

# POLITECNICO DI TORINO

**Department of**

**Control and Computer Engineering (DAUIN)**

Master Degree in Mechatronic Engineering


# Reinforcement Learning approach for cooperative UAVs exploration of critical environments


**Supervisor**
Giorgio Guglieri

**Co-supervisor**
Simone Godio

**Candidate**
Gianpietro Battocletti



Academic Year 2020/2021

14th July, 2021

# Abstract

Unmanned Aircraft Systems (UASs) have become an important and promising field of study in the aerospace industry. Their versatility and efficiency have led them to be used in a considerable number of different applications. Research in this field is constantly increasing their capabilities and with them the number of tasks they are able to perform. For instance, it is only recently that developments in autonomous driving, often supported by artificial intelligence algorithms, have allowed them to work independently from human intervention. This advancement has greatly improved the possibility of using UASs in critical environments where it would be difficult or dangerous for a human to intervene. One of the most challenging problems for UAS is the collaborative operation of multiple Unmanned Aerial Vehicles (UAVs) in the same environment to perform a common set of tasks. The possibility for a fleet of UAVs to collaborate in the execution of the same objective would greatly increase the capabilities of UASs. The ability to work together efficiently would speed up operations in many situations, and allow to specialise each UAV for a specific task. This could open up a whole new set of applications where autonomous UAV fleets could be employed. Different solutions are currently being proposed and studied to address this challenge.

In this thesis, a new approach for collaborative exploration of critical environments using a small fleet of UAVs is proposed. The goal is to design an Artificial Intelligence-based algorithm able to guide an autonomous drone fleet in the exploration of an unknown environment. This kind of task presents several different challenges. In fact, each drone must be capable of moving in space without hitting any obstacle or other drones. At the same time, it has to continue the exploration task - or any other task assigned to it. While performing these tasks, the drones must also communicate with each other in order to coordinate the exploration following a common strategy and share useful information to optimise the execution of the task. All these issues have to be solved and their solutions merged in an organic algorithm. The focus of this thesis will be on the sections of algorithm regarding path planning, obstacle avoidance and exploration operations. The other parts listed here will be only briefly discussed. The proposed solution consists of a combined approach of different methods that are merged in an innovative way that allows to exploit the strong points of each of them. Some methods used, like the Artificial Potential Field, have been used for many years in the engineering field and are widely studied. Others, like Deep Reinforcement Learning, are far more recent and their capabilities are still being explored and tested. The combination of these methods allows to increase the capabilities of the "classical" ones, enhancing their capacities beyond those achieved so far.

# Contents

# Table of Acronyms

| | |
|---|---|
| UAS | Unmanned Aerial System |
| UAV | Unmanned Aerial Vehicle |
| AI | Artificial Intelligence |
| ML | Machine Learning |
| RL | Reinforcement Learning |
| APF | Artificial Potential Field |
| NN | Neural Network |
| MDP | Markov Decision Process |
| DQN | Deep Q-Learning |
| DDPG | Deep Deterministic Policy Gradient |
| EMB | Environment Model Builder |
| SGD | Stochastic Gradient Descent |

# 1. Introduction

## 1.1 Thesis overview

The goal of this thesis is to develop an algorithm to perform autonomous exploration tasks with a fleet of Unmanned Aerial Vehicles (UAVs). UAVs have become an important area of the aerospace industry thanks to their versatility, efficiency and cheapness. The development of autonomous UAVs represents a very interesting research topic, with several critical challenges as well as many potential application fields. This thesis fits in this large research field, and in particular in the area concerning the development of autonomous navigation algorithms through Artificial Intelligence (AI) techniques. This research area has grown large in the last decades, following the rapid development of AI techniques, in particular for what regards those based on Neural Networks. The proposed algorithm is based on Reinforcement Learning (RL) methods, and it revolves around the training of two Neural Network agents capable of taking real-time optimal decisions in order to explore an unknown environment and accomplish a given task. The basic idea is to equip each UAV with Neural Network (NN) agent that, given the information currently available, is able to choose the optimal path to continue the exploration process. In doing so, the presence of other UAVs is also taken into account in order to promote the collaboration of the fleet components toward the completion of the exploration. For simplicity, the exploration task is split into a *Coverage* task and a *Path Planning* task, as will be illustrated in detail later. Each part of this task is carried out by a dedicated RL agent. The training of these agents is the core of this thesis work. Some other parts of the document concern the development of the training and simulation framework, as well as the development of the additional parts of the algorithm (e.g., the code that manages the environment model that is used by the agents to take their decisions).

The proposed algorithm has been written mostly in the Python language, with the exception of some parts written in MATLAB. The development and testing have been made in a custom-designed environment. The code was written in Python 3.8.3. Some relevant libraries used to develop the code are Tensorflow 2.3.0, Keras 2.4.3 and OpenAI Gym 0.17.3. As already mentioned, some sections of the code have been written in MATLAB 2020b. The most relevant parts of the code can be found in the GitHub repositories https://github.com/gbattocletti and https://github.com/gbattocletti-riccardoUrb. In the second one, a more complete version of the algorithm can be found, taking into account also the work of Riccardo Urban. The thesis work was developed in close collaboration with Riccardo Urban, whose MSc thesis revolved around very similar topics to this one. In particular, almost the entire simulation and training framework was developed commonly since the algorithms proposed in the two theses were meant to work in the same type of environment. In both theses, the goal was to perform exploration tasks in an unknown environment, and given the many common traits, it was decided to build

a common development framework to speed up both works. The two theses can be consulted together to get the description of the complete exploration algorithm, composed of both the path planning and coverage parts.

## 1.2 Thesis outline

The thesis is organised as follows. In chapter 2 the state of art of autonomous UAVs research is presented, along with some examples of possible applications for the algorithm that will be designed. In chapter 3 the mathematical and engineering methods employed in the design of the algorithm will be presented and discussed. The presentation of the theoretical tools made in this chapter is completed by the in-depth discussion that can be found in the appendices. In particular, appendix A contains a discussion of Neural Networks structure and functioning, along with a brief illustration of Stochastic Gradient Descent and Backpropagation methods, which are the two fundamental tools used in Neural Network training. Appendix B contains the mathematical discussion of the k-means clustering method which has been used to build some sections of the coverage algorithm. In chapter 4 the structure of the algorithm is finally presented, getting into detail for what regards the model with which the environment is represented, the Path Planning algorithm and the strategy employed to coordinate the drone fleet in the exploration process. As mentioned in the introduction, these last two parts of the algorithm are implemented using RL agents. The training process carried out to obtain those agents is described in chapter 5. The performances obtained with these agents are discussed and compared with other similar algorithms in chapter 6. Here simulations performed with the implemented algorithm are analysed and discussed. In chapter 7 an extension of the algorithm to the 3D case will be presented. This chapter contains the first iteration of the implementation of the code in a 3D environment, and there is much space for refinement. However, since almost every real-world application is in a 3D environment, it is worth showing the potential of the algorithm in that scenario. The conclusions of the work, along with the discussion of the algorithm issues and possible future development, will be presented in chapter 8.

# 2. State of art

Before diving into the algorithm design and development process, it is worth dedicating a few words to the scientific and technical framework in which this work is developed. In fact, as autonomous UAVs have garnered significant interest in the academic and scientific community, a considerable number of papers and publications has been produced on the topics touched by this thesis. In this chapter, a brief review of these works will be performed in order to picture the state of art of this research field. In the first two sections of the chapter (sections 2.1 and 2.2) the state of art of two areas of interest in the autonomous UAVs world is discussed. The last section (section 2.3) is devoted to the description of some relevant application areas.

## 2.1 Path planning and navigation

A considerably large number of algorithms has been developed in the past to perform path planning and path following operations [1, 2, 3]. Each of these methods presents its own points of strength and weaknesses, and has its own distinctive characteristics. Before discussing in detail some of the path planning strategies developed in the past, it is worth making a brief digression about the meaning of *path planning*. Path planning is an operation performed to compute a kinematically-feasible curve in space connecting a point A with a point B. Strictly speaking, path planning does not take into account the presence of obstacles, which are taken care of by an *obstacle avoidance* algorithm [1, 4]. However, most of the algorithms that will be described here, as well as the algorithm developed in this thesis, perform both the operations, making no clear distinction between the two: the objective is to compute a flyable path which avoids any collision with obstacles between point A and point B. Therefore, throughout the thesis the term *path planning* will be used to indicate the *path planning and obstacle avoidance* operations. Another distinction should be made between the word *path* and the word *trajectory*. The first is simply a geometric curve $\Phi(x, y, z)$ in space, whereas the second is a curve parametrised in time, i.e. $\Phi(t) = (x(t), y(t), z(t))$. Therefore, a *path following* operation is different from a *trajectory following* one: the first requires simply to stay close to the curve, while the second requires to track a point in space that moves over time. The algorithm developed in this thesis regards only the path planning operation; all through the document the words *path* and *trajectory* will be used interchangeably with the meaning of *path*.

A final remark should be made about the fact that all the path planning algorithms considered work *online* and in *real-time* (or can be adapted to do so). An online algorithm has the possibility to integrate new information over time and update its result according to them. On the contrary, an offline algorithm computes the path on the base of some initial information and has no way to update the path taking into account new information acquired by real-time

sensing. The second characteristic, *real-time*, only regards online algorithms and depends on the algorithm computational time, as well as on the speed requirements. If the computational time required by the algorithm to produce its result is inferior to the speed requirement (i.e. the maximum time allowed for the computations) then the algorithm works in real time. If the algorithm cannot respect this time constraint, it is not real-time. Only real-time algorithms can effectively react to sudden events in the environment.

Some examples of algorithms designed to perform path planning operations are Dijkstra's algorithm, A* [5], D* [6], and Artificial Potential Field [7]. All these algorithms are fairly light from the computational point of view, allowing them to be run online (by updating over time the environment model they use) and in real time in most applications, even with demanding time constraints. The common issue with all the methods listed above is that they require that the rules used to compute the trajectory are explicitly coded. This is quite simple if the only goal is to find the shortest path to a desired location. However, if the path requires some more complex characteristics (e.g. minimum travel time, take into account the agent dynamics, maximum possible distance from the obstacle), those algorithms become quite limited. In fact, the additional constraints on the path computation can result extremely difficult to explicitly code, and so the algorithms listed result insufficient to compute a path that meets all the desired conditions. To overcome these limitations, optimisation-based path planning algorithms are often used.

Optimisation-based path planning algorithms allow to compute a route optimised with respect to any desired constraint. The only condition is that each constraint must be written as an expression and integrated inside a *cost function* that will be then minimised in order to find the optimal path. This kind of approach allows to find very efficient paths. Different methods can be exploited to perform the optimisation process. One possible strategy is to write the trajectory planning problem as a convex optimisation problem (e.g. Linear or Quadratic Programming) and solve it. This solution is fairly fast and the solution existence is guaranteed, but it has some drawbacks. In fact, it is not always easy to cast trajectory planning as a convex optimisation problem. Moreover, all the constraints (collision avoidance, minimum travel distance...) also need to be expressed in an algebraic form suitable to be included in the optimisation problem.

A different approach to obtain optimisation/based trajectories is to use stochastic optimisation algorithms. In this case the problem formulation can be easier since the cost function does not need to have a specific shape like e.g. in Linear Programming. Examples of algorithms that can be used to implement a stochastic path planning algorithm are Particle Swarm Optimisation and Stochastic Gradient Descent [8]. A evolutionary algorithm can also be used, as discussed in [2] and [9]. It is worth noting that, in most cases, the optimisation process performed by these algorithms is not a *global* one, but is limited to coming close enough to the optimum. In fact, the main limitation of optimisation-based path planning algorithms is that their computational cost is much higher than the one of the algorithms introduced in the sections above. Since the microcomputers on which these algorithms usually run have limited computational power, it is difficult to have them run in real time, and usually their time constraints are more relaxed than the ones of the deterministic algorithm. De facto, to keep the computational cost sufficiently low, often these algorithms end up finding a sub-optimal solution (which is, however, generally acceptable for the path planning purpose). To overcome the time limitations posed by optimisation algorithms, different strategies can be used. A first possibility is to use a mixed approach of deterministic and optimisation-based methods, as in [10] and

[11]. Here the trajectory is first computed by a deterministic algorithm and then optimised by a dedicated routine. This way, the optimisation algorithm only has the task of refining the trajectory, significantly lowering the computational time required. A second possibility is to use Artificial Intelligence-based algorithms, as will be discussed next.

