POLITECNICO DI TORINO

Master of Science degree in ICT for Smart Societies

Politecnico di Torino

UNIVERSITÉ DE LORRAINE

"In-The-Air" ML-Driven Optimization of UAV Coverage and Resource Utilization

Thesis Supervisors:

Prof. Carla Fabiana Chiasserini

Prof. Enrico Natalizio

Candidate:

Lorenzo Bellone s258340

Academic Year 2020-2021

Abstract

Unmanned Aerial Vehicles (UAVs) are being deeply investigated because of the several services and improvements they can provide in different application fields. They have been considered as enablers of several applications, especially when the ground users require them to complete specific tasks, such as target tracking, event identification or areas monitoring. Although several works have addressed numerous challenges in using UAVs, their communication capabilities, their trajectories definition and their constrained on-board computational resources still need a deeper investigation. Furthermore, the need for distributed control solutions to foster UAVs' cooperation in large scale scenarios is sharply increasing in the research community. Reinforcement Learning (RL) has drawn a lot of interest within this context. The application of this framework in decision-making problems has brought sensational results in both single and multi-agent systems. Furthermore, state-of-the-art Deep Reinforcement Learning (DRL) algorithms are constantly enlarging the application domains in which this approach overcomes benchmarking solutions.

In this work, a scenario where ground users assign tasks to UAVs is considered, each task requiring a certain computational effort. The aim is thus to envision a distributed control solution for UAVs trajectories, which jointly maximizes the coverage of ground users and the computational resource utilization of the individual UAVs. A multi-agent deep reinforcement learning approach is used for the trajectory definition, which will be based on a partial knowledge that UAVs can gather from the environment. More in details, two different observation scenarios are considered and compared: (i) a centralized observation, where each UAV knows the positions of all the others, no matter where they are, and (ii) a more distributed observation, where each UAV gather information only from its neighbors and from the ground users it is covering. The performance evaluation based on the training of the models in prelim-

inary simulations presents a remarkable increase in terms of coverage and resource utilization of the UAV network. Furthermore, the trained models are leveraged in a realistic network simulation, observing the challenges that the DRL approach has to face in a more sophisticated environment. Although the implementation of the models presents main limitations in the application to a realistic scenario, the results show the potentials of the proposed approach, and encourage to look into it for defining further improvements.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

UAV networks is an emerging technology that is drawing a lot of attention due to the broad set of application domains it can provide. According to their role in a wireless network, UAVs can be used for different purposes: on one hand, they can operate as users of the network, and be exploited in applications such as surveillance, package delivery or precision agriculture. On the other hand, they can serve as aerial base stations, thus supporting and extending/replacing the existing communication infrastructures in application fields where these facilities are hard to be deployed due to the remote or inaccessible locations. Thanks to their ability to adjust their position in the three-dimensional space, they can increase the likelihood of establishing a line of sight link with the ground users, thus providing an actual enhancement of the wireless network coverage and capacity.

Recent works on UAV networks have demonstrated how the communication challenges and open problems related to the deployment and utilization of this technology are still in an early stage. However, their interest is sharply increasing in the research community due to the enormous advantages they can bring, especially towards the next generation wireless communication networks [1].

One of the main challenges that has been considered in the proper deployment of UAV networks regards position related aspects. Since the main innovation brought

by UAV networks is the possibility to have highly dynamic nodes, their proper placement in the space has to be correctly configured in order to take full advantage of the system. Placement and trajectory design problems in UAV networks have been notably studied in a wide range of applications. Nonetheless, there is still lack of contributions for distributed control solutions that, according to a local understanding of the current environment, drive each UAV towards an optimal position. In this context, the use of Machine Learning (ML) frameworks can bring a level of intelligence in the deployment of UAV networks that allows to identify valid information and patterns, usually too complex to be derived by humans.
More in details, reinforcement learning approaches have acquired a considerable interest in control design applications. RL is a field of machine learning whose aim is to solve Markov Decision Processes (MDP), where an agent learns how to make decisions according to observations of the surrounding environment and a correspondent value of reward. In the past few years, this paradigm has achieved sensational results by learning how to play games such as Go [2], chess [3] or Atari games [4]. Furthermore, it has been found to be particularly suited for other typologies of applications, such as collaborative tasks in multi-agent systems [5] and performance enhancement in communication networks [6].

1.1 Context and Problem Statement

In this thesis, we propose a multi-agent reinforcement learning approach for the placement of the nodes of a UAV network within a specific context that has rarely been considered: besides extending/replacing communication infrastructures, fleets of UAVs can provide computational support for ground users' tasks, thus acting as computational aerial servers and potentially reducing the delay the ground users experience for their tasks to be processed. Our main contribution aims at proposing a distributed framework based on multi-agent DRL that, by exploiting the UAVs as agents of the system, leverages their mobility in order to provide ground users with

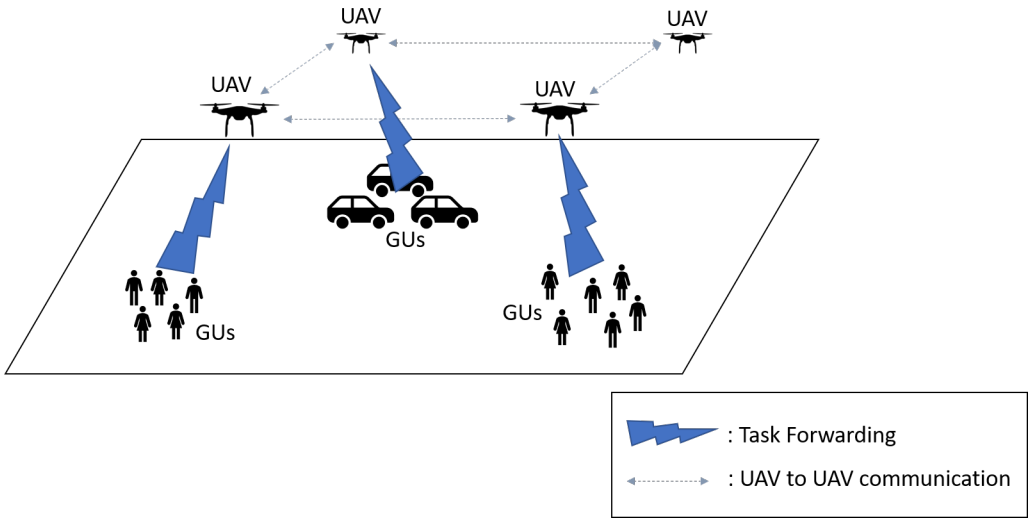


Figure 1.1: Scenario depicted in this work: a set of ground users send tasks requiring computational effort towards a UAV network. The UAVs can share their tasks to a more idle UAV in order to reduce the processing time.

a proper quality of task execution. Inspired by the work of Hu et al. [7], better presented in chapter 2, the proposed scenario consists of the following components:

- UAVs: they are the flying nodes of the network. The communications between them occur over a wireless ad-hoc network. The UAVs under analysis have also sufficient on-board computing capabilities to process tasks that require computational effort, which are forwarded from a set of ground users. However, due to the limited computing capabilities of UAVs, we considered the possibility to apply a **resource sharing algorithm** in order to relax the computational stress.
- Ground Users (GUs): they are mobile terminals located on the ground, with the ability to offload computational tasks directly to UAVs. GUs are not spatially fixed and they are characterized by a **level of activity**, that is a number representing the number of tasks they have to offload.

Figure 1.1 provides a visual representation of this scenario. According to this architecture, it is important to identify the metrics that mostly influence the performance

of the network and that can be improved by an effective placement of the UAVs. As will be discussed in chapter 2, the placement and trajectory design problems in UAV networks try to mostly deal with the optimization of **coverage** and **connectivity** of the network, while other works introduce the possibility to exploit the UAVs in order to **offload tasks** generated from the ground, using their mobility to directly affect the task processing performance.
The contribution we want to give to the research consists in the formalization of a dynamic and decentralized multi-agent decision-making problem addressed through a multi-agent RL approach. Through this approach, two performance metrics that, to the best of our knowledge, have never been tackled together, will be jointly optimized. The aim is thus twofold:

1. maximize the probability that, when a new GU appears in the network, it is already covered by one UAV;

2. maximize the resource utilization of UAVs, in order to properly cover the GUs that require a higher computational effort, thus affecting the task processing performance of the network.

Our algorithm is trained in a relatively simple simulated environment and then tested in a more sophisticated network simulator in order to measure its performance within a more realistic scenario.

1.2 Structure of the thesis

This section aims at presenting how this thesis has been organized.

Chapter 1 - Introduction

The current chapter contains an overview of the main motivations and contributions of this work. Furthermore, the organization of the thesis is presented.

Chapter 2 - Literature Review

Chapter 2 aims at presenting the researches this work has been inspired from, highlighting our position and innovation with respect to the current approaches. It presents the most recent works dealing with position and trajectory design problems in UAV networks, and organizes them into two main sections: coverage and connectivity optimization for the capacity enhancement of a terrestrial network, and computation offloading in UAV networks for the optimization of task processing performance.

Chapter 3 - Theoretical Background

Chapter 3 provides a description of the main background concepts of this work. It is mainly divided into two parts: firstly, an overview of the main applications and architectures of UAV networks is provided. Secondly, a detailed description about reinforcement learning is presented, in order to provide the reader with the basic concepts to have a full comprehension of our approach.

Chapter 4 - Problem Formalization and Algorithm Setup

Chapter 4 presents the problem formalization through a mathematical framework used to model the scenario described in section 1.1. Furthermore, our solution to the decision-making problem is proposed and thoroughly described, motivating the choice of the leveraged approach.

Chapter 5 - Tools and Implementation

Chapter 5 describes the tools implemented along this work and outlines the methodology followed to design our framework. The interaction between the different components of the system, together with the description of the simulation implementations, is presented in detail. Furthermore, the limitations of the approach are also explained and possible solutions to mitigate them suggested.

Chapter 6 - Experimental Results and Discussion

In chapter 6, the results obtained from the application of our algorithm in different simulated scenarios are shown. Firstly, the training of the model is performed in a simplified simulated environment. Secondly, the trained models are applied in the more realistic scenario. All the results are discussed and interpreted, pointing out the strengths but also the main limitations in the obtained outcomes.

Chapter 7 - Conclusions and Future Works

The final chapter includes a summary of the proposed approach and obtained results. A dedicated section provides also a more detailed discussion on the main limitations and future improvements to this work.

Chapter 2

Literature Review

An extensive literature review has been performed in order to understand which are the main UAV networks related issues and how to cope with them with Artificial Intelligence (AI) methods. The survey written by Bithas et al. [8] was truly inspiring to have an overview of the current state-of-the-art and to orient our research. The authors, asserting how the adoption of UAVs in communication-based applications is going to become an integral part of the next generation wireless communication, and how ML frameworks will be exploited to solve most of the problems related to them, provide a very detailed survey on past research works that enlighten how different AI/ML techniques have been applied to UAV networks (as also depicted in Figure 2.1). They identify four main application areas in which these frameworks have proved to bring innovative solutions: **Physical Layer Issues**, **Security and Safety Issues**, **Resource Management and Network Planning** and **Position Related Aspects**. Our interest was mainly captured by the latter described application: the determination of proper placements of UAVs as well as their trajectories in order to achieve proper network performance. The authors argue how this is one of the most challenging applications in UAV networks. Starting from this survey, we explored several other studies to acquire proper knowledge about how the Placement and Trajectory Design problem has been recently handled. We performed a meticulous

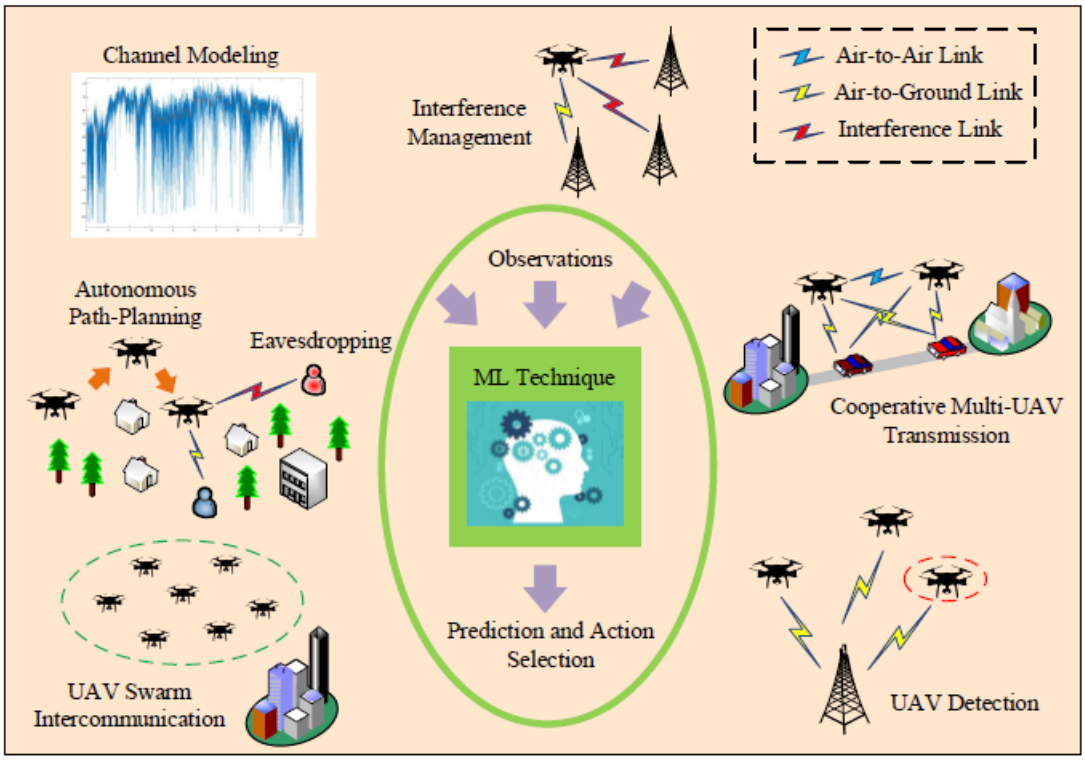


Figure 2.1: "Applications of Artificial Intelligence/Machine Learning in UAV-based communication networks". Image by "A Survey on Machine-Learning Techniques for UAV-Based Communications" from Bithas et al. [8].

selection of the existing works that could direction us towards the definition of a proper scenario and a methodology to deal with it. The existing works that led us to this achievement can be grouped into four main collections:

- Coverage and Connectivity optimization in UAV networks through RL approaches;
- Coverage and Connectivity optimization in UAV networks through Evolutionary Computation (EC) approaches;
- Task offloading from UAVs to fixed infrastructures (UAV-to-Infrastructure), such as Mobile Edge Computing (MEC) [9] servers or fog nodes [10];
- Task offloading from ground users to UAVs (GU-to-UAV) for on-board pro-

cessing.

2.1 Coverage and Connectivity Optimization

In the first two collections, we can find works whose aim is to define the best placement for a set of UAVs in order to maximize the coverage of ground users (i.e. the number of users that are in range with at least one UAV) and/or to ensure a proper connectivity between UAVs.

2.1.1 Reinforcement Learning Approaches

Klaine et al. [11] propose a RL-based approach in order to find the optimal positions for multiple UAVs and to recover a cellular communication network whose main base stations were partially destroyed. The authors propose a distributed Q-learning [12] algorithm to optimally place the UAVs in the target area: the agents of the environment correspond with the UAVs; the state that the agent observe is given by its own position; the actions are given by seven possible three-dimensional discrete movements and, finally, the reward is given by the summation of all the users under coverage. In order to evaluate the network performance, the authors consider the percentage of ground users in outage and the throughput that these users are able to obtain. They compare their results with three other algorithms (Fixed Random Position, Fixed Circular Position and Fixed Hotspot Position), showing how their approach outperforms these methods in terms of number of users under coverage and total achieved throughput. The work proposed by Venturini et al. [13] tries to deal with the concept of scalability with respect to the number of possible states a UAV can observe. The environment consists in a set of UAVs, flying within a squared grid, whose goal is to track and hover over fixed targets that are randomly positioned in the area. The system is modeled as a Partially Observable Markov Decision Process (POMDP) [14]. A Multi Agent Reinforcement Learning (MARL) [5] scheme is proposed through a multi-agent Deep Q-Learning (DQL) [15] approach,

where the partial observation is represented by three images and the Q function is approximated by a Convolutional Neural Network (CNN). All the state information is shared between the UAVs; hence, all the agents of the system have the same observation of the environment. The reward depends on the distance between the agent and a given target, and a penalty is applied when more UAVs move towards the same cell. As presented in the results, this approach brings benefits especially to the scalability of the environment (in terms of both number of agents and state-space). Liu et al. [16] present a similar scenario proposed in [11], with the goal of achieving a "long-term communication coverage" for ground users. The scenario identifies a set of UAVs that are used as aerial base stations, which have to cover a set of fixed Point of Interests (PoI) while keeping the UAV network connected. They propose a distributed version of the Deep Deterministic Policy Gradient (DDPG) [17] algorithm, modeling the scenario as a POMDP. Each agent only has a partial observation that consists in its energy consumption, position, flying direction and distance traveled. The action is given by two continuous parameters: the decision from the agent regarding its flying trajectory and the distance that it will travel in the next step. In the reward function, the algorithm encourages the increase of coverage and geographical fairness while penalizing the decrease of connectivity and high values of energy consumption. Their approach is compared with other four baseline algorithms, namely DRL-EC^3 [18] (presented in one of the previous authors' work), Greedy, mTSP [19] and Random, showing how their algorithm can obtain better results with respect to the baseline approaches in terms of energy efficiency and coverage score, while no metrics are provided for the connectivity of the network.

This first group of researches only deals with coverage and/or connectivity in a UAV network, considering UAVs as data relays. They were fundamental to understand how modern methodologies can face a portion of the placement and trajectory design problems with a DRL approach. However, all these researches present a scenario in which the positions of the ground users that have to be covered are fixed along each

episode, and the state-space is either very limited or requires a potentially large exchange of information in the network, without taking into account the wireless communication of data between nodes. In our work, we include a more dynamic scenario regarding the ground users' behavior and we exploit the learned policy in a network simulator (ns-3) in order to observe how the wireless communications between the nodes of the network can affect the performance of the algorithm.

2.1.2 Evolutionary Computation Approaches

Although we decided to exploit a multi-agent RL-based approach, another point of view was provided by the second collection of works, whose aim is still to improve the coverage and connectivity of the network, but with an EC approach. The researches carried out by Giagkos et al. [20] and Leu and Tang [21] are worth to be mentioned. They both explore the concept of survivable networks, trying to maximize the two metrics (coverage and connectivity) through Genetic Algorithms (GA). The former presents a centralized approach, where one UAV executes the algorithm, sending instructions about which manoeuvre to perform to all the others. The latter presents a decentralized GA, where each agent decides on its own how to tune the parameters of its mobility model. These approaches have very interesting results, since they show near-optimal outcomes in terms of coverage and connectivity, exploiting simple rules that lead to fast convergence and scalability. The discussion regarding whether it is better to use a RL or EC based approach is not in the purpose of this thesis. In fact, the investigation of the applications of evolutionary computation algorithms in UAV networks is one of our objectives for future researches.

2.2 UAV Networks for Task Offloading

Given the recently increasing need for the offloading of delay sensitive and computationally demanding applications, which is moving from the user equipment to the edge of the network, the other focus of this thesis is represented by the role that

UAVs can play in these scenarios.

2.2.1 UAV-to-Infrastructure

In the third collection of analyzed works we investigated the concept of computation offloading from UAVs to a fixed infrastructure at the edge of the network, in order to process delay sensitive tasks. In these scenarios, one or more UAVs gather data from the ground through their sensors and generate tasks that, due to the limited computational resources and battery lifetime, are offloaded to a near fixed infrastructure with sufficient computing resources. Kim et al. [22] propose an offloading solution based on k-means and RL. They analyze a scenario where a set of UAVs fly over a target area and acquire data from a set of ground locations. Since these data should be analyzed through computationally intensive techniques, the UAVs will have to select the best MEC server for the data offloading according to: the CPU requirements of the task, the queue status of the MEC server and the quality of the channel between the UAV and the MEC server itself. The objective of this work is twofold: firstly, the authors propose a k-means algorithm to associate each UAV to a set of ground locations, exploiting, subsequently, a Q-learning algorithm to decide the order of visiting for each task of this set; secondly, they exploit another Q-learning algorithm to select the MEC server that can process the task fastest. In the reward functions of the two RL approaches, the authors consider the total distance traveled by the UAV and the total waiting time for a task to be processed, respectively. The obtained results show how the proposed approach brings better outcomes in terms of energy consumption and total task processing time, when compared to a greedy algorithm. A similar scenario is investigated by Yao and Ansari [23]. A "Fog-aided Internet of Drones (IoD)" is presented. The fog nodes have to provide computational resources to the UAVs. In this scenario, a UAV has to visit a fixed number of locations in an already provided order, collect data from these locations, generate the tasks and offload them to the proper fog node. The authors formulate a mixed-integer non-linear programming problem, with the goal of minimizing

the overall journey completion time, considering the speed of the UAV, the wireless transmission time, and the fog node processing time. Given the complexity of the problem, the authors propose an online learning algorithm by splitting the main objective function into two sub-problems: an integer linear programming problem for the task allocation, which is solved with a heuristic approach, and a continuous optimization problem for the flying speed control between two locations. The proposed online algorithm is compared with baseline approaches, namely "energy-only" and "delay-only". The first one fixes the speed of the UAV to the minimum value and assigns each task to the fog-node that is providing the minimum energy consumption at each location. In the second one, the flying speed is fixed to the maximum one and each task is assigned to the fog-node that is providing the minimum task completion delay. The online algorithm performance has proved to be closer to the delay-only baseline, which opportunely minimizes the overall journey completion time, always keeping the energy consumption below the UAV's battery capacity.
To the best of our knowledge, these two works provide the most recent methodologies to deal with the proper placement and trajectory design in a UAV network where tasks have to be offloaded to the edge of the network from the UAVs. It is important to notice how these works do not deal with the coverage and connectivity, but only with the computation offloading of tasks directly created by the UAV, which has an a-priori knowledge of the locations for the data collection.

