UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

telecos BCN

# Applying Reinforcement Learning to Network management

---

Master Thesis

submitted to the Faculty of the

Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

by

Raúl Gonzalez Infante

In partial fulfilment

of the requirements for the master in

**TELECOMMUNICATIONS ENGINEERING**

Advisor: Prof. Juan Luis Gorricho Moreno

Barcelona, July 2022

# Abstract

This project seeks to find if in the actual scenario Reinforcement Learning could help Vehicle Networks to get better performances, concretely applied in the field of resource allocation. It would be tried to allocate a varied number of requests in a network with multiple datacenters, modeling an actual road and city track. To do so, 4 algorithms were implemented, a heuristic and 3 RL approaches, in which we defined a simple DQN and the remaining two that run the same DQN but also include a parameter sharing method. It will be seen that a more sophisticated model must be done in order to demonstrate that Reinforcement Learning is worthwhile, and also, that parameter sharing is a tool that would be very useful for these types of networks as it could work in a very efficient manner.

# Acknowledgements

I would like to thank my tutor Juan Luis Gorricho Moreno for giving me the opportunity to do this project and for the large amount of good advice and ideas given during this last semester.

Also 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 | 16/05/2022 | Document creation |
| 1 | 18/06/2022 | Document revision |
| 2 | 22/06/2022 | Document revision |
| | | |

| Written by: | | Reviewed and approved by: | |
|---|---|---|---|
| Date | 22/06/2022 | Date | 23/06/2022 |
| Name | Raúl González Infante | Name | Juan Luis Gorricho Moreno |
| Position | Project Author | Position | Project Supervisor |

# Table of contents

# List of Figures

# List of Tables

# 1. Introduction

Machine learning (ML) is a field of computer science and statistics that includes multiple algorithms and methods to learn from data and make predictions, which became one of the most important research directions of multiple areas due to its big potential and good results. For example, it is widely used in image processing, speech recognition, robot control, and telecommunications having proved their efficiency and sometimes, getting better results than humans. ML can be categorized into the following categories depending on how learning is done [2] :

- Supervised learning
    - “Supervised learning algorithms build a mathematical model of a set of data that contains both the inputs and the desired outputs.” [1]
- Unsupervised learning
    - “Unsupervised learning algorithms take a set of data that contains only inputs, and find structure in the data, like grouping or clustering of data points.” [1]
- Semi-supervised learning
    - “Some of the training examples are missing training labels, yet many machine-learning researchers have found that unlabeled data, when used in conjunction with a small amount of labeled data, can produce a considerable improvement in learning accuracy.” [1]
- Reinforcement learning (RL)
    - Reinforcement learning consists of a learning process where an agent makes decisions and interacts with the environment to obtain a reward, with the objective of finding the best policy to maximize the profit [3].

Nowadays, RL is being used and investigated in multiple areas of networking such as Internet of Things, Heterogeneous Networks, Unmanned Aerial Vehicles and these types of applications that connects multiple devices (that can be interpreted as agents) and interact with each other vía Internet, private networks, or similars (interpreted as the environment). As in the last few decades, there is no doubt that communications demands and network complexity has grown rapidly due to the necessity of higher throughput, lower latency and high security requirements, networks became unmanageable for other techniques such as Dynamic programming and Markov Decision Process modeling [4].

Instead of the great variety of challenges, the paper only focuses on Vehicular Networks, a topic that is becoming popular as a consequence of the starting implementation of 5G (which will enable faster response times), Vehicular Ad-Hoc Networks (VANETs), Car Manufacturers and Municipal Transport Authorities that are promoting safe navigation [5]. Inside this topic we can find multiple services:

- Content downloading
- P2P location significant advertising
- P2P (driver to driver) interaction
- Sensing the environment

It can be seen that the network would be extremely dynamic and with a big load of charge, so it is important to find a method to distribute the flux of the network in the optimal way in order to make all the devices work correctly and not saturate the whole network.

## 1.1. Objectives

- Create a simulator that can model properly a vehicle network
- Find an algorithm for all the agents that could manage all the requests optimally and maximize the obtained reward.
- Compare Reinforcement Learning with Heuristics algorithms
- Find if DQN parameter sharing could be useful for a Vehicle Network

## 1.2. Work Plan and deviations

The planification of this project was divided into several parts, the first one consists of doing some research and identifying the main issues of Vehicle Networks and understanding the theory inside Reinforcement Learning. The second part includes the design of the simulator, its implementation and the verification that works as specified, followed by setting the conditions of the multiple environments we want to perform and once it is done, execute these parameters on the simulator in order to get the results. Finally, an analysis of the results must be done and be written on the final report, with all the background information and specifications of the system.

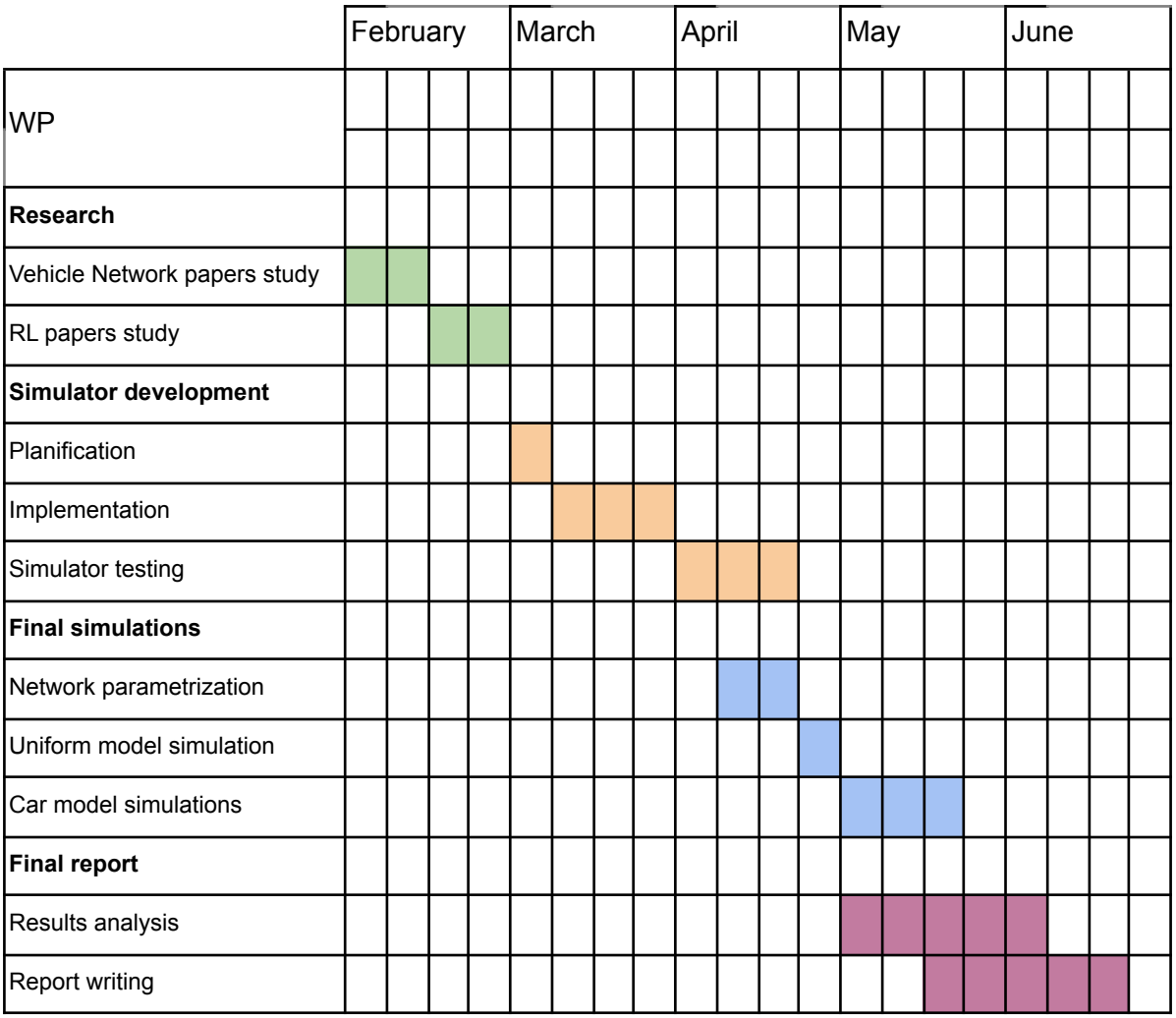


Figure 1. Gantt diagram

# 2. State of the art

## 2.1. Resource allocation

It can be agreed that what we are dealing with in this paper is a resource allocation problem [7], as we are looking to optimally distribute the requested resources of a vehicle network that has limited capacity. There are multiple methods to solve this type of problems, but can be summarized in this three [6]:

- Traditional optimization methods
- Deep Learning assisted optimization methods
- Deep Reinforcement Learning based approaches

### 2.1.1. Traditional methods

Resource allocation problems are commonly solved by mathematical programming, where a model is considered and its parameters are optimized to minimize or maximize (depending on the design) a function of interest. The thing is that “Except in a few simple cases, where we are fortunate enough to end up with convex optimization that admits a systematic procedure to find the global optimum, most optimization problems formulated for wireless resource allocation are strongly non-convex. No known algorithm can solve the problem to optimality with polynomial time complexity.” [6]. To partially solve these big complexities, it is often accepted to find a local optimal solution that does not guarantee an optimal performance but can obtain good results [See Figure 2].