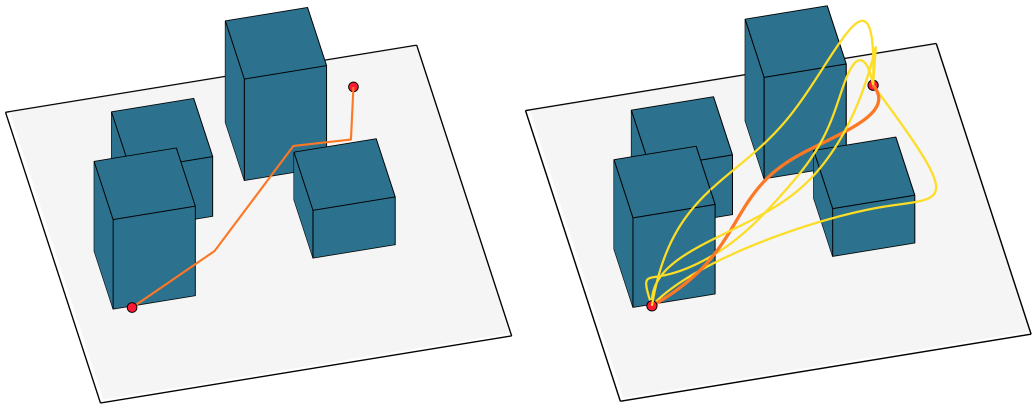

**Figure 2.1:** Example of a deterministic path planning algorithm (left image) compared with a trajectory research performed through a stochastic optimisation process (right image). The final trajectory on the right is more optimised than the other on the left, but the computation time can be high. On the contrary, the trajectory in the image on the left is computed quickly but is not optimised with respect to flight dynamics.

A class of algorithms that could be revolutionary in the path planning field is that of AI-based algorithms. This class of methods allows to compute optimal or nearly-optimal solutions (like in the case of optimisation-based algorithms) but at a fraction of the computational cost. This is possible thanks to a training process, separated from the actual algorithm run phase, that is performed offline and results in a function capable of generating optimal solutions with a very low computational cost. A significant advantage of AI-based algorithms is that they allow to overcome the limitation posed by the modelling processes. In fact, one big limitation of "classical" and optimisation-based algorithms is that the models used have intrinsic limits (due to the modelling process), which affect the accuracy with which they can represent real-world phenomena. AI-based algorithms are usually learning-based, which means they do not need a model to learn how to perform their operations. The independence from models allows them to be effective in many situations where the modelling process would introduce uncertainties and approximations. This kind of algorithms usually involves Neural Networks, that can be trained to compute the optimal trajectory and run at a fairly low computational cost. In this sense, AI-based algorithms behave like an optimisation function, but without the heavy computational cost commonly associated to the optimisation process. AI-based path planning algorithms are just at the beginning of their history, and there is plenty of room for experimentation in this field. Some works that have been taken as a reference while developing the path planning algorithm are [12, 13].

## 2.2 Fleet coordination

In the past decade, the technological capabilities of UAVs significantly grew, while their cost dropped. This lead to a growing interest in the possibility of creating UAV swarms or fleets, capable of coordinating in the pursue of a common goal. In this section, a brief overview of the state of art of fleet coordination algorithms will be discussed. Up to now, the majority of the research in this field has focused on hierarchically organised fleets, which means they have a leader that takes decisions and is followed by the other UAVs. Less attention has been given to *leaderless* fleets of UAVs, which present higher individual independence from the UAVs. The leaderless fleet case is more interesting for what regards the exploration of unknown environments, since the independence of the individual agent can lead to a more optimal organisation of the task. However, it is worth to go through some of the coordination algorithms developed in the past, since some interesting ideas can be taken from them. Algorithms to implement fleet coordination behaviours are often bio-inspired. Techniques derived from fish or birds gave birth to flocking algorithms, whereas others observed in colonies of insects gave rise to stigmergy algorithms or the so-called "ant colony optimisation" approaches, which together compose the larger class of *swarming* algorithms. The main references for this section are the books *Swarm Intelligence* [14] and *Swarm and Evolutionary Computation* [15]. Some additional resources will be indicated along with the discussion of the single methods. It is worth mentioning that, for what regards the fleet coordination problem, only a few research works have been found involving Reinforcement Learning (or, more in general, Artificial Intelligence) techniques [16].

Bio-inspired swarming algorithms represent a very interesting approach to the fleet coordination problem, as well as to the exploration problem. They are based on so-called *emerging behaviours*. In this kind of approach, each UAV has a very simple set of rules to follow. When applied by a single UAV, these rules are usually not very effective. However, when they are applied by the whole fleet, they become extremely powerful. In fact, when the swarm is considered as a whole, it behaves in an efficient way, managing to perform complex operations that were not explicitly coded in the basic rules. These complex behaviours are called *emerging behaviours* or *collective behaviours* and their study is very promising, especially for large swarms of cheap and simple units. These algorithms are surprisingly powerful when applied to swarms of UAVs with limited computational power. Even though every single UAV can only perform very simple operations, the swarm as a whole becomes capable of behaving in complex ways, presenting powerful self-organisation properties [17]. These algorithms are suitable to develop *leaderless* UAV swarms, which does not need hierarchical organisation and exploit emerging behaviour to obtain complex and efficient exploration policies [18, 19].

A particular class of swarming algorithms that can be exploited to perform exploration tasks is that of stigmergy-based algorithms [20, 21]. In these types of bio-inspired methods, each agent is able to modify the environment around himself and - in doing so - to indirectly communicate with the other members of the fleet. In the most common implementation of stigmergy algorithms, each agent bases its behaviour on the presence - or not - of pheromone around it. This substance is constantly released by all the agents, and therefore its presence and concentration give information about their present and past position. During an exploration task, each agent tries to path away from the regions with the highest concentration of pheromone in order to discover new areas and get information about them. An opposed strategy is employed to find and follow the most efficient way to cross a given area. In this scenario, each agent follows the highest pheromone concentration, following the path used by the highest

number of the other agents (this approach is usually called *ant-colony optimisation*). The issue with this approach lies in the management of the pheromone. In fact, insects use a physical substance to mark their path and "indirectly" communicate. The use of a physical substance is infeasible for a swarm of UAVs, and therefore the solution commonly used is to exploit a *digital* pheromone. However, when following this approach the release of the pheromone must be *directly* communicated between the UAVs, requiring that they constantly are in communication range and visibility. Moreover, the pheromone diffusion in the environment must be simulated, making this approach quite heavy from a computational standpoint. These two drawbacks render this method quite inefficient, especially since this approach (as well as many other based on emerging behaviours and collective intelligence) is well-suited for swarms of UAVs having low computational power.

A different approach to the fleet coordination problem is that of formation-based algorithms (also called *flocking* algorithms). Formation-based algorithms can be very efficient to drive a whole group of agents toward a desired goal [22, 23, 24]. When following this approach, fleets are usually hierarchically organised; all the agents follow a given leader that chooses the goal to reach and moves toward it. This allows to significantly reduce the computational cost for all the "follower" agents, since they just need to maintain a given distance or position with respect to the leader, and to modify it only to avoid obstacles (all while maintaining a line of sight with at least one other UAV in order to keep communication lines opened [25]). The fact that the fleet needs to maintain a constrained formation can lead to a loss of versatility in some environments, where a wider or closer distribution of the UAVs, as well as their independent movement, could significantly speed up operations. In addition to this, UAVs must always be able to communicate with each other to know where to go, having little independence in case of loss of communication. These two drawbacks make flocking algorithms inefficient when used for exploration purposes.

## 2.3 Applications

It is worth devoting some space to discuss some of the possible applications of the algorithm presented in this thesis. As already said, autonomous UAVs fleet could be employed in a large number of fields. An exposition of some possible areas of application can be found in [4]. However, it is worth pointing out some specific applications where the usage of fleets of UAVs and automatic coverage and exploration algorithms could be particularly valuable.

- an application discussed in [26] is to use UAV fleets to map the tree population of forest environments. In this application, the UAV fleet moves through the forest guided by a leader drone, and maintaining a given formation it surrounds a tree and gets all sorts of information about it such as age, species, dimension, state of health etc. This sort of application would be helpful to monitor forest resources, especially in large areas difficult to reach. Moreover, it would be possible to build a database containing all the data about the forest, allowing to monitor its evolution over the years and to plan in advance resource management, both in terms of logging and reforestation. The proposed algorithm could be suitable to drive the fleet in the exploration of the forest, spreading the UAVs to map the location and disposition of trees. The fleet would likely switch periodically between the proposed exploration algorithm and a formation-based one, where the latter would be used for the analysis operations around a desired tree;

- UAVs can be employed in forest environments also for surveillance tasks. In particular, an application of interest is that of spotting and monitoring forest fires. UAVs can be employed first of all for detection tasks, and in particular for the reduction of false positives as in [27]. More effective techniques, such as forest monitoring with satellites from LEO, can be used for the early detection of forest fires, but UAVs can be used to confirm the observation and rapidly assess the extent of the fire. An even more effective application of UAV fleets is found in the monitoring of such fires [28, 29]. UAVs can move rapidly and safely around the emergency area, constantly collecting data that can be crucial for an effective response and containment of the fire. In this case, the use of a fleet of UAVs instead of a single UAV has many advantages [18]. First of all, a more wide area can be covered and monitored, increasing the speed and efficiency of the emergency response. Moreover, a wider set of sensors and instruments can be employed, having them carried by different UAVs and allowing a more accurate and detailed mapping of the fire;
- another field in which UAVs can significantly help is in emergency response [30]. After a disaster, like an earthquake or flooding, swarms of UAVs can be extremely efficient for reconnaissance tasks, to map the extent of the damages or to look for survivors. Moreover, they can be used to evaluate the damages on buildings or civil structures by using dedicated sensors or techniques like photogrammetry, which allows to build a 3D model of an object starting from a collection of images taken from a different point of view. The autonomy and high manoeuvrability of UAVs allow them to reach virtually any location, even in critical environments where it would be dangerous for human operators to go;
- UAV swarms can also find application in the agriculture sector. The increasing use of automated machines in precision agriculture has gained significant attention of farmers and industries to minimise human work load to perform tasks such as land preparation, seeding, fertilising, plant health monitoring and harvesting [31]. The management of resources like water and soil will be of paramount importance in the near future to reduce their waste and lower the impact of crops on the environment. The use of aerial imagery obtained through UAVs would allow to map the different crops, observe their state of maturation and the possible presence of plant diseases [32]. This can help optimise the use of resources employed for their growth, and even reduce the use of unnecessary chemical agents. UAVs could even be employed to assist ground robots in autonomous harvesting operations, by supplying images and data to support their tasks such as path planning and movement between the different crops [31];
- one of the most fascinating and challenging applications of autonomous UAVs is in the field of planetary exploration. It is with the landing of the Perseverance rover on the surface of Mars that the first human-made propelled flying machine has reached another planet [33]. Ingenuity is the first UAV deployed in an environment different from Earth. Since radio signals take an average of 11 minutes to reach Mars, it is mandatory for Ingenuity to be completely autonomous in the flight operations, as there is no way to control it in real time from an Earth-based ground station. It is only thanks to recent development in autonomous flight and control that this kind of mission has become possible [34]. The Ingenuity helicopter has been designed to be able to fly autonomously in the challenging martian atmosphere for about 90s to perform exploration tasks thanks to a set of dedicated sensors. The mission is designed to be primarily a technology

demonstrator, but in case of success the employment of autonomous UAVs (both alone and organised in fleets) could become one powerful tool to rapidly explore large martian regions, where the ground rovers currently used would take years to fulfil the same task.