2.2.2 GU-to-UAV

Starting from this specific application, we wonder whether UAVs could be exploited to directly process computationally intensive tasks that are generated by ground users, instead of relying on fixed infrastructures. In fact, given the constantly improving computational platforms that can be embedded in UAVs, the on-board processing of relatively complex tasks with stringent delay constraints is an option that can be taken into account. As argued by Hu et al. [7], real-time applications such as vehic-

ular Virtual Reality (VR) and Augmented Reality (AR) gaming are not suited for the previously described UAV networks. The interactions between UAVs and MEC servers considerably increase the system's overhead, consequently causing a low quality of experience of the ground users. Furthermore, the MEC servers might not be properly positioned, causing failures when ground users are placed in unreliable locations. The authors propose a new computing architecture called "UAV-M3T", whose aim is to enhance the cooperation between UAVs, detaching the network from the need of external and fixed infrastructures. This architecture relies on the coordination of computing, caching, and communication resources between UAVs in order to efficiently exploit their limited capabilities. Figure 2.2 represents the architecture of this system. A proper AI decision framework based on the use of historical and

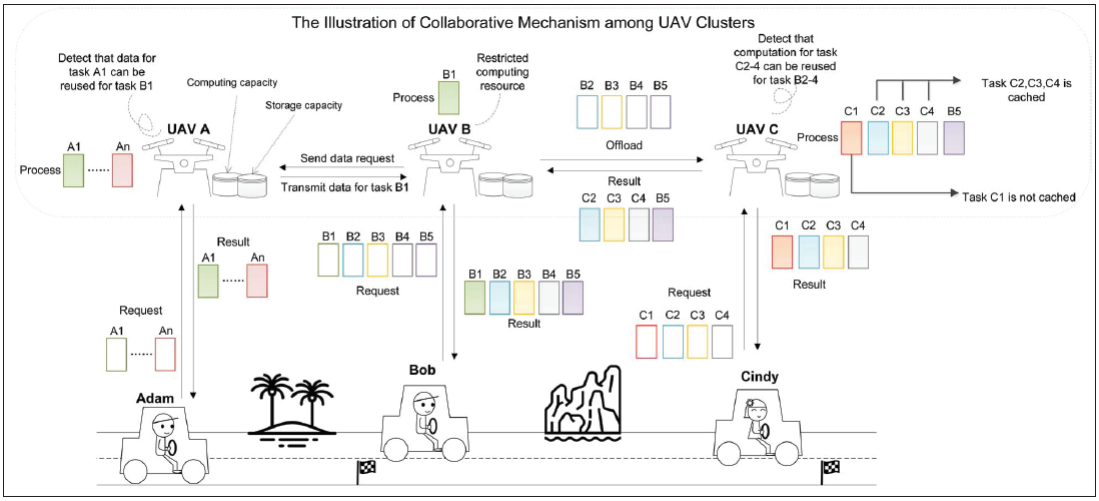


Figure 2.2: "Architecture of UAV-M3T". Image by "Ready Player One: UAV-Clustering-Based Multi-Task Offloading for Vehicular VR/AR Gaming" from Hu et al. [7]. Three possible scenarios can occur: 1) one task is executed by one single UAV, which has sufficient 3C capabilities; 2) Multiple similar tasks should be processed by one single UAV since they could have very similar results; 3) multiple UAVs can process one single task by sharing their resources.

social data to forecast the future demand is proposed to facilitate this cooperation. However, this approach is not very reliable in more dynamic scenarios, as the UAVs

cannot properly adapt with respect to the real current users' demand. Hence, more efficient algorithms for a dynamic resource coordination between UAVs have still to be implemented. In our work, we decided to exploit a similar architecture, where no MEC servers are present. The UAVs share their computing capabilities, thus trying to efficiently utilize the network resources, enabling cooperation between neighboring UAVs, and possibly increasing the users' perceived quality of experience.

Two other interesting works that consider the possibility to offload a computing task directly to a UAV are Ti and Bao Le [24] and Li et al. [25]. The former presents a hierarchical Fog Computing System (FCC) where a set of mobile users can decide whether to locally process their computational tasks or to offload them to the cloud or the cloudlets (UAVs with integrated computing platforms), with the aim of minimizing the total power consumption of all the users. The problem is formalized as a non-convex mixed-integer non-linear programming problem, and it is solved by separately and iteratively optimizing the continuous variables and the integer variables, thus solving the underlying non-convex and integer linear programming subproblems. The latter proposes a framework in which UAVs can be seen as MEC servers (so-called "UAV-assisted MEC networks"). Their system consists in one single UAV and multiple users that can upload computing tasks to it. The possible positions that the UAV can have are fixed, and the goal is to maximize the throughput migration of user tasks by letting the agent decide which one of the fixed points to cover. The problem is formulated as an MDP, and it is solved with a DRL-based scheme, having as a state the locations of UAV and users, the energy level of the UAV and the Channel Gain State between all the users and the UAV. The reward is given by the gain obtained for the computation of a certain task, then penalized by its energy consumption.

While in the just presented works each UAV could process one task at a time, an alternative solution is suggested by Yang et al. [26]. In this scenario, a UAV-assisted MEC network has to provide an offloading strategy to a set of IoT ground nodes, aiming at minimizing the total average tasks processing time, while achieving a proper

offloading fairness among UAVs and satisfying coverage constraints. One UAV can receive more tasks at a time, and a proper scheduling algorithm is carried out, with the objective of task average waiting time minimization. The overall optimization problem formalization leads to an NP-hard problem. Hence, the authors, model the task allocation between IoT nodes and UAVs as a Generalized Assignment Problem (GAP), solving it with an approximate algorithm; the tasks are then scheduled within each UAV through a DRL approach, and finally, the near-optimal locations for UAVs are found through different iterations of a Differential Evolution (DE) algorithm. The just presented scenario is similar to the one proposed in this work: while providing a proper level of coverage, UAVs have to process tasks coming from a set of ground nodes, trying to minimize the waiting time for the task processing. However, while in [26] the authors present a centralized scenario, in which all the environment state is well known and the positions of the ground nodes are fixed, we deal with a decentralized approach, where each UAV has a partial knowledge of the environment state and takes decisions accordingly. Furthermore, the ground nodes that require tasks to the UAVs are mobile users, able to change their position in time, thus providing a much more dynamic scenario.

This chapter presented the most relevant works we considered in order to direction our research and to explain our main contributions with respect to the already existing works. To the best of our knowledge, there are no current researches that try to jointly optimize the coverage of ground users with unknown and dynamic locations, together with the utilization of the computing resources of UAVs, able to process computational demanding tasks forwarded from the ground users. We formalized the coverage and resource optimization problem as a Decentralized POMDP, and applied a multi-agent DRL approach, taking into account the advantages of these algorithms in similar scenarios.

Chapter 3

Theoretical Background

In this chapter, we provide an overview of the basic concepts this thesis is dealing with. Firstly, an outline on what exactly a UAV is intended to be and how UAV networks are deployed and used nowadays is given. Secondly, an analysis of the main algorithms of RL and DRL is provided in order to have a better understanding on our final approach.

3.1 Unmanned aerial vehicles and multi-UAV Networks

Unmanned aerial vehicles are aircrafts capable of fly without the on-board control of any human operator. The level of control they need to fly establishes their degree of autonomy, that can be remote, semi-autonomous and autonomous control. Apart from the level of autonomy, several classification proposals were suggested in order to properly differentiate the various typologies of vehicles. Typically, the most used classification is based on the altitude they can reach, dividing them into High Altitude Platforms (HAP) and Low Altitude Platforms (LAP) [1]. The former usually fly at altitudes above 17km and are used for long term operations, while the latter can fly at altitudes between tens of meters up to few kilometers. The deployment of

LAPs is much more rapid, given their higher flexibility and dynamicity [27]. For this reason, they are usually exploited for time-sensitive applications (e.g., natural disaster scenarios). Another common classification for UAVs splits them between "fixed-wings" and "rotary-wings". Fixed wing UAVs have higher speeds and they can only move forward in order to fly. In contrast, rotary-wings have the ability to hover on a certain area and perform quick change of directions.
An important limiting factor for the deployment of UAV systems is given by the regulation. Several concerns regarding privacy, security, safety and data protection have been expressed in these years, and new regulations continue to be developed to restrict the operations with UAVs according to the country and the typology of vehicle [28].
In this work, a low altitude platform will be considered, made of UAVs that can dynamically move in the space (as the rotary-wings) in order to achieve a given goal. LAPs have already been considered in a wide range of applications. The most common are:

- **Aerial Photography**: UAVs can be equipped with nice cameras able to capture footage and register videos that otherwise would require expensive infrastructures.
- **Shipping and Delivery**: the delivery of different items (packages, groceries, medicine and so forth) exploiting UAVs is already used by different major companies, saving time and relieving the traffic congestion.
- **Disaster Management**: after a natural disaster, UAVs can efficiently be exploited to gather information on the target area thanks to different sensors they can be equipped with.
- **Precision Agriculture**: UAVs can help farmers to monitor the health of their crops through properly tuned sensors.
- **Search and Rescue**: when equipped with cameras or thermal sensors, UAVs are a powerful tool for surveillance. They can discover the location of lost or

injured people in harsh environments, as well as drop supplies to unreachable places.

Other significant applications of LAPs concern their use in wireless networking. As discussed in chapter 2, UAVs can be used as flying aerial base stations, supporting or replacing an existing terrestrial communication network and improving its level of coverage and capacity. The possibility to have a dynamic deployment of base stations, also varying their position in the three-dimensional space, allows for a better communication link between ground users and UAVs, thus having high chances to enhance the network performance. Another example is given by the use of UAVs with IoT networks. The flying nodes can provide an efficient uplink communication to the energy-limited devices, and, as already seen in chapter 2, forward this data to the cloud (or to the edge of the network) to process them. The main differences between these UAVs applications and the applications described in the previous list is given by the role the UAV has in the network. While in the former case the UAVs are seen as users of a wireless network (e.g. delivery or surveillance), in the latter applications they act as infrastructures to provide ground users with additional and enhanced services.

3.1.1 FANETs

One of the most common architectures for UAV networks consists in the centralized communication with a ground station. Each UAV is directly connected with a fixed infrastructure and all the messages must be routed through it. More distributed systems can rely on the ad-hoc networking between UAVs, so-called, Flying Ad-Hoc Networks (FANETs).

In the infrastructure-based approach, each UAV must be equipped with a sophisticated hardware in order to communicate with the correspondent base station. The stability of this communication link is not reliable, due to the nodes' movement and the typology of terrain. Furthermore, the dynamicity of the flying nodes is strictly constrained by the coverage range of the fixed infrastructure, and the gen-

eral performance of the network is correlated to one single entity, which represents a vulnerability for the network acting as single point of failure. As illustrated in Figure 3.1, FANETs relax the coverage constraint between UAVs and ground base stations by providing a direct communication between the flying nodes of the network.

FANETs are still deeply investigated in current researches; hence, it is difficult to

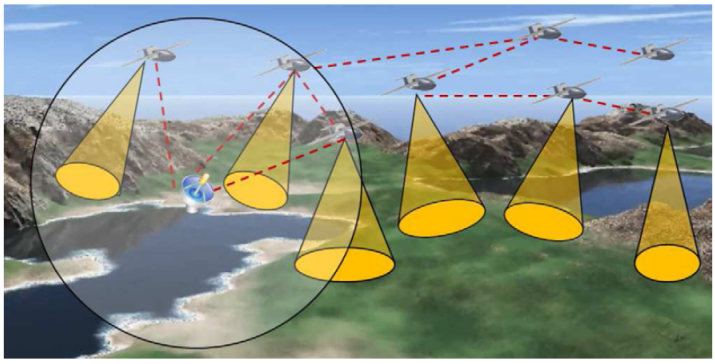


Figure 3.1: Example of FANET application extending the scalability of a multi-UAV system. Image by "Flying Ad-Hoc Networks (FANETs): a Survey" from Bekmezci et al. [29].

provide a proper background for this particular subject. After the highlight of the major applications in which flying ad-hoc networks can be exploited, we limit to give the reader an overview of the main FANET communication architectures.
As represented in Figure 3.2, UAV communications approaches can be different: direct communication with a ground station, communication in a satellite network, communication in a cellular network and, finally, communication via an Ad-Hoc network can be exploited. Despite in FANETs the communications between UAVs should occur over an ad-hoc network, other infrastructures can still be leveraged in order to keep the network connected to existing terrestrial wireless systems.

In **UAV direct communication** with a ground station, each flying node is connected to a ground control station through a star topology. This centralized approach brings several benefits to the network: fault tolerance when any of the UAVs incurs in a failure, easy synchronization of the entire network and global knowledge

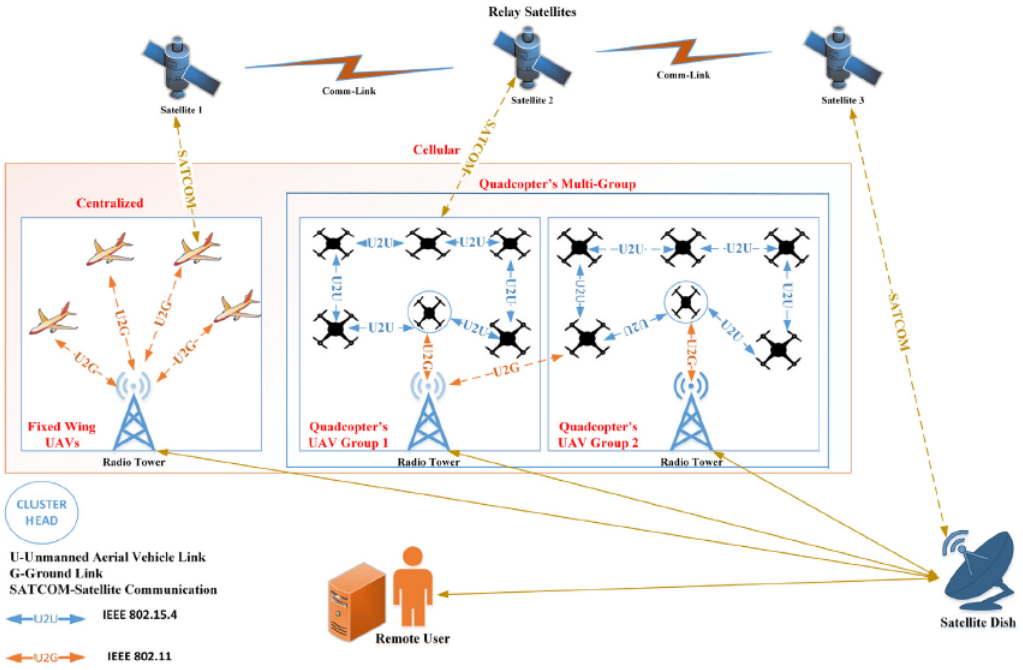


Figure 3.2: "Impression of a complete FANET architecture". Image by "Future FANET with application and enabling techniques: Anatomization and sustainability issues" from Srivastava and Prakash [30].

of the state of each node. On the other hand, the approach is not scalable with the growing number of UAVs, as the ground station becomes the bottleneck of the network. Another drawback is related to the presence of a single point of failure, whose breakdown would cause the failure of the entire flying network [31].

Communications in satellite networks are considered when the presence of a ground station is not possible due to the harsh terrain conditions (like mountains or oceans) [32]. These networks provide communication over long ranges and line of sight communicating nodes.

Connecting UAVs with the **cellular network** is a nice option to exploit already existing infrastructures and allow the incorporation of UAVs in LTE networks. Although the several advantages this solution brings, UAVs cannot be seen as normal ground user equipment, since they undergo a very different radio propagation. The

Third Generation Partnership Project (3GPP) has recently concluded its work to produce measurements aiming at the integration of UAVs communication in LTE networks [33].

UAV communication via ad-hoc networks is the main concept defining a FANET. The UAVs directly communicate between each other through an ad-hoc wireless network in a flat or hierarchical mesh topology, which brings scalability advantages with respect to a star topology. Furthermore, while single-UAV systems or centralized approaches have single point of failure, FANETs provide network survivability: if one UAV is no more capable of operating in the network, the FANET mission can still proceed with the remaining nodes.

These scalability and flexibility features make FANETs suited for working in the distributed scenario we want to propose, where each node can take decisions based on a local exchange of information.

3.2 Reinforcement Learning

Reinforcement Learning is a branch of Machine Learning that have always aroused the interest of many researchers. Recently, this interest has increased due to the progresses of Deep Learning (DL), which enabled the utilization of function approximators in RL algorithms; this technique is better known as deep reinforcement learning. Together with supervised and unsupervised learning, RL represents a machine learning paradigm that, instead of exploiting historical data as the previous two, uses a trial and error rule in a decision-making problem to gather information from the previously taken decisions. In these problems, an agent has to face a series of valuable decisions in a given environment according to a *reward function* that is returned after each action.
The aim of this section is to provide a complete understanding of the RL paradigm, in order to have a smooth transition towards the description of our approach.

3.2.1 Fundamentals of Reinforcement Learning

Reinforcement learning is a computational approach whose aim is to solve decision-making problems through a sequence of actions based on the interaction between an agent and the environment.

Agent, environment, and reward

The **agent** is an entity that interacts with the environment by performing actions (a_t) that are based on what it can observe of the current state s_t of the environment. The agent can choose over a set of possible actions, called *action space* ($\mathcal{A}$). The action space can be either discrete, thus the agent can decide over a finite number of possible actions, or continuous, where the actions are vectors of real values. The action space can strongly affect the performance of the algorithm, and different frameworks have been proposed according to the need for discrete or continuous action spaces.
The **environment** represents everything that is outside the agent. It provides an

observation of the current state and a value of reward every time the agent performs an action.

The **reward** (r_t) is identified by a scalar value that allows the agent to distinguish positive and negative actions. It defines the objective of the RL problem. It is very important to underline how the reward only refers to the current action performed in the current state; hence, it has not a global meaning. The achievement of a large immediate reward does not exclude the possibility to obtain very low rewards in the next steps. The agent has not to learn how to maximize the immediate reward, and sometimes it has to give up to large immediate rewards in order to achieve the global objective of the problem. Therefore, the main purpose of the agent is to maximize a value of cumulative reward, called **return** (G_t), defined as the total discounted reward starting from timestep t.

$$G_t = r_{t+1} + \gamma r_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \quad \gamma \in [0, 1) \tag{3.1}$$

In Equation 3.1, the concept of return is represented. The parameter $\gamma \in [0, 1)$ is the *discount factor*; it is mathematically useful towards the convergence of an infinite-horizon sum of rewards to a finite value. Furthermore, it shows the natural behavior of agents to prefer immediate rewards rather than future ones.

Markov Decision Process

Markov Decision Process (MDP) is a mathematical framework to model decision-making problems. RL is widely adopted to solve problems formalized as MDPs, which are defined by a set $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma >$ where:

- $\mathcal{S}$ is a finite set of states of the environment;
- $\mathcal{A}$ is the set of actions;
- $\mathcal{P}$ is a state transition probability matrix, which describes the probability of

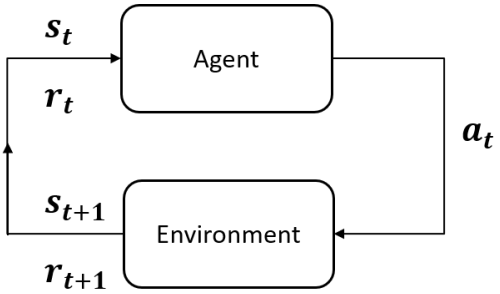


Figure 3.3: Interaction loop between the agent and the environment. Every action the agent performs results in a state and a reward, which are the input for the next iteration.

transition to a new state given the previous state and action: $\mathcal{P}_a(s, s') = \mathbb{P}[s_{t+1} = s' \mid s_t = s, a_t = a]$;

- $\mathcal{R}$ is the reward function associated to the action performed in a given state: $\mathcal{R}_a(s) = \mathbb{E}[r_{t+1} \mid s_t = s, a_t = a]$;
- γ is the discount factor.

Figure 3.3 shows how the agent interacts with the environment in a MDP problem. Formally, the probability for a given state to occur only depends on the previous state of the environment and not on the entire history of transitions. This property is commonly known as **Markov property**, formalized in eq. 3.2.

$$\mathbb{P}[s_{t+1} \mid s_t] = \mathbb{P}[s_{t+1} \mid s_1, \ldots, s_t] \tag{3.2}$$

Partially Observable MDP (POMDP)

Partially observable MDP is a generalization of MDP, where the state of the environment is not directly observable. At each step, the agent can only acquire a partial observation o_t that includes only a portion of the information about the current state. A POMDP is defined by a seven-tuple $\mathcal{T} = (S, A, P, R, \gamma, \Omega, \mathcal{O})$. The first five elements have the exact same meaning they have in a MDP. O is the set of possible observations o_i while Ω is the set of conditional observation probabilities $\mathbb{P}(o \mid s', a)$,

that is, the probability for the agent to observe o, given the new state s' after action a has been performed.

Model, policy, value function

The agent is basically defined by these three components: the a-priori **model** of the environment, the **policy** (π) and the **value function** (V). The model consists in the a-priori information the agent has with respect to the environment. It can influence the behavior of the agent by providing a belief about the environment's state transition, thus biasing from the beginning the agent decisions. A model of the environment can be provided or not. In the former case, the problem is tackled with a *model-based* approach. Otherwise, the method is called *model-free*-based. The policy is the mapping between the observed state and the action the agent will perform. It can be either deterministic ($a_t = \pi(s_t)$) or stochastic ($\pi(a_t|s_t) = \mathcal{P}[a_t|s_t]$). Naturally, the aim of RL is to allow the agent to find the optimal policy (π^*) in order to maximize the return, and hence, to find the best possible action to perform according to the current observation. The value function represents an expected value of the return g_t starting from the current state. While the reward is just an immediate feedback from the environment, the value function shows how the current action will also affect the future rewards.
There are two typologies of value functions:

1. the *State Value Function* ($V^\pi(s)$) (eq. 3.3) is the return that is expected by being in a certain state of the environment following the current policy π;

$$V_\pi(s) = \mathbb{E}_{\tau\sim\pi}\left[g_t \mid s_0 = s\right] \tag{3.3}$$

2. the *Action Value Function* ($Q^\pi(s)$) (eq. 3.4) is the return that is expected by taking action a starting from a state s, following the policy π.

$$Q_\pi(s,a) = \mathbb{E}_{\tau\sim\pi}\left[g_t \mid s_0 = s, a_0 = a\right] \tag{3.4}$$

Bellman Equations

Bellman equations (3.5) give the possibility to split equations 3.3 and 3.4 into two terms: the immediate reward r and the discounted future value functions $(\gamma V^{\pi}(s'), \gamma Q^{\pi}(s', a'))$. This gives the opportunity to solve those equations through simple and recursive sub-problems.

$$\begin{aligned} V^{\pi}(s) &= \sum_{a \in \mathcal{A}} \pi(a \mid s) \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} P(s', r \mid s, a) \left[r + \gamma V^{\pi}(s') \right] \\ Q^{\pi}(s, a) &= \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} P(s', r \mid s, a) \left[r + \gamma \sum_{a' \in \mathcal{A}} \pi(a' \mid s') Q^{\pi}(s', a') \right] \end{aligned} \tag{3.5}$$

The demonstration of the Bellman equations is beyond the purpose of this thesis and can be found in [34]. From Bellman equations, the Bellman optimality functions can be derived. Since the goal is to find the optimal policy π^*, at each step our agent has to choose the action that brings him to the state with the highest state value function $(V^*(s))$ or that carries the highest action value function $(Q^*(s, a))$ (eq. 3.6).

$$\begin{aligned} V^{*}(s) &= \max_{a} \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} P(s', r \mid s, a) \left[r + \gamma V^{*}(s') \right] \\ Q^{*}(s, a) &= \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} P(s', r \mid s, a) \left[r + \gamma \max_{a'} Q^{*}(s', a') \right] \end{aligned} \tag{3.6}$$

The Bellman optimality equations do not present a linear or closed-form solution; hence, they are solved with iterative methods. In the next section, a brief introduction on the most common iterative methods used in RL problems will be provided.