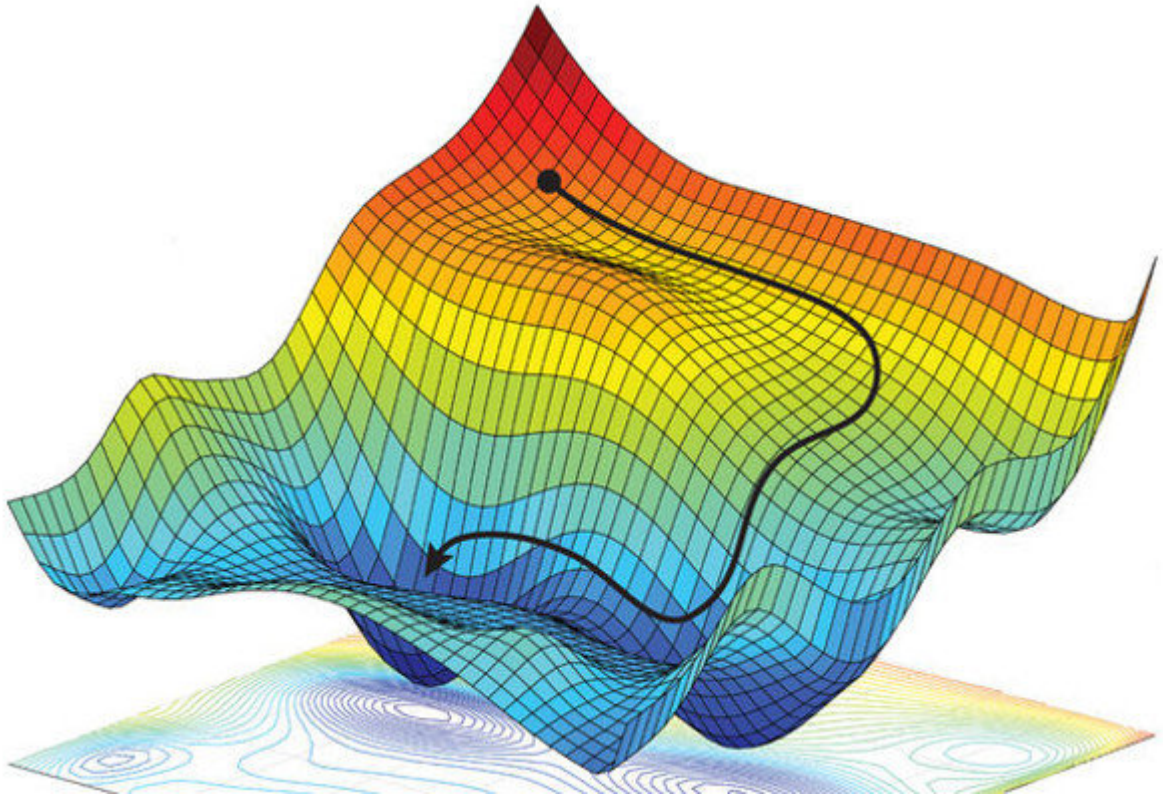

Figure 2. Non-convex optimization. Stochastic gradient descent used to find a local optimum in a loss landscape.

But apart from complexity, the world of networking is very extended and versatile, which means that not all the performance metrics are reliable for all the applications and sometimes do not admit an obvious formulation, so it is difficult to obtain an accurate model which lacks the performance of the found solution.

### 2.1.2. Deep learning assisted

This follows the same idea as traditional methods, but in this case, Neural Networks are used to obtain the optimal parameters. Knowing that neural networks are trained to minimize an output from the ground truth given an input, if we set our objective function as the loss function from the Neural Network, we can directly maximize or minimize our optimization objective. The main advantage of this approach is that, from new parameters, a good solution can be calculated almost instantly, so it can be implemented in real-time.

"Alternatively, deep learning can be embedded as a component to accelerate some steps of a well-behaved optimization algorithm, such as the pruning stage of the branch-and-bound (B&B). This method leverages the theoretical models developed with expert knowledge and achieves near-optimal performance with significantly reduced execution time." [6]

### 2.1.3. Deep Reinforcement Learning

"RL addresses sequential decision making via maximizing a numerical reward signal while interacting with the unknown environment, as illustrated in Figure 3. Mathematically, the RL problem can be modeled as a Markov decision process (MDP). At each discrete time step t, the agent observes some representation of the environment state St from state space S, and then selects an action At from the action set A. Following the action, the agent receives a numerical reward Rt+1, and the environment transitions to a new state St+1, with transition probability p(s, r|s, a). In RL, decision-making manifests itself in a policy π(a|s), which is a mapping from states in S to probabilities of selecting each action in A. The goal of learning is to find an optimal policy π∗ that maximizes the expected accumulative rewards from any initial state s." [6]

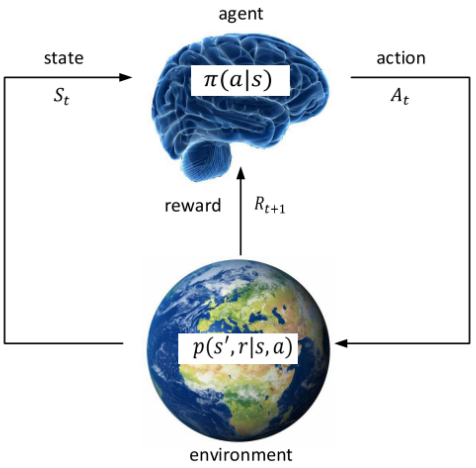


Figure 3. Agent - Environment interaction in RL

Unless there are multiple RL algorithms, here we briefly explain the one that will use on this paper:

#### 2.1.3.1. DQN

This algorithm is based on the Q-learning algorithm, in which we compute the Q-table which contains the Q-values of any state-action, and when an action has to be taken, we look at the table and see which action gives us a bigger reward. But the Q-learning algorithm works well for finite states and action spaces because, since we store every state-action pair, this would mean that we need a huge amount of memory to store all of them and much more iterations for the Q-table to converge [17]. That is why the idea of applying Neural Networks is interesting because for each state we can approximate the Q-values and choose the highest one without the necessity of storing all the state-action pairs. The algorithm follows Figure 4.

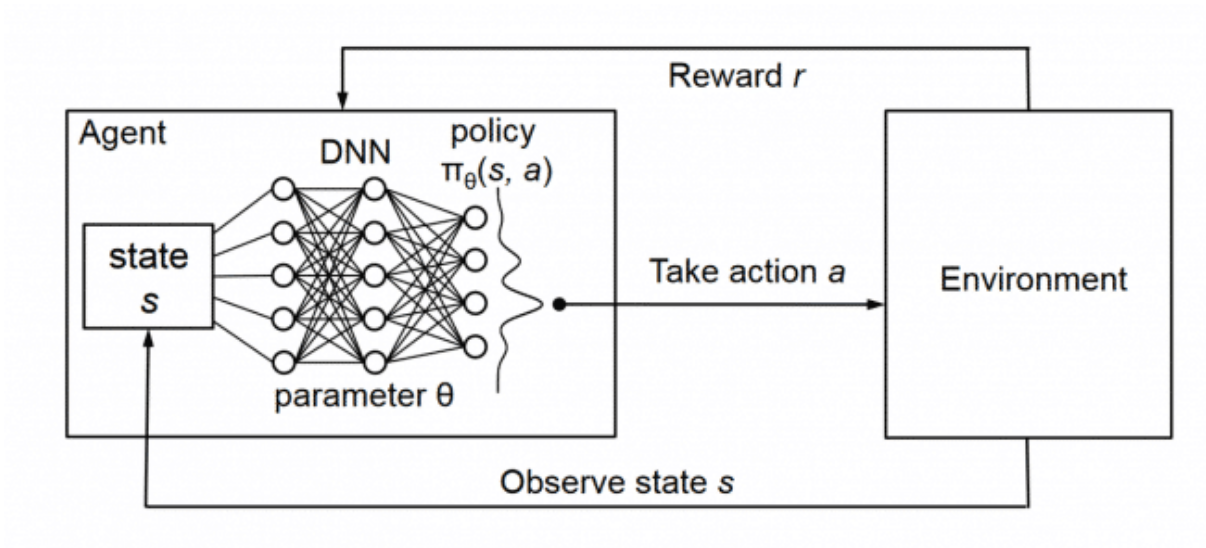


Figure 4. DQN algorithm

And how is the DNN updated? To train our DNN, we use a technique called Replayed Memory. The idea is that the agent stores all its experiences in a memory buffer called a replayed memory buffer. At time step t, the experience is a tuple containing the current state of the environment, the chosen action, the reward, and the next state of the environment:

$$e_t = (s_t, a_t, r_{t+1}, s_{t+1})$$

After each episode, the agent samples a batch of experiences from the replayed memory and uses them to train the DNN. During the training, we use as a loss function the Temporal Difference error function (TD function), which is the difference between the Q-value of a state-action pair and its Q-Target.

As the Q-Target is unknown, we use once again the Bellman optimality equation that we recall:

$$q_*(s,a) = E\left[R_{t+1} + \gamma \max_{a'} q_*(s',a')\right]$$

Where s' is the next environment state, R is the reward, and finally, to compute the highest Q-value from state s', we only need to forward s' through the DNN and get the highest output value. [17]

## 2.2. Multi Agent System (MAS)

"The world is witnessing a sudden shift in the paradigm of technology moving from centralized to a decentralized approach. A centralized approach leads to a single point of failure if any fault occurs and hence a whole system comes to rest. Hence, a decentralized approach like the Multi-Agent System is trending nowadays. A MAS is several software entities (agents) working together in pursuit of specified tasks." [14] In our case, we will study a MARL, a specific case where agents use Reinforcement Learning to refine their policies.

### 2.2.1. Multi Agent Reinforcement Learning (MARL)

"The goal of MARL algorithms is to learn a policy for each agent such that all agents together achieve the goal of the system. Particularly, the agents are learnable units that aim to learn an optimal policy on the fly to maximize the long-term cumulative discounted reward through the interaction with the environment. There are several properties of the system that is important in modeling a multi-agent system:" [13]

- Centralized or decentralized control
    - Centralized: A central unit takes the decision for each agent
    - Decentralized: Each agent takes the decision itself
- Fully or partially observable environment
    - Fully observable: Agents can see all the information from the environment
    - Partially observable: Agents only are allowed to see some information from the environment

- Cooperative or competitive environment
    - Cooperative: Agents collaborate to achieve a goal
    - Competitive: Agents competitive between them to achieve a goal

The system on this paper would be decentralized as each agent would take its own decision, partially observable as agents only would see their servers capacity and summarized information from the other agents and cooperative, as the goal of the system is to process the maximum amount of data.

Apart from that, as said before, each agent will try to send a simplified report of its state, so we need to introduce a new field of research called “Learning to communicate”

#### 2.2.1.1. Learning to communicate

Communication is an important factor for the big multi-agent world to stay organized and productive, indeed, for applications where an individual agent has limited capability, it is particularly critical for multiple agents to learn communication protocols to work collaboratively.

##### *2.2.1.1.1. Reinforced Inter-Agent Learning (RIAL)*

Think about how you would make multiple DQN communicate between them, probably you would think about making them generate a message and send it to the other one each time you take an action. So that way is how RIAL was implemented, a DRQN (a variation of the DQN algorithm that implements memory thanks to the use of Recurrent Networks [20]) each time takes an action, it generates and later sends a message to the other DRQNs. In the next step, this message will be used as an additional input.

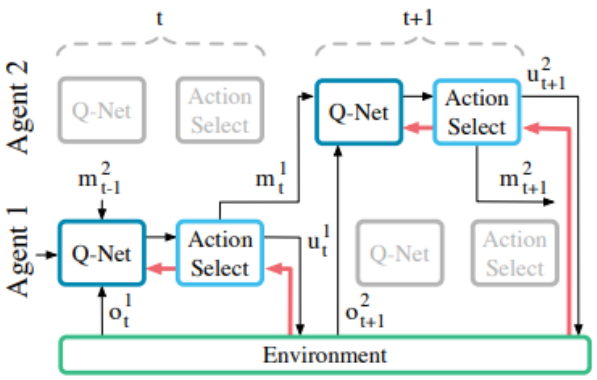


Figure 5. RIAL - RL based communication

*2.2.1.1.2. Differentiable Inter-Agent Learning (DIAL)*

"While RIAL can share parameters among agents, it still does not take full advantage of centralized learning. In particular, the agents do not give each other feedback about their communication actions. Contrast this with human communication, which is rich with tight feedback loops. For example, during face-to-face interaction, listeners send fast nonverbal queues to the speaker indicating the level of understanding and interest. RIAL lacks this feedback mechanism, which is intuitively important for learning communication protocols.