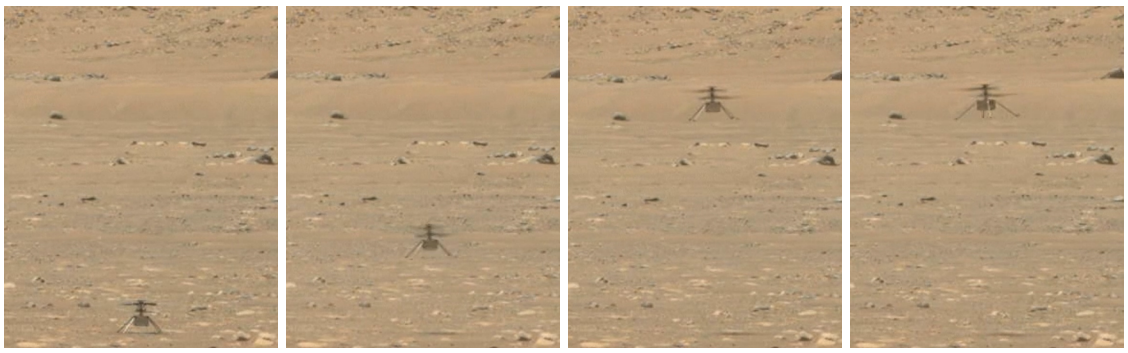

**Figure 2.2:** Frames from the Perseverance video of Ingenuity first flight (from [33]).

- an even more exciting example of the employment of autonomous UAVs for planetary exploration comes from the Dragonfly mission [35]. The Dragonfly mission is designed to be deployed on Titan (the largest moon of Saturn) by 2035, and will be a completely autonomous UAV capable of flying over Titan surface. Dragonfly will map Titan surface from above through its cameras and collect and study soil samples with dedicated instruments in search of clues of life presence [36]. Radio signals take an average of 1.5 hours to reach Titan from Earth, rendering it even more crucial that Dragonfly is completely autonomous in almost every operation, including exploration planning, path planning and flight control operations. Artificial intelligence algorithms will be crucial for the success of such a complex mission, especially since the environment where Dragonfly will move is almost completely unknown, with the only available information coming from satellite photographs.

# 3. Theoretical background

In this chapter, the methods used in the development of the exploration algorithm will be briefly presented. In section 3.1 Artificial Potential Field will be illustrated and discussed, along with some modifications made to the original formulation to better fit the problem. In section 3.2 the theoretical background of Reinforcement Learning will be introduced along with the learning algorithm used to train the *Path Planning* and *Coverage* agents.

## 3.1 Artificial Potential Field

Artificial Potential Field (APF) is an algorithm presented in 1985 as a path planning method [7]. APF is based on the creation of a virtual force field in the environment where the robot (or, more in general, the *agent*) has to move. The main idea is that the field is generated so that, by following the negative direction of the potential gradient, the agent is able to reach the goal. This behaviour is obtained by associating a repulsive force to obstacles (higher close to them and decreasing to zero after a certain distance) and an attractive force to the goal. At every time instant, the agent computes the potential field value in the point where it is located and computes the gradient of the field. Then, it moves in the direction of the negative gradient at the desired speed (or, as in the original formulation, the gradient can be used to generate a velocity vector, taking into account also its magnitude). The potential, given agent position $x$, is found as:

$$U(x) = U_{attr}(x) + \sum_{i=1}^{n} U_{rep,i}(x) \tag{3.1}$$

where $U_{attr}(x)$ is the attractive potential generated by the goal and $U_{rep,i}(x)$ is the repulsive potential generated by the $i$-th obstacle. The two contributions are defined as follows. The attractive potential is usually defined as a cone or a paraboloid with the vertex in the goal, obtaining, in the 2D case, a potential shaped as in figure 3.1. When the attractive potential is defined as a cone, the equation describing its intensity in a generic point $x$ is:

$$U_{attr}(x) = k_a \|x - x_g\| \tag{3.2}$$

where $k_a$ is a positive constant and $x_g$ is the goal position. The negative gradient of this potential always points toward the goal $x_g$ and is the component of the gradient that allows the agent to reach the goal. The obstacle avoidance part of the path planning algorithm is implemented by the repulsive potential. A high potential value is associated to each obstacle following a hyperbolic or gaussian distribution (depending on the definition - the former is more common). This way, the closer a point is to the obstacle, the higher is the potential associated to it. When computing the negative gradient of this potential distribution, the resulting vector points away from the obstacle itself, allowing the agent to avoid any collision. The equation

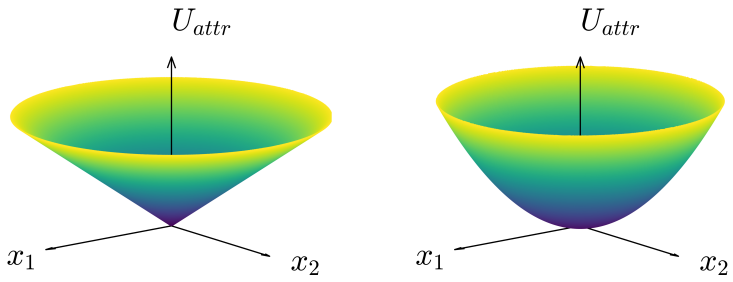


**Figure 3.1:** Attractive potential distribution over a 2D space. On the left, the shape resulting from conic definition (as the one of equation 3.2); on the right, the one obtained from parabolic definition.

implementing the repulsive potential distribution, when defined as a gaussian curve,[1] is:

$$U_{rep,i}(x) = \begin{cases} k_{rep}\ \frac{1}{\sigma\sqrt{2\pi}}\ e^{-\frac{1}{2}\left(\frac{d(x)}{\sigma}\right)^2} & \text{if } d(x) < d_0 \\ \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

where $x$ is the point where the repulsive potential is computed and $d(x) = |x - x_{obs}|$ is the distance from the obstacle. It is worth noting that the repulsive potential goes to zero after a certain distance $d_0$ from the obstacle. This distance is called *distance of influence* or *safe distance*. The constants $\sigma$ and $k_{rep}$ are used to tune the shape of the repulsive potential field around obstacles. Once all the potential contributions are summed, the agent can compute the potential gradient in order to obtain a versor $\vec{v}$ pointing toward the direction of maximum steepness of the potential, corresponding to the direction of movement to follow.

$$\vec{v} = -\frac{\nabla U(x)}{|\nabla U(x)|} \tag{3.4}$$

The traditional APF algorithm main issue is the presence of local minimum points. Local minima easily emerge when the complexity of the environment grows in terms of number and shape of the obstacles. Those points can be very difficult to deal with, since their presence cannot be easily predicted and they trap the agent into a region where the gradient vector is null. In fact, when the gradient magnitude is zero (i.e., when $U$ is maximum or minimum) the versor of eq. (3.4) does not exist and so the agent does not have a direction to move toward. Since this problem has been known for a long time, different solutions have been developed to deal with it [13, 37, 38, 39, 40].

[1] It must be said that very few implementations of the APF use this equation to define the repulsive field, at least according to the available literature. In fact, the majority of APF implementations use a hyperbolic repulsive potential distribution, like in the original proposal of the APF algorithm [7]. The reasons for the choice of the gaussian shape will be discussed in chapter 4. For completeness, the equation defining the hyperbolic potential field is reported here: $U_{rep,i}(x) = \frac{k_{rep}}{\gamma}(\frac{1}{d_i(x)} - \frac{1}{d_0})^\gamma$ if $d_i(x) < d_0$, $U_{rep,i}(x) = 0$ otherwise. It is worth noting that, while the equation is different, in general the shape of the repulsive potential is the same - growing quickly approaching the obstacle and decaying to zero after a certain distance from it.

## 3.2 Reinforcement Learning

Reinforcement Learning (RL) is a branch of Artificial Intelligence, and in particular it is one of the main ramifications of Machine Learning, along with Supervised Learning and Unsupervised Learning. The goal of Reinforcement Learning is to train an agent to take optimal actions inside a certain environment. To reach this goal, an iterative trial-and-error process is performed inside a training environment. The basic idea in RL is to deploy the untrained agent in this environment and have it perform actions, to which a reward is assigned. If the reward is positive, the agent will increase the probability of repeating the same action when faced with the same situation. On the contrary, if the reward is negative the likelihood of repeating the same action is decreased. At the end of the training process, the trained agent will consist of a function called *policy* able to choose the optimal action for every situation the agent is faced with.

### 3.2.1 Markov Decision Process (MDP)

The mathematical framework through which most Reinforcement Learning problems are formalised is the Markov Decision Process (MDP) [41, 42]. A Markov Decision Process is a discrete-time stochastic process that is mathematically defined as a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$. $\mathcal{S}$ is a set of states: a state $s \in \mathcal{S}$ fully represents the environment at a certain time instant. Its definition resembles the one used in control theory, where states are defined as "the smallest possible subset of system variables that can represent the entire state of the system at any given time". An important property of the state variables is that they allow to fully know the system configuration without requiring any information about the system past history. $\mathcal{A}$ is the set of actions, and contains all the possible actions that the agent can take from any state. In some instances this set could be indicated with $\mathcal{A}_s \subseteq \mathcal{A}$, which represents the set of actions that can be taken from state $s$. Both $s \in \mathcal{S}$ and $a \in \mathcal{A}$ can be continuous or discrete variables, depending on the problem definition. $\mathcal{P}$ is a probability transition function, which determines the probability to reach state $s'$ from state $s$ when action $a$ is taken, namely $\mathcal{P}(s, s', a) = \mathrm{P}(s_{t+1} = s' | s_t = s, a_t = a)$. The probability transition function implies that, in general, the MDP is a stochastic process, since the transition from state $s_t$ to state $s_{t+1}$ is computed through a probability distribution. However, in some situations (like the one that will be analysed later in the algorithm) the environment is fully deterministic and the state at time $t+1$ can be derived from $s$ and $a$ at time $t$, so $\mathcal{P}(s, s', a) = 1$ only for state $s'$ and 0 for all the others. Lastly, $\mathcal{R}$ is a reward function associated to the transition from state $s$ to $s'$ due to action $a$. For any combination of $s$, $a$, $s'$ the function returns $r = \mathcal{R}(s, s', a)$. The block scheme in figure 3.2 represents the relationship between all the variables involved in an MDP: given a state $s_t$, the agent takes an action $a_t$ which modifies the environment leading to a new state $s_{t+1}$ and to a reward $r_{t+1}$. State $s_{t+1}$ is then fed back to the agent to compute $a_{t+1}$.