3.2.2 Tabular Methods

Tabular methods are iterative algorithms used to obtain an approximation of the value functions presented in eq. 3.5. These iterative algorithms are efficient when the state and action space of the MDP problem is small enough in order to represent

the value functions as arrays or tables.

Dynamic Programming

Dynamic Programming (DP) applies only to model-based problems, where the transition probability matrix $\mathcal{P}$ is defined. The general purpose of dynamic programming is to separate a single problem into several simpler subproblems. Once each of these subproblem is solved, their solutions are merged in order to obtain a result for the original problem. DP approaches can be divided into Policy Iteration and Value Iteration.

Policy Iteration consists of two steps: *evaluation* and *improvement*. In the first step, the algorithm evaluates how good the current policy is according to the estimation of the state value function $V^\pi(s)$ for each state. Since $V^\pi(s)$ is recursive, the algorithm starts from an initial value (e.g., 0) and performs several updates until a good approximated value, $V_k^\pi(s)$, with $k \in \mathbb{N}$, is found.

The second step is called *policy improvement*. Once the current policy is evaluated, the objective is to find a better policy by taking actions that do not belong to the current policy and that lead to an improvement of $V^\pi(s)$. The new action can be evaluated through the action value function $Q^\pi(s, a)$ (eq. 3.5). If $Q^\pi(s, a)$ is higher than $V^\pi(s)$, then the current action a is better than the one chosen by the policy, that now has to be updated. Each policy is iteratively updated every time the action value function associated to the new action is higher than the state value function related to the previous policy in a given state. The iterations stop when the improved policy does not differ from the previous one.

While Policy Iteration switches between the evaluation and the improvement step in order to return the best policy, the **Value Iteration** updates the $V^\pi(s)$ at each state without passing by the evaluation of any $Q^\pi(s, a)$. In this case, the Bellman Optimality functions are exploited (eq. 3.6). At each iteration, the algorithm updates the V function of all the states according to the equation of $V^*(s)$, until all the V functions converge towards the optimal one. The pseudocodes for policy iteration

and value iteration algorithms are presented in the Appendix at section A.1.

Monte Carlo Method

DP methods are based on the knowledge of the probability transition matrix $\mathcal{P}$ of the environment. However, this knowledge is not always easy to achieve, given that in real-world scenarios it is very hard to understand how a system may evolve. In these cases, the agent can try to learn this information directly interacting with the environment, exploiting *model-free* based algorithms.
Monte Carlo (MC) Method is one of them. The agent interacts with the environment by learning states and rewards. Then, according to the average return, it is able to evaluate the value functions. This method only works with episodic problems, where there is not an infinite time-horizon. Two learning policy evaluation methods exist:

1. First Visit: the evaluation of the value function for state s is given by the ratio between the total return over N episodes and the number of "first visits" of the agent in s. The pseudocode of this approach is presented in section A.2.

2. Every Visit: the evaluation of the value function for state s is given by the ratio between the total return over N episodes and the total number of visits of the agent in s.

Once the value function of each state is estimated, the policy is updated by making it greedy with respect to the value function ($V^*(s)$ in eq. 3.6) and another step of value estimation is performed, until the convergence of value function is met.
Monte Carlo methods are very simple to implement and they present a good convergence property. However, they can only be applied in episodic environments and the variance related to the value function is usually large.

Temporal Difference

Temporal Difference (TD) is another model-free approach that combines ideas from both DP and MC methods. It does not need a model, learning from sampled experience as in MC method, while it does not need to wait for the end of an episode to learn the estimates, like in DP. TD makes use of *temporal error*, which is the difference between two value function estimates of the same state. In its simplest form, TD updates the value function by combining it with the reward and the value function of the next state, as it is shown in Eq. 3.7.

$$V(s_t) \leftarrow V(s_t) + \alpha(r_t + \gamma V(s_{t+1}) - V(s_t)) \tag{3.7}$$

Eq. 3.7 represents how the value function of each state is updated every time the agent visits state s. The member inside the brackets is called *TD-error*, and, through a proper learning coefficient α, the value function is updated towards that error until convergence. The most common TD algorithms are **SARSA** and **Q-learning**.
SARSA is an on-policy method where the agent passes from a state-action pair to another state-action pair collecting a reward, thus generating the tuple (s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}) from where the name of the approach comes from. In this TD method, the agent tends to learn the action value function $Q^\pi(s,a)$ instead of $V^\pi(s)$, using an ϵ-greedy policy [35] in order to guarantee a proper trade-off between exploitation of the policy and exploration of the environment. At each step, the $Q^\pi(s,a)$ is updated according to eq. 3.8.

$$Q^\pi(s_t,a_t) \leftarrow Q^\pi(s_t,a_t) + \alpha(r_t + \gamma Q^\pi(s_{t+1},a_{t+1}) - Q^\pi(s_t,a_t)) \tag{3.8}$$

Finally, **Q-learning** is an off-policy method that differs from SARSA in the update of the $Q(s,a)$ function. At each step, the agent selects the action a according to an ϵ-greedy policy and updates the current action value function through a maximum function that always select the action returning the highest value of $Q(s_{t+1},a_{t+1})$, as better presented in eq. 3.9

$$Q\left(s_t, a_t\right) \leftarrow Q\left(s_t, a_t\right) + \alpha \left[r_t + \gamma \max_a Q\left(s_{t+1}, a\right) - Q\left(s_t, a_t\right)\right] \tag{3.9}$$

The main difference of Q-learning with respect to SARSA is that, while the latter updates the current $Q(s, a)$ according to a given policy (on-policy method), the former always performs this update through the action a_{t+1} that maximizes the action value function in the next state (off-policy method).
The advantages in using TD methods lay in their ability to work in infinite time-horizon environments with a lower variance with respect to MC methods. In literature, also combinations of these two methods have been proposed (TD(λ) algorithms), but they are out of the scope of this chapter, as the description of tabular methods learned so far just aims at clarifying the need for DRL approaches. The pseudocodes for the TD methods can be found in section A.3.

3.3 Deep Reinforcement Learning

Tabular methods are suited for those MDP problems with a well-defined state-space and action-space, where the main strategy is to create a look-up table in order to define the state value function for each state of the environment or the action value function for each state-action pair. It is straightforward that tabular methods do not scale with the potential high number of states and actions the problem may have. In environments presenting large or even continuous state and/or action space, problems regarding memory requirements for the look-up table storage as well as the slowness of convergence for the definition of the value functions in each possible state arise. Tabular methods are efficient when the agent is able to visit all the states once at least, thus updating the correspondent value functions. This is hard to achieve in continuous problems.
DRL was introduced for the first time by DeepMind, with the work "Playing Atari with Deep Reinforcement Learning" [36]. The main idea behind DRL is the exploitation of function approximators (i.e., neural networks), that, through a vector

of parameters θ, map each state or state-action pair into the correspondent value function (eq. 3.10).

$$V(s,\theta) \approx V_\pi(s)$$
$$Q(s,a,\theta) \approx Q_\pi(s,a) \tag{3.10}$$

The use of neural networks as function approximators reduces the training time and the memory required by the algorithm. In the next sections, an overview of neural networks will be presented together with the theory behind deep reinforcement learning, in order to explain the main algorithm exploited in the experiments of this thesis: the Deep Q-Network.

3.3.1 Fundamentals of Neural Networks

A neural network is a function approximator whose structure is made of artificial neurons. An artificial neuron, similarly to the biological one, consists in a set of input (dendrites) and one output (axon). The aim of the neuron is to process the input x_i though a **weighted sum** $\sum x_i w_i$, add a given **bias** b and apply an **activation function** f. The output of the neuron y is given by: $y = f(\sum_n w_i x_i + b)$. Figure 3.4 represents the architecture of a single neuron.

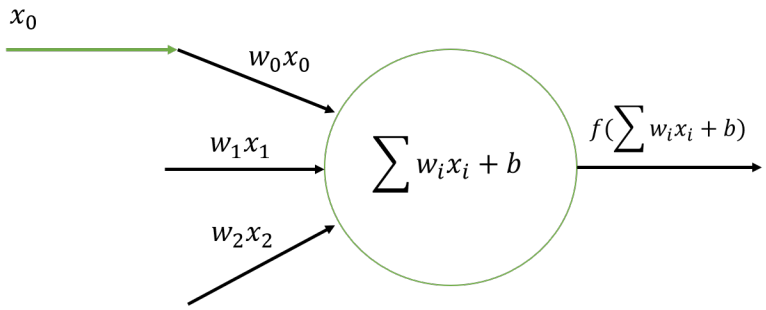


Figure 3.4: Architecture of a single neuron.

An artificial neural network is a structure of neurons organized in processing layers. In order to approximate the function of our interest (e.g., $Q^\pi(s,a)$), the parameters of all the artificial neurons, w and b, must be properly adjusted through

a *learning* process.

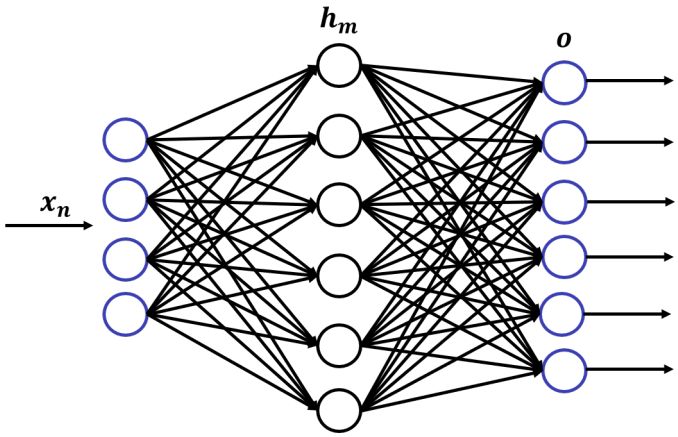


Figure 3.5: Fully connected feed forward neural network with one hidden layer.

In the simple feed forward neural network shown in Figure 3.5, the input layer receives the input vector of size $n \in \mathbb{N}$ with features x_i. Each feature is passed to the hidden layer and processed according to Eq. 3.11

$$h = f(w_1 x + b_1) \tag{3.11}$$

where h refers to the column vector of size $m \in \mathbb{N}$, correspondent with the number of neurons in the hidden layer. The matrix w_1 of size $m \times n$ corresponds to the weights of all the neurons present in the hidden layer, $b1$ is the bias vectors of dimension m and f is the activation function used in the hidden layer. Similarly, the output vector will be obtained through eq. 3.12

$$o = g(w_2 h + b_2) \tag{3.12}$$

where this time, o is the vector of the output dimension, g is the output activation function, and w_2 and b_2 are the weights and biases of the output layer.

Activation Functions

The activation function determines the output of each neuron. A proper choice must be done in the activation function of a given layer, since they can affect the convergence speed as well as the ability of the model to converge in the first place. The proper role of the activation function is to "fire" the neuron, which means to give an idea of how much the neuron's input is relevant for the model's prediction. The most used activation functions are listed below.

- Sigmoid $\rightarrow f(x) = \frac{1}{1+e^{-x}}$
- Hyperbolic Tangent $\rightarrow f(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
- Rectified Linear Unit (ReLU) $\rightarrow f(x) = max(0, x)$

Activation functions does not show discontinuous behaviors since they have to smoothly respond to input variations.

Learning Process

The learning process aims at setting the best set of parameters θ of the neural network in order to better approximate the function of interest. In traditional supervised learning, the learning process is split into three phases:

1. training;
2. validation;
3. testing.

For each input of the network $\mathcal{X}$, the correspondent output $\mathcal{Y}$ is already known and, during the training phase, it is used to update the parameters of the neural network. The validation phase is an intermediate step, where the current model is evaluated with input it never saw in order to fine-tune the hyper-parameters. In the testing phase, the final evaluation of the model is provided once it is completely trained.
The learning process for neural networks consists in the four steps that are going to be presented in the following sections.

Forward Propagation

The training data $\mathcal{X}$ are passed as input to the network and cross the entire structure until the predicted output $\hat{\mathcal{Y}}$ is evaluated.

Loss Function

A loss function $\mathcal{L}(\mathcal{Y}, \hat{\mathcal{Y}})$ estimates the error between the true value $\mathcal{Y}$ correspondent to input $\mathcal{X}$, and $\hat{\mathcal{Y}}$. The lower the value of this estimation and the better the prediction of the model. In the learning process, parameters θ are adjusted until sufficiently small values of loss function are obtained. A common example of loss function is given by the Mean Squared Error (MSE) (eq. 3.13)

$$L(\mathcal{Y}, \hat{\mathcal{Y}}) = \frac{1}{N} \sum_{n} (y_n - \hat{y_n})^2 \tag{3.13}$$

where N is the total number of input samples.

Backpropagation

Once the loss function is evaluated, this information is backpropagated in the network to update each weight and bias. The backpropagation algorithm [37] explains how the gradient of the loss function $\nabla L(\theta)$ is evaluated with respect to the parameters of the neural network. This global gradient is then propagated backwards in order for each single neuron to evaluate the local gradient of the loss function with respect to its weights and bias (i.e., the relative contribution of the neuron in the loss function).

Update

The final learning step consists in the update of θ. The most commonly adopted approach is the **gradient descent**. In simple terms, gradient descent tries to update the parameters of the loss function moving towards the negative direction of its gradient. In this way, the function will get closer to its minimum at every step,

according to a given learning coefficient α. The main update step is presented in Eq. 3.14.

$$\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta) \tag{3.14}$$

In the gradient descent method, the update step is performed every time the entire training set has been fed into the neural network, thus evaluating the gradient of the loss function as the gradient of the average loss obtained from each of the input samples (eq. 3.13).
Several approaches have been proposed over the years to improve the update step of the simple gradient descent algorithm. An example is given by the Stochastic Gradient Descent (SGD), that instead of using the whole training data to perform the update step, uses one training sample at a time to compute the loss and update θ. In the Mini-Batch Gradient Descent (MBGD), which lays in between the two previous methods, the gradient of the loss function is evaluated with a mini-batch of a given number of samples. Other common and more sophisticated optimizers are ADAM, AdaGrad or AdaDelta [38]. They improve the convergence of the gradient descent methods by introducing an adaptive learning coefficient.

3.3.2 Convolutional Neural Networks

Convolutional Neural Networks are a specific type of feed-forward neural network typically used to process and classify images or, more in general, any data structure with a two-dimensional topology. One of the pioneers of CNN is Yann LeCun, who, in 1988, proposed LeNet [39], a neural network for digits recognition. A typical CNN has the following architecture:

- Convolution Layer;
- ReLU Activation Function;
- Pooling Layer;
- Fully Connected Layer;

Convolution Layer

In a convolution layer, different filters (also called "kernels") are shifted over the image in order to perform a convolution operation. This operation consists in a dot product of the input data with the weights defined in the kernel, followed by the summation of the results. Different kernels learn different features of the input vector (e.g., edges, patterns), and they update their weights at every update step of the learning process. The output of the convolution step consists in as many feature maps as the number of defined kernels, containing all the results of the convolution operations.
When dealing with CNN, the following terms must be defined:

- Stride S: the number of input pixels the kernel shifts after every convolution operation;
- Padding P: consists in adding 0 values to the input vector in order to make the feature map's dimension of the same size of the input map.

Given these two definitions, and provided the dimension of the input vector, (W, H), and the size of the filter, $F \times F$, the size of the feature map is evaluated as follows:

$$\begin{aligned} W_{map} &= (W - F + 2P)/S + 1 \\ H_{map} &= (H - F + 2P)/S + 1 \end{aligned} \tag{3.15}$$

A representation of the convolution operation is shown in Figure 3.6

ReLU Activation Function

After the creation of the feature maps, the ReLU activation function is usually applied to all the generated values, in order to see if a given feature has been detected or not. Eq. 3.16 represents the activation function applied to the hidden neurons of a feature map, in a case where the kernel has dimension 5×5, with b being the bias associated to that feature map, σ the activation function, $w_{l,m}$ the kernel weights at

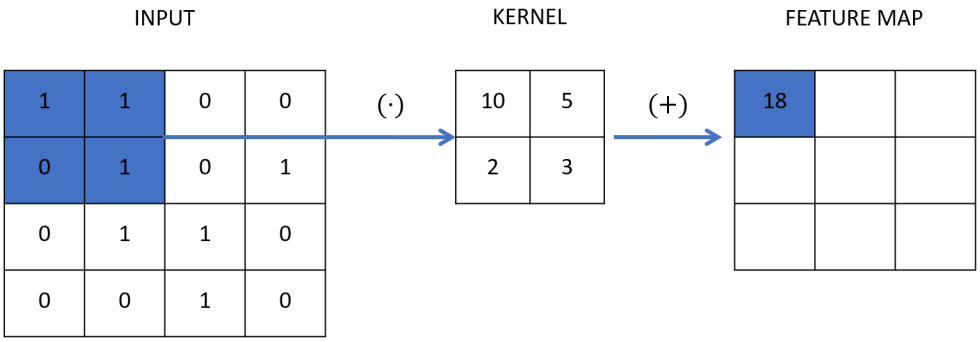


Figure 3.6: Convolution operation on a 4x4 input vector through a 2x2 kernel. According to eq. 3.15, the feature maps have dimension 3x3.

position (l, m), and $a_{j+l,k+m}$ the input values on which the convolution operation is performed resulting in the hidden neuron at position (j, k).

$$\sigma\left(b + \sum_{l=0}^{4}\sum_{m=0}^{4} w_{l,m} a_{j+l,k+m}\right) \tag{3.16}$$

The great innovation brought by CNN consists in the possibility to take into account the spatial structure of the image, thus finding inner patterns that would be impossible to determine with a traditional neural network.

Pooling Layer

It is good practice to insert a pooling layer after a convolution layer. Its main objective is to simplify the information brought by the output of the convolution layer by condensing the feature map in a smaller dimension. This operation is useful to lighten the computational cost of the learning process as well as to prevent overfitting. Each unit of the pooling layer summarizes a given sub-region of the previous feature map by keeping only the maximum value of that sub-region (*max-pooling*, Figure 3.7) or the average (*average-pooling*).
The main idea behind pooling layers is that, after a given feature is detected, its exact position is not as important as its relative position with respect to the other

detected features.

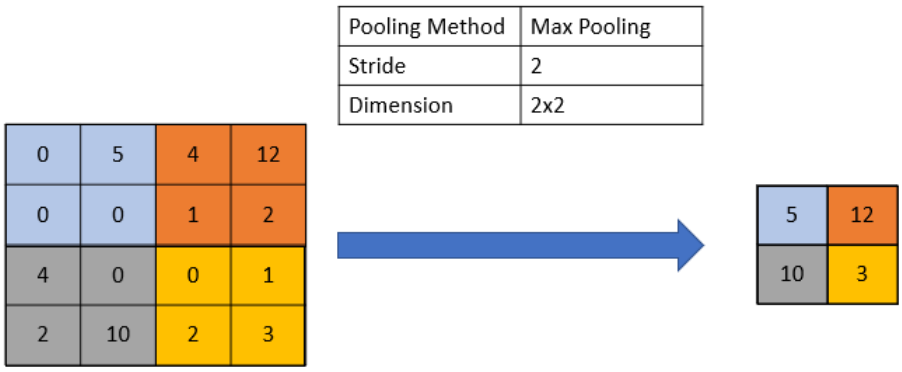


Figure 3.7: Example of max-pooling applied on a 4x4 matrix with stride factor equal to 2 and filter dimension of 2x2.

Fully Connected Layer

The last part of a CNN architecture is a fully connected layer, which is used to flatten the results. Every neuron of the last pooled matrix is connected to every flat neuron. The outcome of the neural network at this point is exactly the same of a traditional neural network, and the learning process does not differ.
Figure 3.8 shows the specific architecture used for LeNet [39], where all the layers of a CNN can be appreciated.

3.3.3 Value-based DRL

Value-based DRL are the first class of DRL algorithms, putting together the strengths of reinforcement learning and deep learning. This application is not straightforward, given the different learning processes of the two paradigms: while DL requires large amounts of already labeled data, RL learns from a noisy and delayed reward that follows every action. Furthermore, DL assumes every input data to be independent

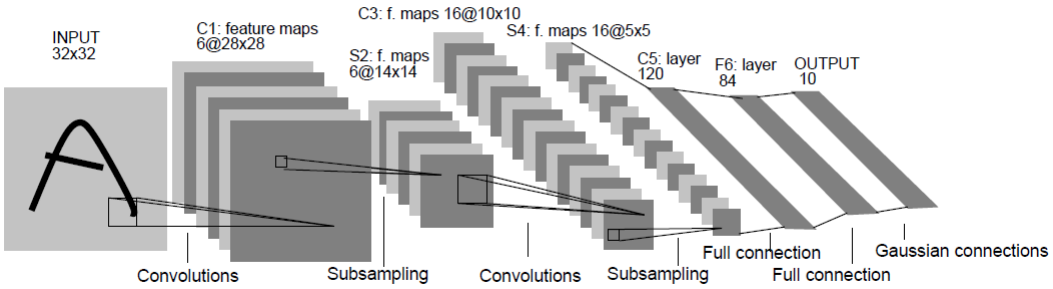


Figure 3.8: Architecture of LeNet, a CNN for digit recognition. It presents two sequential convolution and pooling layers and two fully connected layers.

from the others, while the observations of RL are highly correlated between them. The strategy behind value-based DRL overcomes these challenges by learning an approximator $Q_\theta(s, a)$ to infer the optimal action value function $Q^*(s, a)$. The approximator is a neural network with parameters θ, generally called *Q-Network*.

