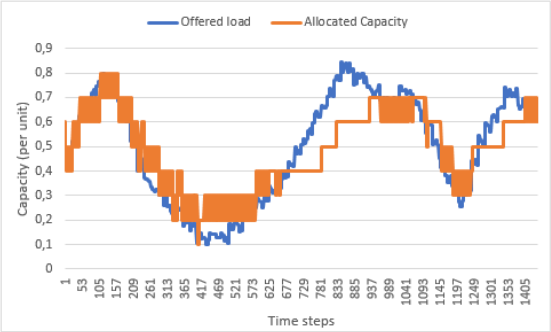


Figure 119. Allocated Capacity of tenant 1. Evaluation 25 with collect steps = 25000

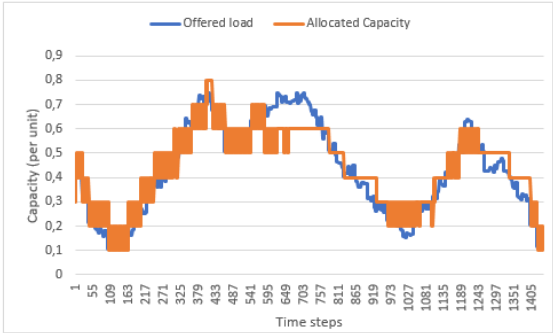


Figure 120. Allocated Capacity of tenant 2. Evaluation 25 with collect steps = 25000

This hyperparameter impact less on the system, but if it is initialized too small the model will need more training to arrive to a good policy. We can see that there is a big difference between the results with 1 and 2500, but increasing more the value does not improve the behaviour. We conclude that we need of some steps at the beginning to gain experience when updating the network, but as we are taking a limited number of samples to update it, collecting so much steps at the beginning is not profitable. Therefore, 5000 could be an appropriate value for this hyperparameter.

### 6. Buffer Experience Replay

This hyperparameter tells the maximum capacity of samples that can be stored in the replay buffer. The analysis has been done for 64 (all the elements of the buffer are taken to update the network), 1.500 and 100.000.

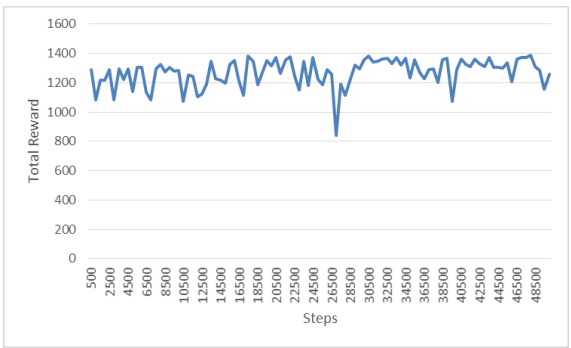


Figure 121. Reward evolution with buffer = 64

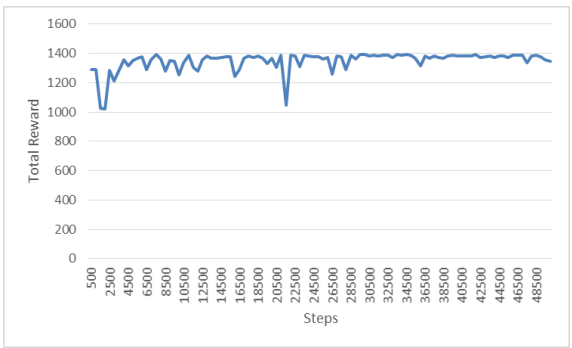


Figure 122. Reward evolution with buffer = 1.500

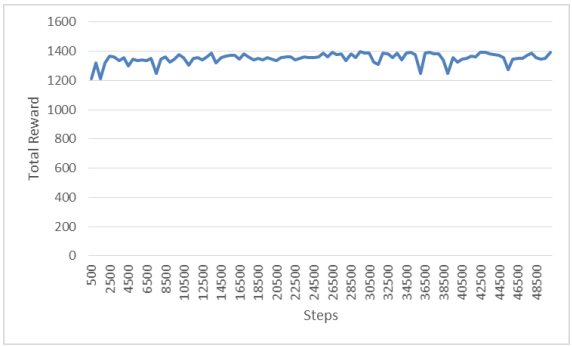


Figure 123. Reward evolution with buffer = 100.000

When initializing this parameter to 64, we always are updating the network with the last 64 samples of experience, so they are correlated and the results are not good. Then, when increasing the value, the buffer has more capacity and can save the experience of the random process of the beginning, so it makes no sense to limit this hyperparameter when we can have a lot of experience stored. Therefore, we decide to set it up to 100.000.