DIAL works as follows: during centralized learning, communication actions are replaced with direct connections between the output of one agent's network and the input of another's. Thus, while the task restricts communication to discrete messages, during learning the agents are free to send real-valued messages to each other. Since these messages function as any other network activation, gradients can be passed back along the channel, allowing end-to-end backpropagation across agents." [19].

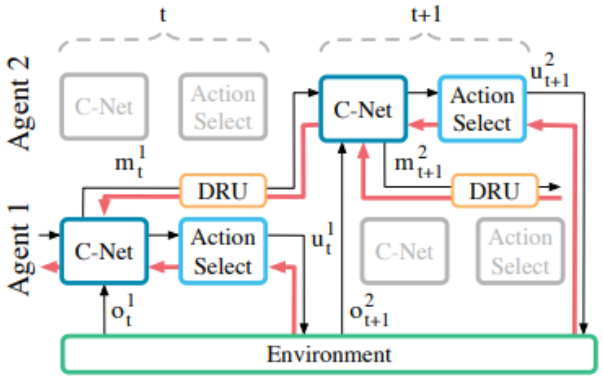


Figure 6. DIAL - Differentiable communication

Recognise that DIAL uses the same idea as RIAL but with the only difference that here the gradient flows between agents.

# 3. System model and simulator development

In this chapter we will see how the simulator was though and how it works, for better understanding it is divided into two sections:

- System model
    - Explains the main idea of the environment behavior, with the basic requirements and basic parameters that will be explained in detail at the following section.
- Simulator development
    - Detailed description of the simulator structure, organization of the multiple environments, model parameters, and organization of the final code.

## 3.1. System model

Let's start with the explanation of the main idea behind all the environments and some important aspects that will make the understanding of the whole simulator easier.

### 3.1.1. Idea

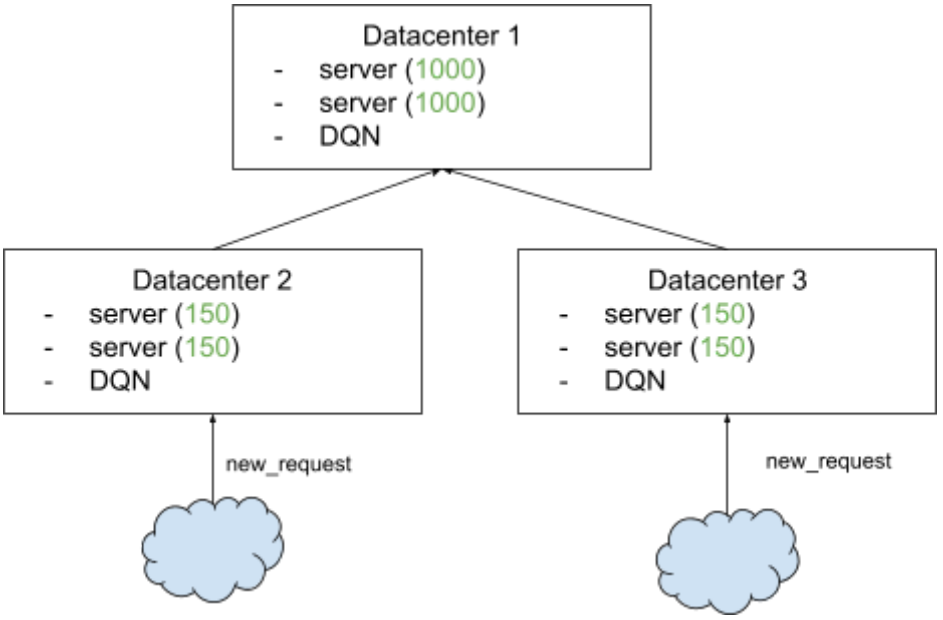


Figure 7. Network example

The basic idea is to have a simulator able to generate a network of multiple Datacenters (or devices) with one or multiple layers, easy to configure, and with the following features:

- The number of datacenters per layer and the number of layers must be configurable.
- New requests are generated by the environment and are passed to all the datacenters of the first level.
- Each datacenter must contain a configurable number of servers with individual configurable capacities.
- A bottom datacenter can resend a request, if needed, to any top device in which there exists a link between them.
- Each Datacenter must implement an heuristic or a DQN (the method must be configurable) that decides what to do with a new request.

### 3.1.2. Requests

Each petition has 3 parameters:

- Size: Number of resource units that need from the server. For example, if a server has available 1000 resource units (as the example in Figure 7) and it decides to accept the requests with a size of 100, the left available resources are 900.
- Duration: Number of steps that this request keeps the resources occupied.
- Max hops: Number of maximum hops the request can do (default is set as the maximum possible).

### 3.1.3. Request flow

For each received request, the DQN has to decide between 3 different options:

- Accepted the request

  After accepting the petition, it is sent to the servers to try to allocate the petition
  - If there are enough available resources and it is allocated, the petition is confirmed
  - If there are no available resources, the request is rejected
- Discard the request
  - Discarded request
- Request redirected to an upper datacenter
  - The request is sent to an upper datacenter that has to decide between accepting, discarding or resending the request again (if it is possible). Meanwhile, the datacenter waits for an answer (the reward or the penalty) from the top entity.

## 3.2. Simulator development

Now that we have a general idea of how everything works, let's explain in detail how each environment works, how we defined the algorithms and how we set the multiple testing zones.

### 3.2.1. Maps - Environments

At first, the two testing environments are explained. The first one was used to check that everything was working as expected, try different environment parameters and multiple algorithm implementations in order to know if it could be useful or not. And the second one that it was used to test, get statistics and results from the final algorithms.

#### 3.2.1.1. Pilot map

This map has only 3 datacenters, the first one on the top and the two remaining are at the bottom, both connected with the highest level, following Figure 8. The number of servers for each datacenter and its capacities are parameters that were changed during multiple experiments. The multiple variations and results can be found in Section X.

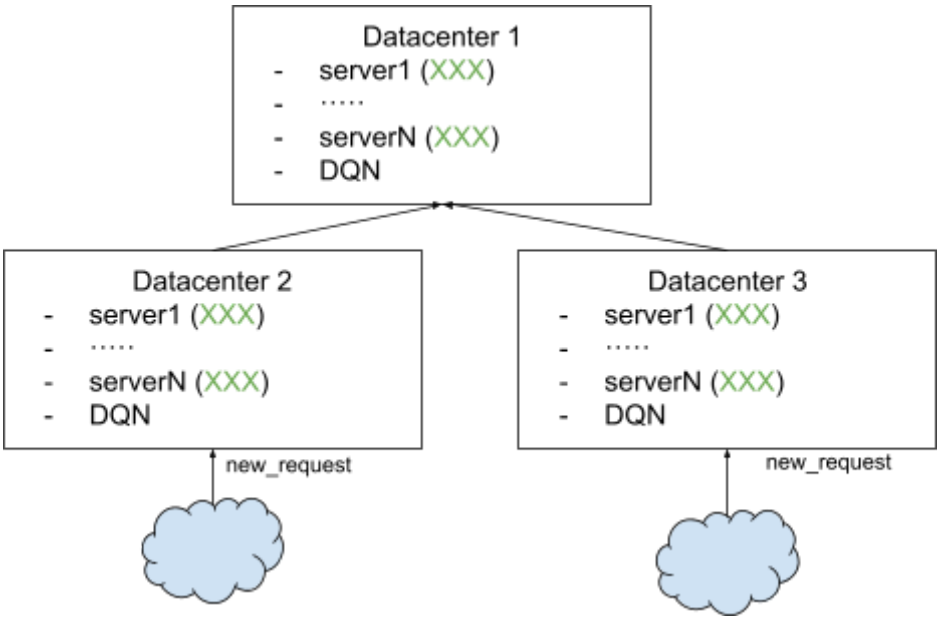


Figure 8. Pilot map schema

#### 3.2.1.2. Final map

It is composed of 9 datacenters, 1 on the top, 2 in the middle, and the remaining 6 at the bottom. Connections are the same as Figure 9 indicates, with all the parameters of the network architecture fixed.

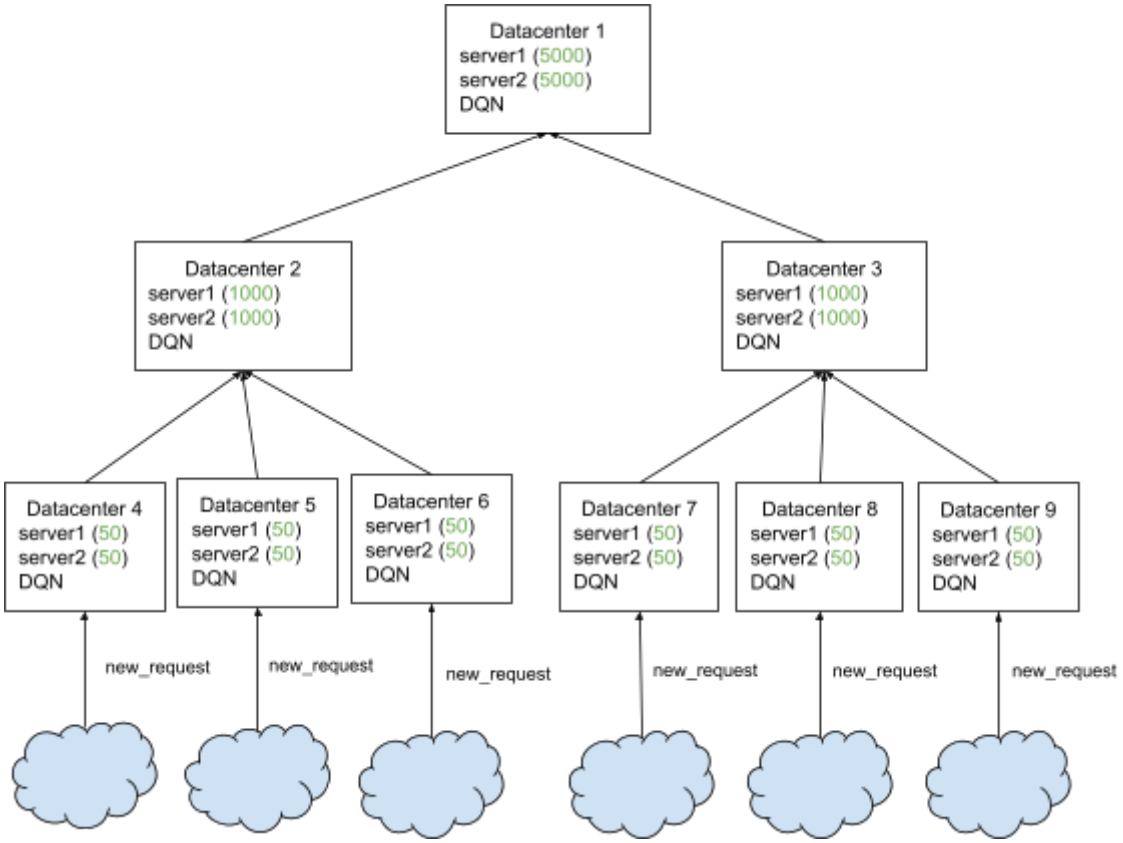


Figure 9. Final map schema

### 3.2.2. Request models and statistics

Once it is clear how we set the multiple environments, here we define how requests are generated and which statistics models they follow.