Deep Q-Network (DQN)

DQN is a value-based DRL method that can be seen as an improvement of the Q-learning algorithm (section 3.2.2). At every step of the DQN approach, the Q-Network takes as input the observation of the agent and outputs one value per each possible action, correspondent with its action value function. Figure 3.9 shows an intuition of a Q-Network architecture. The learning process of the Q-Network consists in the minimization of the loss function $L(\theta)$ defined as follows:

$$L\left(\theta_i\right) = \mathbb{E}_{s,a\sim\pi}\left[\left(y_i - Q\left(s, a; \theta_i\right)\right)^2\right] \tag{3.17}$$

The loss function depends on the current predicted $Q_\theta(s, a)$ and the target value y, which is presented in eq. 3.18.

$$y_i = \mathbb{E}_{s'\sim E}\left[r + \gamma \max_{a'} Q\left(s', a'; \theta_i\right) \mid s, a\right] \tag{3.18}$$

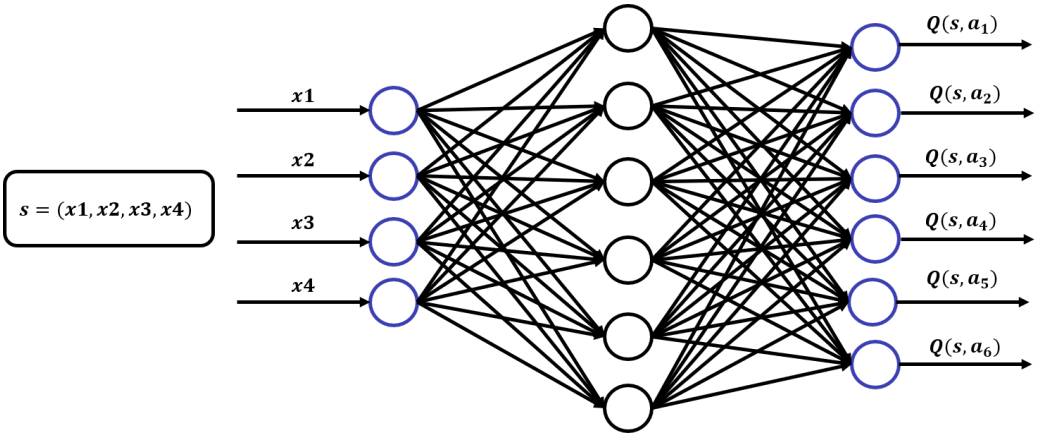


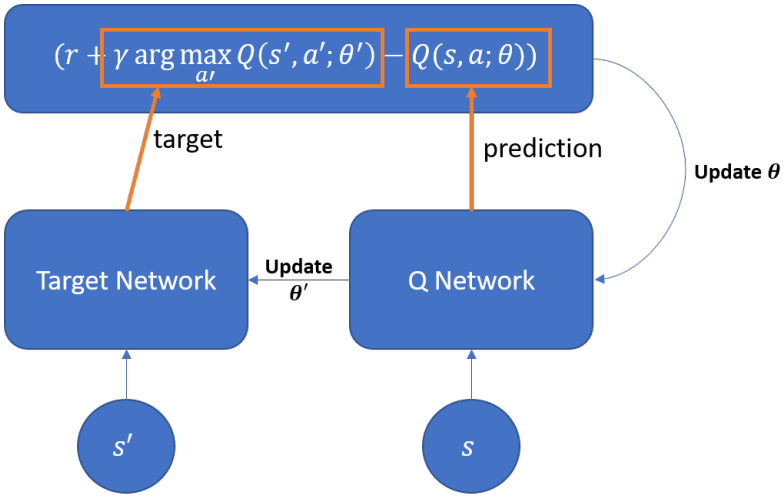
Figure 3.9: Example of a Q-Network. The input vector corresponds with the observation of the agent. The network has one output per each possible action, correspondent with the Q function associated with that action.

The argument of the loss function has exactly the same meaning of the TD-error presented in section 3.2.2. It is interesting to highlight how the target value is not fixed, but strongly depends on the parameters of the Q-network, in contrast with the learning process exploited in deep learning.
The continuous changing in the target parameters brings high instability to the convergence of the loss function. The first solution to improve the learning process is to evaluate the target value y with a different neural network called **Target Network**. This network has the exact same architecture as the actual function approximator, but its parameters θ' are updated much more slowly than the function approximator's ones θ, usually through a Polyak averaging (eq. 3.19).

$$\theta' = \tau \cdot \theta' + (1 - \tau) \cdot \theta, \qquad \tau \in [0, 1] \tag{3.19}$$

As shown in Figure 3.10, when the Q-Network has to be trained, the target value is evaluated through the target network, while the prediction through the Q-Network.

Another fundamental concept of DRL is the **experience memory replay buffer**. The learning process performed with data obtained as soon as they are acquired from the agent observations is not a proper strategy, since, while neural networks expect



$$transition = (s, a, r, s')$$

Figure 3.10: Training step of the Q-Network exploiting a target network with parameters θ'.

their input samples to be independent between them, subsequent observations are temporally correlated. At every new step performed by the agent, the transition made of the state s_t, the action a_t, the obtained reward r_t and the next state s_{t+1} is stored in a buffer of a given size. During the learning process, a random mini-batch of transitions is sampled from the buffer and used to perform one training step. In this way, a wider and independent set of possible state-action pairs is taken into account, thus improving the network update.

The DQN algorithm applying the concept of target network and experience memory replay buffer is presented in Algorithm 1. Once the replay buffer and the two neural networks are initialized, the agent gets the first observation from the environment. According to an ϵ-greedy policy, the next action is selected and the transition (o_t, a_t, r_t, o_{t+1}) is stored in the replay buffer. When the latter is full, a mini-batch of transitions are sampled at every step and used to update the parameters of the Q-network according to Eq. 3.17. Finally, also the parameters of the target network are updated through the Polyak averaging presented in Eq. 3.19. The loop continues

Algorithm 1 Deep Q-Network with Experience Replay

Initialize Replay Memory $\mathcal{D}$ to capacity $\mathcal{N}$
Initialize Q-Network Q_θ with random parameters
Initialize the target network $Q_{\theta'}$ with $\theta' = \theta$
for episode = 1, M **do**
  Initialize the state s_0 and the observation of the agent o_0
  for t=1, T **do**
    with probability ϵ select a random action a_t
    otherwise select $a_t = arg\max_a Q(o_t, a; \theta)$
    execute action a_t and get the next observation o_{t+1} and reward r_t
    store the transition (o_t, a_t, r_t, o_{t+1}) in $\mathcal{D}$
    sample a random mini-batch of transitions (o_j, a_j, r_j, o_{j+1}) from $\mathcal{D}$
    set $y_j = r_j + \gamma arg\max_a Q(o_{j+1}, a; \theta')$
    perform gradient descent step on $(y_j - Q(o_j, a_j; \theta))$ updating θ
    update θ' through eq. 3.19.
  end for
end for

over several episodes, until a maximum number of episodes is reached.
Two variants of DQN that are worth to be mentioned have been proposed over the years: Double DQN [40] and Dueling DQN [41], but they have not been considered in this work.

3.3.4 Policy Gradient Methods

Although the approach used in this thesis is based on the DQN algorithm (a value-based approach), it is necessary to deal with the main concepts of another approach of DRL given its importance: the policy gradient methods [42].
Policy gradient methods aim at directly finding the best policy $\pi(a \mid s)$ instead of focusing on the action value functions. They operate by optimizing a policy performance metric, $J(\theta)$, which depends on the current policy and the expected reward (eq. 3.20).

$$J(\theta) = \mathbb{E}\left[\sum_{t=0}^{N} r\left(s_t, a_t\right); \pi_\theta\right] \tag{3.20}$$

The maximization of the objective function $J(\theta)$ is performed through gradient ascent. It updates the parameters θ towards the direction of the gradient of $J(\theta)$, according to eq. 3.21.

$$\theta_{t+1} \leftarrow \theta_t + \alpha \nabla_\theta J(\theta_t) \tag{3.21}$$

Actor-Critic Architecture

One of the most adopted policy gradient methods is the Actor Critic [43]. In DRL, two neural networks, namely *actor* and *critic*, are the components of this architecture. The actor has to define the proper policy π_θ, while the critic estimates the action value function (similarly to the Q-network) associated to a state-action pair. In simple terms, the critic estimates the value function and the actor updates its policy in the direction suggested by the critic through a policy gradient approach. Both networks train their parameters through the *TD error* discussed in section 3.2.2. Figure 3.11 represents the typical actor-critic architecture.

Deep Deterministic Policy Gradient (DDPG)

DDPG is a policy gradient algorithm that exploits the actor-critic architecture. The actor learns the policy π while the critic learns the $Q(s, a)$ given a specific state-action pair. It is applied to those environments presenting a continuous state and action space, where a DQN algorithm cannot be exploited given the structure of its Q-network. In fact, when the optimal value-function $Q^*(s, a)$ is known, the best action the agent can perform is given by: $a^*(s) = arg\max_a Q^*(s, a)$. When the action space is continuous, this operation is not trivial anymore.
DDPG exploits four neural networks:

- actor π with parameters θ;
- critic Q with parameters σ;
- target actor π' with parameters θ';
- target critic Q' with parameters σ'.

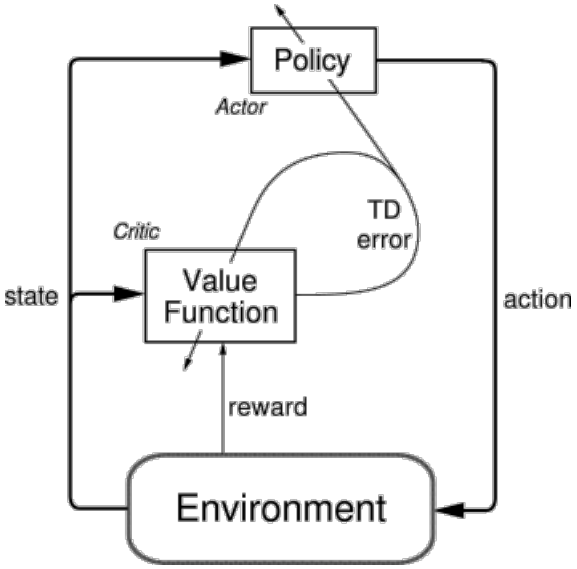


Figure 3.11: The actor critic architecture. The actor is the policy that directly returns the action to take according to the input state. The critic is represented by the "Value Function". The critic is used only during the training. Once the policy is trained, it will be the only one to infer the actions that the agent has to perform.

At the beginning, all the networks have random parameters, and the agent starts to explore the environment by exploiting its actor network and populating the replay buffer. When the buffer contains sufficient samples, the learning process can start: a mini-batch of size N is sampled and used to update the parameters of the critic (σ) computing the loss function L between the predicted Q and the target value y obtained from the target critic and target actor.

$$\begin{aligned} L &= \frac{1}{N} \sum_i \left(y_i - Q\left(s_i, a_i \mid \sigma \right) \right)^2 \\ y_t &= r\left(s_t, a_t \right) + \gamma Q'\left(s_{t+1}, \pi'\left(s_{t+1} \mid \theta' \right) \mid \sigma' \right) \end{aligned} \tag{3.22}$$

Eq. 3.22 represents the loss function obtained in order to train the critic. It shows how the target value y_t is obtained from the target actor to derive the action to be performed in the next state $a' = \pi'(s_{t+1} \mid \theta')$ and from the target critic to get the Q'.

After the update of σ, also the actor performs a training step. Its aim is to select the action that maximizes the Q value given the state. Hence, the policy performance metric $J(\theta)$ that has to be maximized is evaluated as the $Q(s, a)$ value, where s is the current state and a is the action chosen by the current policy π. In order to calculate the policy loss, the gradient of the policy performance metric can be obtained with respect to the policy parameters θ (eq. 3.23). Finally, the weights and biases of the network are updated using the gradient ascent as optimizer.

$$\begin{aligned} J(\theta) &= \mathbb{E}\left[Q(s,a)|_{s=s_t, a=\pi(s_t)} \right] \\ \nabla_\theta J(\theta) &\approx \frac{1}{N} \sum_i \left[\nabla_a Q(s, a \mid \sigma)|_{s=s_i, a=\pi(s_i)} \nabla_\theta \pi(s \mid \theta) \Big|_{s=s_i} \right] \end{aligned} \tag{3.23}$$

A complete explanation of DDPG and demonstration of the equations presented in this section can be found in the original paper presented by Google DeepMind: "Continuous Control with Deep Reinforcement Learning" [17].
The main advantages in using this method rely on the possibility to deal with continuous action spaces and to learn stochastic policies. However, DQN is the best option when the environment presents a large state-space but a limited and discrete action-space. An idea of DDPG algorithm is presented through its pseudocode in section A.4.

Chapter 4

Problem Formalization and Algorithm Setup

The aim of this chapter is to provide the mathematical framework exploited to model the decision-making problem described in section 1.1 in terms of observations from the environment, actions of the agents and gained rewards. Subsequently, the reinforcement learning approach allowing each agent to map a given observation to a reasoned action is presented and described in detail.

4.1 Problem Formalization

As stated in section 3.2.1, MDPs are one of the most common and convenient methods to model a decision-making problem. Also, all the previous research on similar scenarios showed how the formalization of a problem through a MDP or a POMDP is the proper approach when RL is the chosen methodology to tackle it.
Since multiple agents are considered in this scenario, the proper model that has to be chosen is a Decentralized-POMDP, which is the multi-agent variant of a POMDP. Multiple agents acquire partial observations of the environment and may share their local knowledge with the others in order to learn how to cooperate and reach a

common goal.

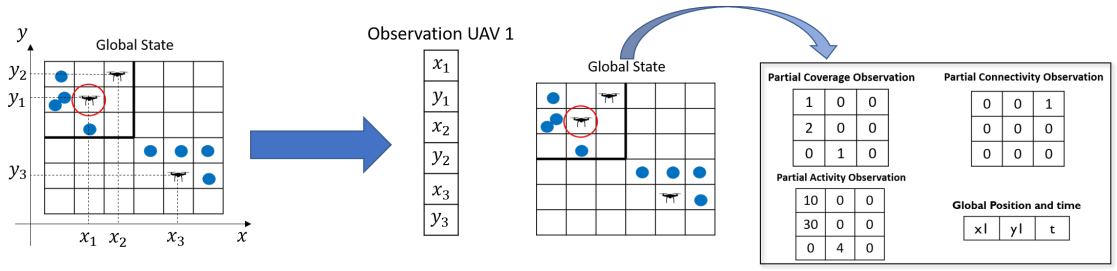
4.1.1 Observations

The observation consists in the partial knowledge that the agent is able to acquire about the entire state. The decision regarding the typology of observation for each agent has been tackled according to what actually a UAV is able to observe thanks to its communication capabilities. We decided to propose two slightly different POMDP models that only change in their observation:

1. in the first model, the partial observation of the agent consists only in its position and the position of all the other agents of the system. No information is provided regarding the covered GUs and their level of activity. This observation can be seen as a linear vector of coordinates (Figure 4.1a);

2. in the second model, the observation is more similar to the actual perception that the UAVs can have in the network. It consists of: the relative positions of the detected neighbors, the relative positions of the GUs that are in range with that UAV, and the level of activity of the GUs that the UAV is covering at the very moment. This observation can be seen as a sequence of relative maps of the environment (Figure 4.1b) whose size is limited by the agent's observation range. Each relative map is a discretization of a part of the environment in a given number of cells; hence, the partial observation will return the summation of neighbor UAVs, GUs and levels of activities detected in each cell.

The main differences between the two typologies of observations lie in the number of input that have to be passed to the learned policy. While in the first case this input is fixed, assuming that the number of UAVs does not change in time, the second scenario presents a much more variable situation. The relative positions of neighbor UAVs, as well as the relative position of near GUs, has not a constant size; hence, the storage of this information in discrete maps significantly helps in dealing with the observation of these variable-size input.



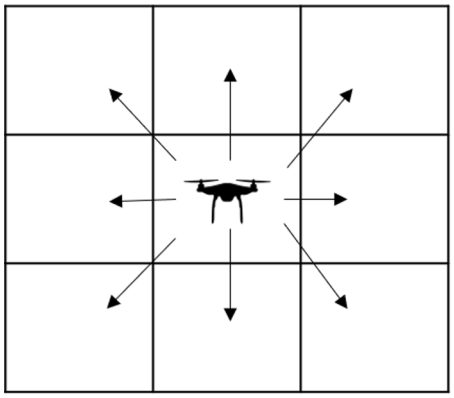
(a) (b)

Figure 4.1: Two different observation scenarios: in (a) each agent has the complete knowledge about the position of all the other UAVs. The observation is represented by a scalar vector. In (b), the observation is represented by a sequence of maps, each map bringing the relative information of: (*i*) relative position of neighbor agents, (*ii*) relative position of near GUs and (*iii*) relative level of activity of the covered GUs.

The main motivation behind the decision of two different typologies of observations is to investigate whether a partial observation with only relative information can achieve similar results when compared to a more global and centralized information about the environment. Despite both observations consist of a partial knowledge of the environment, in this dissertation we will refer to the first observation case as **centralized** while the second observation case will be indicated as **partial**.

4.1.2 Actions

The formalization of the action space is another fundamental step of the designing process, in order to properly define how the agent interacts with the surrounding environment. Given the wide degree of freedom a UAV can have in terms of mobility, we decided to reduce the space of possible actions by defining nine different and discrete movements the UAV can perform at each time step. This limit allows the agent to take simple decisions in the environment and to reduce the complexity that a continuous action space would have brought. The nine possible actions that the agent can take correspond with nine possible discrete movements in a two-dimensional



	Actions
0	[0,0]
1	[1,0]
2	[0,1]
3	$[-1,0]$
4	$[0,-1]$
5	[1,1]
6	$[-1,1]$
7	$[-1,-1]$
8	$[1,-1]$

Figure 4.2: Discrete action space of the modeled POMDP. At every time step, the agent can decide between nine possible actions corresponding with a movement in the two-dimensional space.

space (Figure 4.2), while the altitude of the UAVs is considered as a fixed parameter.

4.1.3 Reward

The reward function represents the most crucial feature in the formalization of a MDP. After the study of how previous works (proposed in chapter 2) modeled their reward functions, and a consistent trial and error procedure, we defined it as the summation of two components returned every time an action is performed:

- the number of GUs that the current UAV is associated with, normalized by the total number of GUs present in the network;
- the resource utilization of the current UAV, normalized by the total workload required from the GUs.

With the term *resource utilization*, we refer to the total level of GUs' activity that the UAV is handling at the moment. Given this value of reward function, each agent will try to maximize its own level of cumulative coverage and resource utilization over time. The UAVs should learn either to find undiscovered GUs, in order to improve their level of coverage, or to place themselves near other UAVs in order to

share computing resources through the implemented resource sharing algorithm, and hence, improve their level of utilization.

4.2 Algorithm Setup

The next step in the design of the system is the formalization of the exploited reinforcement learning algorithm. The aim is to make all the agents in the environment select the optimal *action* at every timestep, in order to maximize the return in terms of coverage and resource utilization.
DQN has been the proposed RL approach to deal with this problem. This choice was driven by the astonishing results obtained by deep reinforcement learning in the past few years, together with the intrinsic structure of the problem itself: a large observation space, in addition to the limited action space, make DQN the perfect approach to solve this decision-making problem. Even though several already implemented algorithms are available in the open-source community, none of them can be directly applied on the specific environment previously presented. Hence, starting from the main idea of how DQN works, we decided to implement the algorithm from scratch, using the ready-made implementations as guidelines. The reason behind this choice is twofold: (*i*) the implementation of an algorithm from scratch is the best way to understand how the algorithm actually works in every aspect; (*ii*) the algorithm can be specifically designed for the architecture of the proposed environment.

4.2.1 Neural Networks Design

The two different observations the agents can get from the environment, described in sub-section 4.1, necessitate two different neural network structures. The first observation type only returns the positions of all the agents. Considering the guidelines provided by already implemented algorithms, and, more importantly, the original work presenting DQN [4], we decided to implement a shallow fully-connected Q-network with one single hidden layer (same structure presented in Figure 3.9). This

simple architecture is also justified by a trial and error procedure, that revealed how the performance of the training has not a significant impact with respect to the number of implemented hidden layers, especially if compared to the additional training time needed as the number of layers increases. Furthermore, we also investigated the proper activation function and loss function. We decided to apply the ReLU as non-linearity, given its computational simplicity and the reduced likelihood of vanishing gradients. The adopted loss function has been the *smooth l1 loss* (Eq. 4.1), which combines the advantages of the *l1 loss* and the *l2 loss* (Eq. 4.2). Finally, the Adam optimizer has been exploited for the update of the network's parameters.

$$L_{1;\,\text{smooth}}\,(y,\hat{y}) = \begin{cases} |y-\hat{y}| & \text{if } |y-\hat{y}| > \alpha \\ \frac{1}{|\alpha|}(y-\hat{y})^2 & \text{if } |y-\hat{y}| \leq \alpha \end{cases} \tag{4.1}$$

$$\begin{aligned} L_1(y,\hat{y}) &= |y-\hat{y}| \\ L_2(y,\hat{y}) &= (y-\hat{y})^2 \end{aligned} \tag{4.2}$$

The second observation type is closer to a more realistic perception a UAV can have in the environment. Given that the number of covered GUs and neighbor UAVs may change over time, we organized this observation into two-dimensional maps (Figure 4.1b), passing them as the input of the Q-Network with a CNN architecture. This approach allows to overcome two main problems:

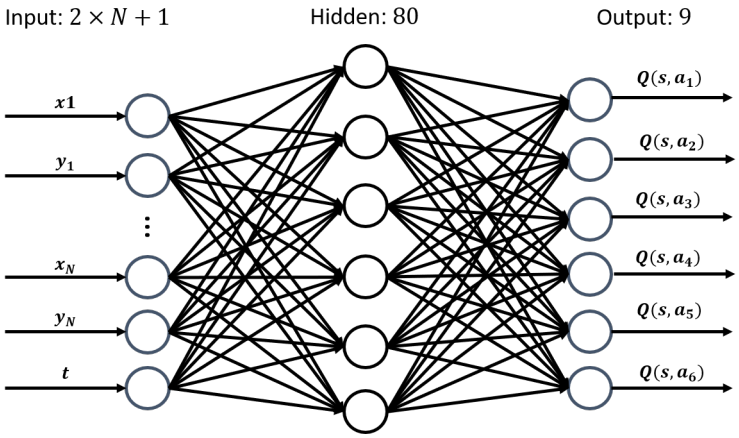
1. dealing with input of variable dimensions;
2. considering the spatial correlation of features in the neural network.

The first problem emerges when a flat neural network architecture is considered. The input neurons should vary according to the variable number of neighbor UAVs or covered GUs. If this information is organized in two-dimensional maps and passed to a CNN, this problem is completely addressed. Furthermore, the use of a CNN allows the automatic detection of spatial features and the relative orientation between them, thus making this structure particularly suited for our purposes. The design of

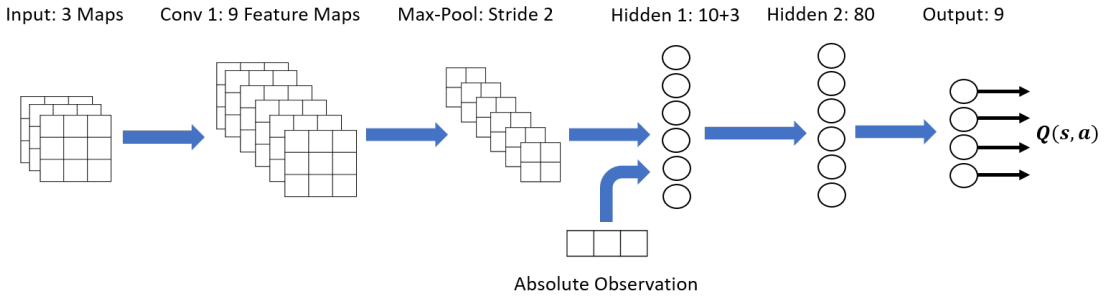
the CNN architecture has been inspired by [4] and [13], in addition to an extensive trial and error procedure. We opted for a CNN with one convolution layer with 9 filters of 3x3 dimension and one *max-pooling* layer with kernel size 2x2 and stride of 2 for computational efficiency and robustness enhancement. After these operations, the results are flattened and given as input to the last part of the CNN, which is made of three fully connected layers. The first one is made of $10+3$ hidden neurons, where the additional three elements correspond with the vector (x_u, y_u, t), containing the global coordinates of the agent and the value of timestep. This information has to be added since the only knowledge of relative information is not enough for the agent to orient itself in the target area. The last hidden layer's size is 80 neurons and the output layer has exactly the action space dimension. The graphical representation of the two implemented neural network architectures is provided in Figure 4.3.