After all the studies of the different hyperparameters, an appropriate value for each of them could be:

| Hyperparameter | Value |
|---|---|
| Learning rate | 1e-4 |
| Batch size | 128 |
| Discount factor | 0.9 |
| Hidden layers | (100,0) |
| Initial collect step | 5.000 |
| Replay buffer max length | 100.000 |

**Table 39. First approach of DDQN hyperparameters**

# Annex 2. DQN convergence study

In this annex is explained the study done to determine the number of iterations needed to consider that the algorithm has converged. In the chapter 4.1 we have seen that it is very important to stop the training when the optimum results are achieved, because if not the model will lead to a problem of overfitting and we will end up with a bad behaviour of the algorithm.

To do this study, first we run a simulation of 250.000 steps to see the evolution. The hyperparameters configuration for this simulation is the obtained from Annex 1, Table 39. From the figure below, it is pretty obvious the necessity of finishing the training. The interval that seems that reach the convergence is what is marked with the red circle, after these steps the policy starts getting worse and worse evaluations.

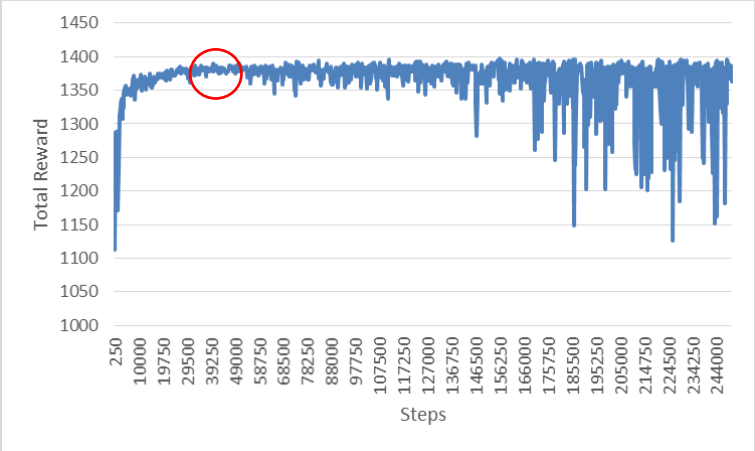


**Figure 124. Total reward evolution**

We enlarged the graph to focus on these steps and to see the total reward more clearly. In the red circle of the figure below are the evaluations that perform a good simulation, they obtain a total reward between 1375 and 1390, which will be the range that we will use to determine that the algorithm has converged.

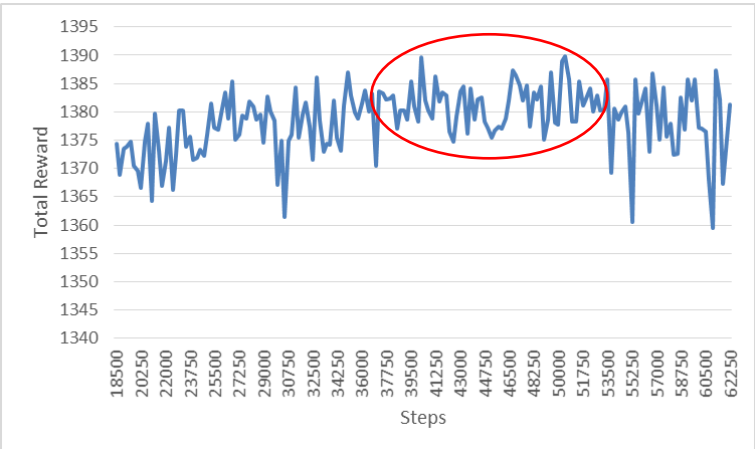


**Figure 125. Expand of total reward evolution**

At this point, we know the values that the total reward has to take. Specifically, the criteria to consider the convergence is: all the evaluations done every 250 steps in an interval X have to be bigger than 1375 and the variation between the last evaluation and all the others inside the interval must be lower than 1% (more or less it is the 1375-1390 interval). After that, we need to know for how much steps the evaluations must be inside this range to consider the convergence. In order to know this, three different simulations have been executed with three different intervals: 5.000, 7.500 and 10.000.