All the requests have two main parameters:

- REQUEST_MAX_SIZE: Maximum size a request can have
- REQUEST_MAX_DURATION: Maximum duration a request can have

And attributes *size* and *duration* from requests are generated with this constraint following different statistics and models explained below.

#### 3.2.2.1. Uniform model

Following a uniform distribution [See Figure 10], both size and duration attributes are generated between 0 and its maximum value.

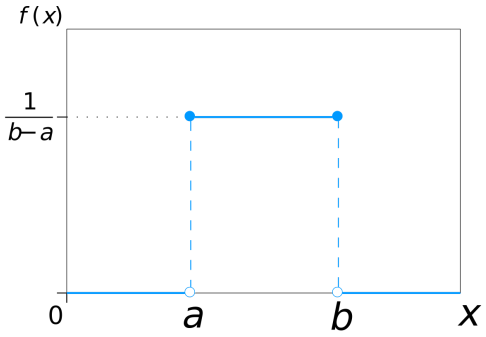


Figure 10. Uniform distribution

#### 3.2.2.2. Car model

It searches to simulate a realistic model for the requests to get more realistic results. To do that we add an additional parameter called max_hops which indicates the maximum number of hops the request can do, if this number is exceeded, the request is deleted. With that, we try to model this request that needs fast reply. The rule to generate this new attribute is the following:

- max_hops: The default value of this parameter is equal to the size of the network but, if the size of the request is <100, it has a certain probability (look at the end of Road Model and City Model sections) of having only 1 hop permitted.

For the rest of the attributes, we have the following changes:

- size: Uniform distribution
- duration: It was differentiated 2 different models, the Road model, and the City model, in which the first one models a track of the road that is near to a city and the second one that corresponds to the city center. More information is given in the following section.

##### *3.2.2.2.1. Duration statistic*

In order to model the duration, we differentiate between a normal traffic situation (called “Free” state) and a traffic jam situation (called Jam state). After analyzing multiple distributions, it was agreed to use the 1-beta distribution with the following parameters:

- Free state: beta distribution with α= 2 and β = 8 [See Figure 11]
- Jam state: We use the complementary distribution of the free state.

Going from one state to another is not instantaneous, we decrease the β parameter accordingly until we arrive at β = 2 [See Figure 12], the function is negated, and then we start increasing the β again until it has arrived at 8, which means that the transition is completed.

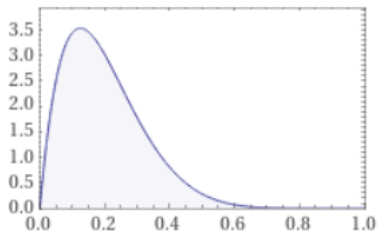


Figure 11. beta distribution(2,8)

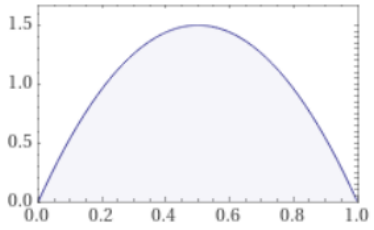


Figure 12. beta distribution(2,2)

And now we have to decide how many steps we want to be in a free state on how many in a Jam state for a complete day. For that purpose, we considered two different models, the road model and the city model.

#### *3.2.2.2.1.1. Road model*

It searches to simulate a realistic model for requests that will be found on a track near a city like Barcelona.

With the help of Google Maps, it was calculated which percentage of the time the road was busy and which was free [See Figure 13].

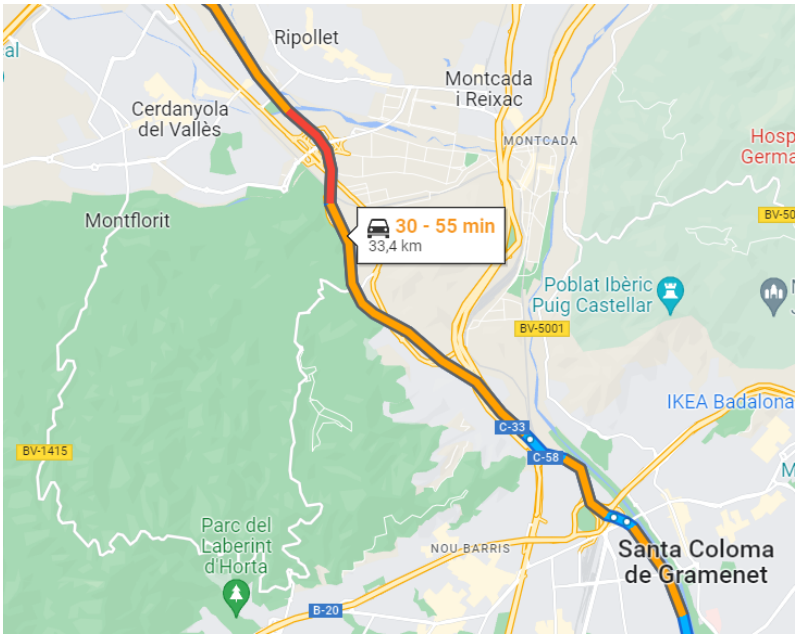


Figure 13. Google Maps indicating a traffic jam

It was determined the following time slots during a complete effective day:

- 5 - 8 → Free state
- 8 - 11 → Jam state
- 11 - 17 → Free state
- 17 - 19 → Jam state

Note that the day starts at 5 AM and finishes at 7 PM, because it was considered that out of this schedule it is not useful to have an intelligent system, as everything is quiet.

With the data below, it is considered that 70% of the time we are in Free state and 30% we are in Jam state, so the final distribution appears in Figure 14.

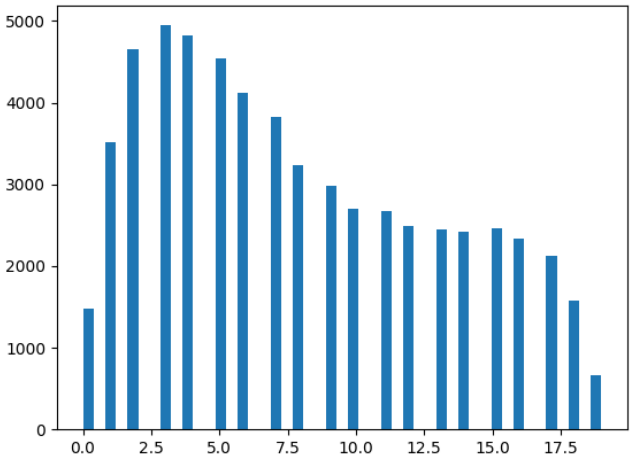


Figure 14. Road model duration statistics

Finally, the probability that a request has max_hops = 1 is determined to 50%

*3.2.2.2.1.2. City model*

The same was done with a track on the city center and the following slots were found:

- 5 - 7 → Free state
- 7 - 12 → Jam state
- 12 - 16 → Free state
- 16 - 20 → Jam state

Approximately it is 50% Jam state and 50% Free state.

Finally, the probability that a request has max_hops = 1 is determined by the 20%

### 3.2.3. Case studies

With all the requests statistics and environments explained, now it is time to explain all the reasons behind the study cases.

#### 3.2.3.1. Background

As said in the Objectives section, one of the main purposes of this study is to find an algorithm or an architecture that enables datacenters to manage optimally all the requests and try to improve the performance that we obtain from the implementation of a simple heuristics or a DQN. In order to do that, as seen in the State of the art section, it was realized that using communication between DQNs or parameters sharing was a good approach to get better performance, so a new architecture was designed based on the parameters sharing idea taking into account the data privacy factor, because in a vehicular network not always the users would allow giving their data to the other users, both from the customer's side and the provider's side.

In order to compare this new architecture, called by us Shared DQN, with the rest of the algorithms, we also have 3 more implementations:

- A simple heuristic
- A basic DQN typically used for OpenAI Gym tasks
- A “optimal” DQN implementation which is able to see all the parameters of the network

Each algorithm will work in two different scenarios, the first one that is little and useful for testing and checking that everything is well programmed and the second one that is much bigger and more difficult to train, but in which we can get more realistic results. In addition, different request statistics were made in order to get different simulation conditions.

#### 3.2.3.2. Heuristic

In the state-of-the-art section we saw that Deep Reinforcement Learning is very helpful for these types of cases, but it was also implemented a simple version that does not include Deep Learning in order to see the real gain.
This algorithm checks if it has enough resources to process the new request, if so, it accepts the request, if the resources are not enough, the request is sent to the top datacenter.

#### 3.2.3.3. DQN

A simple DQN algorithm whose hyperparameters (learning rate, batch_size…) were optimized to solve multiple OpenAI Gym environments, but it was adapted to work on this environment.

The main features of the DQN Neural Network are:

- Layers
  - 1rst dense layer
    - 10 neurons
    - Activation function: Hyperbolic tangent function
  - 2nd dense layer
    - 10 neurons
    - Activation function: Hyperbolic tangent function
  - Final dense layer
    - Same number of neurons as outputs
- Squared error
  - $(target - observation)^2$
- RMSProp algorithm as optimizer

A summary of the inputs and outputs is described below:

- state: The state is composed by the information of the local server's occupations (note that in this case each datacenter has 2 servers) and all the parameters of the requests.
  - [server1_occupation, server2_occupation, req_size, req_duration, req_max_hops]
  - All the data is normalized before entering into the DQN
- action: The number of actions is 2 plus the number of top datacenters in which have a connection, as it corresponds to Accept, Deny or resend to the corresponding datacenter. Normally, as we only have 1 top datacenter, the number of actions is 3.
  - 0 means accept the request and try to allocate on its servers
  - 1 means deny request
  - 2 means resend the request to the first visible datacenter
  - 3 means resend the request to the second visible datacenter
  - ….
  - N means resend the request to the (N-1) visible datacenter

- Reward: Reward depends on the size of the request and it is positive if the request is accepted and negative if it is denied. In addition, the reward also depends on the redirected fact, because for each time it becomes redirected, for the local datacenter it reduces its value to its half if accepted and doubled if rejected. For example, in case of having Datacenter1 (top) and Datacenter2 (bottom) and a request being redirected, if the top one accepts the petition, the reward would be equal to the size of the request for DC1 and a half of req_size for DC2. In case of being rejected, the penalty is - req_size and - (req_size * 2) respectively, everything with a max value of 1 and a minimum of -1.