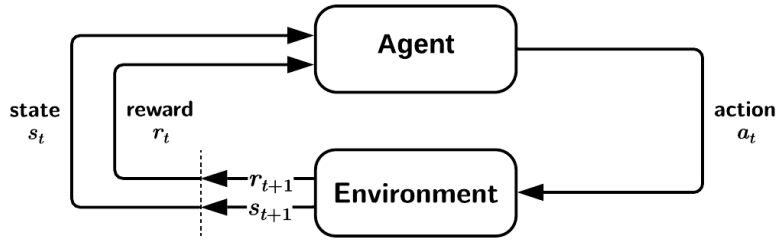


**Figure 3.2:** Simple block-scheme representation of MDP (image taken from [42]).

From the block scheme representation the equations governing the time evolution of an MDP can be derived. The evolution over time of this system, given an initial state $s_0$, is described by the system of equations (3.5). An important property that the equations point out is that an MDP benefits of the Markov Property, which states that the future state $s_{t+1}$ depends only on the current state $s_t$ and on the action $a_t$.

$$\begin{cases} s_t \in \mathcal{S} \\ a_t = \arg\max_{a \in \mathcal{A}} \pi(a|s = s_t) \\ s_{t+1} \text{ is derived from } s_t, a_t \text{ through } \mathcal{P}(s, s', a) \\ r_{t+1} = \mathcal{R}(s, s', a) \end{cases} \qquad \forall t = 0...N. \tag{3.5}$$

In the MDP mathematical framework, the agent takes its decisions based on a function called *policy* which is usually indicated with $\pi$. For the sake of simplicity, it could be said that the policy *is* the agent.[2] Once correctly trained, for any input state the policy returns a probability distribution over the action set (i.e. it returns $\pi(a|s) = \mathrm{P}(a_t = a|s_t = s)\ \forall a \in \mathcal{A}$) that indicates how likely is every action to return the highest reward. From this distribution, the action with the highest probability of returning the greatest return is selected. A simplified way to write the selection of the best action by the agent is $a = \pi(s_t)$ where the research of the probability distribution maximum is hidden. A policy that always points to the action that returns the highest reward is called *optimal policy* and is denoted as $\pi^*$. The goal of a Reinforcement Learning problem is to find (or, more likely, *approximate*) the optimal policy through a suitable training process. As mentioned, the optimal policy is the one that maximises the reward. Usually, the *discounted cumulative reward* is considered, in order to take in account long-term strategy by the agent. The *discounted cumulative reward* function (also called *discounted return*) is defined as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} = r_{t+1} + \gamma\, G_{t+1} \tag{3.6}$$

where $r_i$ is the reward obtained at time step $i-1$ selecting an action through the policy. The *discount factor* $\gamma < 1$ progressively reduces the importance of future rewards in the sum. Depending on the approach followed, the maximisation of the expected return is obtained following different strategies, as will be discussed in more detail later.

### 3.2.2 Optimal policy and Bellman Equation

A brief discussion about optimal policy properties is helpful before getting to the discussion of the RL learning process. As already said, the goal of an RL problem is to obtain a policy $\pi$ that is as close as possible to the optimal policy $\pi^*$ (managing the fact that the function$\pi^*$ is not explicitly known). To reach this goal, it is useful to study the properties of the optimal policy function. To mathematically define the optimal policy, two functions have to be introduced first. The first is the state-value function:

$$v_\pi(s) = \mathbb{E}[G_t|s_t = s] \tag{3.7}$$

[2] This definition is correct in the MDP framework. On the contrary, in the RL framework this is not completely true since the agent also includes the reinforcement learning training algorithm.

where $\mathbb{E}$ is the *expected value* function. The state-value function computes the expected return when starting from a given state and following policy $\pi$ for all the subsequent action choices. The second function used to measure the optimality of a policy is the action-value function:

$$q_\pi(s, a) = \mathbb{E}[G_t | s_t = s, a_t = a] \tag{3.8}$$

This function measures the goodness of action $a$ in state $s$ by evaluating the expected return when starting from state $s$, choosing action $a$, and following policy $\pi$ thereafter. These two functions, and in particular the action-value function $q(s, a)$, are useful to measure the optimality of a policy. The optimal policy $\pi^*$ can be defined as:

$$\pi^* : \pi^* \geq \pi \quad \forall\, \pi \in \Pi \tag{3.9}$$

where $\Pi$ is the set of all possible policies, while the operator $\geq$ is defined as:

$$\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s) \qquad \forall s \in \mathcal{S} \tag{3.10}$$

The optimal policy definition can be rewritten as a function of the action-value and state-value functions. In fact, the optimal policy is that where the state-value function is maximum for every state and, for every state, the action chosen is the one that leads to the maximum action-value function, as expressed in equations (3.11) and (3.12):

$$v^*(s) = \max_\pi v_\pi(s) \tag{3.11}$$

$$q^*(s, a) = \max_\pi q_\pi(s, a) \tag{3.12}$$

Equation (3.12) is the most interesting, since it allows to connect the action choice to the optimal policy. An interesting idea about the policy approximation could emerge here. Knowing the $q$ function, one could choose the optimal action by evaluating all actions available in state $s$ and perform the one which returns the highest value of $q$. This would allow to perform optimal actions without having to *actually* approximate the policy function. This means that the agent would not approximate the function $\pi^*$. Instead, the function approximated would be the action-value function $q$, which would allow to evaluate the quality of all the available actions and choose the best one.[3] An agent that approximates the action-value function is called a *critic*,[4] and is at the base of some learning algorithms like DQN (Deep Q-Network). In those algorithms, the agent is trained to approximate the $q^*$ function and use it to choose the best action. The issue is that function $q^*$ is not known, and so an approximation of it is used during the training. In the majority of critics, the $q^*$ function is approximated starting from the Bellman equation, reported below in (3.13). This equation is obtained by substituting (3.8) in (3.12) and manipulating the equation starting from the expression of $G_t$ seen in equation (3.6).

$$q^*(s, a) = \mathbb{E}\Big[r_{t+1} + \gamma \max_{a'} q^*(s', a')\Big] \tag{3.13}$$

---

[3] This statement assumes that the number of actions is finite. If, on the contrary, the number of actions is infinite - e.g. in case of a *continuous* action space - the use of the $q$ function approximation is a bit different, as will be shown in the DDPG algorithm in section 3.2.5.

[4] On the contrary, an agent that directly approximates $\pi^*$ is called an *actor*.

### 3.2.3 Policy approximation with Neural Networks

As introduced above, the goal of a Reinforcement Learning problem is to find a good approximation of the optimal policy $\pi^*$ or the optimal action-value function $q^*$, depending on the type of agent that is being trained. As already shown in equation (3.8), $q^*$ is a function of $a$ and $s$. Under the assumption of stationary agents[5] the policy $\pi$ can also be written as a function of $s$. In both cases, the goal of the training process is to approximate a function; in the first case, the function to be approximated is a mapping from the state space $\mathcal{S}$ to the action space $\mathcal{A}$, while in the second the approximation is from set $\mathcal{S} \times \mathcal{A}$ to the scalar value $q \in \mathbb{R}$. Different strategies can be exploited to approximate those functions. The one that will be discussed in this section - which is one of the leading ones in the RL world - is to approximate the agent functions with an Artificial Neural Network (ANN or just NN). The Universal Approximation Theorem guarantees that any function between two sets can be approximated with arbitrarily small error through a NN [43], guaranteeing that such an approximation is legitimate. At this point the new goal is to find the NN that approximates the desired function.

A Neural Network is a function composed of a layered structure of *neurons* connected between them. An example of NN is represented in figure 3.3. The function takes $n_i$ inputs, which are stored in an equal number of neurons collected in the *input layer*, and returns $n_o$ output values in the *output layer*. Between the input and output layer, a series of *hidden layers* are placed.

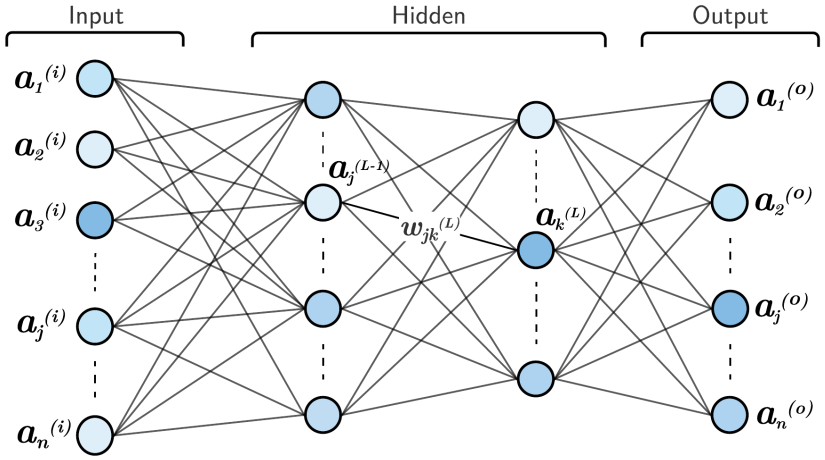


**Figure 3.3:** Simple Neural Net scheme. The input, output and hidden layers are put in evidence. Each neuron holds a value $a_j^{(L)}$ and a bias $b_j^{(L)}$, while each connection holds a weight $w_{jk}^{(L)}$. The colours represent fictitiously the higher or lower value of the neurons.

The basic working principle of a NN is that each neuron performs a specific mathematical operation and stores its result $a_j^{(L)}$. The result is then passed to the neurons in the following layer through some connections. Each connection is associated to a weight, which multiplies the neuron value before passing it to the following neuron. The operation performed by each

[5] A *stationary agent* is an agent that chooses the best action only on the basis of the current state [41, 42]. Every agent that will be considered in this discussion and in the algorithm development fits in this category, coherently with the properties of the MDP with which RL problems are formalised.

neuron includes a usually very simple nonlinear function called the *activation function*. The usage of the layered structure and the nonlinear functions allows to approximate any desired function by properly tuning some parameters. A more detailed discussion of the functioning of Neural Networks can be found in appendix A. The only relevant information, for the moment, is that a NN is a *parametric* function. The vector of parameters is indicated with $\theta$. Being the NN a parametrized function, the approximated policy can be written as $\pi_\theta(s)$ and the action-value function as $q_\theta(s, a)$. These two functions, built through a NN, must get as close as possible to their optimal correspondents $\pi^*$ and $q^*$. In the end, the objective of an RL problem is *to find the parameter set $\theta$ to build a NN that approximates the desired optimal function as well as possible*,[6] i.e. to find $\pi_\theta(s) \approx \pi^*(s)$.

### 3.2.4 Policy training in the Reinforcement Learning framework

In the previous section it was defined that the goal of an RL training process is to obtain a set of parameters $\theta$ so that the policy NN function $\pi_\theta(s)$ approximates the optimal policy $\pi^*$ as close as possible.[7] The issue is that the function to be approximated is not known. In fact, the mapping from $\mathcal{S}$ to $\mathcal{A}$ is typically a complex function, quite difficult to be described analytically. It is at this point that the Reinforcement Learning main idea comes into play. As shown in the block scheme of figure 3.4, the RL training process is based on an iterative proceeding which is derived from the MDP structure. As illustrated in the figure, the policy is trained by having it

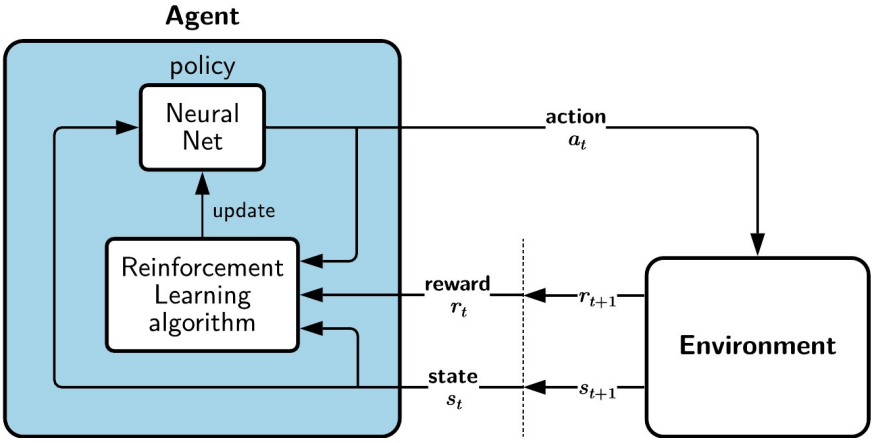


**Figure 3.4:** Block scheme representation of a RL training process. The basic structure is the same of the MDP represented in figure 3.2. However, in this case the agent internal structure and the presence of the RL training algorithm are put in evidence.

choose an action, performing it in the environment and updating the policy on the basis of the reward obtained. The basic idea is that, given a state $s_t$, if the reward obtained for action $a_t$ is positive, the policy will be updated to encourage the choice of the same action when state $s_t$ -

[6] It is worth mentioning that, in general, many NN can approximate the same function, so the goal is to find the optimal set of parameters for a specific NN. The NN architecture choice is important to reduce the training time and computational complexity of the net.