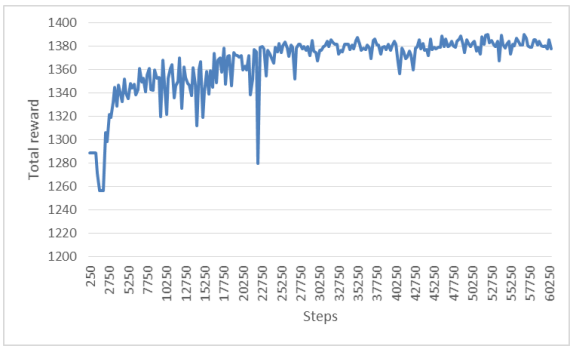


Figure 126. Total reward evolution with convergence interval of 5.000 steps

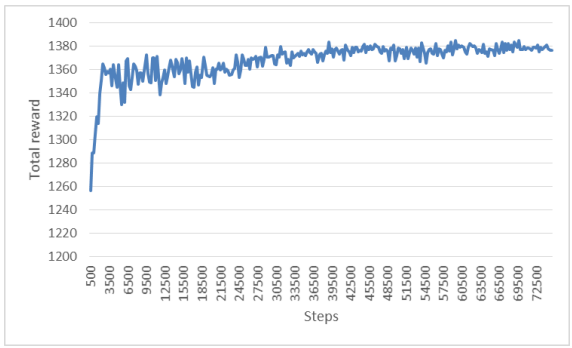


Figure 127. Total reward evolution with convergence interval of 7.500 steps

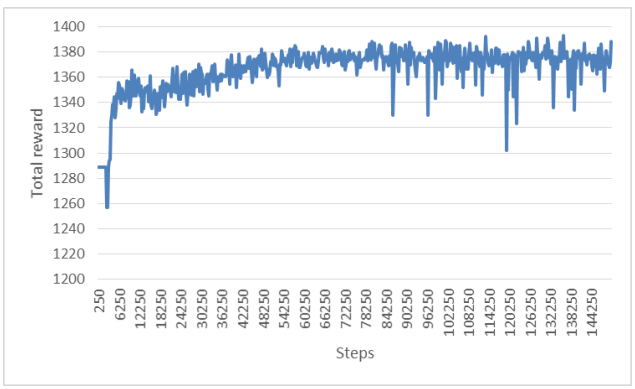


Figure 128. Total reward evolution with convergence interval of 10.000 steps

With both 5.000 and 7.500 intervals the algorithm has reached convergence but with 10.000 hasn't. Therefore, between the two intervals that reach convergence we take the most restrictive, which is 7.500 steps.

# Annex 3. DDPG simulations

In this annex is presented the first simulations done with DDPG to have an idea of which range of values is appropriate for each hyperparameter. The problem with DDPG has changed with respect the DQN/DDQN problem, now the action space is continuous, so the reference values we had for DQN cannot be used.

As a starting point, we take the reference configuration from [29]:

| Hyperparameter | Value |
|---|---|
| Actor learning rate | 1e-4 |
| Critic learning rate | 1e-3 |
| Batch size | 64 |
| Discount factor | 0,99 |
| Hidden layers (L1,L2) | (400,300) |
| Soft update ($\tau$) | 0,01 |

Table 40. Reference hyperparameters for DDPG

From our experience with DQN and DDQN, we have seen that 64 samples to update the network are not enough to achieve good results in terms of convergence time. Thus, we have changed the value of the batch size to 256.