4.2.2 Training Process

The algorithm proposed for the designed Q-networks training has been presented in Alg. 1. The only difference is that multiple agents interact with the environment at the same time, mutually affecting their decisions and fostering their cooperation. Given the homogeneous capabilities of the agents considered so far, they will leverage the same policy through the training of a single Q-network. The use of the same policy fosters the scalability and the time complexity of the algorithm, by avoiding to train one Q-network per each agent of the system. Figure 4.4 schematizes the DQN training procedure over the simulation. The agents interact with the environment following the current policy and creating transitions. These transitions are appended to the replay buffer, whose batches are used to train the Q-network through the loss function presented in Eq. 3.17. The target network's parameters are then updated using the Polyak averaging (Eq. 3.19).



(a)



(b)

Figure 4.3: Neural Networks design for the multi-agent decision making problem. (a) represents the structure of the shallow neural network, which takes as input the positions of all the agents in the systems. (b) represents the design of the CNN, where the input is the concatenation of the three relative maps that the agent can build together with its absolute observation. For both the architectures, the output is the Q-value for each of the nine possible actions.

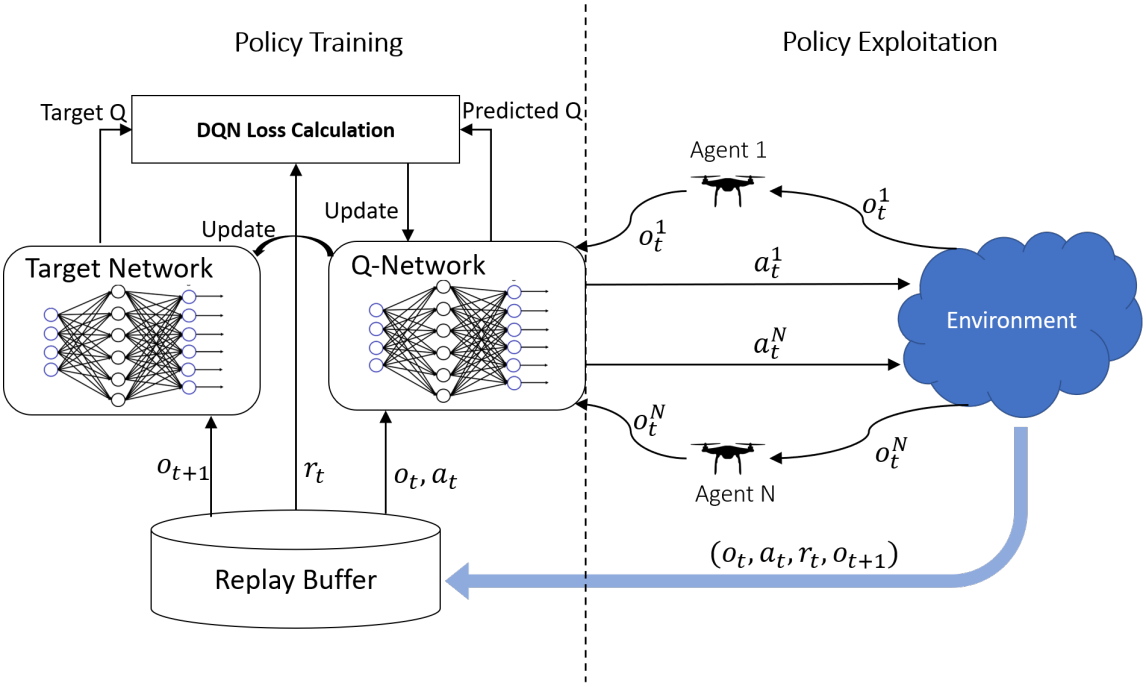


Figure 4.4: DQN Training Process with Multiple Agents. On the right part, the agents exploit the learned policy by using Q-networks with the same parameters. While they interact with the environment, the Q-network is trained (left part) according to the DQN algorithm (1), leveraging on the replay buffer and target network.

Chapter 5

Tools and Implementation

This chapter aims at describing the main tools that have been exploited in this work and how they have been merged together in order to implement the proposed approach. Once each tool has been properly presented, the followed methodology will be outlined in order to provide a detailed overview of the designed framework. Its main strengths as well as some relevant problems that we encountered will be also highlighted towards the end of the chapter.

5.1 Tools

Before explaining how the scenario has been simulated, the main tools used in this work should be described. They allowed us to: (*i*) develop the multi-agent RL algorithm, (*ii*) to train it on a simplified scenario and, (*iii*) to test it in a more realistic environment.

5.1.1 PyTorch

PyTorch is a fast machine learning library developed by Facebook's AI Research Lab [44]. It provides a very intuitive framework in a Pythonic programming style for the development of deep learning models. The two most important features that

distinguish PyTorch from other deep learning libraries consist in the computation of tensors with a strong GPU acceleration and the automatic differentiation for the creation and the training of the neural networks.
Tensors are data structures (like arrays) that can run on GPU. In this way, the time for numerical computations is speed up by a factor of fifty at least.
The **automatic differentiation** consists in the possibility to numerically compute the derivative of a function. It allows to adjust all the parameters of a neural network during the backpropagation and update steps (section 3.3.1). The *"autograd"* feature of a tensor object supports this automatic differentiation, thus allowing the network training.
Apart from tensors and automatic differentiation, other fundamental features of PyTorch are:

- **nn.Module Class**: module for the neural network creation. It defines the structure of the model by providing layer architectures (linear, convolutional, recurrent), activation functions and loss functions.
- **Optimizer**: the *optim* package defines the optimizer used to update the parameters of the network. It automatically implements common optimization algorithms (ADAM, gradient descent, AdaGrad) on the tensor that is storing them.

The choice of using PyTorch as deep learning framework has been based on all these convenient features. Furthermore, PyTorch provides a strong community support, which is fundamental to increase the developers' productivity.

5.1.2 Network Simulator 3 (ns-3)

Ns-3 is a free and open source discrete-event network simulator that runs on Unix and Linux-based systems and is developed in C++. It provides models for packet data networks, tools for their analysis and an engine to handle the simulation experiments. This simulator is based on the following key simulation objects:

- **Node**: it is the connection point of the network, that can be associated with network devices, protocol stacks, mobility models and applications. It is defined by the class *Node*;
- **Channel**: it represents the propagation medium of the signal. It defines the condition of propagation in terms of delay and propagation loss.
- **NetDevice**: it represents the physical network interface of the node. It is defined by the class *NetDevice* and has to be associated with a *Node Pointer* and attached to a Channel.
- **Application**: it is the user defined process that generates the traffic and guide the whole simulation. Several already prepared frameworks for different types of applications are present in the simulator. However, by inheriting from the class *Application*, it is possible to customize traffic patterns.
- **Mobility Models**: they allow the nodes to have a mobility pattern. There are several already defined mobility models, such as the Constant Position, Random Position, Random Velocity, Waypoint. One can always define a custom mobility model through the class *Mobility*.

The software infrastructure allows the development of sufficiently realistic simulation models, which can be even used as network emulators and interconnected with the real-world.
This simulator has been useful towards the end of this work, in order to define a more realistic scenario where UAVs exchange packets with ground nodes and between themselves while exploiting the policy learned in Python. It has been useful to have an idea of how the learned model would behave in a more realistic scenario, without changing the structure of the observation that the agents could get.

5.1.3 Ns-3 - Gym

The last tool that has been implemented in this thesis is called ns-3-Gym [45]. This toolkit provides a framework to apply RL in the networking environments that are

simulated with ns-3. It allows to represent an ns-3 simulation as an environment in the OpenAI Gym framework [46], a toolkit for developing and comparing RL algorithms. Ns-3 - Gym has been developed in order to foster the application of ML approaches to modern networking environments, by allowing the feeding of RL models with data generated by the simulator. The motivations are multiple: the training of novel RL-based networking algorithms needs for several experiments, that usually are not affordable in real-world scenarios. Furthermore, provided the realistic networking environments that are implementable in ns-3, the policy learned in simulation may be sufficiently robust also in real-world implementations.
The architecture of ns-3 - Gym (Figure 5.1) is made of three main components: the network simulator, OpenAI Gym and a middleware that handles the exchange of messages (i.e., the observations and the actions) between the simulation and a Python agent.

The middleware is made of two elements: the **Environment Proxy** and the

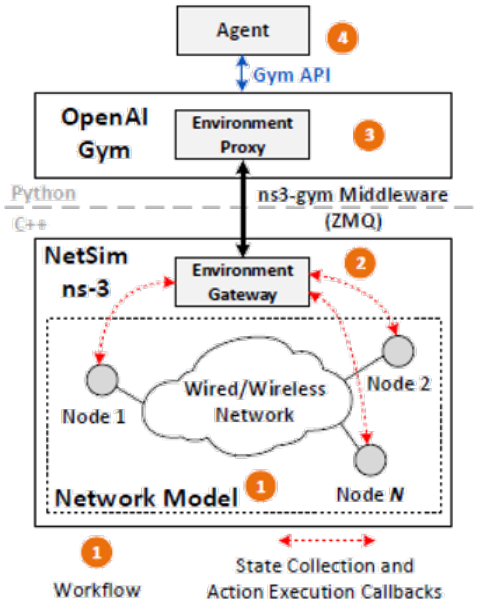


Figure 5.1: Architecture of ns-3 - Gym framework [45].

Environment Gateway. The core of this toolkit consists in these two software

components, who take care of the exchange of the messages and the management of the ns-3 simulation timings in a completely transparent way for the developer.
The **environment gateway** is responsible for the gathering of states from the simulation into numerical structures and the translation of received numerical actions into simulation functions. The main functions that have to be exposed in ns-3 in order to properly implement the gateway are:

- *GetObservationSpace()*: defines the environment observation space;
- *GetActionSpace()*: defines the environment action space;
- *GetObservation()*: collects the observed simulation variables or parameters in predefined data-structures that are sent to the Python agent;
- *GetReward()*: evaluates the reward achieved during the last step;
- *GetGameOver()*: checks whether the episode is terminated or not;
- *ExecuteActions(action)*: maps the numerical action received by the Python agent into the proper simulation action.

The communication between the two frameworks is implemented through the ZMQ socket [47]. The data structures supported in this communications can be: a discrete number (type *Discrete*), a matrix of single-type values (type *Box*) and a tuple or a dictionary of discrete or matrices structures (type *Dict* or *Tuple*).
The **environment proxy** receives the observations from the ns-3 environment and exposes them to the Python agent through the Gym API. Then, it translates the Gym function calls into messages, sending them to the gateway through the ZMQ socket. From the Python perspective, the creation of the environment has exactly the same interface functions of the standard Gym API. The exchange of messages between C++ and Python is completely automatic and transparent to the developer once that the previous functions are correctly implemented. This makes ns-3 - Gym a very easy to use tool for the implementation of RL algorithms on ns-3 simulations.

5.2 Outline of the Methodology

This section provides an overview of the methodology that has been followed for the framework design. The first step consists in the *problem formalization*, already described in section 4.1. After the problem formalization, a Python simulation is implemented in order to train a set of Python agents via the proposed DQN algorithm. The policies learned by these agents are then saved, allowing their direct application in similar environments without the need for a re-training procedure. Finally, the trained agents are interfaced with an ns-3 simulation that implements: a specific **network architecture**, **mobility models** for the nodes, and **applications** installed on each of them. The interface between the network simulation and the trained agents is handled by the ns-3 Gym middleware. The ns-3 environment provides the exact observation and action space defined in the problem formalization, thus avoiding to adjust the agents' interface. Figure 5.2 displays a visual representation of the designed framework, together with the interactions that occur between the different components. A more detailed explanation of the presented methodology is described

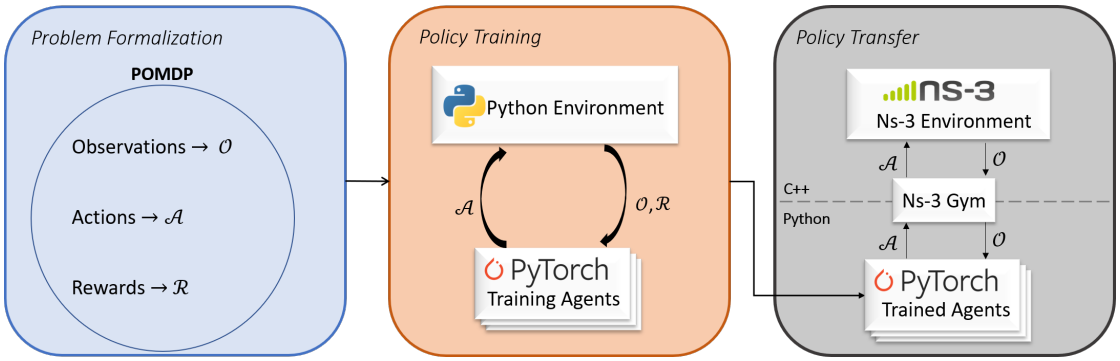


Figure 5.2: Overview of the methodology for the framework design. The three blocks represent the main steps we performed in order to formalize the problem and implement two simulations: (*i*) the Python simulation for policy training, and (*ii*) the ns-3 simulation for the policy transfer in a more realistic scenario.

in the next sections.

5.3 Python Simulation Implementation

The first simulation of the environment has been developed entirely on Python. This choice has been made in order to train the model on a simplified scenario with respect to a more sophisticated simulation that could take days in order to properly train the agents. Therefore, we propose a Python environment emulating the scenario described in section 1.1 and implementing the observations, actions and rewards described in the problem formalization (section 4.1.
The Python environment consists in M clusters of GUs that are created according to a bivariate gaussian distribution within a two-dimensional grid, whose number of cells is limited by given bounds. The GUs do not have a mobility within the grid, but they can appear and disappear from the environment according to a probability that depends on: the cluster in which they are and the episode timestep. Each GU is characterized by a four-dimensional tuple (x_g, y_g, c, act), where:

- $(x_g, y_g) \in \mathbb{R}^+$ represent the geographical coordinates of the GU;
- $c \in \mathbb{Z}$ is the identifier of the UAV that is covering the given GU. If no UAVs are in range with this user, then $c = -1$;
- $act \in \mathbb{R}^+$ is the level of activity of that GU. It can be interpreted as a number proportional to the data rate of the GU.

In the same grid, N_u UAVs discretely move in a two-dimensional space. While GUs are characterized by the four-dimensional tuple, the N_u agents are described by a two-dimensional one (x_u, y_u), where x_u and y_u are the geographical coordinates.
The simulation behaves as follows: at each timestep, the agents can individually perform one of the nine actions previously described in the formalization of the problem. Following this action, each agent will receive a reward, which is based on its level of coverage and resource utilization achieved in the new state. The coverage is evaluated according to the number of GUs the agent can observe within a given range. It is important to underline that two or more agents cannot cover the same GU; the resource utilization is evaluated as the summation of the level of activities of the

covered GUs for each UAV. Furthermore, if the resource utilization of a UAV is above a certain threshold, the agent offloads a part of this value to a more idle neighbor agent until either the utilization of the former comes back below the threshold or the latter reaches it. In this way, even if a UAV is not covering any GUs, it still gets the chance to be rewarded by being close to another agent.
While at every timestep each agent can decide where to move, the size of the clusters tend to change according to the current timestep. A given cluster could be very populated in a given interval, while nearly desolated during another one. Furthermore, the activity of each GU can change over time as well. Hence, we expect the agents to find a reasonable trade-off between level of coverage and utilization of their computational resources. This behavior allows to emulate a scenario where variable hot-spots have to be covered in a dynamic way over time according to the number of users and their level of activity. The variability of the clusters' size and level of activity only depends on the timestep of the simulation and does not change over different episodes. In this context, we expect the agents to dynamically cover different clusters over time, in order to maximize their cumulative reward. The parameters that completely describe the implemented simulation are resumed below:

- **Number of UAVs**: the total number of agents in the simulation. It remains fixed over the episodes;
- **Number of GUs Clusters**: the number of clusters of GUs. Each of them has a mean position and a given variance in order to place new GUs;
- **Average Cluster Size**: the variable average number of GUs that populate a cluster;
- **Bounds**: determine the limits of the grid topology;
- **Coverage Range**: the maximum distance between a GU and a UAV in order to be associated;
- **Connectivity Range**: the maximum distance between two UAVs in order to be connected;

- **Resource Utilization Threshold**: the total level of activity after which the UAV starts sharing its resources with a neighbor;
- **GUs' level of activity**: the variable average value of a GU's level of activity;
- **Total Timesteps**: number of steps within an episode;

The interface functions that have been implemented for the agents to interact with the environment, thus allowing the Q-network training, are:

- *def step(self, action)*: through this function, the agents can perform the selected discrete movement. The input *action* is a vector of bi-dimensional movements, one per each agent, with dimension $(N_u, 2)$. The value returned by this function consists in a complete observation of the next state of the environment, made of both the agents and GUs tuples, together with the value of reward achieved by each agent. The observation is then limited for each agent according to the considered level of observability.
- *def reset()*: while the agents take actions, the internal variables of the environment change accordingly. The *reset()* function allows to restart the episode, by resetting all the environment variables. While the position of the agents is re-initialized randomly, the one of the GUs clusters does not change.

5.4 Ns-3 Simulation Implementation

In this section, the implemented ns-3 simulation is described. This simulation has been realized to propose a more realistic wireless network scenario, where the observations that the agent acquire depend on actual messages exchanged by the nodes of this network. The policies, which are learned in the Python environment, are tested also in this simulation, in order to observe how a more realistic environment affects the performance of the implemented algorithm. The observations acquired by the ns-3 agents have the same structure and meaning with respect to the ones observed in Python, but the dynamics are more sophisticated, affecting the quality

of the observations that the agents can get. The first step towards the implementation of the ns-3 simulation is the definition of the typologies of nodes that are going to be considered: how they communicate, how they move and what messages they exchange during the simulation.
Given the scenario presented at the beginning of this chapter, the network has two typologies of nodes: UAVs and GUs. These nodes have been characterized by key simulation objects (described in section 5.1.2), that completely define the **Network Architecture**, the nodes' **Mobility** and the **Applications** installed on each of them.

5.4.1 Network Architecture

Two main typologies of communications occur in our scenario: (*i*) UAV to UAV and (*ii*) UAV to GU. The architecture of the network can be implemented through a flying ad-hoc wireless network that is able to directly communicate with GUs devices. This architecture has been simulated by deploying a Wi-Fi network in Ad-Hoc mode for the communications between UAVs. The communications between UAVs and GUs, instead, present a different structure. In this case, taking a leaf from the work of Mayor et al. [48], we implemented an infrastructure-based Wi-Fi network, exploiting UAV-mounted IEEE 802.11 Access Points (APs). This configuration allows the UAVs to communicate with the other UAVs over an ad-hoc wireless network on one interface, and with the GUs that are associated to that AP on the other one.

5.4.2 Mobility Models

As explained in the outline of the scenario, the mobility of the two typologies of nodes is well defined; hence, in order to reproduce that behavior, the installed mobility models must present specific characteristics. The GUs should appear in the network according to a bivariate gaussian distribution at different rates over time. This

mobility behavior has been implemented by installing a Constant Position mobility model on the GU nodes, and grouping them in clusters according to a **Disc Position Allocator**, which assigns to each node random positions within a disc according to a normal distribution applied to the polar coordinates (angle and radius). Then, in order to emulate the rates at which the GUs appear and disappear from the network, the nodes' interface is periodically deactivated or activated according to different probabilities depending on the cluster and timestep of the simulation.
The mobility of the UAVs depends on the *action* that the learned policy will generate according to the input observation. As explained in section 4.1.2, the actions are defined by nine possible discrete movements in the two-dimensional space. Hence, the mobility model implemented for the UAV nodes is the **Waypoint Mobility Model**. This model allows to set, at every moment of the simulation, the next position of the node. Whenever an agent has to decide the action to perform, a new waypoint is added in the model according to the chosen direction of movement and distance to cover. Since in the ns-3 simulation the space is continuous, the distance that the agent covers at each action corresponds to the chosen cell size obtained after a space discretization.

5.4.3 Applications

The applications that have been installed provide abstractions of user programs that generate the activities of the nodes. UAVs and GUs have different roles in the simulation; hence, different applications have been installed on them.

GUs Applications

The GUs are characterized by two fundamental actions they have to perform in this framework: (*i*) sending tasks to the UAV they are connected with, and (*ii*) periodically broadcast information regarding their position and level of activity. These two behaviors are implemented by installing on each GU node two specific applications that will handle the **task forwarding** and the **information sharing**.

The *task forwarding* application consists in a client application that sends UDP packets with a given payload. Each of these packets corresponds with one task whose payload contains information about: the required processing time, the time it has been sent, and a task identifier. The parameters that completely define this application are:

- **Socket Address**: the combination of IP address and port that completely identify the destination of the UDP packets. This destination corresponds with the AP the GU is connected with;
- **Packet Size**: a random variable that defines the size of each task in bytes;
- **Processing Time**: a random variable that identifies the computational effort of the task in terms of time that will be required by the UAV to process it;
- **Time Interval between Packets**: a random variable that defines the frequency at which the GUs forward tasks;
- **Maximum Number of Packets**: the maximum number of tasks that the GUs can send.

The task forwarding application also handles the answers from the network correspondent to each task. For each GU, this application will provide three interesting output that can be exploited to analyze the performance of the network: the number of tasks sent, the number of tasks processed and the average waiting time per each task.
The second application that has been installed on the GU nodes provides a periodic broadcasting, towards the associated UAV, of information regarding their position and the level of activity. This message is useful for the UAVs in order to acquire the partial observation of the covered users' position and the correspondent level of activity, which has been identified as the average number of bytes sent per unit of time for task forwarding purposes. The tunable parameter for this application consists in the time interval between two packets. The higher this frequency is and the more updated the observations of the UAVs will be with respect to the GUs

they are covering. This frequency parameter tuning should be properly set in order to guarantee an updated observation for the agents without excessively increase the network overhead.

UAVs Applications

The UAVs are provided with four applications that define their behavior in the network. Three of them handle the periodic exchange of network information and observation acquirement while the other one characterizes the procedure that the agents follow when a new task is received. The first UAV application is a server application that handles the periodic information received by the covered GUs. This server will receive packets, describing the GU's state, with the following payload:

- *identity*;
- *position*;
- *level of activity.*

This information is stored in a local database that constantly updates the entries regarding the GUs that are in range with the UAV and their characteristics. From this database, the agent is able to acquire the information regarding its level of coverage, the position of the GUs and their level of activity, thus having all it needs to create the partial observation with respect to the covered GUs. Other two applications deal with the exchange of network information between UAVs. They are a client and a server application respectively. The client will periodically broadcast UDP packets over the flying ad-hoc network, in order to share information with neighbor UAVs. The payload of these packets contains:

- the **UAV position**;
- the **number of tasks** that the UAV currently has to process;
- the **expected time** that a new task should have to wait in queue in order to be processed by that UAV.