[7] For simplicity, in this chapter the optimal policy $\pi^*$ will be used as function to be approximated. All the discussion and results hold also when approximating the action-value function $q^*$ with a critic agent.

or a state similar to it - is received as input. On the contrary, if the reward is negative, the choice of the same action will be discouraged. This learning strategy, characteristic of the RL framework, is implemented through a family of learning algorithms. Each algorithm has its own distinctive characteristics, but the general working routine is the same:

1. given the current environment state $s_t$, choose action $a_t$ through policy $\pi_\theta(s)$ or randomly, depending on the exploration/exploitation strategy adopted (see below for details);
2. perform action $a_t$ in the environment to get reward $r_{t+1} = \mathcal{R}_{a_t}(s_t)$ from the environment, along with the next state $s_{t+1}$. Here, it is assumed that the state transition function is deterministic (as in the deterministic case of the MDP introduced in section 3.2.1) so, given $s_t$ and $a_t$, $s_{t+1}$ can be exactly determined;
3. compute the cost value (i.e., the estimation error) through an arbitrarily defined *cost function*. The cost function commonly used is:

$$L = \sum_{j=1}^{n_o} \left( a_j^{(o)} - y_j \right)^2 \tag{3.14}$$

   where $a_j^{(o)}$ is the value of the $j$-th output layer neuron, $n_o$ is the number of neurons in the output layer and $y_j$ is the desired ("correct") value of the output;
4. use the cost value to update the NN parameters and get a closer approximation of the optimal policy $\pi^*$ (this step is performed by following the Stochastic Gradient Descent algorithm explained in detail in appendix A);
5. repeat from the beginning updating the state with the one obtained in step 2 ($s_t \leftarrow s_{t+1}$).

The issue of this training routine lies in step 3, since - from the moment that the function to be approximated is not analytically known - the correct value of the output is unknown too. One of the RL training algorithms main goals is, therefore, to find a way to compute the loss without knowing the "real" value of $y$. Different methods can be used to approximate or estimate $y$ and the loss value, mainly by starting from the reward $r$. One of those methods will be presented in section 3.2.5 along with the corresponding training algorithm.

The last consideration before getting to the learning algorithms regards step 1 of the routine above. In fact, one key aspect of RL training is to try as much different strategies as possible, in order to be sure that the one implemented by the trained NN is the best possible. To do so, during training the action $a_t$ is chosen through a strategy that balances *exploration* and *exploitation*. Exploration means that the actions are selected independently from the current NN output, in order to try new strategies and possibly find better ones with respect to the current ones. Exploitation, instead, means that the action is selected only on the base of the NN and is necessary to verify that the NN action choice is truly efficient, as well as to consolidate good strategies already learned. There are several ways to promote exploration. A common one is the $\varepsilon$-greedy strategy.[8] Another one, which involves the use of a Ornstein-Uhlenbeck random noise process, will be presented in the DDPG learning algorithm discussed below.

[8] The $\varepsilon$-greedy strategy uses a time-varying probability distribution to decide whether to sample a random action from the action space or to use the action chosen by the agent i.e., $a = \pi(s)$.

### 3.2.5 Deep Deterministic Policy Gradient (DDPG)

The learning algorithm that has been used during all the thesis development is Deep Deterministic Policy Gradient (DDPG).[9] As can be seen in figure 3.5, DDPG is a model-free and off-policy learning method. It is suitable to work with *continuous state spaces*, *continuous action spaces* and is an *actor-critic* method.

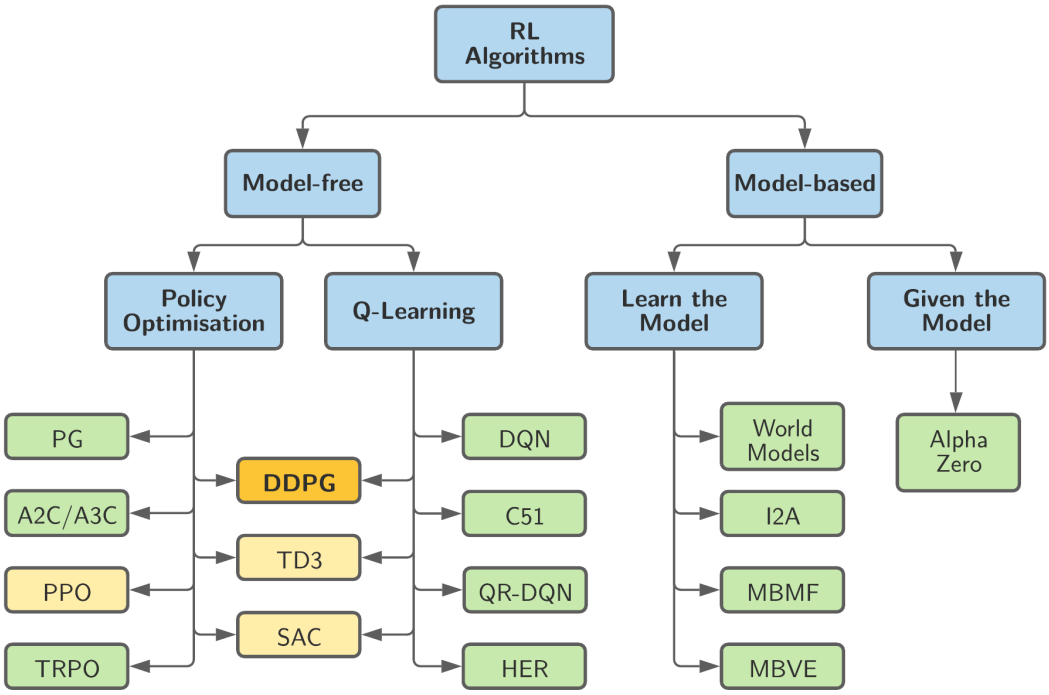


**Figure 3.5:** Scheme of the categorisation of some of the most popular RL algorithms. DDPG is the one chosen for the development of the path planning and coverage algorithms. The other three algorithms evidenced in yellow (PPO, SAC, T3D) are also suitable to work with a continuous action space, but are more advanced and complex to implement. They could be used for future developments of the algorithm. Image is taken from [44].

The fact that it is an actor-critic method means that, in the end, this algorithm produces an actor, i.e. a function $\mu_\theta$ that approximates the optimal policy $\pi^*(s)$. This function can be directly used to find the optimal action $a$ for state $s$ by computing $a = \mu_\theta(s)$. To train the actor, DDPG first trains a critic network $Q_\phi(s, a)$ capable of approximating the Q-value function. The network $Q_\phi(s, a)$ becomes able to evaluate the goodness of action $a$ in state $s$ by approximating the optimal Q-value function introduced in equation (3.13) of section 3.2.2. The fact that DDPG trains both an actor and a critic network is the reason why, in the image above, it is categorised both as a Policy Optimisation method and a Q-Learning based one. In the training process, the critic network learns the Q-value function directly from the sampled data; the Q-value function approximation obtained in this way is then used to train the actor.

[9] At the beginning of the development process DDQN (Double Deep Q-Network) was also considered. However, the fact that DDQN only works with a discrete action space severely limited its capabilities, and in the end it was discarded in favour of DDPG.

In fact, in DDPG the goal of the actor is to find the action $a$ that maximises the Q-value, i.e., $a = \mu_\theta(s) = \arg\max_a Q_\phi(s, a)$, so the data provided by the Q-value function are the ones used to train it. The complete learning routine of DDPG is illustrated in algorithm 3.1 [45, 46].

**Algorithm 3.1** DDPG learning algorithm

initialise the actor and critic $\mu(s|\theta)$ and $Q(s, a|\phi)$ with random parameters $\theta$ and $\phi$
initialise the target networks $\mu'(s|\theta')$ and $Q'(s, a|\phi')$ with parameters $\theta' = \theta$ and $\phi' = \phi$ [10]
initialise the experience buffer $B$ with size $N_B$
**for** $e$ in $0...N_{episodes}$ **do**
    initialise a random Ornstein-Uhlenbeck noise process $\mathcal{N}$ for action exploration[11]
    reset episode initial condition (i.e., obtain state $s_0$)
    **for** $t$ in $0...N_{steps}$ **do**
        select action according to the current actor policy and exploration noise as:

$$a_t = \mu(s_t) + \mathcal{N}_t \tag{3.15}$$

        execute action $a_t$ in the environment and collect reward $r_t$ and the new state $s_{t+1}$
        store the experience sample $(s_t, a_t, r_t, s_{t+1})$ in the experience buffer $B$
        randomly pick $N_m$ samples from $B$ and store them in the minibatch $M$
        compute the critic target $\forall i$ in $1...N_m$ as:

$$y_i = r_i + \gamma\, Q'\Big(s', \underbrace{\mu'(s')}_{a'}\Big) \tag{3.16}$$

        compute the minibatch *cost function* and use it to update the critic parameters $\phi$ by performing one step of Stochastic Gradient Descent (as described in appendix A). The cost function gradient is computed as:

$$\nabla_\phi C = \frac{1}{N_m} \nabla_\phi \sum_{i=1}^{N_m} \Big(y_i - Q(s_i, a_i)\Big)^2 \tag{3.17}$$

        update the actor policy performing one step of Stochastic Gradient Ascent:

$$\begin{aligned} \nabla_\theta J &= \frac{1}{N_m} \sum_{i=1}^{N_m} \nabla_\theta Q\left(s, \mu(s_i)\right) \\ &= \frac{1}{N_m} \sum_{i=1}^{N_m} \nabla_a Q(s, a)|_{s=s_i, a=\mu(s_i)} \nabla_\theta \mu(s)|_{s=s_i} \end{aligned} \tag{3.18}$$

        update the target networks parameters following the smoothed-upgrade law:

$$\theta' = \tau\theta + (1 - \tau)\theta' \tag{3.19}$$

$$\phi' = \tau\phi + (1 - \tau)\phi' \tag{3.20}$$

        update the current state $s_t \longleftarrow s_{t+1}$
    **end for**
**end for**

It is worth dedicating a few words to the description of the main features of the learning process.

- as already introduced, two NNs are trained simultaneously in a DDPG learning process. The first is a *critic* NN, whereas the second is called the *actor* network. The former has the task of evaluating the goodness of an action $a$ when it is selected in state $s$. To do so, the critic $Q_\phi$ learns to approximate the Q-value function $q^*(s,a)$. The actor $\mu_\theta$, instead, uses the evaluation produced by the critic to learn to predict which action will lead to the highest Q-value, i.e., the actor learns to maximise the Q-value and therefore it learns to select the best actions in each state. Only the actor NN is deployed after the training process is completed;

- the critic network $Q(s,a)$ is trained using the data sampled in the environment. During each step, a training sample $(s,a,r,s')$ is stored in memory. These data can be used to train $Q_\phi$ by exploiting the Bellman equation. At each critic learning step, the gradient of the cost function

$$C(\phi) = \frac{1}{N_m} \sum_{i=1}^{N_m} \Big( y_i - Q(s_i, a_i) \Big)^2 \tag{3.21}$$

  is computed (as the gradient of the average cost of the minibatch). The value of $y_i$ (i.e., the "real" value of the action-value function, which is, in general, unknown) is approximated by the Bellman equation in the form written at line 12 of the algorithm. The resulting equation can be substituted in equation (3.21) to obtain the cost function:

$$C(\phi) = \frac{1}{N_m} \sum_{i=1}^{N_m} \Bigg( \underbrace{r_i + \gamma\, Q'\Big(s', \mu'(s')\Big)}_{\substack{\text{critic target value } y_i \\ \text{computed by using} \\ \text{the target network } Q'}} - Q(s_i, a_i) \Bigg)^2 \tag{3.22}$$

  It is worth noting that in equation (3.22) the Q-value of the action $a'$ is not computed directly by the critic network, but is instead obtained from a "delayed" copy of it called *target network* (since its goal is to compute the target value $y_i$ that the function has to reach). The usage of a target network highly increases the training process stability, since it decouples the network computing the target from the network being updated, and so it avoids a "network chasing its own tail"-like situation. The target network output is different from the one of the "main" network since the former one is updated through a smoothed update law and so the value of its parameters is "delayed" in time with respect to the "main" one. The smoothed update law, as already seen in the algorithm, is:

$$\phi' = \tau\phi + (1-\tau)\phi' \tag{3.23}$$

  where the value of $\tau$ (which is called *smoothing factor* and is smaller than 1, e.g. $\tau = 0.05$) determines how fast the target network is updated to match its "main" counterpart;

---

[10] In the algorithm description the expression $\mu'(s)$ and $Q'(s,a)$ are used instead of $\mu'(s|\theta')$ and $Q'(s,a|\phi')$ in order to streamline the notation. The same is done for $\mu(s)$ and $Q(s,a)$. The difference between the actor/critic networks and their target counterparts is expressed by the $'$ superscript.