Before studying the rest of hyperparameters, a simulation is done to see its performance and study the steps needed to consider the convergence. The simulation will be running during 125.000 steps, and an evaluation will be performed every 250 steps. The total reward evolution obtained is the following:

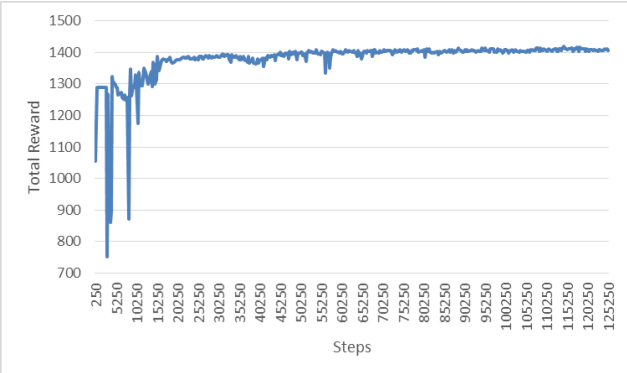


Figure 129. Total reward evolution with DDPG

It needs of some steps to start improving the policy, but then the results are quite good considering that it is the first simulation done. Then, we make a zoom in the steps where the high rewards are achieved to have an idea of the rewards obtained and the steps needed to converge (we plot the last 40.000 steps).

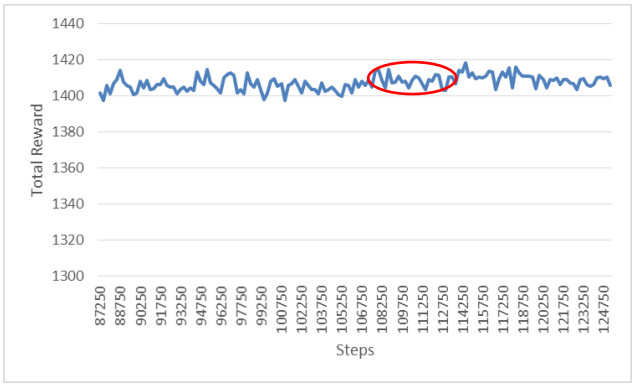


Figure 130. Total reward evolution with DDPG

With DDPG the total reward has been increased to 1400 and more, without exceeding 1420. We take a random evaluation with a total reward of 1405 to study how the capacity has been assigned to each tenant.

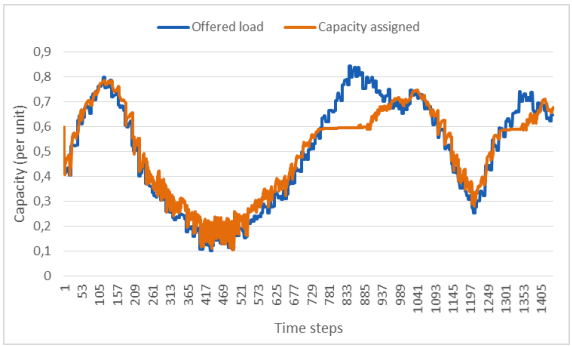


Figure 131. Policy behaviour of tenant 1

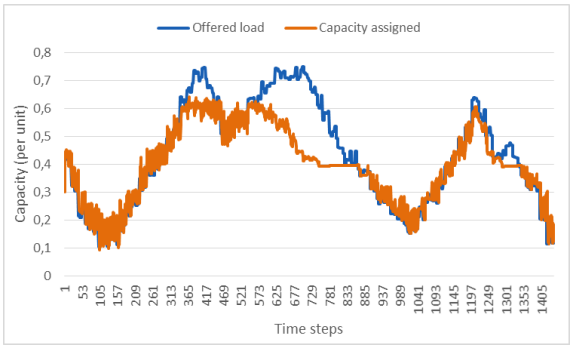


Figure 132. Policy behaviour of tenant 2

The policy obtained is the desired, so a total reward from 1405 upwards can be considered optimum. Therefore, we can consider that the algorithm has converged when the last evaluation is above 1405 and the variation between this evaluation and each of the others within an interval of 12.500 steps is less than 0.75% (±10 points of the total reward more or less). In the figure 128, this interval could be the red circle.

Once we know the criteria to study the convergence time, a methodology is created to study the value of the hyperparameters. The objective, as we said, is to find a good initial set of values for the following studies.