  A summary of the reward protocol is described below:
  - Direct request (Request that comes from the lower level)
    - Accepted and allocated by the server
      - request_size / request_max_size [Reward]
    - Accepted but denied by the servers
      - - (request_size / max_request_size) [Penalty]
    - Deny
      - 0
  - Indirect request (Request processed by the top datacenter)
    - Accepted
      - up_reward / 2 [“up_reward” → reward from the top device]
    - Rejected
      - up_reward * 2 [“up_reward” → penalty from the top device]

#### 3.2.3.4. Optimal DQN

It is the same implementation as the normal DQN but this new type of DQN has information about the whole system, concretely all the servers occupation, so it is a way to get an “optimal” solution with parameters sharing. In this case, the input of the DQN contains the information of all the servers.

- State
  - [global_servers_occupation, local_servers_occupation, request_state]

Observation: request_state includes size, duration and max_hops.

#### 3.2.3.5. Shared DQN

I got the Shared DQN idea from a couple of papers that use Reinforcement Learning techniques for autoencoding information[21][22], so it was thought to autoencode the environment information with a global DQN in order to send this information to the datacenters and make them know the general state without giving them the real information and fulfill with the privacy requirements.

As said before this new Shared DQN architecture [See Figure 15] is different from the others, here we have 2 types of DQN:

- Shared DQN: This DQN belongs to the environment and is in charge of encoding the global information.
- Datacenter DQN: Normal DQNs that are on datacenters taking decisions about accepting requests or not.

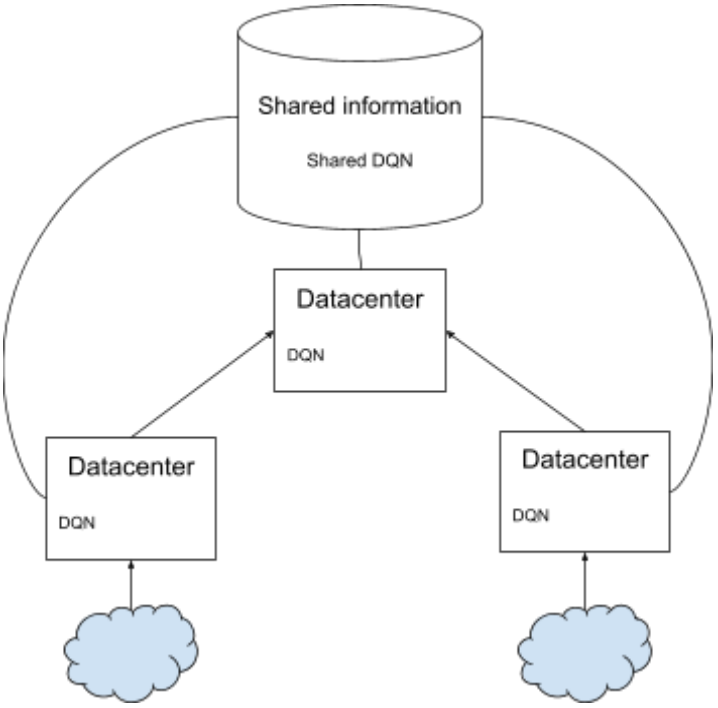


Figure 15. Shared DQN architecture with only 3 datacenters

The Shared information is an array with a length equal to the number of datacenters, where each one has its position assigned. For example in Figure 15, we have 3 devices, so the shared information would be:

shared_info = [shared_number_dc1, shared_number_dc2, shared_number_dc3].

Each time a datacenter receives a request, its shared number is updated.

The new inputs and outputs for these two DQNs are:

- State
  - Shared DQN
    - [global_servers_occupation, dc_req_server_occupation, req_state]
      Where global_servers_occupation is the occupation of all the servers, dc_req_server_occupation is the occupation of the server whose shared number is being updated, and req_state the state of the new request
  - DQN
    - [local_occupation, shared_info, req_state]
- Reward
  - Remains the same
- Action
  - Shared DQN
    - As the action will be the number that is assigned to the shared number, and we defined a resolution of 20 steps, there are 20 possible actions. For example, if we get a 3 as action, the shared number for that datacenter would be a 3/20.
  - DQN
    - The same as the normal DQN
      - 0 Accept
      - 1 Deny
      - 2 Resend

To sum up, imagine that a new request arrives at datacenter 2. The shared DQN will wake up, see the actual global state, and if it outputs a 9, it would change the shared information from [shared_number_1, shared_number_2, shared_number_3] to [shared_number_1, 9/20, shared_number_3] and finally waking up the DQN of the datacenter 2. Which will take the new shared information, concatenate it with the rest of the input local data, and will decide what to do with the new request.

### 3.2.4. Classes

A brief description of all the simulator classes is described below.

#### 3.2.4.1. Request

Model of a simple request that for each time step it decreases its duration by one (once it arrives at 0 it returns that is completed).

##### *3.2.4.1.1. Attributes*

- **size**: Occupied resources
- **duration**: Remain time steps until the task is finished
- **max_hops**: Maximum number of hops the request can be resent

##### *3.2.4.1.2. Functions*

- **state**: Returns the request state
- **normalized_state**: Returns the normalized state
    - size: Normalized by the maximum environment request size
    - duration: Normalized by the maximum environment duration size
    - max_hops: Normalized by the maximum environment number of hops
- **get_num_params**: Returns the number of attributes
- **processed**: Decrease the request duration and returns a boolean indicating if the request is completed

#### 3.2.4.2. DummyServer

Server implementation.

##### *3.2.4.2.1. Attributes*

- **capacity**: Maximum number of available resource
- **server_state**: Number of occupied resources (starts at 0)
- **queue**: List with all the requests that are being processed

##### *3.2.4.2.2. Functions*

- **empty_queue**: Returns if the queue is empty
- **try_to_allocate**: Heuristic that checks if there is space for a request
    - If this is the case, it adds the request to the queue and returns True
    - On the contrary, it returns False

- **time_step**: Time step simulation, it decreases all the requests durations from the queue and delete the ones that are completed (duration == 0)

#### 3.2.4.3. DQN

Implementation of DQN algorithm.

#### 3.2.4.4. DataCenter

##### *3.2.4.4.1. Attributes*

- **env**: Environment where is placed
- **name**: Datacenter name
- **servers**: List with the datacenter servers
- **visible_DC**: List with the name(s) of the datacenter(s) in which it has a link
- **n_actions**: Number of DQN actions
- **DQN**: DQN object

##### *3.2.4.4.2. Functions*

- **servers_state**: Returns the servers state
- **state**: Returns the input for the DQN
- **epsilon_greed_policy**: Epsilon greedy policy implementation
- **try_to_allocate**: Function that tries to allocate a request and returns if it was allocated
- **try_allocate_in_server_n**: Function that tries to allocate a request in a specific server and returns if it was allocated. (Not used - Explained in Section X )
- **send_request**: Receives a request, call the DQN and (if necessary) tries to allocate it (calls try_to_allocate)
- **training_step**: Training step
- **performance_step**: Performance step

#### 3.2.4.4.5. DC_env

Environment implementation.

##### *3.2.4.4.5.1. Attributes*

- **map**: Array with all the environment datacenters
- **n_DC**: Number of datacenters

- **first_line**: Array with all the datacenters that are on the lowest level of the map, so for each time step, a new request is generated for each member of this line.
- **max_req_size**: Maximum permitted request size, used to normalize data
- **max_req_duration**: Maximum permitted request duration, used to normalize data)
- **shared_info**: Array with all the datacenters shared numbers
- **req_historic**: Array with all the generated requests, used to get statistics
- **dqn_precision**: Precision of the shared numbers, or which is equivalent, number of actions of the Shared DQN
- **DQN**: Shared DQN object

*3.2.4.4.5.2. Functions*

- state: Returns the state of all the datacenters, or what is the equivalent, the state of the environment.
- modify_shared_info: Modifies the shared number of a specific datacenter
- get_shared_info: Returns the shared number of a specific datacenter
- get_shared_info: Returns all the shared info, with all the shared numbers
- get_dqn_features: Returns the size of the Shared DQN input
- epsilon_greed_policy: Epsilon greedy policy implementation
- get_map_size: Return the map size
- generate_request: Generates a new request
- generate_random_request: Generates a random request following uniform distribution
- generate_car_model_seq: Generates car model statistical distribution
- generate_car_model_req: Generated a random request following the car model statistics
- get_occupation: Returns the occupation of all the environment servers
- resend_request: Receives a request from a datacenter and resend it to another one
- jointly_training_step: Training step
- random_performance_step: Performance step
- train: Trains datacenter DQNs
- shared_train: Trains datacenter DQNs and Shared DQN
- save_models: Save all the DQN models

# 4. Studies

## 4.1. General algorithm

For all the simulations, we have some aspects to consider:

- An epoch starts when all the datacenters that are on the bottom receive a new request and that request is completed (processed or rejected).
- First the algorithm starts the training part, and when it has finished all the training steps, it freezes all the models and starts the performance steps, in which we get all the system statistics.
    - The amount of training steps has varied during the investigation, so for each of the simulations training steps are indicated.
    - The number of performance steps is always set to 20.000

## 4.2. Results indicators

- **Reward**: Mean reward obtained by the system
- **Accepted ratio**: Percentage of accepted requests from the total
- **Discarded ratio**: Percentage of discarded requests, the ones the DQN directly discarded, from the total
- **Rejected ratio**: Percentage of rejected requests, the ones the DQN accepted but the server did not find a place to allocate it, from the total
- **Servers usage**: Mean usage for each server

## 4.3. Pilot map

As mentioned in previous sections, this map was created to test different ideas and features in order to decide which idea was finally implemented and later make heavier trainings with the final map. So let's see which ideas were proposed, the results they gave and the reasoning behind the decision of finally implementing them or not.

### 4.3.1. Heuristic vs DQN

In order to see if the parameters were well implemented, a first comparison was made between the heuristic and the DQN.

The system was trained two times, with an environment that had:

- Requests
    - MAX SIZE = 200
    - MAX DURATION = 20

- Steps
  - 60.000
- Datacenter 1
  - server 1 → 1000
  - server 2 → 1000
- Datacenter 2 and 3
  - server 1 → 150
  - server 2 → 150

At Figure 16 it can be seen the error from the 3 DQNs, one for each datacenter where red corresponds to DC1 and, as it has more capacity, the error is lower than the other two.

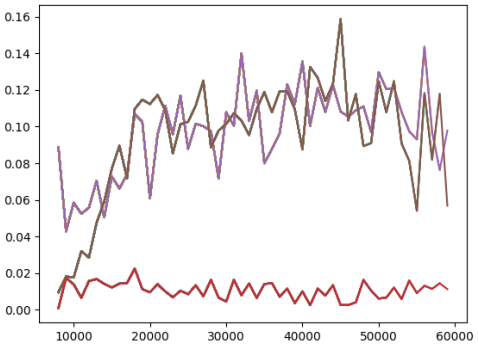


Figure 16. DQN training error evolution (Error - steps)

#### 4.3.1.1. Results

It can be seen that with little training better results can be found, having an increase of 40% in the reward and a 5% on the accepted request.

| Parameter | Heuristic | DQN | Increase |
|---|---|---|---|
| Reward | 0,1178 | 0,1667 | 41,51% |
| Accepted ratio | 0,7927 | 0,8340 | 5,21% |
| Discarded ratio | 0,0000 | 0,1239 | - |
| Rejected ratio | 0,2073 | 0,0421 | -79,68% |

Table 1. Heuristic vs DQN results

Apart from that, the server usage is much better, as in the DQN case, the second server is tiny used. Note that blue and red are Heuristic and yellow and green are DQN.