The server application that receives these packets, stores and updates this information in a "neighbor table", which allows to determine the agent's level of connectivity, the positions of its neighbors and the status of their queue, whose relevance will be explained in the following application.
The last installed application handles the task receipt at the UAVs, allowing them to decide whether to locally process the task or offload its computation to a neighbor. When a UAV node receives a task, the decision of where to process it is driven by a simple *Least Load* task dispatching algorithm. The current UAV checks the expected waiting time at its queue and compares it with the waiting time at the queues of its neighbors. The chosen UAV that is charged to process the current task is the one with the shorter waiting time. A finer approach could have also considered the additional time for the extra hop that the task has to take, but we consider that the sending time of tasks between two UAVs is much shorter than the correspondent processing time.
Once the task is stored in a queue, its scheduling is performed through a *First In First Serve* (FIFS) algorithm and the status of the queue, identified with the number of waiting tasks and the expected waiting time for a new task, is updated. The proposal of an optimal task scheduling and dispatching approach is out of the scope of this work. FIFS is a standard task scheduling algorithm in which the tasks are processed according to the order they arrive. It represents the proper baseline approach to start with, since, despite more sophisticated algorithms can be implemented, we consider all the tasks to carry a similar computational effort without any priorities.
Once a task has been processed, the UAV network has to send an answer back to the GU. This answer will always come from the UAV who received the task in the first place, either by processing the task and sending back the answer or by forwarding the answer coming from the neighbor node where the task was offloaded to. Task retransmission algorithms have not been implemented; if a task, or the correspondent answer, is lost, it cannot be retrieved. Furthermore, when the UAV loses connectivity with a GU that is waiting for a task's answer, there is no possibility to route

this answer through multiple UAVs to reach it. These choices have been made in the simulation implementation phase in order to simplify and speed up the realization of the simulated environment.

5.4.4 Policy Transfer on ns-3

The ns-3 Gym toolkit takes care of the exchange of messages (observations and actions) between ns-3 and the python agents. In order to properly apply this interface, we had to implement the environment proxy and environment gateway that have been discussed in section 5.1.3. The environment gateway has to expose the functions that define: (i) how observations are gathered from the simulation and (ii) how actions are applied on it:

- *GetObservationSpace()*: the observation space is defined as a two-dimensional tuple (U, G). The first entry of this structure $U \in \mathbb{R}^{N_u \times 2}$ contains UAV-related information while the second entry $G \in \mathbb{R}^{N_g \times 4}$ describes the status of the GUs according to the knowledge of the UAVs. The observations collected from the ns-3 simulation have exactly the same structure of the observations returned in the simpler Python implementation;
- *GetActionSpace()*: the action space is given by an array $\mathbf{a} \in \mathbb{R}^{N_u}$. Each entry is an index corresponding with one of the nine possible discrete actions the agent can perform in the space;
- *GetObservation()*: the U matrix is filled with the position of each agent (x_u, y_u), while the GUs' status G includes their position (x_g, y_g), the AP they are connected with (c) and their level of activity (act);
- *GetGameOver()*: the episode terminates only after a determined number of timesteps which depend on the duration of the simulation and the frequency of observation acquirement;
- *GetReward()*: this function has not to be implemented for the moment, since the policy the agents exploit is entirely trained on Python.

- *ExecuteActions(action)*: this function acquires the action each agent has to perform and turns it into an actual movement. Firstly, each index is mapped with the correspondent two-dimensional vector (Figure 4.2). Secondly, a new waypoint is added to the mobility model of the correspondent UAV, by summing its current position with the action vector. The agent will move towards this new position at constant velocity.

The definition of these functions is fundamental to use a Python agent on the ns-3 simulation. The agents are implemented exactly as in the Python simulation, interacting with the environment through the OpenAI Gym functions *step(action)* and *reset()*. The only difference is that the Q-networks have not to be trained again. They can observe both a centralized scenario, where it is possible to know the positions of all the other agents independently on the actual connectivity (matrix U), or the partial scenario, where the observations depend on information that the nodes in the network are exchanging along the simulation. For this second case, each observation is given by the knowledge that the UAV currently has through the periodic broadcasting of information among the nodes. The partial connectivity, coverage and level of activities are retrieved from the matrices U and G.

5.5 Limitations

The design and implementation of this framework has not been free of questions and problems. Despite our original intention was to propose a reinforcement learning approach able to learn good policies for an agent directly in a realistic environment, the steps towards this goal took more time than expected. Firstly, the learning curve for an implementation of a multi-agent reinforcement learning algorithm together with an ns-3 simulation, from scratch, is quite long. Furthermore, one single episode of the ns-3 simulation can last minutes, without considering the additional time required for the training process. Hence, the learning of a policy through RL directly on this framework, without a proper computational power, could take days. This has been

the main reason why we decided to keep the training of the DQN entirely on the simpler Python environment and, subsequently, to apply the already learned policy directly on the network simulation. The policy transfer from an environment to a slightly different one brings the performance of the approach to the sub-optimality. Even though the agents observe the same input, the way in which these observations are acquired is very different, causing a reduction in the achieved return.
Other limitations regard the implementation of the algorithm in the Python environment. The first problem we faced has been the hyper-parameters tuning. Since previous works that consider similar scenarios do not present any comprehensive hyper-parameters tuning approach, we had to start from known configurations that are successfully applied in completely different environments. Starting from here, through several runs with different parameters, we were able to orient ourselves on a discretely suited set of parameters, despite the long time it took to find them. Furthermore, the learned policy has shown a low level of scalability with respect to the bounds of the target area where the agents operate. This problem needs to be addressed through a deeper study of methods that could make the training of the DQN more robust to these variations without spending too much time and money in a careful hyper-parameter tuning. The other main limitation relies in the resource sharing approach and its contribution in the reward of the agent. As discussed in section 5.3, the resource sharing between UAVs should allow an agent to increase its reward by being connected to another agent. Hence, in the case in which the number of UAVs is higher than the number of clusters, their behavior should allow more agents to converge towards the most active or populated clusters in order to share their computational resources. Unfortunately, keeping the same structure of reward and neural networks, the policy is not able to learn such behavior. Therefore, the resource sharing approach does not improve any metric and its contribution in the learning process of the policy is limited. However, recalling the limited on-board computing capabilities of UAVs, the implementation of a resource sharing approach enhances the realism of the simulation, and we plan that it will play an active role

in future improvements to this work.
In the next chapter, the results obtained from specific scenarios will be presented, being aware of the limitations that have been discussed so far, which bring the learning of the policy to quite poor performance in some of the presented scenarios.
In chapter 7 some possible solutions to these limitations will be proposed as future works of this thesis.

Chapter 6

Experimental Results and Discussion

So far, the description of the simulated environments together with the designed reinforcement learning control system has been carried out. We also discussed about the main limitations of the system, which have to be taken into account in order to make this approach more robust and, hopefully, to exploit it in a real-world use case. The aim of this chapter is to present the adopted experimental methodology in order to clearly describe the scenarios on which we applied our approaches.

The first section focuses on the training of our models. Keeping the same neural network architectures and the same reward, we wanted to observe whether the agents are able to learn a proper policy in three slightly different scenarios, namely "static clusters", "dynamic clusters" and "dynamic activity", which differ according to the level of dynamicity of the clusters' size and GUs' level of activity. Furthermore, a comparison between the policies learned by the two different POMDP models (section 4.1) is going to highlight the main differences and challenges brought by a partial and limited observation. The results are going to be presented and compared in terms of reward (section 4.1.3) achieved by the agents along the policy training.

The second section is dedicated to the policy transfer on ns-3. The main limitations will be highlighted again, together with a more detailed description of the parameters used for the network simulation. Once again, the comparison between the different

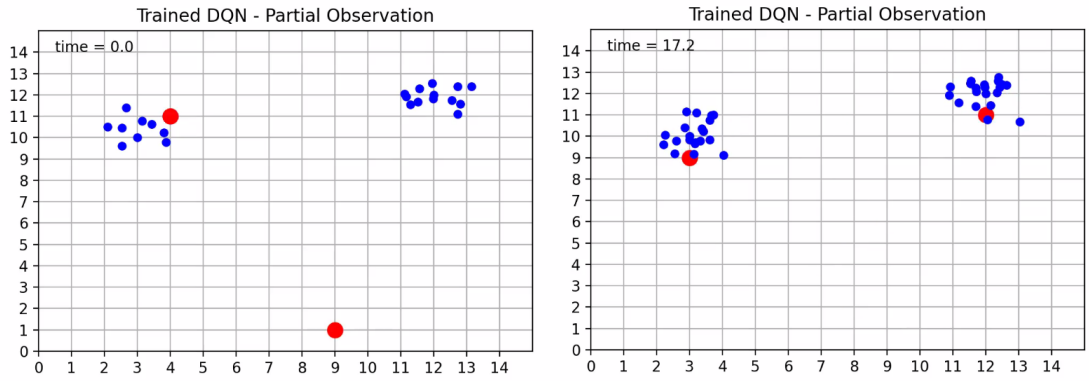
results obtained from the two policies will be presented in terms of achieved network coverage and task completion ratio, focusing on the problems that come out when the trained policies have to be applied on the more realistic environment.

6.1 Python Experiments

In section 5.3 we already provided an overview of how the Python environment has been developed and what are the observations, actions and rewards that completely characterize the reinforcement learning approach. However, our goal is to observe how the policy that the agents learn can adapt to variations in the scenario according to different behaviors that the GUs might have. In this section, these environments are going to be carefully described, together with a schematic presentation of the parameters characterizing them.

6.1.1 Simulation with "Static Clusters"

In the first scenario, we wanted to test the simple case where the agents have to cooperate in order to cover all the clusters that are present in the area and that have, on average, a constant size. Furthermore, the number of agents is equal to the number of clusters. This is a coverage problem that has been widely discussed in literature, and several algorithms can be found to efficiently solve it. The objective was to start applying our reinforcement learning approach in a simple environment in order to orient ourselves towards an initial set of proper hyper-parameters, neural network architectures and reward function.
The parameters of this scenario are resumed in Table 6.1. Two clusters appear in the target area. The average size of each cluster is constant, meaning that all of them should be equally covered by the agents over the entire duration of the episode. Two agents are considered in the environment. Each of them does not have any a-priori knowledge about the cluster positions. They can only move in the space according to nine possible actions (Figure 4.2) and get their observation of the environment



(a) (b)

Figure 6.1: Screen shot acquired from the "static clusters" scenario. (a) represents the initial configuration, where the agents (red dots) are placed in random position. The agents tend to move towards the clusters (blue dots) and cover them (b) for the entire duration of the episode.

according to the centralized or partial approach that is being considered. The level of activity of each GU is not so important in this scenario since the agents have just to learn to statically cover the two different clusters. Figure 6.1 displays a Screen shot taken from this scenario.

The position of the clusters is fixed over different episodes, while the starting state of the agents is random every time the environment is reset. One episode terminates when the total number of epochs is reached.

The hyper-parameters we exploited for this scenario are represented in Table 6.2. The tuning of this first set of hyper-parameters has been quite straightforward. We started by taking a leaf from [4], that already gave us good performance of the algorithm in terms of convergence and reward. Given the relatively simple environment we are testing as our first scenario, very simple architectures have proved to achieve proper results.

Figure 6.2 shows the mean value and the 99% confidence interval of the running rewards (Eq. 6.1) obtained by each agent during the training of the policy, for five

Parameters	**Value**
Number of Agents	2
Number of Clusters	2
Average Cluster Size	[20, 20]
Bounds	15x15
Coverage Range	2
Connectivity Range	4
Resource Utilization Threshold	100
Level of Activity	$\mathcal{N}(4, 1)$
Epochs	180
Seeds	5

Table 6.1: Parameters used in the Python simulation with "static clusters".

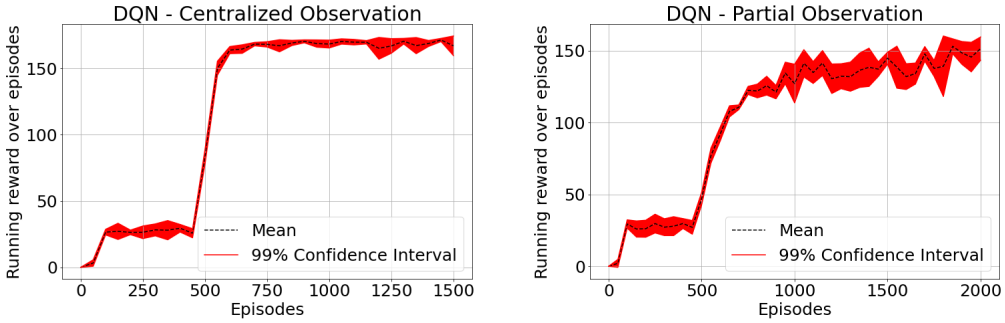
different runs of the simulation.

$$\text{running reward} = 0.9 \times \text{running reward} + 0.1 \times \text{episode reward} \tag{6.1}$$

The sharp increase of the running reward after the 400^{th} episode (when the noise starts to decrease) represents how fast the agents tend to learn the clusters that have to be covered. The two agents will never occupy the same cluster, no matter their initial position. The satisfying results obtained in this first simple scenario pushed us to train the Q-networks in more complicated and dynamic environments.

Hyper-parameters	**Value (Centralized)**	**Value (Partial)**
Q-Network	**Learning Rate**: 10^{-3} **Architecture**: 1 hidden layer of size = 80; 9 output values;	**Learning Rate**: 10^{-3} **Architecture**: 1 CONV Layer $3 \times 3 \times 9$, stride=1, padding=1; 1 MaxPooling Layer; 2 FC Layers of size = 13 and 80; 9 output values;
Epsilon Decay	**Start**: 1, **End**: 0.1 $\epsilon \leftarrow \epsilon \times 0.999$, after $400^{th} ep.$	**Start**: 1, **End**: 0.1 $\epsilon \leftarrow \epsilon \times 0.999$, after $400^{th} ep.$
Gamma (γ)	0.95	0.95
Tau τ	0.05	0.05
Observation	**Input Size**: 5 **Input Structure**: 1×5	**Input Size**: 4 **Input Structure**: 3 of 8×8 and 1 of 1×3
Batch Size	65	100
Replay Buffer	12×10^3	12×10^3
# of Episodes	1500	2000

Table 6.2: DQN hyper-parameters setup for the "static clusters" scenario.



(a) (b)

Figure 6.2: DQN Static Clusters Environment Running Reward Plot. The two graphs report the running reward achieved by the agents with centralized (a) and partial (b) observations in the "static clusters" scenario.

6.1.2 Simulation with "Dynamic Clusters"

In the second simulated scenario, we wanted to increase the dynamicity of the ground users, in order to observe if the training of the policy can adapt to variations in the behavior of the GUs. The size of the clusters can change over time and the number of agents is smaller than the number of clusters. Hence, the UAVs should cover different clusters over time, by learning which are the hot-spots in the target area according to the simulation timestep. The parameters of this second scenario are presented in Table 6.3. In this case, three clusters have been deployed, with their size changing over the episode according to the timestep. In detail, the simulation time has been divided in three equally large intervals. In each of them, a different couple of clusters will be the most populated one. The two agents have to learn how to dynamically cover the hot-spots, in order to maximize the coverage of the GUs. Apart from this slightly different GUs behavior, the other characteristics of the environment remain exactly the same.

By keeping the initial configuration of the hyper-parameters that we used for the

Parameters	**Value**
Number of Agents	2
Number of Clusters	3
Average Cluster Size	[2,20, 20]
Bounds	15x15
Coverage Range	2
Connectivity Range	4
Resource Utilization Threshold	100
Level of Activity	$\mathcal{N}(4, 1)$
Epochs	180
Seeds	5

Table 6.3: Parameters used in the Python simulation with "dynamic clusters".

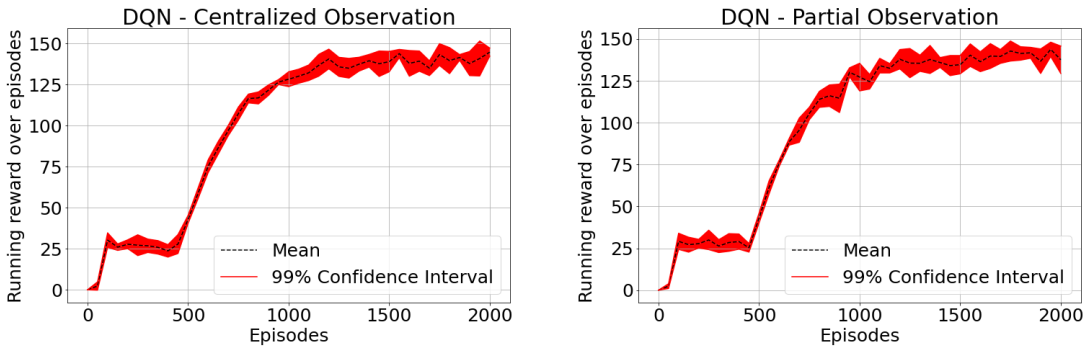
static scenario, the policy learned by the agents is sub-optimal. Either the agents find

two clusters and statically cover them for the entire duration of the simulation, or both agents cover the same cluster. This issue was addressed by increasing the level of initial noise applied to the policy, in order to let the agent explore the surrounding environment for a longer time. All the hyper-parameters used for this scenario are presented in Table 6.4. The architectures of the two neural networks, together with

Hyper-parameters	**Value (Centralized)**	**Value (Partial)**
Q-Network	**Learning Rate**: 10^{-3} **Architecture**: 1 hidden layer of size = 80; 9 output values;	**Learning Rate**: 10^{-3} **Architecture**: 1 CONV Layer $3 \times 3 \times 9$, stride=1, padding=1; 1 MaxPooling Layer; 2 FC Layers of size = 13 and 80; 9 output values;
Epsilon Decay	**Start**: 1, **End**: 0.1 $\epsilon \leftarrow \epsilon \times 0.999$, after $400^{th} ep.$	**Start**: 1, **End**: 0.1 $\epsilon \leftarrow \epsilon \times 0.9995$, after $400^{th} ep.$
Gamma (γ)	0.95	0.95
Tau τ	0.05	0.05
Observation	**Input Size**: 5 **Input Structure**: 1×5	**Input Size**: 4 **Input Structure**: 3 of 8×8 and 1 of 1×3
Batch Size	65	100
Replay Buffer	12×10^3	12×10^3
# of Episodes	2000	2000

Table 6.4: DQN hyper-parameters setup for the "dynamic clusters" scenario.

the other parameters, are exactly the same as before, except for the epsilon decay noise and the batch size, whose higher value shown better results for the partial observation approach.



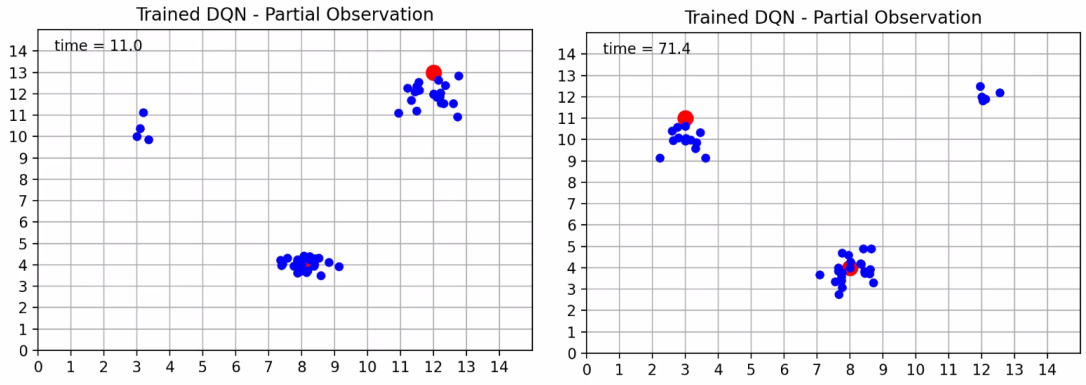
(a) (b)

Figure 6.3: DQN Dynamic Clusters Scenario Running Reward Plot. The two graphs report the running reward achieved by the agents with centralized (a) and partial (b) observations in the "dynamic clusters" scenario over 2000 episodes.

The plots reported in Figure 6.3 describe how the two approaches tend to reach the same level of running reward. The learning curve smoothly increases until reaching convergence on the 1250^{th} episode for both approaches. With these results, it is possible to appreciate how the agents move towards the most populated clusters, which are not fixed over time. Figure 6.4 gives an idea of the positions that the agents should occupy in order to optimize the return.
The "dynamic clusters" scenario has been the one implemented in the ns-3 simulation, thus using the same policy learned in these experiments to let the agents move in the more realistic scenario.

6.1.3 Simulation with "Dynamic Activity"

The first two scenarios have as main goal the optimization of the coverage, while the level of activity of GUs is kept on average constant. As the reward function described in section 4.1 suggests, the agents have to optimize their coverage as well as their resource utilization. Hence, they should also move according to the variable number of tasks that the GUs are sending, thus trying to find a trade-off between



(a) (b)

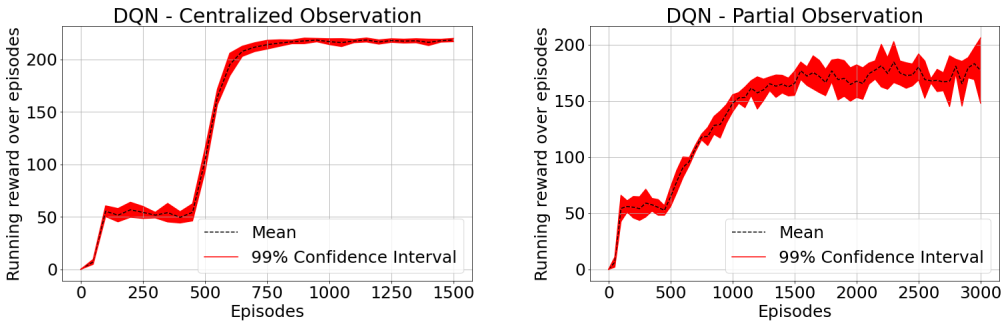
Figure 6.4: Screen shot acquired from the "dynamic clusters" scenario. The three clusters (blue dots) tend to change their average size over the simulation, so the agents (red dots) learn how to dynamically cover different clusters.

coverage and utilized resources. The third scenario aims at testing the flexibility of our approach when the clusters have the same size but different activities over time, with a number of agents that is smaller than the number of clusters. The optimal policy the UAVs should learn corresponds with which GUs not to cover in order to maximize both coverage and resource utilization. The set of parameters used for this simulation is presented in Table 6.5.

This environment has revealed a bit more challenging for the proper policy training. Neither increasing the noise nor changing the batch size brings the agents to learn how to cover the clusters according to their level of activity. After several trials, where we performed a grid search over other hyper-parameters, we decided to try to slightly change the value of reward, by doubling the term that reflects the utilization factor. This small change, together with a proper hyper-parameters re-tuning, brought the policy to better adapt with respect to different activities of the GUs. The hyper-parameter setup and the results obtained from this third scenario are represented in Table 6.6 and Figure 6.5 respectively.