Figure 133. DDPG methodology

The results are shown in the table below It is a big table where all the combinations of the different configurations are presented, no matter if it has converged or not. From this table, we can draw some conclusions about each hyperparameter and it gives an idea of possible good values.

| Step | Actor learning rate | Critic learning rate | Batch size | Discount factor | L1 | L2 | τ | Reward | Dev. | Iterations | Time (s) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0,001 | 0,01 | 256 | 0,9 | 400 | 300 | 0,01 | 1394,80 | 20,76 | 100000 | 7762,43 |
| **0** | 0,0001 | 0,01 | 256 | 0,9 | 400 | 300 | 0,01 | 1368,08 | 39,32 | 62250 | 4845,27 |
| **0** | 0,0001 | 0,001 | 256 | 0,9 | 400 | 300 | 0,01 | 1369,95 | 12,58 | 35500 | 1991,77 |
| **0** | 0,00001 | 0,001 | 256 | 0,9 | 400 | 300 | 0,01 | 1335,48 | 65,23 | 44000 | 3424,03 |
| **0** | 0,00001 | 0,0001 | 256 | 0,9 | 400 | 300 | 0,01 | 1223,74 | 98,75 | 100000 | 7828,43 |
| **1** | 0,0001 | 0,001 | 256 | 0,9 | 200 | 100 | 0,01 | 1334,20 | 42,46 | 99750 | 4311,00 |
| **1** | 0,0001 | 0,001 | 256 | 0,9 | 200 | 300 | 0,01 | 1136,91 | 152,23 | 100000 | 5786,58 |
| **1** | 0,0001 | 0,001 | 256 | 0,9 | 200 | 500 | 0,01 | 1377,17 | 9,98 | 21500 | 1909,52 |
| **1** | 0,0001 | 0,001 | 256 | 0,9 | 400 | 100 | 0,01 | 1390,00 | 24,63 | 85000 | 4283,45 |
| **1** | 0,0001 | 0,001 | 256 | 0,9 | 400 | 300 | 0,01 | 1364,20 | 18,46 | 24500 | 1929,23 |
| **1** | 0,0001 | 0,001 | 256 | 0,9 | 400 | 500 | 0,01 | 1367,38 | 24,26 | 27000 | 3787,79 |
| **1** | 0,0001 | 0,001 | 256 | 0,9 | 600 | 100 | 0,01 | 1317,27 | 62,59 | 57000 | 3870,38 |
| **1** | 0,0001 | 0,001 | 256 | 0,9 | 600 | 300 | 0,01 | 1333,91 | 92,25 | 74250 | 14303,83 |
| **1** | 0,0001 | 0,001 | 256 | 0,9 | 800 | 500 | 0,01 | 1288,58 | 0 | 100000 | 24493,26 |
| **2** | 0,0001 | 0,001 | 256 | 0,85 | 200 | 500 | 0,01 | 1365,27 | 17,37 | 20500 | 1765,01 |
| **2** | 0,0001 | 0,001 | 256 | 0,9 | 200 | 500 | 0,01 | 1366,36 | 18,28 | 26750 | 2308,78 |
| **2** | 0,0001 | 0,001 | 256 | 0,95 | 200 | 500 | 0,01 | 1230,47 | 100,23 | 100000 | 8708,70 |
| **2** | 0,0001 | 0,001 | 256 | 0,99 | 200 | 500 | 0,01 | 1321,10 | 65,40 | 54000 | 4636,71 |
| **3** | 0,0001 | 0,001 | 256 | 0,85 | 200 | 500 | 1 | 1288,58 | 0 | 100000 | 8597,406 |
| **3** | 0,0001 | 0,001 | 256 | 0,85 | 200 | 500 | 0,1 | 1373,46 | 11,85 | 26750 | 2331,48 |
| **3** | 0,0001 | 0,001 | 256 | 0,85 | 200 | 500 | 0,01 | 1300,15 | 96,96 | 33500 | 2911,67 |
| **3** | 0,0001 | 0,001 | 256 | 0,85 | 200 | 500 | 0,001 | 1336,68 | 54,45 | 32250 | 2783,31 |