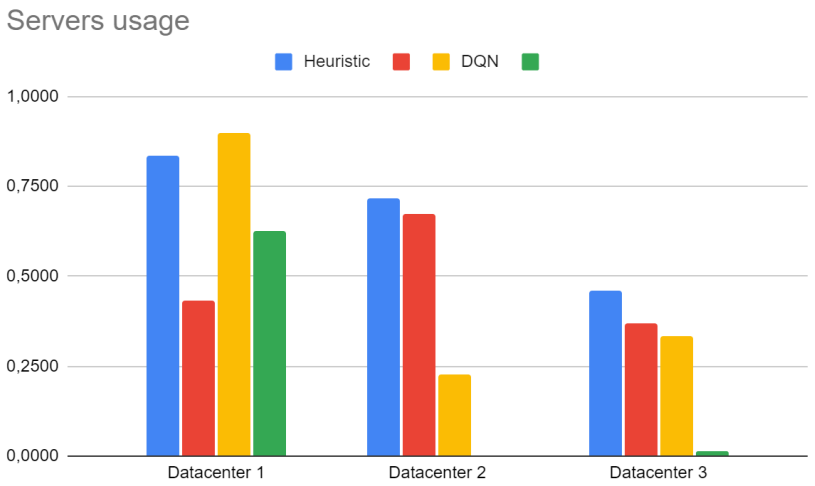


Figure 17. Heuristic vs DQN server usage

### 4.3.2. First-Fit Server Allocation vs Distributed DQN

From the beginning it was clear that DQNs would be used to manage requests, but there was quite a debate about how to integrate the server allocation algorithm. Finally, two different approaches were tested:

- First-Fit server allocation
  - The system allocates the request to the first server that has enough resources.
- Distributed DQN
  - The DQN from each datacenter apart from deciding if they accept the request, decides on which server they want to allocate the coming requests.

The system was trained two times, with an environment that had:

- Requests
  - MAX SIZE = 200
  - MAX DURATION = 10
- Datacenter 1
  - server 1 → 1000
  - server 2 → 1000
- Datacenter 2 and 3
  - server 1 → 150
  - server 2 → 150

The training of Distributed DQN was following Figure 18 and took 600.000 steps, the same as we did for normal DQN with a First-Fit server allocation.

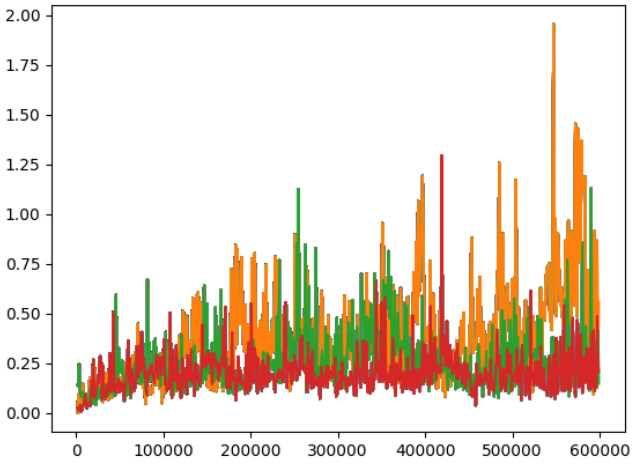


Figure 18. Distributed DQN training error (Error - steps)

#### 4.3.2.1. Results

| Parameter | First-Fit | Distributed DQN | Increase |
|---|---|---|---|
| Reward | 0,2765 | 0,2306 | -16,60% |
| Accepted ratio | 0,9978 | 0,8839 | -11,42% |
| Discarded ratio | 0,0014 | 0,0137 | 914,81% |
| Rejected ratio | 0,0008 | 0,1023 | 12687,50% |

Table 2. First-Fit vs Distributed DQN results

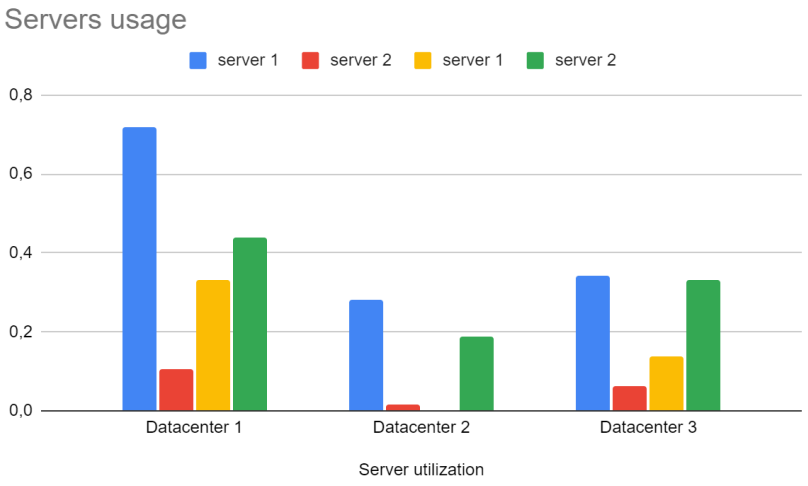


Figure 19. First-Fit vs Distributed DQN servers usage

It can be seen that letting the DQN decide on which server it wants to allocate the requests is not beneficial because neither the reward and the accepted ratio have better results, so it was decided to use the First-Fit solution.

### 4.3.3. DQN vs Shared DQN

The idea of Shared DQN, which has already been explained in previous sections, was validated by simulating both DQN and Shared DQN under the same conditions with a training of 100.000 steps [See Figure 20], with the same servers capacity as the last experiment, and a maximum request size of 500, to add more difficulty.

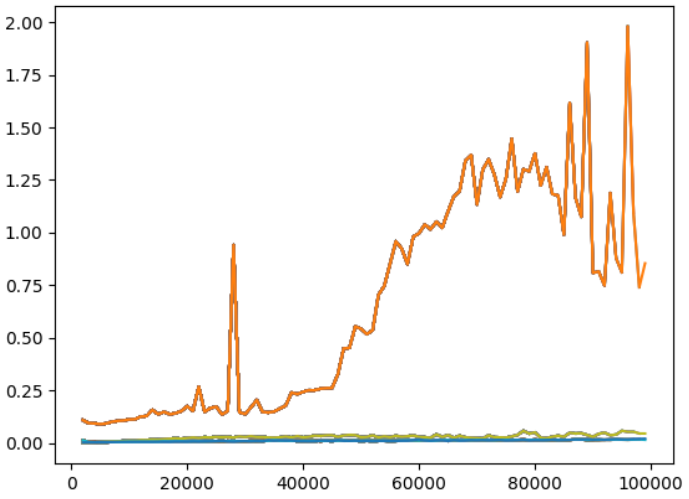


Figure 20. Shared DQN training error (Error - steps)

#### 4.3.3.1. Results

Unless the error of the Shared DQN goes a little crazy, after some steps it goes down again. In addition, unless it is quite a saturated environment, as requests can be huge, we see that the shared DQN gets better reward and accepted ratio values.

| Parameter | DQN | Shared DQN | Increase |
|---|---|---|---|
| Reward | 0,1045 | 0,1645 | 57,42% |
| Accepted ratio | 0,5849 | 0,7194 | 23,00% |
| Discarded ratio | 0,3884 | 0,0000 | -100,00% |
| Rejected ratio | 0,0267 | 0,2806 | 950,94% |

Table 3. DQN vs Shared DQN results

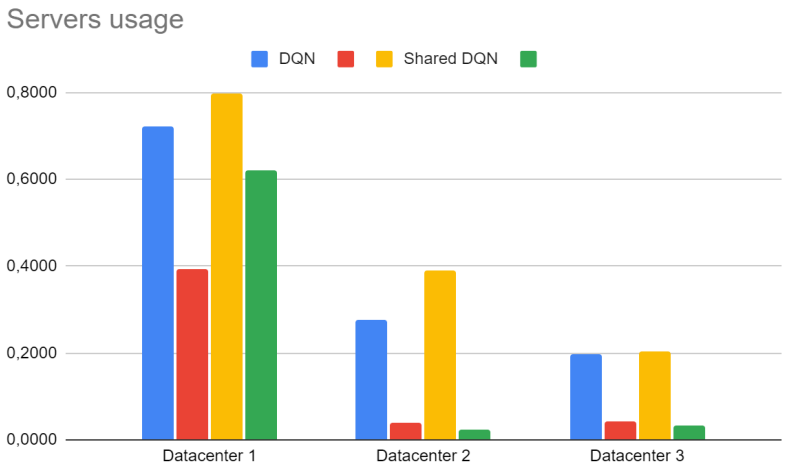


Figure 21. DQN vs Shared DQN servers usage

## 4.4. Final map

With all the lessons learned from the last map, the following parameters were set for this type of environment:

- Architecture (Summary)
  - Top DCs [Big computing centers]
    - Number of servers: 2
    - Capacity: 5000
  - Medium DCs [Edge computing]
    - Number of servers: 2
    - Capacity: 1000
  - Bottom DCs [Cars - Low computational ability devices]
    - Number of servers: 2
    - Capacity: 50
- Requests
  - Max size: 200
  - Max duration: 20
  - Max hops: Depends on the model (default is 3)

The maximum request size was set to 200 and the maximum request duration was set to 20 because this parameters generates good testing server occupation conditions, which means that servers occupation follows a model that could fit with the actual reality that has not many connected devices, 30% for top datacenters, 90% for medium and 50% for the low ones (in uniform model conditions and using the heuristic algorithm) [See Figure 22].

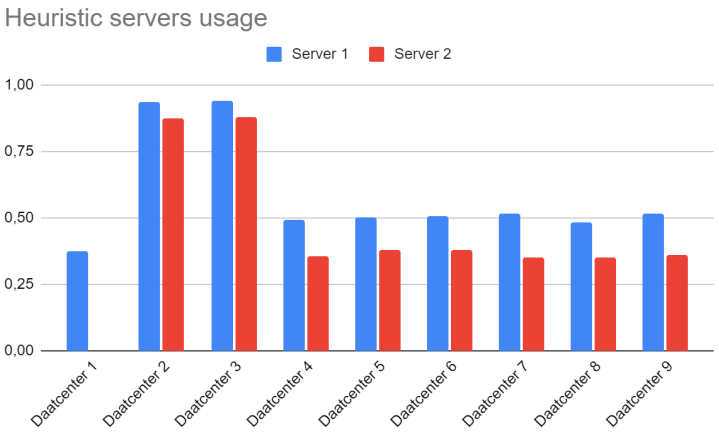


Figure 22. Heuristic algorithm servers usage

### 4.4.1. Uniform model

This simulation was done with random generated requests following a uniform distribution.