Parameters	Value
Number of Agents	2
Number of Clusters	3
Average Cluster Size	20
Bounds	15x15
Coverage Range	2
Connectivity Range	4
Resource Utilization Threshold	100
Levels of Activity	[2,10,10]
Epochs	180
Seeds	5

Table 6.5: Parameters used in the Python simulation with "dynamic activity".



(a) (b)

Figure 6.5: DQN Dynamic Activity Scenario Running Reward Plot. The two graphs report the running reward achieved by the agents with centralized (a) and partial (b) observations in the "dynamic activity" scenario.

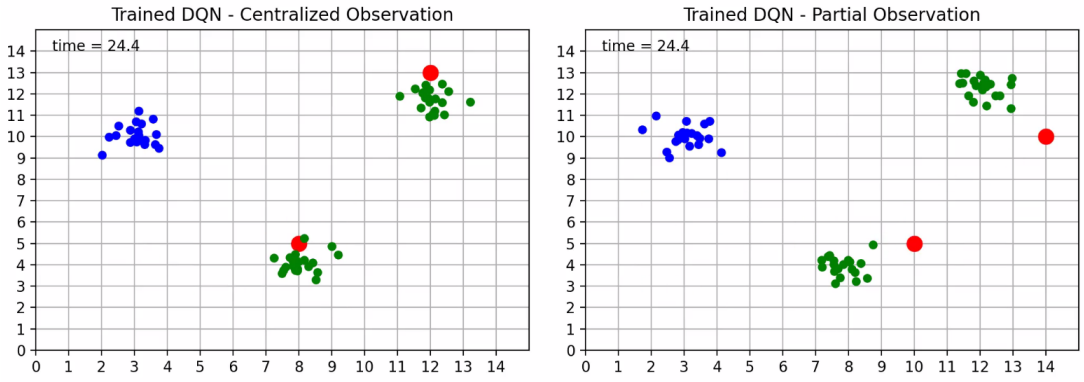
The results obtained in this scenario present how the centralized approach better behaves when the agents have to learn more complicated policies. However, even if the partial observation case shows a lower reward and larger confidence interval, the learned policy guides each agent towards the most active cluster. The lower reward

Hyper-parameters	**Value (Centralized)**	**Value (Partial)**
Q-Network	**Learning Rate**: 10^{-3} **Architecture**: 1 hidden layer of size = 80; 9 output values;	**Learning Rate**: 10^{-3} **Architecture**: 1 CONV Layer $3 \times 3 \times 9$, stride=1, padding=1; 1 MaxPooling Layer; 2 FC Layers of size = 13 and 80; 9 output values;
Epsilon Decay	**Start**: 1, **End**: 0.1 $\epsilon \leftarrow \epsilon \times 0.999$, after $400^{th} ep.$	**Start**: 1, **End**: 0.1 $\epsilon \leftarrow \epsilon \times 0.9995$, after $400^{th} ep.$
Gamma (γ)	0.95	0.95
Tau τ	0.05	0.05
Observation	**Input Size**: 5 **Input Structure**: 1×5	**Input Size**: 4 **Input Structure**: 3 of 8×8 and 1 of 1×3
Batch Size	65	100
Replay Buffer	12×10^3	12×10^3
# of Episodes	1500	3000

Table 6.6: DQN hyper-parameters setup for the "dynamic clusters" Python environment.

is given by the wrong positioning of the UAVs, that do not place themselves in the middle of the cluster but close to it (Figure 6.6). These results suggest how the architecture of the convolutional neural network should be better designed in order to let the agents understand the precise position they have to occupy.

The results achieved so far in the Python environments show that the training of a DQN that runs in a multi-agent system can be used to enhance the collaboration between agents in order to reach the maximization of coverage and resource utilization of the network. However, relatively simple environments have been considered so far, and a deeper study regarding the robustness of the policy with respect to the number of agents and the size of the target area should be carried out. Given the obtained results, we are confident that this approach can be improved through a more intense



(a) (b)

Figure 6.6: Screen shot acquired from the "dynamic activity" scenario. The most active clusters (green dots) are covered by the agents (red dots). The placement obtained in the centralized case (a) is more precise than the one obtained from the partial case (b).

hyper-parameters tuning and neural network design. We did not further focus on this part in order to move to the second goal of our work: the application of the trained policy in the network simulator.

6.2 Ns-3 Experiments

The ns-3 simulation details and implementation have been presented in section 5.4. The objective is to exploit the same policy that has been already trained in Python on a very similar scenario developed in ns-3, which provides a more realistic environment under the wireless communication point of view. For this reason, we had to carefully set the parameters of this simulation in order to be as close as possible to the Python one. We decided to reproduce the "dynamic clusters" scenario presented in section 6.1.2, where the agents have to learn how to optimize the coverage of the network, while the activity of the GUs is on average the same. The parameters that fully describe the implemented ns-3 simulation are presented in Table 6.7. These parameters highlight some similarities as well as major differences with respect to

Parameter	**Value**
Number of UAVs	2
Number of Clusters	3
Cluster Size	10
Switch On/Off Probabilities	[0.1, 0.9, 0.9]
Probabilities Shift (s)	20
On/Off Evaluation Frequency (s)	5
Bounds (m)	90x90
Action Step (m)	6
Observation Timestep (s)	1
Task Size (bytes)	Uniform(1024, 2048)
Task Processing Time (s)	Uniform(0.5, 1)
Task Forwarding Frequency (s)	Uniform(3,5)
Ground Info Frequency (s)	1
UAVs Info Frequency (s)	1
Simulation Time (s)	60

Table 6.7: Parameters used in the ns-3 simulation.

the Python environment. The agents have to learn how to dynamically cover clusters of variable size. The maximum number of GUs per cluster is ten, and every five seconds the size of each cluster is updated according to a *Switch On/Off* probability: depending on the cluster and on the current timestep, the probability for a GU to be activated or deactivated depends on the value of this parameter. The *Switch On/Off* probabilities are then shifted over the clusters every twenty seconds. Given the continuous state space available in ns-3, the bounds are not evaluated according to a number of cells, but according to the size of the target area evaluated in meters. Since the policy does not know how to deal with a continuous state space, we divided this area in cells of dimension $6m \times 6m$, thus discretizing the continuous bounds into a grid with dimension 15×15. The bounds and actions discretization allows the pol-

icy to deal with the same state and action space it was trained with in the Python environment. The other simulation parameters describe the time interval between the agent's observations/actions (*Observation Frequency*), the behavior of the GUs in terms of task forwarding, and the frequency of information broadcasting between the nodes of the network, thus specifying the application parameters seen in section 5.4.

Every *Observation Timestep*, the observation of the environment is sent to the Python agents through the *GetObservation()* function exposed in the environment gateway. The Python agents receive this observation and, according to the centralized or partial scenario that is being considered, they get the correspondent policy input. Once the input has been processed by the DQN, the actions are gathered in the action vector and sent back to the ns-3 environment through the function *step(action)*. Finally, the ns-3 nodes perform the movements returned by their policies through the *ExecuteActions(action*) function.

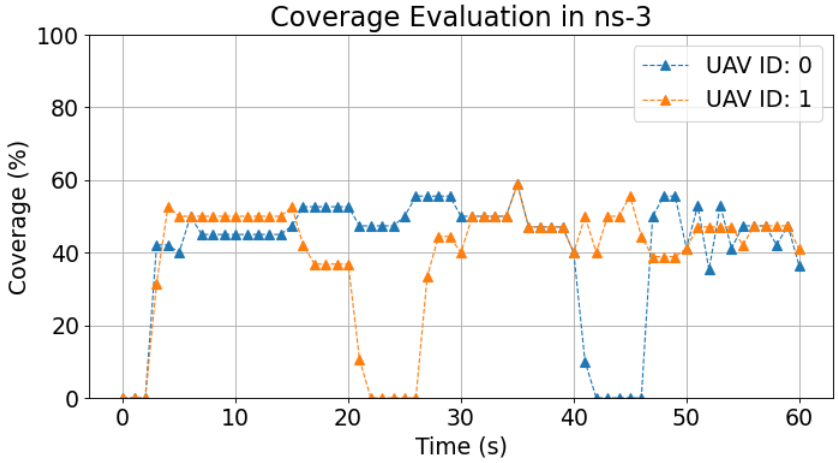
6.2.1 Centralized Observation Experiments

In the first experiment, the centralized observation was tested. The initial positions of the clusters and the variability of their size are the same used in the Python environment, and, since the observation that the agents get are the same, also the positions and the order of visited clusters returned by the policy remains identical. The correspondent movements do not depend on the actual GUs positions and activities; hence, the policies tend to move the UAVs towards the same positions they visited in the Python environment. Figure 6.7 shows the level of coverage and task completion ratio achieved by the UAV network along the ns-3 simulation, when the policy exploiting the centralized observation is used. It is possible to appreciate how the two UAVs always reach around the 50% of coverage each, meaning that nearly the 100% of GUs are covered along the entire duration of the simulation. The uncovered GUs left correspond with the active ones that are still present in the less

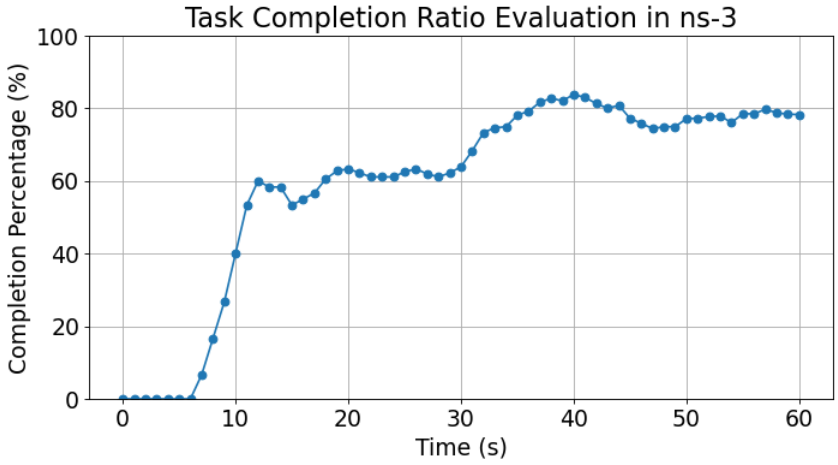
populated clusters. The level of task completion ratio gives an idea of the resource utilization of the network. It corresponds with the ratio between the tasks that have been processed by the UAVs and the total number of tasks sent by the GUs. The actual value of this metric is not very meaningful, but it gives an idea of the different network resource utilization when compared with other approaches.

6.2.2 Partial Observation Experiments

When the agents observe the partial information about coverage, level of activity and connectivity directly from their current knowledge of the environment, differently from the centralized observation scenario, the results happen to be worse. While in the Python environment, each agent learns how to equally split the clusters and hover on the most populated ones, the same policy applied on ns-3 returns a sub-optimal outcome. In fact, both agents tend to converge towards the same cluster, leaving the other active one completely uncovered. Figure 6.8 perfectly represents this situation: the summation of the coverage level for both UAVs is always between the 40% and 60%, never reaching the same values of coverage appreciated in Figure 6.7. Furthermore, the tasks completion ratio is much lower, given that the uncovered clusters keep sending tasks that will never be received and processed.
This outcome was not surprising. In the centralized scenario, all the decisions are based only on the positions of the agents, which are supposed to be instantaneously known at every moment. In the partial scenario, each agent relies only on the partial observation it can create out of its local databases. The observations are not synchronized with the information broadcasting and database update, meaning that some observations may contain out-of-date network information. Furthermore, the level of activity associated to the GUs plays a very important role in this scenario. Just like the position of agents and GUs is a well-defined observation that has been made coherent to the Python observation by simply organizing the ns-3 space in a grid area, so the level of activity must have the same meaning and range of values in



(a)

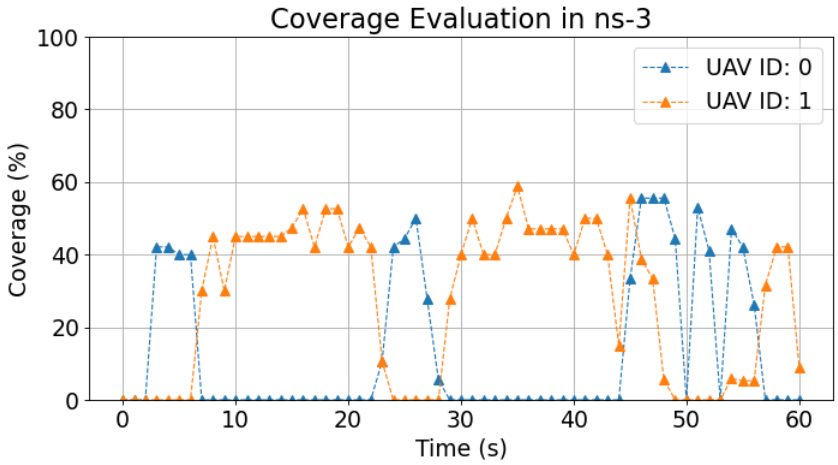


(b)

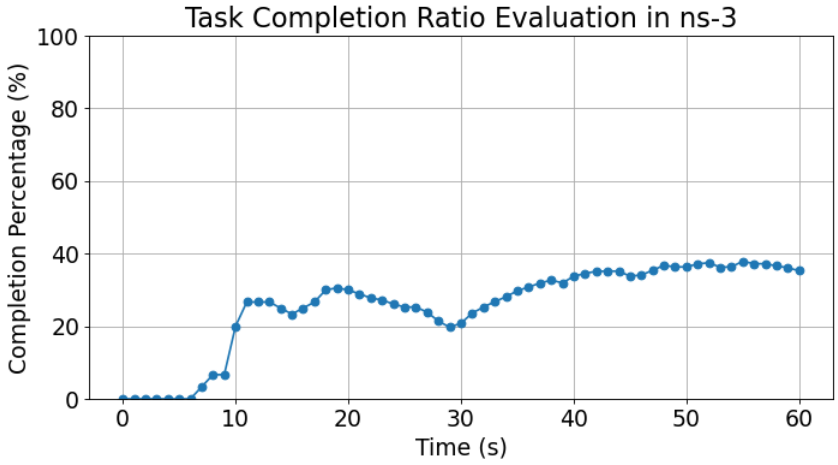
Figure 6.7: DQN ns-3 Centralized Environment Coverage and Task Completion. The two plots represent level of coverage (in percentage) reached by the two UAVs (a) and the task completion ratio achieved in the UAV network (b) in the ns-3 simulation when the centralized observation policy is applied.

both environments. In ns-3, we have actual packets that are sent from the ground to the network; hence, the level of activity has been interpreted and set as the average number of bytes sent per unit of time, differently from the meaning it has in the Python environment.
These two issues have been tackled by setting the information broadcasting frequency higher than the *GetObservation()* frequency and by re-training the policy with the proper range for the levels of activity: given the interval between two packets I_p and the average size of these packets S_p, this parameter can be defined as $act = \frac{S_p}{I_p}$. Furthermore, in order to add some noise, this value has been considered as the mean of a normal distribution $\mathcal{N}(act, 100)$, whose samples correspond with the GUs levels of activity. However, the solution obtained still remains sub-optimal, which is probably given by: (i) the lack of synchronization between the observation acquired by the agents and the update of its local databases, and (ii) the more unpredictable trend that the levels of activity may have on the network simulator with respect to the much more predictable ones applied in Python.

The convolutional neural network exploited for the partial observation case has already shown a higher variance in terms of performance concerning the "dynamic activity" scenario described in section 6.1.3. The poorer performance obtained in ns-3 could potentially be solved by better designing the neural network, making it more robust to more noisy and delayed observations. Furthermore, a deeper tuning of hyper-parameters and setup of the ns-3 simulation could also improve the level of coverage and resource utilization of the network. These preliminary results obtained from the ns-3 simulation aimed at proposing an alternative approach for the application of multi-agent deep reinforcement learning in a UAV network, taking into account the possibility for the agents to acquire their observations directly through the periodic exchange of messages with the other nodes.



(a)



(b)

Figure 6.8: DQN ns-3 Partial Observation Environment Coverage and Task Completion. The two plots represent the level of coverage (in percentage) reached by the two UAVs (a) and the task completion ratio achieved in the UAV network (b) in the ns-3 simulation when the partial observation policy is applied.

Chapter 7

Conclusions

The wide set of applications in which UAV networks have proved to be effective in wireless communication networks represents the main motivation that inspired this work. Once investigated on the several challenges that still have to be addressed, we decided to focus on a placement and trajectory design problem in order to foster the performance of the network through the mobility and flexibility of the UAVs. We considered a scenario in which a set of GUs dynamically assigns tasks to UAVs, each task requiring a certain computational effort. Through their proper placement, UAVs have to maximize the coverage of the network and their resource utilization, to guarantee a proper quality and reliability in task execution. A DRL approach was leveraged in order to address this decision-making problem, taking into account the promising results achieved by these frameworks in multi-agent systems and intelligent communications.

The main contribution of this thesis consisted of the design of a distributed control framework in order to properly place the UAVs of the network and jointly optimize the previously mentioned metrics. Firstly, we formalized the multi-agent decision-making problem as a POMDP and implemented a relatively simple simulated environment in Python, able to interact with the agents by exchanging observations, rewards, and actions. This simulation allowed us to perform reinforcement learning

experiments according to the centralized or partial observations that were considered and compared. A multi-agent DQN algorithm with two different neural network structures was proposed: a shallow neural network with a flat topology for the centralized observations, and a convolutional neural network for the partial observations. Furthermore, three slightly different scenarios were tested while keeping the same DQN architectures and parameters, in order to assess the flexibility of the algorithm according to different GUs behaviors. We run experiments of more than 1500 episodes for both of the observation typologies in order to train the respective policies. As we expected, the results obtained from the centralized observations tended to converge faster and with a lower variance than the partial observation case. Despite their worse performance, the more limited observations enable the scalability and flexibility of the algorithm, which is no further constrained to an a-priori knowledge of the total number of agents in the environment. In addition, the knowledge of the current environment state can rely on other important information that are derived from variable sized observations. The trained models were then tested in the ns-3 environment. The differences in terms of coverage and resource utilization achieved by the UAV network between the two observation scenarios were evident. While in the centralized case, the UAVs tend to equally cover the most populated and active clusters, always maintaining a total level of coverage close to 100%, the policy reaches a sub-optimal behavior in the partial observation case, providing a lower network coverage and task completion ratio.

The results obtained from these experiments showed main limitations that should be considered and addressed. Firstly, the implemented neural network architectures present a very simple structure. A deeper examination of more robust structures and hyper-parameters tuning would benefit our approach to adapt to different GUs' behavior and, eventually, to a variable number of agents and clusters. Furthermore, the reward function has also to be better designed. We noticed how, in the "dynamic activity" scenario, a higher weight associated to the resource utilization in the reward function improved the performance of the algorithm. A better trade-off

between coverage and resource utilization should be found, in order to make the reward more suitable for a larger set of possible environments. Finally, the poor results achieved in the ns-3 simulation for the partial observation scenario have also to be analyzed. Apart from the need for a better tuning, the range of observation values acquired in ns-3 should be more coherent with the observations the agents acquire in Python. Furthermore, the lack of synchronization between observation acquisition and network message broadcasting for the establishment of the agent's partial knowledge leads the agents to take sub-optimal actions. A broad set of improvements can be injected in this approach in order to develop a much more scalable and realistic framework. Considered the limitations encountered along this project, the last section of the thesis is dedicated to promoting some of the many possible future works that can contribute to the enhancement of this framework.

7.1 Future Works

Starting from the just described limitations, a set of future improvements is worth to be proposed in the last part of this thesis. As already discussed, our DRL approach has weaknesses in terms of robustness with respect to small changes in the simulated environment and its application in a more realistic one. It is important to highlight that our proposal for agents' observations only represents one example among many others that might be implemented. Therefore, together with more sophisticated neural network architectures, other typologies of observations related to the environment state may be tested. Furthermore, the reward function should also be better investigated in order to allow the agents finding the best trade-off between coverage and resource utilization. Another fundamental contribution that could improve the current methodology from the DRL side relies on the implementation of recent upgrades of DQN, such as Double DQN or Dueling DQN. They would help the algorithm to be trained faster and have a more stable learning procedure, other than provide a comparison with the current baseline methodology.

The second group of future possible advancements includes ns-3 simulation improvements. Firstly, the simulation realism can be enhanced by including proper channel models for the Air-to-Air (A2A) and Air-to-Ground (A2G) communications. This implementation would make the simulated environment much more similar to a real-world scenario, thus allowing to fit the DRL algorithm to a more realistic scheme. The proper definition of channel models can also improve the resource sharing algorithm reliability. Even though its implementation was not fundamental in the proposed environments, the importance of a resource sharing approach in UAV networks is well motivated by their limited on-board computation resources. The task offloading criterion should not only be based on the least load server concept, but also consider the cost of sending the task over the network, which strongly depends on the quality of the channel.
The last two ideas regard the interface between the ns-3 and DRL frameworks. Firstly, our ns-3 simulation returns observations that were suited to our DQN algorithm. However, this makes the implementation of other algorithms to the same environment challenging, thus preventing a comparison between different solutions. A proper generalization and standardization of this environment would allow to easily implement benchmarking or innovative solutions in order to compare the performance and track the algorithm progresses. Secondly, the policy exploited in ns-3 was already trained in a simpler environment and just applied in the network simulation. However, thanks to *transfer learning*, we can allow the agents to leverage the previously acquired knowledge in the more accurate network simulation, while the policy itself keeps learning in the new environment. This approach would be a proper trade-off between the amount of time needed to train the model entirely on ns-3 and the level of realism the policy learns to deal with.

The proposed multi-agent DQN approach implemented in a simulated UAV network showed promising results in the joint optimization of the coverage and resource utilization of the network. However, further work is still needed in order to propose

this approach as a baseline solution. The suggested improvements of the algorithm, the standardization of the environment and the proper comparisons with benchmarking solutions, could bring an important contribution in order to foster the application of distributed DRL approaches in the placement and trajectory design of UAV networks.