**Table 41. Results of first configurations studied**

From the table, it is appreciated that some configurations have performed really bad, having an average reward very small or a deviation too big. With other configurations, the policy has not changed in all the simulation, taking always the action [0%, 0%]. Some considerations can be taken from each hyperparameter:

- Actor and critic learning rates seem to be well configured to 1e-4 and 1e-3 respectively.
- Batch size has been considered well defined to 256 and we think that increasing it more will lead to a very long simulation. Nevertheless, its impact has been studied in the methodologies of 4.5.1.
- Discount factor needs to be further studied, but its impact on the model is lower than the other hyperparameters.
- Hidden layers is the hyperparameter that has changed more with respect the other approaches. When defining them very big, it is more difficult to change the policy as there a lot of weights to configure, and the simulation is slowed down. It seems that is better to define L2 with more layers than L1, and (200,500) has been the value obtained as the best. Therefore, the starting point for the following methodologies will be (200,500).
- Soft update hyperparameter has changed the performance very little when testing different values. The only value that shows that it is not a good option, is when setting to 1 (the soft update disappears), as no action has been taken in all the evaluations performed. .

After these aspects, the configuration that could be appropriate as the starting point for the hyperparameters optimization is:

| Hyperparameter | Value |
|---|---|
| Actor learning rate | 1e-4 |
| Critic learning rate | 1e-3 |
| Batch size | 256 |
| Discount factor | 0,9 |
| Hidden layers (L1,L2) | (200,500) |
| Soft update ($\tau$) | 0,01 |

**Table 42. Hyperparameters improvement**

# Glossary

| | |
|---|---|
| **3GPP** | 3rd Generation Partnership Project |
| **5G** | 5th Generation of mobile communications |
| **5G PPP** | 5G Public Private Partnership |
| **5GC** | 5G Core Network |
| **5QI** | 5G QoS Identifier |
| **ARP** | Allocation Retention and Priority |
| **BS** | Base Station |
| **CMC** | Connection Mobility Control |
| **CN** | Core Network |
| **CP** | Control Plane |
| **DRB** | Data Radio Bearers |
| **DDPG** | Deep Deterministic Policy Gradient |
| **DDQN** | Double Deep Q Network |
| **DNN** | Deep Neural Network |
| **DP** | Dynamic Programming |
| **DPG** | Deterministic Policy Gradient |
| **DQN** | Deep Q Network |
| **DRL** | Deep Reinforcement Learning |
| **eMBB** | enhanced Mobile Broadband |
| **FDMA** | Frequency Division Multiple Access |
| **FR** | Frequency Range |
| **GBR** | Guaranteed Bit Rate |
| **gNB** | New Generation Node |
| **IP** | Internet Protocol |
| **KPI** | Key Performance Indicator |
| **LTE** | Long Term Evolution |
| **L1** | Layer 1 |
| **L2** | Layer 2 |
| **L3** | Layer 3 |
| **M1** | Methodology 1 |
| **M2** | Methodology 2 |
| **M3** | Methodology 3 |
| **MABR** | Maximum Aggregated Bit Rate |
| **MAC** | Medium Access Control |
| **MDP** | Markov Decision Process |
| **mMTC** | Massive Machine Type Communications |
| **NF** | Network Functions |
| **NG-RAN** | New Generation RAN |
| **NN** | Neural Network |
| **OFDMA** | Orthogonal Frequency Division Multiple Access |
| **PLMN** | Public Land Mobile Network |
| **PRB** | Physical Resource Block |
| **PS** | Packet Scheduling |
| **RAC** | Radio Admission Control |
| **RAN** | Radio Access Network |
| **RB** | Radio Bearer |
| **RBC** | Radio Bearer Control |
| **RL** | Reinforcement Learning |
| **RRC** | Radio Resource Control |
| **RRM** | Radio Resource Management |
| **QoS** | Quality of Service |
| **SAGBR** | Scenario Aggregated Guaranteed Bit Rate |
| **SARSA** | State Action Reward State Action |
| **SIB** | System Information Block |
| **SLA** | Service Level Agreement |
| **TD** | Temporal Difference |
| **UE** | User Equipment |
| **UP** | User Plane |
| **URLLC** | Ultra Reliable and Low Latency Communications |
| **VNF** | Virtualized Network Functions |