[11] In DDPG this self-correlated noise is used to promote action exploration instead of the $\varepsilon$-greedy approach mentioned earlier.

- in DDPG, the role of the actor is to learn a deterministic policy $\mu_\theta(s)$ that maximises the function $Q_\phi(s, \mu(s))$. In fact, the critic network approximating the Q-value function can be used to find the best action to perform by solving the maximisation problem

$$a = \arg\max_a Q(s, a) \tag{3.24}$$

  However, in a continuous action space this maximisation problem is not easy to be solved, since $Q$ is - in general - non linear and its analytical expression is not known. Therefore, the actor network is specifically trained to compute an action $a = \mu(s)$ that maximises equation (3.24). To do so, a *Stochastic Gradient Ascent* problem is solved:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J \tag{3.25}$$

  where $\alpha$ is the actor learning rate (some more details about gradient descent/ascent methods are discussed in appendix A). The function $J$ whose gradient is computed is simply the Q-value function, as shown in equation (3.18). The passages performed to derive equation (3.18) are:

$$\begin{aligned} \nabla_\theta J &= \nabla_\theta Q(s, a) \\ &= \nabla_\theta Q\Big(s, \mu(s|\theta)\Big) \\ &= \nabla_a Q(s, a|\phi)_{\phi=\text{const}} \nabla_\theta \mu(s|\theta) \end{aligned} \tag{3.26}$$

  where the last passage is performed by just applying the chain rule to the derivation of $Q$. The subscript $\phi = \text{const}$ means that, during the computation of the gradient of $J$, the critic parameters are considered constant and so $Q$ is not derived with respect to them;

- an *off-policy* learning strategy is used to train the NNs. In fact, in step 11 of algorithm 3.1 a random set of data samples are extracted from the *experience buffer* $B$ and stored in the *minibatch* $M$ to be used for the training part of the routine. The fact that these data are randomly sampled from $B$ means that the data with which the NNs (and in particular the critic one) are trained are not the data generated by the agent in its current state of training, but they could have been produced by an "older" version of it. This is not a problem in terms of training efficiency. On the contrary, since the Bellman equation used to train the critic network must work for any given set of data (i.e., it must be able to correctly evaluate the goodness of any action $a$ taken in any state $s$), the usage of older data guarantees the robustness of the Q-value function approximation. The approach of using older data to train the NNs (i.e. data which have not necessarily been produced by the agent in its current state) is called - as already mentioned - off-policy learning;

- the use of *minibatches* $M$ instead of single data samples during the learning steps (i.e., during the gradient computation and parameter update operations) is introduced to increase the stability of the training process. In fact, the minibatch allows to obtain a better approximation of the gradient, introducing less noise in the Stochastic Gradient Descent operation. A more in-depth discussion of the use of minibatches can be found in appendix A;

- as mentioned at the end of section 3.2.4, during the learning process it is necessary to promote the exploration of the action space. This is required to try new possible actions -

different from the ones computed by the agent in its current training state - which could lead to the discovery of new strategies through which higher rewards could be obtained. In the case of DDPG, the exploration of new actions is obtained by summing a noise (produced through a Ornstein-Uhlenbeck process) to the action computed by the agent. This kind of noise process, differently from random Gaussian noise, is not uncorrelated and keeps track of its past value. Therefore, the noise shows a trend toward an arbitrary direction, and this helps the exploration of that specific direction of the action space.[12] Over time, the trend changes so that the action space is fully explored after a sufficient amount of steps.

[12] Some critics have been made against the use of an Ornstein-Uhlenbeck noise process instead of a random Gaussian one. In fact, some papers discussing DDPG state that the Ornstein-Uhlenbeck noise process does not improve training performances and is actually just an over-complication of the algorithm [47, 48]. During the training of this thesis agents, an Ornstein-Uhlenbeck process was used nonetheless.

# 4. Algorithm design

In this chapter, the algorithm structure is presented. The chapter starts with a section about the assumptions done prior to the design process (section 4.1). The chapter continues with the description of the algorithm structure, where the relationship between the algorithm and the simulation environment (as well as with the real world) is put in evidence (section 4.2). After this section, the algorithm is broken down into sections and each of them is described in detail. In particular, the focus is on the model with which the environment is represented (section 4.3), the path planning algorithm (section 4.4) and the coverage algorithm (section 4.5). The chapter ends with the description of the dynamic model of the UAV that was used during the simulations as well as some less relevant parts of the algorithm (section 4.6).

## 4.1 Assumptions and simplifications

A few assumptions have been made before starting the design of the exploration algorithm. It is worth discussing them to clearly circumscribe the field of applicability of the proposed method, along with its points of strength and its weaknesses. Some of these observations are better understood by reading the other parts of the chapter, but it is worth collecting all of them in a single place for ease of reference. The assumptions and simplifications done in the algorithm design are:

1. **2D environment:** to simplify the design process and the RL agent training algorithm has been implemented in a 2D environment. Two solutions have been considered to extend the algorithm to the 3D case. They are discussed in chapter 7;

2. **environment dimensions are known:** each drone stores an internal model of the environment, which is built starting from an empty data structure initialised using the environment dimensions. Therefore, this data has to be known. An upper bound on the dimensions is sufficient in most situations. An issue linked to the environment dimension is that, for environment that is very large, the map stored in the drone memory could become quite big and potentially slow to be accessed and manipulated. Therefore, map dimensions should not be too large (100m x 100m could be a good upper limit). A possible solution to manage larger environments would be to store two maps, one at a lower resolution that considers the whole environment and one at high resolution that represents only a neighbourhood of the drone (this could potentially be done even with more than two levels of resolution);

3. **map resolution is limited:** the environment model is stored with a given resolution, which means the space is discretized and information are saved for areas of space of size $r$ x $r$ (with $r$ being the spatial resolution with which the map is represented). Resolution

is approximately in the range 5cm-20cm. Any obstacle smaller than this size must be either represented as a bigger obstacle or neglected (quite obviously, the former solution is better since it is more conservative);

4. **the exact position of the UAVs is always known:** this assumption has been introduced to simplify the development ignoring positioning errors and uncertainties. In general, this assumption is not true since any method employed to estimate the position of the UAVs would be affected by some error. Exact position knowledge also includes the initial UAV location. In fact, when initialising the environment model data structure the absolute initial position of the UAV with respect to the environment is required;

5. **only static obstacles are considered:** obstacles position is stored in the local model of the environment. At the moment, an algorithm able to manage moving obstacles has not been implemented, and therefore all obstacles are assumed to be static (with the exception of the other drones, which are the only mobile obstacles considered since their position over time is assumed to be known);

6. **communications are instantaneous and have infinite bandwidth:** the communication between drones is not analysed in the algorithm development and is simplified as much as possible. Therefore, communications between drones are considered to be always possible, have infinite speed and infinite bandwidth, so any amount of data can be transferred instantaneously between any two UAVs. A better model of the communication system will be implemented in the future to obtain a more realistic simulation of the algorithm.

Any other information about the environment that has not been specifically listed here is unknown. This means the number, shape, and disposition of any obstacle is not known, i.e., at the beginning of the exploration the environment is represented with an empty map.

## 4.2 Algorithm structure

In this section, the structure of the algorithm is presented, as well as its relationship with the simulation environment and with the real world. First, the focus is put on the algorithm workflow and the main blocks that compose it. Each block is briefly described. An in-depth analysis of each of them is carried out in the following sections (sections 4.3, 4.4, and 4.5). The relationships amongst the different parts that compose the algorithm, i.e. how the different sections interact with each other, are also put in evidence in this section.

First of all, it is necessary to introduce the algorithm workflow. Figure 4.1 shows a block scheme of the algorithm pipeline. The goal of the algorithm is to compute a suitable trajectory to drive the UAV in the exploration of the environment. This task is split into two steps. The first is the computation of a temporary goal in the environment, and is carried out by the *coverage agent*. The temporary objective is placed so that, while reaching it, the UAV is able to discover as much information as possible about the environment (i.e., about the part of the environment it travels through). After this first step, the actual trajectory has to be computed. The trajectory computation is taken care of by the *path planning agent*, which also performs the obstacle avoidance operations. The link between the coverage agent and the path planning agent is the *environment model builder*. This piece of code has the task of managing the UAV internal representation of the environment. At each time step, the environment model builder takes

in input the temporary goal position and all the known obstacles positions and updates the environment model accordingly. The goal position is supplied by the coverage agent, whereas the obstacles positions can be obtained both from the on-board sensors or be communicated by the other UAVs of the fleet.

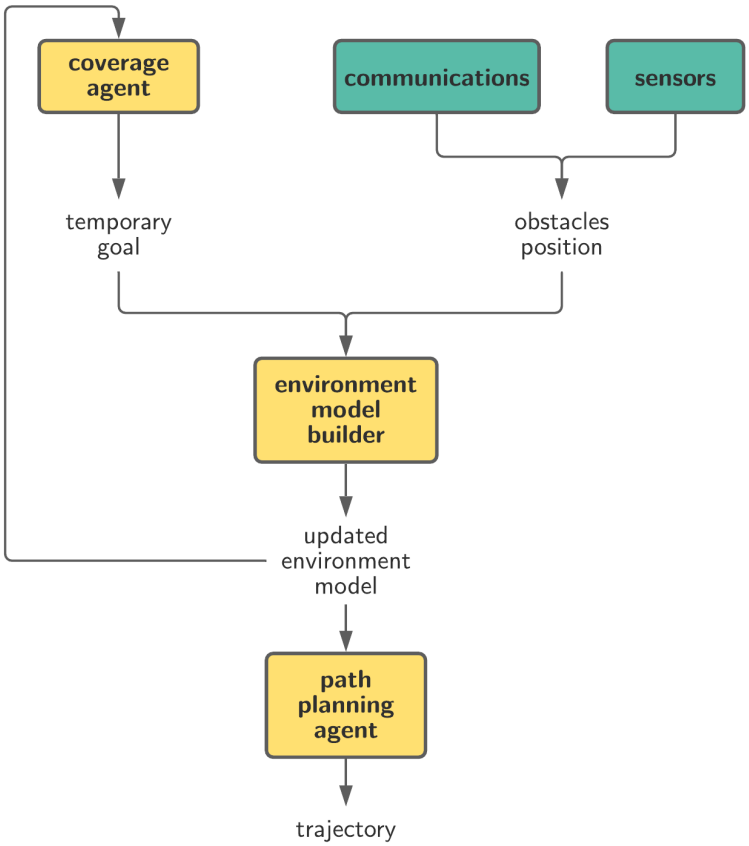


**Figure 4.1:** Block scheme of the algorithm workflow. The yellow blocks are the three core parts of the proposed algorithm, whereas the green ones represent some minor sections that are used to gather information about the environment.