Appendix A

Reinforcement Learning Algorithms

A.1 Dynamic Programming

Algorithm 2 Value Iteration Algorithm

Initialize $V(s)\ \forall s \in \mathcal{S}$ and $\pi(s) \in \mathcal{A}$
repeat
  $\Delta \leftarrow 0$
  for each $s \in \mathcal{S}$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s \in \mathcal{S}, r \in \mathcal{R}} P\left(s', r \mid s, a\right)\left[r + \gamma V\left(s'\right)\right]$
    $\Delta \leftarrow max(\Delta, |v - V(s)|)$
  end for
until $\Delta < \theta$ (a small positive number)
$\pi(s) = \arg\max_a \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} P\left(s', r \mid s, a\right)\left[r + \gamma V\left(s'\right)\right]$

Algorithm 3 Policy Iteration Algorithm

Initialize $V(s)\ \forall s \in \mathcal{S}$ and $\pi(s) \in \mathcal{A}$
repeat
  // Policy Evaluation
  repeat
    $\Delta \leftarrow 0$
    for each $s \in \mathcal{S}$ **do**
      $v \leftarrow V(s)$
      $V(s) \leftarrow V(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(a \mid s) \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} P\left(s', r \mid s, a\right)\left[r + \gamma V\left(s'\right)\right]$
      $\Delta \leftarrow max(\Delta, |v - V(s)|)$
    end for
  until $\Delta < \theta$ (a small positive number)
  //Policy Improvement
  policy_stable $\leftarrow$ *true*
  for each $s \in \mathcal{S}$ **do**
    previous_action $\leftarrow \pi(s)$
    $\pi(s) \leftarrow \arg\max_a \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} P\left(s', r \mid s, a\right)\left[r + \gamma V\left(s'\right)\right]$
    if *previous_action* $\neq \pi(s)$ **then**
      policy_stable $\leftarrow$ *false*
    end if
  end for
until *policy_stable* is *true*

A.2 Monte Carlo Methods

Algorithm 4 First Visit Monte Carlo Prediction

Initialize $V(s) \forall s \in \mathcal{S}$ and $\pi(s) \in \mathcal{A}$
Initialize $Return(s) \leftarrow [\quad] \quad \forall s \in \mathcal{S}$
while true **do**
  Generate a episode following π: $s_0, a_0, r_1, s_1, a_1, r_2 ... s_{T-1}, a_{T-1}, r_T$
  $G \leftarrow 0$
  for each step $t = T-1, T-2 ... 0$ **do**
    $G \leftarrow G + \mathcal{R}_{t+1}$
    if $s_t \notin s_0, s_1 ... s_{t-1}$ **then**
      $Return(s_t).append(G)$
      $V(s_t) \leftarrow average(Return(s_t))$
    end if
  end for
end while

A.3 Temporal Difference Methods

Algorithm 5 SARSA Algorithm

Initialize $Q(s, a)$ $\forall s \in \mathcal{S}$ and $\forall a \in \mathcal{A}$
for episode=1, M **do**
  Receive initial state s_0
  for t=1, T **do**
    Select action a_t from s_t using policy derived from Q (e.g., ϵ-greedy)
    Get the next state s_{t+1} and reward r_t
    Select action a_{t+1} from s_{t+1} using policy derived from Q
    Update $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$
    $s_t \leftarrow s_{t+1}$
    $a_t \leftarrow a_{t+1}$
  end for
end for

Algorithm 6 Q-learning Algorithm

Initialize $Q(s, a)$ $\forall s \in \mathcal{S}$ and $\forall a \in \mathcal{A}$
for episode=1, M **do**
  Receive initial state s_0
  for t=1, T **do**
    Select action a_t from s_t using policy derived from Q (e.g., ϵ-greedy)
    Get the next state s_{t+1} and reward r_t
    Update $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$
    $s_t \leftarrow s_{t+1}$
  end for
end for

A.4 Deep Deterministic Policy Gradient

Algorithm 7 DDPG Algorithm

Initialize critic network $Q(s, a|\sigma)$ and actor network $\pi(s|\theta)$
Initialize target critic Q' and actor π' with parameters $\sigma' \leftarrow \sigma$ and $\theta' \leftarrow \theta$
Initialize Replay Buffer $\mathcal{D}$
for episode = 1, M **do**
  Initialize Random Process $\mathcal{N}$ for action exploration
  Receive initial state s_0
  for t=1, T **do**
    Select action $a_t = \pi(s_t|\theta) + \mathcal{N}_t$
    Execute a_t and observe s_{t+1} and r_t
    Store the tuple (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
    if $\mathcal{D}$ is full **then**
      Sample mini-batch of N transitions (s_j, a_j, r_j, s_{j+1}) from $\mathcal{D}$
      Set the target $y_j = r_j + \gamma Q'(s_{j+1}, \pi'(s_{t+1}|\theta')|\sigma')$
      Update the critic minimizing the Loss $L = \frac{1}{N}\sum_i \left(y_j - Q\left(s_j, a_j \mid \sigma\right)\right)$
      Update the actor using the sampled policy gradient:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_j \left[\nabla_a Q(s, a \mid \sigma)|_{s=s_j, a=\pi(s_i)} \nabla_\theta \pi(s \mid \theta)\Big|_{s=s_j} \right]$$

      Update target critic $\sigma' \leftarrow \tau \cdot \sigma + (1-\tau) \cdot \sigma'$
      Update target actor $\theta' \leftarrow \tau \cdot \theta + (1-\tau) \cdot \theta'$
    end if
  end for
end for

Bibliography

[1] M. Mozaffari, W. Saad, M. Bennis, Y.-H. Nam, and M. Debbah, "A Tutorial on UAVs for Wireless Networks: Applications, Challenges, and Open Problems", en, *arXiv:1803.00680 [cs, math]*, Mar. 2019. [Online]. Available: `http://arxiv.org/abs/1803.00680`.

[2] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search", *Nature*, vol. 529, pp. 484–489, Jan. 2016. DOI: `10.1038/nature16961`.

[3] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, *Mastering chess and shogi by self-play with a general reinforcement learning algorithm*, 2017. arXiv: `1712.01815 [cs.AI]`.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning", en, p. 9,

[5] L. Busoniu, R. Babuska, and B. De Schutter, "Multi-agent reinforcement learning: An overview", in. Jul. 2010, vol. 310, pp. 183–221, ISBN: 978-3-642-14434-9. DOI: `10.1007/978-3-642-14435-6_7`.

[6] N. C. Luong, D. T. Hoang, S. Gong, D. Niyato, P. Wang, Y.-C. Liang, and D. I. Kim, "Applications of Deep Reinforcement Learning in Communications and Networking: A Survey", en, *IEEE Communications Surveys & Tutorials*, vol. 21, no. 4, pp. 3133–3174, 2019, ISSN: 1553-877X, 2373-745X. DOI: 10.1109/COMST.2019.2916583. [Online]. Available: https://ieeexplore.ieee.org/document/8714026/.

[7] L. Hu, Y. Tian, J. Yang, T. Taleb, L. Xiang, and Y. Hao, "Ready Player One: UAV-Clustering-Based Multi-Task Offloading for Vehicular VR/AR Gaming", en, *IEEE Network*, vol. 33, no. 3, pp. 42–48, May 2019, ISSN: 0890-8044, 1558-156X. DOI: 10.1109/MNET.2019.1800357. [Online]. Available: https://ieeexplore.ieee.org/document/8726071/.

[8] P. S. Bithas, E. T. Michailidis, N. Nomikos, D. Vouyioukas, and A. G. Kanatas, "A Survey on Machine-Learning Techniques for UAV-Based Communications", en, *Sensors*, vol. 19, no. 23, p. 5170, Nov. 2019, ISSN: 1424-8220. DOI: 10.3390/s19235170. [Online]. Available: https://www.mdpi.com/1424-8220/19/23/5170.

[9] P. Mach and Z. Becvar, "Mobile Edge Computing: A Survey on Architecture and Computation Offloading", en, *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1628–1656, 2017, ISSN: 1553-877X. DOI: 10.1109/COMST.2017.2682318. [Online]. Available: http://ieeexplore.ieee.org/document/7879258/.

[10] S. Yi, Z. Hao, Z. Qin, and Q. Li, "Fog computing: Platform and applications", in *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, 2015, pp. 73–78. DOI: 10.1109/HotWeb.2015.22.

[11] P. V. Klaine, J. P. B. Nadas, R. D. Souza, and M. A. Imran, "Distributed Drone Base Station Positioning for Emergency Cellular Networks Using Reinforcement Learning", en, *Cognitive Computation*, vol. 10, no. 5, pp. 790–804, Oct. 2018, ISSN: 1866-9956, 1866-9964. DOI: 10.1007/s12559-018-9559-8.

[Online]. Available: http://link.springer.com/10.1007/s12559-018-9559-8.

[12] B. Jang, M. Kim, G. Harerimana, and J. Kim, "Q-learning algorithms: A comprehensive classification and applications", *IEEE Access*, vol. PP, pp. 1–1, Sep. 2019. DOI: 10.1109/ACCESS.2019.2941229.

[13] F. Venturini, F. Mason, F. Pase, F. Chiariotti, A. Testolin, A. Zanella, and M. Zorzi, "Distributed reinforcement learning for flexible uav swarm control with transfer learning capabilities", in *Proceedings of the 6th ACM Workshop on Micro Aerial Vehicle Networks, Systems, and Applications*, ser. DroNet '20, Toronto, Ontario, Canada: Association for Computing Machinery, 2020, ISBN: 9781450380102. DOI: 10.1145/3396864.3399701. [Online]. Available: https://doi.org/10.1145/3396864.3399701.

[14] G. E. Monahan, "A survey of partially observable markov decision processes: Theory, models, and algorithms", *Management Science*, vol. 28, no. 1, pp. 1–16, 1982, ISSN: 00251909, 15265501. [Online]. Available: http://www.jstor.org/stable/2631070.

[15] H. Y. Ong, K. Chavez, and A. Hong, *Distributed deep q-learning*, 2015. arXiv: 1508.04186 [cs.LG].

[16] C. H. Liu, X. Ma, X. Gao, and J. Tang, "Distributed Energy-Efficient Multi-UAV Navigation for Long-Term Communication Coverage by Deep Reinforcement Learning", en, *IEEE Transactions on Mobile Computing*, vol. 19, no. 6, pp. 1274–1285, Jun. 2020, ISSN: 1536-1233, 1558-0660, 2161-9875. DOI: 10.1109/TMC.2019.2908171. [Online]. Available: https://ieeexplore.ieee.org/document/8676325/.

[17] 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*, 2019. arXiv: 1509.02971 [cs.LG].

[18] C. H. Liu, Z. Chen, J. Tang, J. Xu, and C. Piao, "Energy-efficient uav control for effective and fair communication coverage: A deep reinforcement learning approach", *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 9, pp. 2059–2070, 2018. DOI: 10.1109/JSAC.2018.2864373.

[19] M. Assaf and M. Ndiaye, "Multi travelling salesman problem formulation", in *2017 4th International Conference on Industrial Engineering and Applications (ICIEA)*, 2017, pp. 292–295. DOI: 10.1109/IEA.2017.7939224.

[20] A. Giagkos, M. S. Wilson, E. Tuci, and P. B. Charlesworth, "Comparing approaches for coordination of autonomous communications UAVs", en, in *2016 International Conference on Unmanned Aircraft Systems (ICUAS)*, Arlington, VA, USA: IEEE, Jun. 2016, pp. 1131–1139, ISBN: 978-1-4673-9334-8. DOI: 10.1109/ICUAS.2016.7502551. [Online]. Available: http://ieeexplore.ieee.org/document/7502551/.

[21] G. Leu and J. Tang, "Survivable Networks via UAV Swarms Guided by Decentralized Real-Time Evolutionary Computation", en, in *2019 IEEE Congress on Evolutionary Computation (CEC)*, Wellington, New Zealand: IEEE, Jun. 2019, pp. 1945–1952, ISBN: 978-1-72812-153-6. DOI: 10.1109/CEC.2019.8790353. [Online]. Available: https://ieeexplore.ieee.org/document/8790353/.

[22] K. Kim, Y. M. Park, and C. Seon Hong, "Machine Learning Based Edge-Assisted UAV Computation Offloading for Data Analyzing", en, in *2020 International Conference on Information Networking (ICOIN)*, Barcelona, Spain: IEEE, Jan. 2020, pp. 117–120, ISBN: 978-1-72814-199-2. DOI: 10.1109/ICOIN48656.2020.9016432. [Online]. Available: https://ieeexplore.ieee.org/document/9016432/.

[23] J. Yao and N. Ansari, "Online Task Allocation and Flying Control in Fog-Aided Internet of Drones", en, *IEEE Transactions on Vehicular Technology*, vol. 69, no. 5, pp. 5562–5569, May 2020, ISSN: 0018-9545, 1939-9359. DOI:

10.1109/TVT.2020.2982172. [Online]. Available: https://ieeexplore.ieee.org/document/9043589/.

[24] N. T. Ti and L. Bao Le, "Joint Resource Allocation, Computation Offloading, and Path Planning for UAV Based Hierarchical Fog-Cloud Mobile Systems", en, in *2018 IEEE Seventh International Conference on Communications and Electronics (ICCE)*, Hue: IEEE, Jul. 2018, pp. 373–378. DOI: 10.1109/CCE.2018.8465572. [Online]. Available: https://ieeexplore.ieee.org/document/8465572/.

[25] J. Li, Q. Liu, P. Wu, F. Shu, and S. Jin, "Task Offloading for UAV-based Mobile Edge Computing via Deep Reinforcement Learning", en, in *2018 IEEE/CIC International Conference on Communications in China (ICCC)*, Beijing, China: IEEE, Aug. 2018, pp. 798–802, ISBN: 978-1-5386-7005-7. DOI: 10.1109/ICCChina.2018.8641189. [Online]. Available: https://ieeexplore.ieee.org/document/8641189/.

[26] L. Yang, H. Yao, J. Wang, C. Jiang, A. Benslimane, and Y. Liu, "Multi-UAV-Enabled Load-Balance Mobile-Edge Computing for IoT Networks", en, *IEEE Internet of Things Journal*, vol. 7, no. 8, pp. 6898–6908, Aug. 2020, ISSN: 2327-4662, 2372-2541. DOI: 10.1109/JIOT.2020.2971645.

[27] A. Al-Hourani, S. Kandeepan, and A. Jamalipour, "Modeling air-to-ground path loss for low altitude platforms in urban environments", in *2014 IEEE Global Communications Conference*, 2014, pp. 2898–2904. DOI: 10.1109/GLOCOM.2014.7037248.

[28] C. Stöcker, R. Bennett, F. Nex, M. Gerke, and J. Zevenbergen, "Review of the current state of uav regulations", *Remote Sensing*, vol. 9, no. 5, 2017, ISSN: 2072-4292. DOI: 10.3390/rs9050459.

[29] İ. Bekmezci, O. K. Sahingoz, and Ş. Temel, "Flying Ad-Hoc Networks (FANETs): A survey", en, *Ad Hoc Networks*, vol. 11, no. 3, pp. 1254–1270, May 2013, ISSN: 15708705. DOI: 10.1016/j.adhoc.2012.12.004.

[30] A. Srivastava and J. Prakash, "Future FANET with application and enabling techniques: Anatomization and sustainability issues", en, *Computer Science Review*, vol. 39, p. 100 359, Feb. 2021, ISSN: 15740137.

[31] M. Bacco, P. Cassarà, M. Colucci, A. Gotta, M. Marchese, and F. Patrone, "A survey on network architectures and applications for nanosat and uav swarms", in *Wireless and Satellite Systems*, P. Pillai, K. Sithamparanathan, G. Giambene, M. Á. Vázquez, and P. D. Mitchell, Eds., Cham: Springer International Publishing, 2018, pp. 75–85, ISBN: 978-3-319-76571-6.

[32] M. Marchese, A. Moheddine, and F. Patrone, "Iot and uav integration in 5g hybrid terrestrial-satellite networks", *Sensors*, vol. 19, no. 17, 2019, ISSN: 1424-8220. DOI: 10.3390/s19173704. [Online]. Available: https://www.mdpi.com/1424-8220/19/17/3704.

[33] A. Fotouhi, H. Qiang, M. Ding, M. Hassan, L. G. Giordano, A. Garcia-Rodriguez, and J. Yuan, "Survey on uav cellular communications: Practical aspects, standardization advancements, regulation, and security challenges", *IEEE Communications Surveys Tutorials*, vol. 21, no. 4, pp. 3417–3442, 2019. DOI: 10.1109/COMST.2019.2906228.

[34] R. Bellman, "The theory of dynamic programming", *Bulletin of the American Mathematical Society*, vol. 60, no. 6, pp. 503–515, 1954. DOI: bams/1183519147. [Online]. Available: https://doi.org/.

[35] A. Slivkins, *Introduction to multi-armed bandits*, 2019. arXiv: 1904.07272 [cs.LG].

[36] "Playing Atari with Deep Reinforcement Learning", Dec. 2013.

[37] "Learning representations by back-propagating errors.", *Nature 323*, pp. 533–536, 1986. DOI: https://doi.org/10.1038/323533a0.

[38] E. M. Dogo, O. J. Afolabi, N. I. Nwulu, B. Twala, and C. O. Aigbavboa, "A comparative analysis of gradient descent-based optimization algorithms on convolutional neural networks", in *2018 International Conference on Computational Techniques, Electronics and Mechanical Systems (CTEMS)*, 2018, pp. 92–99. DOI: 10.1109/CTEMS.2018.8769211.

[39] Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio, "Object recognition with gradient-based learning", in *Shape, Contour and Grouping in Computer Vision*, Berlin, Heidelberg: Springer-Verlag, 1999, p. 319, ISBN: 3540667229.

[40] H. van Hasselt, A. Guez, and D. Silver, *Deep reinforcement learning with double q-learning*, 2015. arXiv: 1509.06461 [cs.LG].

[41] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, *Dueling network architectures for deep reinforcement learning*, 2016. arXiv: 1511.06581 [cs.LG].

[42] R. Sutton, D. Mcallester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation", *Adv. Neural Inf. Process. Syst*, vol. 12, Feb. 2000.

[43] V. Konda and J. Tsitsiklis, "Actor-critic algorithms", *Society for Industrial and Applied Mathematics*, vol. 42, Apr. 2001.

[44] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, *Pytorch: An imperative style, high-performance deep learning library*, 2019. arXiv: 1912.01703 [cs.LG].

[45] P. Gawłowicz and A. Zubow, "Ns-3 meets OpenAI Gym: The Playground for Machine Learning in Networking Research", en, in *Proceedings of the 22nd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems - MSWIM '19*, Miami Beach, FL, USA: ACM Press, 2019, pp. 113–120, ISBN: 978-1-4503-6904-6. DOI: 10.1145/3345768.3355908.

[46] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, *Openai gym*, 2016. arXiv: 1606.01540 [cs.LG].

[47] *ZeroMQ*. [Online]. Available: https://zeromq.org/socket-api/.

[48] V. Mayor, R. Estepa, A. Estepa, and G. Madinabeitia, "Deploying a Reliable UAV-Aided Communication Service in Disaster Areas", en, *Wireless Communications and Mobile Computing*, vol. 2019, pp. 1–20, Apr. 2019, ISSN: 1530-8669, 1530-8677. DOI: 10.1155/2019/7521513.

Acknowledgements

Permettetemi di scrivere quest'ultima parte del mio lavoro di tesi nella mia lingua madre, al fine di poter ringraziare in maniera chiara e trasparente le persone che mi sono state vicine, sia fisicamente che spiritualmente, non solo durante la stesura di questo elaborato, ma in tutti i cinque anni del mio percorso universitario.
Il primo ringraziamento lo rivolgo ai miei relatori: la Professoressa Carla Fabiana Chiasserini e il Professor Enrico Natalizio. In realtà, il ruolo che hanno ricoperto è andato ben oltre quello di relatore. Nei miei confronti si sono comportati da vere e proprie guide, un costante punto di riferimento che mi ha orientato e portato non solo fino al raggiungimento di questo obiettivo, ma anche verso una crescita personale, fatta di nuove esperienze e preziosi consigli. Un sentito grazie anche al Professor Claudio Zito, il cui supporto è stato di fondamentale importanza nella realizzazione di questo lavoro. La sua professionalità e passione per la ricerca mi hanno ispirato e accompagnato nell'approfondimento di diversi aspetti di questa tesi.
Adesso tocca a voi, mamma Lucia e papà Raffaele. Grazie per l'amore, la fiducia e il sostegno che mi avete sempre dimostrato, non solo in questi anni universitari, ma da sempre. E comunque, non pensate sia finita qui! Tra me e Marco, ne avrete ancora molte da passare. E a proposito, grazie a te, Marco. Da bravo fratello maggiore mi hai sempre saputo dare la spinta che mi serviva, specialmente nei momenti di forte indecisione. Ci siamo sempre fatti forza a vicenda, tra una litigata e una risata, avendo costantemente la certezza di poter trovare in te la giusta dose di sincerità, saggezza, ma anche stravaganza, a seconda delle situazioni.
Non potrebbe mai mancare il ringraziamento per la mia compagna di viaggio (e di

viaggi ne abbiamo fatti parecchi), Elisa. Trovo difficilissimo racchiudere in qualche riga le cose per cui voglio dirti grazie. Sai già quanto mi ritenga fortunato ad averti avuta accanto in questi anni e quanto tu sia stata per me una costante fonte di ispirazione, per la tua tenacia, intelligenza e schiettezza. Una buona parte di ciò che sono oggi lo devo a te. Nonostante le distanze fisiche che ci hanno costantemente separato, sei sempre riuscita a farmi sentire la tua vicinanza. Adesso però, sappiamo entrambi qual è il prossimo obiettivo.

Un grazie ad Antonio e Loredana, che mi hanno sempre trattato come un figlio, consigliandomi e supportandomi in ogni momento. Grazie a voi per avermi aiutato più volte ad affrontare decisioni importanti con lo spirito giusto.

Grazie a tutti i miei amici sparsi per l'Italia. Senza di voi, dubito sarei riuscito ad arrivare fino a qui (completamente) sano di mente. In particolare, grazie ai miei fratelli adottivi, Fabio e Andrea, la cui distanza non riesce ad influire sul nostro legame. Grazie al mio gruppone Torinese: Angelica e Marco, Aldo e Valeria, Fetta e Fabio, Simona, Elania e Salvatore, Jessica e Donato, Ilaria e Angioletta. Grazie perché ciascuno di voi mi ha regalato momenti indimenticabili nel corso della mia esperienza universitaria che porterò per sempre nel cuore.

Un grazie anche a tutti i miei colleghi universitari, incontrati durante gli anni di triennale e di magistrale. Ho sempre ritrovato in tutti loro dei compagni affiatati, determinati, ma soprattutto generosi ed altruisti. Che fosse una battuta durante una lezione, un ripasso di gruppo prima di un esame o un'uscita la sera, mi avete sempre dato la giusta motivazione, o distrazione, di cui avevo bisogno.

Grazie ai miei zii Luca, Leonardo e Licia, a Giancarlo e Paola, a Davide e Sofia, ai miei cuginetti Federica, Stefano e Gabriele. Grazie per la vostra vicinanza e affetto che non mi avete fatto mai mancare.

Infine, vorrei dedicare l'ultima sezione ai miei nonni: Franco e Teresa, Gino e Clara. I primi due non ho mai potuto conoscerli, ma sono sicuro che mi siano sempre stati vicini, in tutte le strade che ho scelto e che sceglierò di percorrere. Nonno Gino e nonna Clara si sono sempre presi cura di me e Marco ogni singolo giorno da quando

siamo nati, e sono sempre stati l'incarnazione dell'amore che i nonni provano per i propri nipoti. Nonno, tu sei e sarai sempre un esempio per me. Nonna, a te dedico questo mio lavoro. Ti ho persa proprio in questi mesi, e rimpiango di non averti potuto salutare come avrei voluto, ma sento che sei sempre stata al mio fianco da quel momento, e mi hai dato una spinta in più per andare avanti e concludere questo percorso. So che ci sarai sempre.