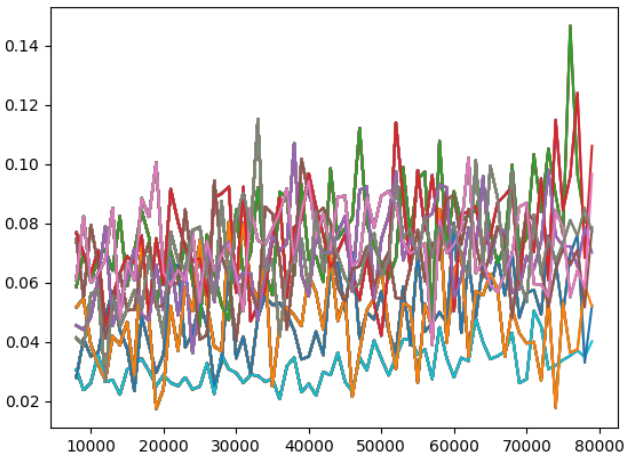


Figure 23. DQN training (Error - Steps)

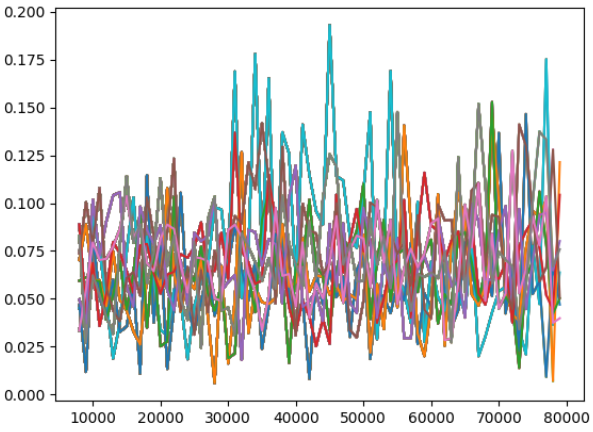


Figure 24. Optimal DQN training (Error - Steps)

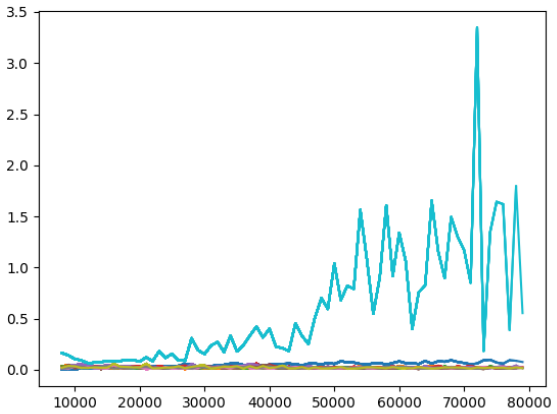


Figure 25. Shared DQN training (Error - Steps)

#### 4.4.1.1. Results

| Parameter | Heuristic | DQN | Optimal DQN | Shared DQN |
|---|---|---|---|---|
| Reward | 0,2195 | 0,1852 | 0,1708 | 0,1881 |
| Accepted ratio | 1,0000 | 0,9323 | 0,8731 | 0,9950 |
| Discarded ratio | 0,0000 | 0,0592 | 0,1148 | 0,0000 |
| Rejected ratio | 0,0000 | 0,0085 | 0,0122 | 0,0050 |

Table 4. Uniform model simulation results

### 4.4.2. Car model with limited hops

This simulation was done with random generated requests following a uniform distribution and with a 50% of probability of having a single limited hop.

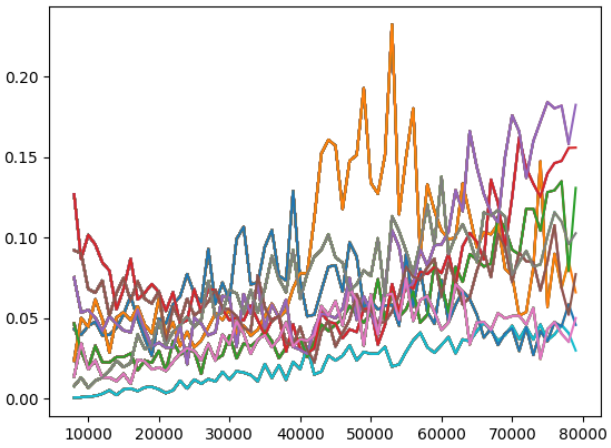


Figure 26. DQN training (Error - Steps)

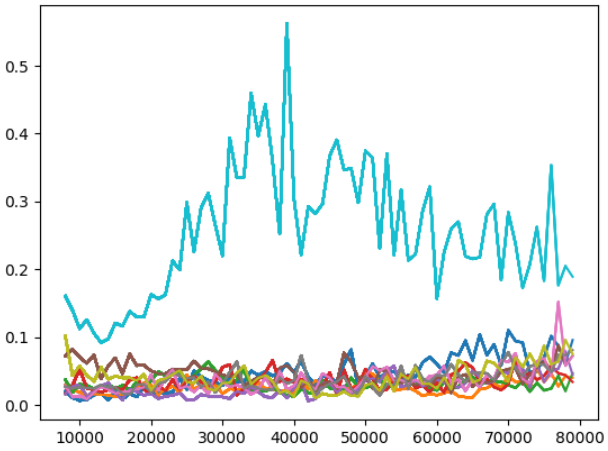


Figure 27. Shared DQN training (Error - Steps)

#### 4.4.2.1. Results

| Parameter | Heuristic | DQN | Optimal DQN | Shared DQN |
|---|---|---|---|---|
| Reward | 0,1889 | 0,1282 | - | 0,1876 |
| Accepted ratio | 0,9822 | 0,7077 | - | 0,9001 |
| Discarded ratio | 0,0000 | 0,2788 | - | 0,0630 |
| Rejected ratio | 0,0178 | 0,0135 | - | 0,0369 |

Table 5. Limited hops simulation results

### 4.4.3. Car model with beta distribution but without limited hops

This simulation was done with beta-distribution random generated requests without hop limitations. Note that a picture with the DQN training is not available, it was lost during the research.

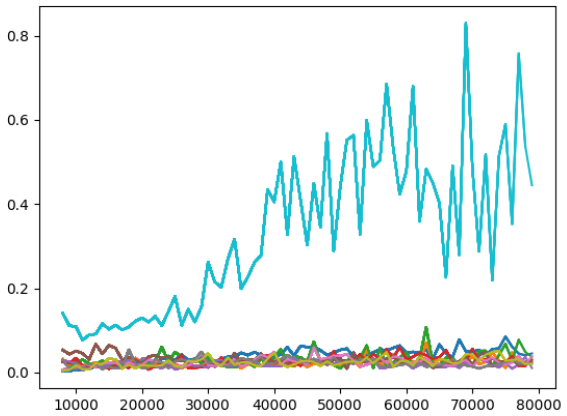


Figure 28. Shared DQN training (Error - Steps)

#### 4.4.3.1. Results

| Parameter | Heuristic | DQN | Optimal DQN | Shared DQN |
|---|---|---|---|---|
| Reward | 0,2388 | 0,1469 | - | 0,1505 |
| Accepted ratio | 1,0000 | 0,8716 | - | 0,9611 |
| Discarded ratio | 0,0000 | 0,1103 | - | 0,0332 |
| Rejected ratio | 0,0000 | 0,0181 | - | 0,0058 |

Table 6. Beta distribution simulation results

### 4.4.4. Complete car model

#### 4.4.4.1. Case 1

This simulation was done following our proposed Road Model [See Section X] that includes beta-distribution and hop limitations.

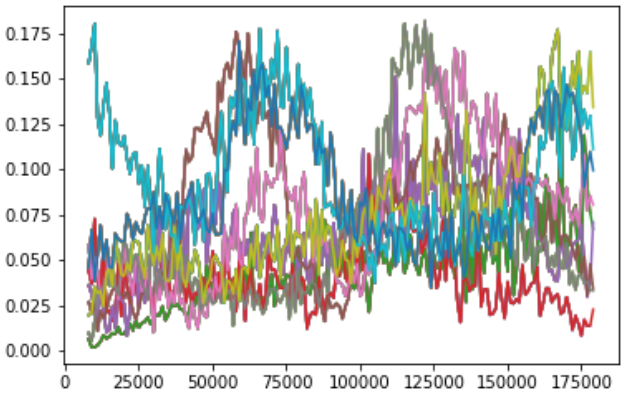


Figure 29. DQN training (Error - Steps)

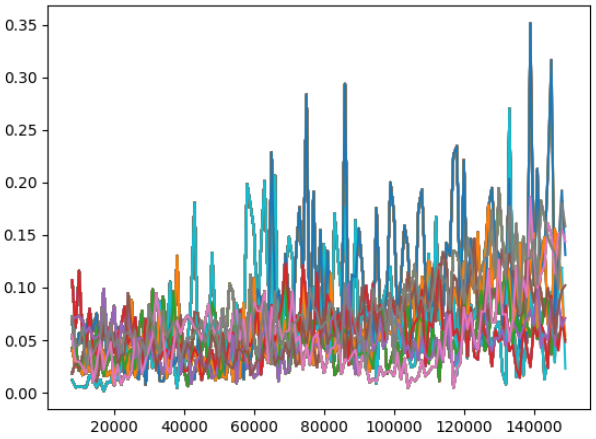


Figure 30. Optimal DQN training (Error - Steps)

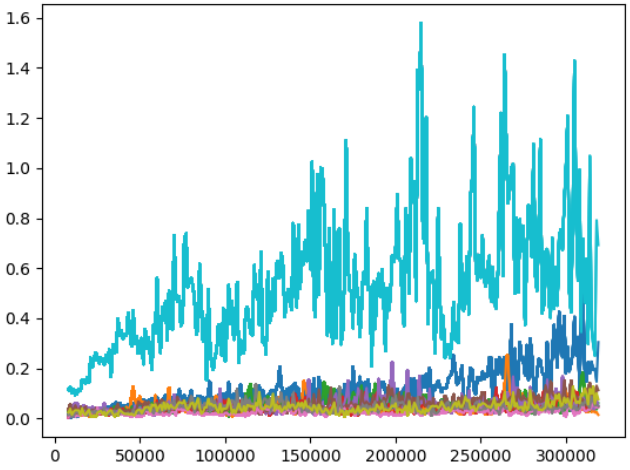


Figure 31. Shared DQN training (Error - Steps)

#### *4.4.4.1.1. Results*

| Parameter | Heuristic | DQN | Optimal DQN | Shared DQN |
|---|---|---|---|---|
| Reward | 0,2110 | 0,1416 | 0,1148 | 0,1554 |
| Accepted ratio | 0,9868 | 0,6464 | 0,7365 | 0,7333 |
| Discarded ratio | 0,0000 | 0,3183 | 0,2241 | 0,1469 |
| Rejected ratio | 0,0132 | 0,0353 | 0,0394 | 0,1199 |

Table 7. Car model (Case 1) simulation results

### 4.4.4.2. Case 2

This simulation was done following our proposed City Model [See Section X] that includes beta-distribution and hop limitations.