The yellow blocks in the workflow of figure 4.1 represent the three core parts of the proposed algorithm. It is worth dedicating a few words for a brief description of these three sections:

- **Coverage planning algorithm**: this part of the algorithm is used by each member of the fleet to plan the exploration operations. It takes in input the environment model and the position of all the UAVs and uses this information to compute a target location (or *temporary goal*). Each UAV computes its own goal location, trying to position it to maximise the quantity of information that it is able to obtain while reaching it. To do this, each UAV has to maintain a sufficient distance from the other members of the fleet, avoid crossing areas already explored and minimise non-useful travels. The coverage planning algorithm is implemented as an RL agent;
- **Environment Model Builder**: this part of the algorithm is the one that manages the UAV internal representation of the environment. It takes in input the data from the on-board sensors of the UAV, the positions of all the other drones and the current location

of the temporary goal. Then, it uses this information to update the environment model. The environment model is stored in each UAV internal memory to be used by the other parts of the algorithm. Periodically, the model is shared with the other UAVs to make available any new information about the environment. This allows each UAV to know a larger portion of the map that the one it explored by itself;

- **Path planning algorithm**: the path planning algorithm uses the environment model to compute the optimal trajectory to reach the temporary goal. The trajectory is constantly updated since the environment model is continuously (in order to take into account new obstacles and changes in the goal location). The trajectory is optimised for what regards distance from obstacles, smoothness and travel time. As for the coverage agent, also the path planning algorithm is implemented as an RL agent.

The three sections just described are developed separately and then integrated in the pipeline of figure 4.1. To work correctly, however, the algorithm needs some additional pieces of code that are used to simulate all the operations and interactions that are not taken care of by the coverage agent, model builder and path planning agent. For this reason, a *simulation framework* is developed. This framework contains a set of functions that perform all the additional operations required by the algorithm to work. The simulation framework has been used to perform all the training sessions necessary to obtain the trained RL agents, as well as to simulate and evaluate their behaviour. The block-scheme structure of the algorithm paired with the simulation environment is shown in figure 4.2.

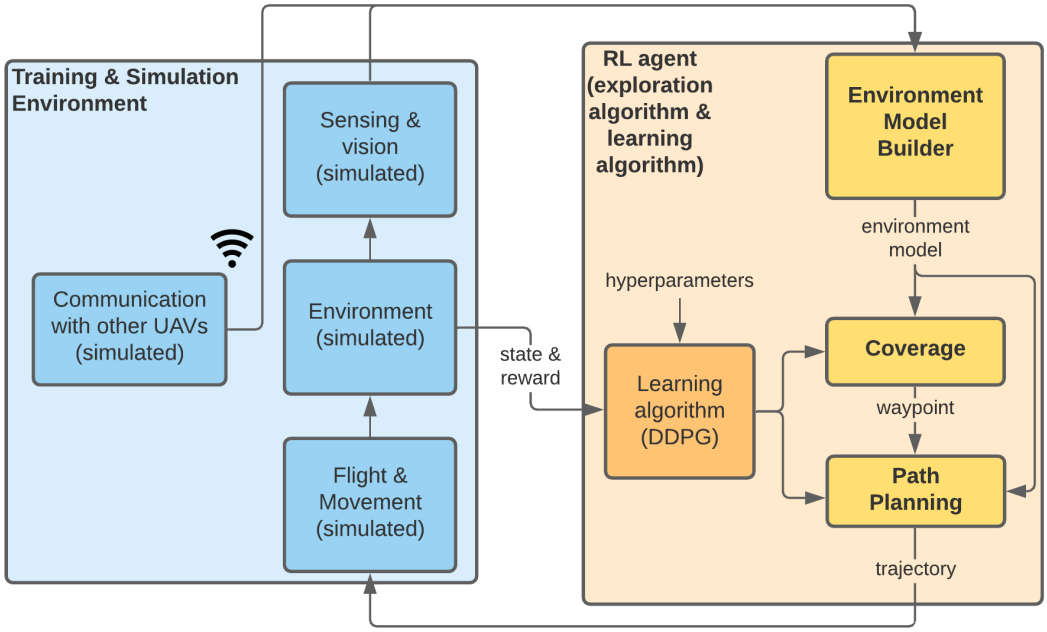


**Figure 4.2:** Block scheme of the proposed algorithm (yellow blocks) paired with the simulation & training framework (blue blocks). The yellow blocks represent the proposed algorithm parts, whereas the orange one represents the learning algorithm.

At its highest level of abstraction, figure 4.2 shows the interaction between the algorithm and the simulation environment. This environment has been created in order to simulate the external environment, as well as the UAV dynamics. For what regards the environment, the

simulation framework reproduces all its properties and the events happening in it. For what regards the UAV, instead, it simulates all the operations that are not part of the exploration algorithm but are needed by it. For instance, the simulation environment simulates the flight dynamics of the UAV while following the trajectory produced by the path planning agent (by using a simulated dynamic model). Moreover, it simulates the communication between the different UAVs, and also the sensing operations performed by the cameras that each UAV has on board. In particular, it simulates the use of a depth camera used to generate a point cloud representing obstacle positions, which is the main input of the environment model builder. The simulation of all these operations is performed through the custom-designed functions that compose the simulation framework. As already mentioned, this framework has been used to perform all the training operations, as will be discussed more in detail in chapter 5, as well as to obtain all the simulations presented in chapter 6.

After completing the training and validation process, the algorithm could be deployed in a real-world application. In this case, it would be loaded on the operative system of an UAV and run along with all the other applications required by the UAV to fly. The block scheme of figure 4.3 represents the proposed algorithm in this situation.

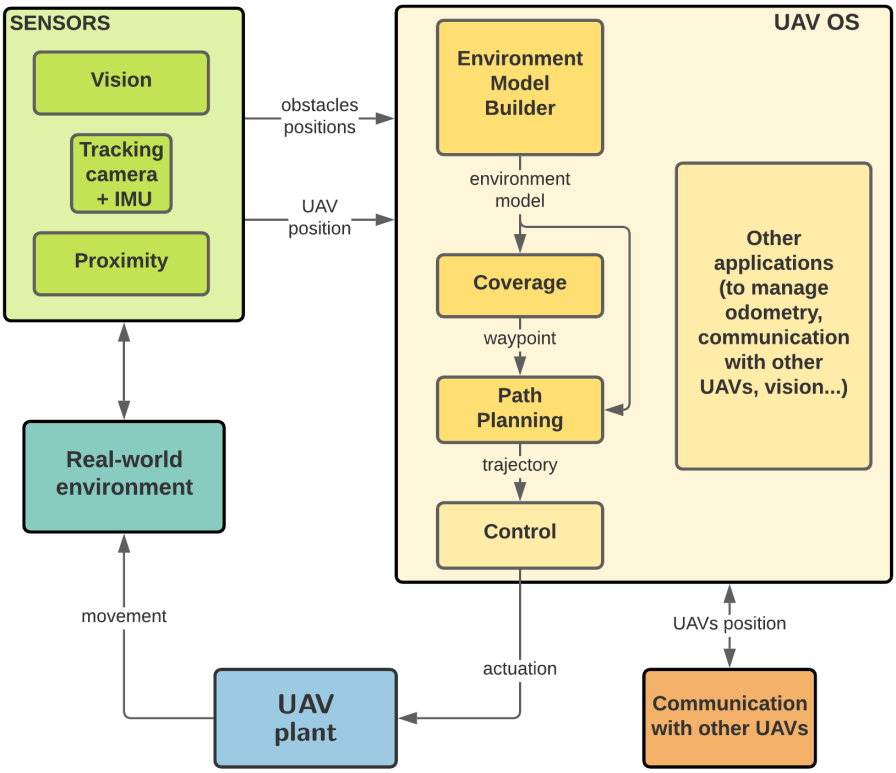


**Figure 4.3:** Block scheme of the algorithm structure deployed for a real-world application. The light-yellow block represents the UAV OS and contains the proposed algorithm. The UAV and the blocks representing the real-world, have to be treated as black boxes, and can be interacted with only through the actuation of the UAVs motors or through sensors.

To obtain the algorithm in the configuration shown in figure 4.3 it is necessary first of all to integrate the proposed algorithm in the UAV operative system. The algorithm parts, i.e., the dark yellow blocks of figure 4.2, are extracted from the training environment. For what

regards the coverage and path planning agent, this means to extract the trained agents models, whereas for the environment model builder this means to collect all the functions necessary for the environment model management and export them. The algorithm may need some minor adaptations to work in the UAV OS, in particular for what regards the input/output data structure compatibility (as it is probable that some of the data used or computed by the algorithm need to be shared with others applications). The block scheme of figure 4.3 represents all the external parts needed by the algorithm. In particular, sensor and communication data are necessary for the environment model to be built. UAV dynamics and the block representing the "real-world environment" have to be treated as black boxes, as there is no way to interact with them except from their input (i.e., the actuation of the UAV motors to modify its dynamical status) and their output (i.e., the environment status which can be observed through the sensors). It is worth noting that an additional block, the *control algorithm*, has been put in evidence in this representation along with the proposed model. The control algorithm is the one which translates the trajectory computed by the path planning agent in actuation signals for the motors, and in the end allows the UAV to move in the desired way. The control algorithm completes the workflow that goes from the sensor data to the actuation of the motors (the only reason for which the control algorithm has not been shown in the pipeline of figure 4.1 is that it has not been designed during the development of this thesis, since many effective control algorithms for UAVs already exist).

## 4.3 Environment model

First of all, it is necessary to explain how the environment is modelled inside the algorithm. The Environment Model Builder (EMB) has the task to translate all the sensor measurements into a numerical model that can be used by the other parts of the algorithm to navigate the environment. The model created is based on a numerical implementation of the Artificial Potential Field (or APF - see chapter 3 for the theoretical discussion of this method). The 2D environment is represented as a "grid world". This means that the map is discretised in cells whose dimension depends on a *resolution* parameter $r$. A graphical representation of the grid world used to create the environment model is shown in figure 4.4.

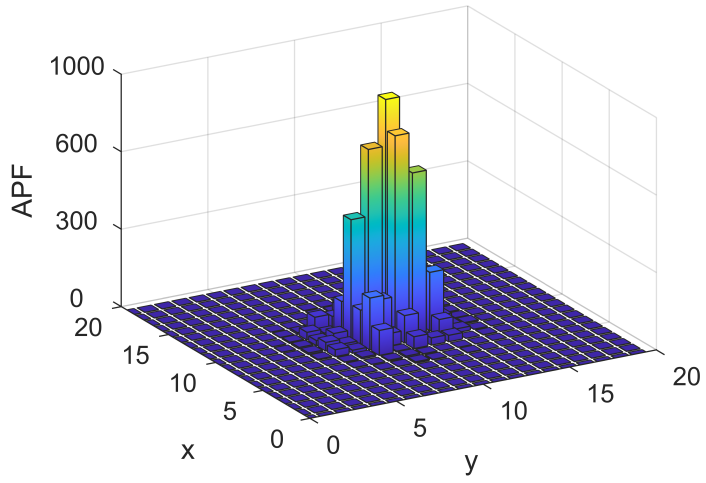


**Figure 4.4:** "Grid world" representation of the environment model. Along the $z$ axis the potential value held by each cell is represented. Each cell has size $r \times r$.

The model is initialised with all the cells holding a potential value of $0$. Then, an APF value is associated with each of the cells on the basis of the information collected about the environment. The basic idea is to associate to each cell a potential value that is higher near obstacles and lower near the goal. The potential can be seen as a representation of the "risk level" associated to each cell. Obstacles are associated to a high risk level, whereas regions distant from them are associated a low level of risk. The potential field can be exploited to reach the goal by moving toward the direction of minimum potential. For each cell, the potential field is computed as the sum of three contributions:

$$U(x) = U_{attr}(x) + U_{rep}(x) + U_{exp}(x) \tag{4.1}$$

The terms of equation (4.1) correspond to different elements in the environment. They are computed and managed separately and summed to obtain the final potential value. The different "layers" of which the APF environment model is composed are shown in figure 4.5. Each layer corresponds to one term of equation (4.1).

$U_{attr}(x)$ is an attractive term. It is defined as a cone having vertex in the goal position:

$$U_{attr}(x) = k_a||x - x_g|| \tag{4.2}$$

The attractive layer $U_{attr}$ is updated every time the coverage planning algorithm computes a new waypoint for the UAV. This is the only layer which is different for all the UAVs.