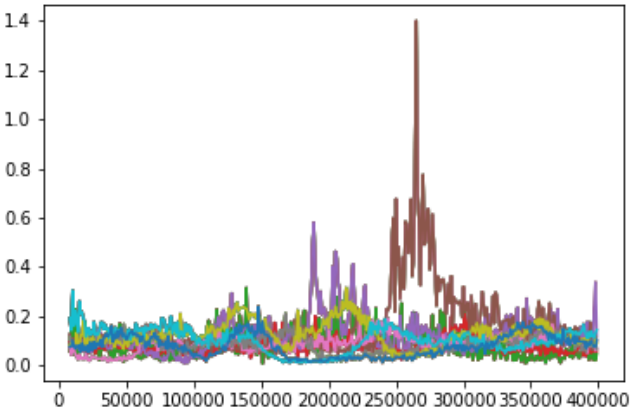


Figure 32. DQN training (Error - Steps)

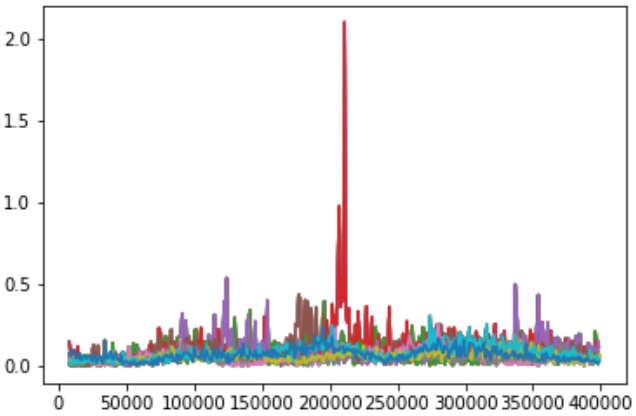


Figure 33. Optimal DQN training (Error - Steps)

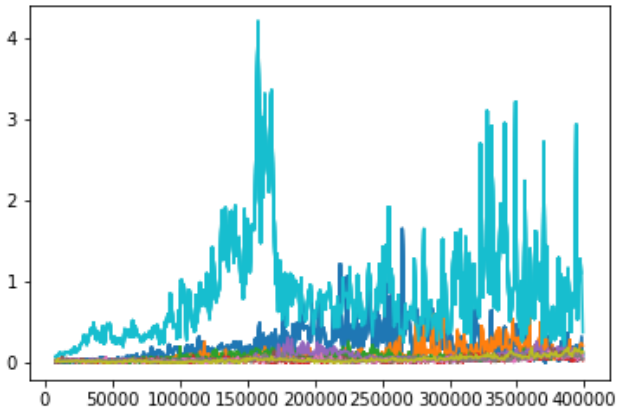


Figure 34. Shared DQN training (Error - Steps)

*4.4.4.2.1. Results*

| Parameter | Heuristic | DQN | Optimal DQN | Shared DQN |
|---|---|---|---|---|
| Reward | 0,2142 | 0,0813 | 0,0581 | 0,1387 |
| Accepted ratio | 0,9927 | 0,5075 | 0,6059 | 0,7483 |
| Discarded ratio | 0,0000 | 0,4237 | 0,3223 | 0,1730 |
| Rejected ratio | 0,0073 | 0,0688 | 0,0718 | 0,0787 |

Table 8. Car model (Case 2) simulation results

# 5. Budget

The estimation realization cost of this project are divided into two different categories:

- Labour cost: The cost of the time spended on working into this project
- Tools cost: The cost of all the tools used to carry out the project

So a specification of how each cost was defined and a final summary is presented below.

## 5.1. Labour cost

Here is included all the costs that are related with the time people spend working on the project, in this case, the project was developed by a person with a degree in telecommunications engineering with the supervision of a professor from the UPC (Universitat Politècnica de Catalunya). The following aspect were considered at the final table:

- Junior engineer:
    - Dedication of 30 hours/week during 23 weeks
        - 690 hours
    - 10€/hour without fees
    - 30% fees
- Master thesis tutor:
    - Dedication of 2 hours/week during 12 weeks
        - 24 hours
    - 30€/ hours without fees
    - 30% fees

## 5.2. Tools cost

Here is included all the cost of the materials and tools used to correctly develop the project, including hardware and software. The following aspects were considered at the final table:

- Laptop
    - Price 999,00€
        - Amortization of 33% for each year
        - 6 months
- Software
    - Python
        - Open source

## 5.3. Total

| Concept | Quantity | Unit Price | Total cost | Amotization | Final Cost |
|---|---|---|---|---|---|
| Junior engineer | 690,00 | 13,00 € | 8.970,00 € | - | 8.970,00 € |
| Thesis tutor | 24,00 | 39,00 € | 936,00 € | - | 936,00 € |
| Laptop | 1,00 | 999,00 € | 999,00 € | 164,84 € | 164,84 € |
| **TOTAL** | | | | | **10.070,84 €** |

Table 9. Budget summary

# 6. Conclusions and future development

After all the work done and simulations we can get the following conclusions:

- Nowadays, in case vehicle networks were implemented, the model would be not complex enough to engage the use of Reinforcement Learning because heuristics are near from the optimal solution.
- Two possible conclusions about the bad performance, in comparison to the Shared DQN, of the Optimal DQN approach
    - As the Shared DQN can see the states of all the datacenters DQNs and introduces values to its states, it has a way to interact with them so, this interaction makes that the results of Shared DQN could be better than the optimal approach.
    - A bad approach or implementation of the Optimal DQN is done.
- Parameter sharing is a very useful tool, as it has better results and helps training better and faster.
- Reinforcement learning could be better at managing congestion conditions.

## 6.1. Future development

in this thesis we have seen a first approach on how to do routing and resource allocation in vehicle networks, but there are some studies that can be done to gain more knowledge and upgrade the simulator:

- Optimize the DQN hyperparameters in order to work optimally with this type of problems.
- Implement and compare other types of Reinforcement Learning algorithms.
- When creating the environment, follow a OpenAI Gym environment structure, so it can be easier to compare and use with other algorithms that other communities do.
- Study and obtain a more realistic parameters for the Car Model
    - Include more parameters
    - Obtain more realistic parameters values
- Check the implementation of the Optimal DQN or change the optimal approach for parameter sharing, as would be interesting to have an optimal reference.
- Study which are the parameters or situations in which Reinforcement Learning could be better than heuristics.

# 7. Bibliography

1. Wikipedia contributors. “Machine Learning.” Wikipedia, Available at: https://en.wikipedia.org/wiki/Machine_learning#Unsupervised_learning .
2. T. M. Mitchell, Machine Learning. New York, NY, USA: McGraw-Hill, 2017
3. R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction. Cambridge, U.K.: MIT Press, 1998.
4. Luong, Nguyen Cong, et al. "Applications of deep reinforcement learning in communications and networking: A survey." IEEE Communications Surveys & Tutorials 21.4 (2019): 3133-3174.
5. Vehicular networks and the future of the mobile internet . Mario Gerla ⇑ , Leonard Kleinrock
6. Luong, Nguyen Cong, et al. "Applications of deep reinforcement learning in communications and networking: A survey." IEEE Communications Surveys & Tutorials 21.4 (2019): 3133-3174.
7. Lebeaux Rachel, “resource allocation definition”. Available at: https://www.techtarget.com/searchcio/definition/resource-allocation
8. HU, T.C. y KAHNG, A.B., 2016. Linear and Integer Programming Made Easy. Cham: Springer International Publishing. ISBN 9783319239996.
9. Ren, J. Wireless Network Virtualization Resource Sharing Based on Dynamic Resource Allocation Algorithm. Available at: https://www.hindawi.com/journals/wcmc/2022/5654188/
10. K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, “Deep reinforcement learning: A brief survey,” IEEE Signal Process. Mag., vol. 34, no. 6, pp. 26–38, Nov. 2017
11. Mao, Hangyu, et al. "Accnet: Actor-coordinator-critic net for" learning-to-communicate" with deep multi-agent reinforcement learning." arXiv preprint arXiv:1706.03235 (2017).
12. Lowe, Ryan, et al. "Multi-agent actor-critic for mixed cooperative-competitive environments." Advances in neural information processing systems 30 (2017).
13. OroojlooyJadid, Afshin, and Davood Hajinezhad. "A review of cooperative multi-agent deep reinforcement learning." arXiv preprint arXiv:1908.03963 (2019).
14. Kamdar, Renuka, Priyanka Paliwal, and Yogendra Kumar. "A state of art review on various aspects of multi-agent system." Journal of Circuits, Systems and Computers 27.11 (2018): 1830006.

15. Mao, Hangyu, et al. "Accnet: Actor-coordinator-critic net for" learning-to-communicate" with deep multi-agent reinforcement learning." arXiv preprint arXiv:1706.03235 (2017).
16. Chen, Tianyi, et al. "Communication-efficient policy gradient methods for distributed reinforcement learning." IEEE Transactions on Control of Network Systems (2021).
17. Amine, A., 2022. Deep Q-Networks: theory and implementation. [online] Medium. Available at: <https://towardsdatascience.com/deep-q-networks-theory-and-implementation-37543f60dd67>
18. Zaïem, Mohamed Salah, and Etienne Bennequin. "Learning to communicate in multi-agent reinforcement learning: A review." arXiv preprint arXiv:1911.05438 (2019).
19. Foerster, Jakob, et al. "Learning to communicate with deep multi-agent reinforcement learning." Advances in neural information processing systems 29 (2016).
20. Hausknecht, Matthew, and Peter Stone. "Deep recurrent q-learning for partially observable mdps." 2015 aaai fall symposium series. 2015.
21. Jiang, Feibo, et al. "Stacked autoencoder-based deep reinforcement learning for online resource scheduling in large-scale MEC networks." IEEE Internet of Things Journal 7.10 (2020): 9278-9290.
22. Allahham, Mhd Saria, et al. "I-SEE: Intelligent, Secure, and Energy-Efficient Techniques for Medical Data Transmission Using Deep Reinforcement Learning." IEEE Internet of Things Journal 8.8 (2020): 6454-6468.

# 8. Glossary

| | |
|---|---|
| **RL** | Reinforcement Learning |
| **DQN** | Deep Q Network |
| **ML** | Machine Learning |
| **VANET** | Vehicular Ad-Hoc Networks |
| **WP** | Work Package |
| **NN** | Neural Network |
| **DNN** | Deep Neural Network |
| **MAS** | Multi Agent System |
| **MARL** | Multi Agent Reinforcement Learning |
| **RIAL** | Reinforced Inter-Agent Learning |
| **DRQN** | Deep Recurrent Q Network |
| **DIAL** | Differentiable Inter-Agent Learning |
| **DC** | DataCenter |