The second contribution, $U_{rep}(x)$, corresponds to the *repulsive layer* or *obstacle layer*. In this layer, a repulsive shape is added each time an obstacle is detected. The way obstacles are added to this layer will be explained in more detail below.

The third term, $U_{exp}(x)$, is called *experience layer*. It is used to store temporary information about the environment or modify it temporarily (e.g. for the management of local minima). In the image, local minima are evidenced in red and a temporary repulsive field associated to them is represented.

The sum of the contributions of the layers above returns the final APF environment model. Each point in space has a potential value corresponding to the sum of the three contributions. The main advantage of using a layered structure to compute the APF model is that in this way it is very easy and quick to remove/modify the value of one of the layers. For example, when the goal position changes, it is sufficient to update just the value of the attractive layer (by subtracting it, re-computing its value and summing it back in the model). The same

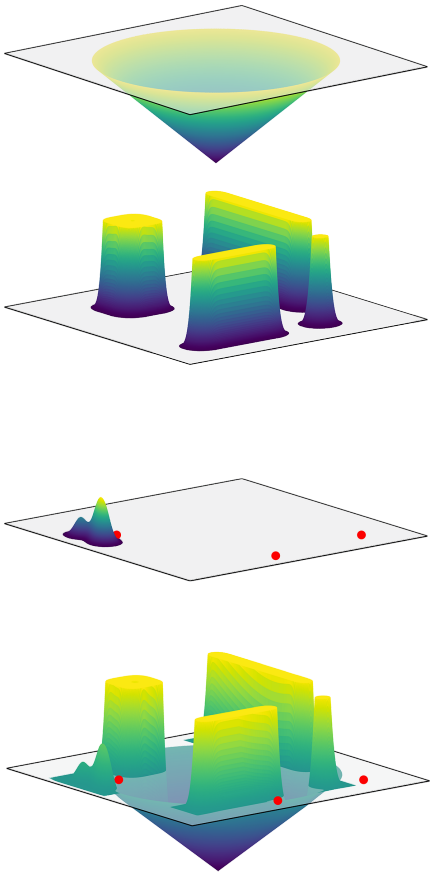

**Figure 4.5:** Layers forming the APF environment model used in the algorithm. The bottom one is the final environment model, corresponding to the sum of all the ones above.

holds for the update of obstacles. All the layers are continuously changing to incorporate new information about the environment in real time. For the attractive layer, the update happens every time the coverage planning algorithm computes a new waypoint for the UAV.

The construction of the obstacle layer proceeds iteratively. This layer is never deleted (the UAV needs to keep track of all the obstacles positions), and information are added to it as the exploration proceeds. As the other layers, the obstacle layer is initialised as an empty matrix and filled over time. Every time the position of a new obstacle is detected through sensors or is communicated by some other UAV, the obstacle layer is locally updated around the obstacle position. The vision function (discussed in more detail in section 4.3.1 below) passes to the EMB a list of obstacle points. Each cell in the model where at least one obstacle point is detected is associated to an obstacle. The obstacle location is associated to a high potential value, and the area around it to a decreasing one. The potential distribution around each obstacle cell is defined as a gaussian distribution (as introduced in the theoretical discussion of APF in chapter 3). The gaussian shape is summed with the centre in the obstacle cell in order to extend some of its influence to the neighbouring cells and keep the UAV at a safe distance from the actual obstacle. The gaussian potential distribution used to represent obstacles is shown in figure 4.6. After a certain distance (called *safe distance* the influence of obstacles drops to 0. In fact, obstacles have only a local effect: after a certain distance their presence can be neglected.

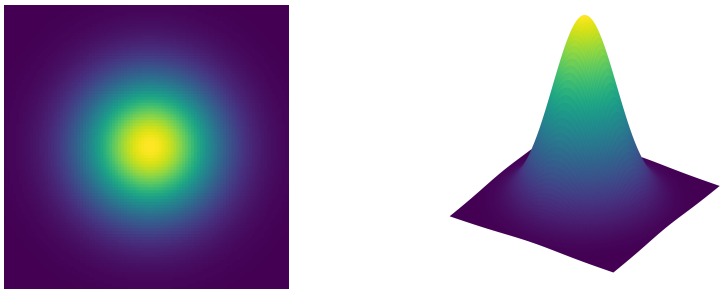

**Figure 4.6:** Local potential distribution used to represent obstacles in the environment model. On the left, the distribution is represented as a 2D matrix. On the right, a 3D view is displayed (in both representation, yellow represents a high potential value whereas blue indicates a low potential.

An additional task of the obstacle layer is that of representing other UAVs as obstacles to avoid collisions between them. UAVs need to be represented as mobile obstacles, since their position changes over time. To do so, they are managed in a separated obstacle layer which, at every time step, is reset and computed again. The mobile obstacles layer is computed by placing a gaussian repulsive matrix in correspondence with each of the other UAVs. The matrix has a shape similar to the one of figure 4.6 but with a larger radius, to keep a safer distance since UAVs are in movement and this must be taken into account in the trajectory planning.

Finally, it is necessary to briefly discuss the role of the experience layer. The experience layer was introduced to have the possibility to modify locally and temporarily the shape of the APF model without having to modify the goal layer or the obstacle layer. For example, an idea that was explored to manage local minima was to detect them using a dedicated routine and "fill" them by creating a proper potential peak in the experience layer. Once the goal was

changed (and therefore also the position of local minima changed) this modification could be erased by resetting the experience layer, leaving no trace in the environment model. In the final implementation of the algorithm the experience layer was used very little, but it was considered worth describing it in the creation of the environment model. In fact, it has a large range of possible uses and in the future it could be exploited to implement some interesting modifications in the APF model management (one among all, the management of mobile obstacles different from other UAVs could be performed inside the experience layer).

Having explained the role of each layer, the complete working routine of the Environment Model Builder is quite straightforward. Each time the environment model builder routine is called it performs the operations listed in algorithm 4.1 to update the environment model.

**Algorithm 4.1** Environment Model Builder routine

```
1:  procedure ENVIRONMENT MODEL BUILDER
2:      # update goal position in the model
3:      if waypoint position x_g has changed then
4:          subtract U_attr from the model
5:          recompute U'_attr for the new x_g
6:          sum U'_attr to the model
7:      end if
8:      # add fixed obstacles potential
9:      if new obstacles are detected or communicated by other UAVs then
10:         check if obstacles are already known
11:         for each new obstacle o do
12:             sum the local potential matrix in the obstacle position x_o
13:         end for
14:     end if
15:     # update other UAVs potential
16:     subtract the mobile obstacle layer from the model
17:     recompute the mobile obstacle layer using the updated UAVs positions
18:     sum the mobile obstacle layer back into the model
19: end procedure
```

### 4.3.1 Vision

It is worth opening a brief parenthesis about the simulation of the sensor vision and the obstacle detection process. It is assumed that the UAVs of the fleet are equipped with a depth camera capable of generating a cloud point representing the location of obstacle points in the vision range. The cloud point is elaborated to integrate the obstacle positions in the environment model, as illustrated above. The obstacle detection process, along with their integration in the environment model, is schematically illustrated in figure 4.7. The image shows well the way in which obstacles are discretized in the Environment Model Builder. The small wall in the leftmost figure is divided in three point-like obstacles, each one located in the centre of one cell of the model. Each of those cells is considered an obstacle, and a local repulsive matrix is summed on top of all of them.

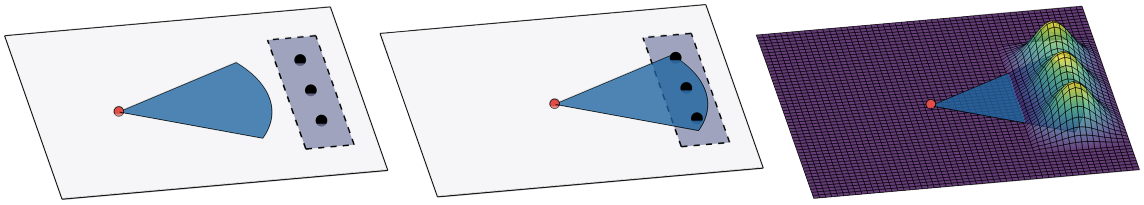

**Figure 4.7:** Obstacle detection and integration in the the environment model builder. In the leftmost image, the obstacles (black dots) are out of vision range. In the image in the middle obstacles are detected by the depth camera. In the rightmost image the potential is raised around the newly discovered obstacles by summing the local repulsive matrix of figure 4.6 in their position.

As already mentioned, the vision model used is that of a depth camera. The camera shoots a number of infrared rays in order to generate a point cloud where each point correspond to an obstacle location. The vision function implemented in the simulation environment imitates this behaviour. It represents with a fair degree of accuracy the working of a real-world dedicated depth camera. An example of its working is shown in figure 4.8. The situation displayed in the figure corresponds to the obstacle detection phase, which is the one represented in the middle frame of figure 4.7.

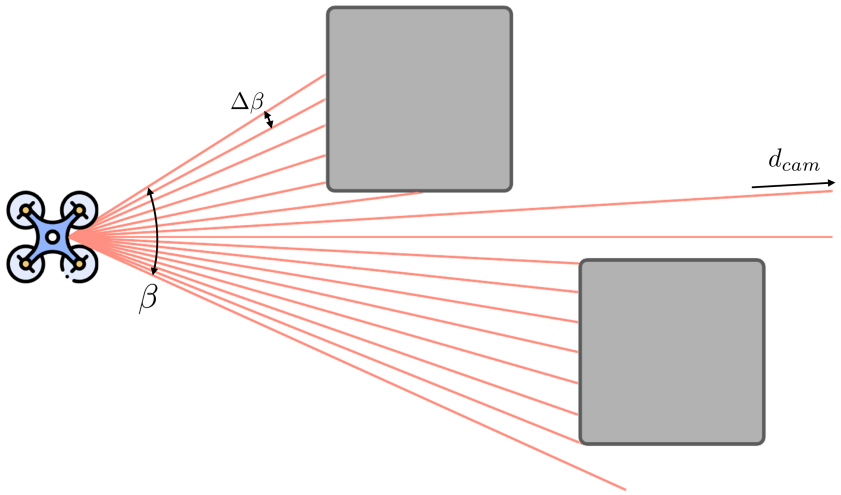


**Figure 4.8:** Implementation of the vision function using a simulated depth camera.

The simulated depth camera has a set of parameters that can be tuned to better represent the behaviour of a real camera. Those parameters are the detection angle $\beta$, the maximum detection distance $d_{cam}$ and the angular resolution $\Delta\beta$. The values of the three parameters are collected in table 4.1.

**Table 4.1:** Parameters of the simulated depth camera.

| Parameter | Variable | Value |
|---|---|---|
| detection angle | $\beta$ | 50° |
| angular resolution | $\Delta\beta$ | 0.25° |
| max. detection distance | $d_{cam}$ | 3m |

### 4.3.2 Obstacle shape prediction

A problem encountered while implementing the vision function to be used in the exploration simulations was that only the surface points of 2D obstacles could be detected by the depth camera. This means that, in the case of a square obstacle, only its edges could be observed and integrated in the environment model. The internal points could not be observed and therefore a hole would remain in the exploration map (the exploration map is an environment representation in which each cell is associated to a boolean value determining if it has already been observed by at least one of the UAVs or not. In the first case, the cell is considered *explored*. In the second, it is not). This would create a problem in the development of the coverage planning agent, since these holes would be seen by it as unexplored regions, even if there was no way to reach or observe them. A possible solution could have been to teach the coverage agent to ignore those regions, but the implementation of this behaviour is not easy and so this approach was abandoned. To solve this issue, instead, an *obstacle shape prediction* function has been implemented inside the Environment Model Builder. This function allows to predict obstacle shapes from the available data about the edges and fill the inside points in the environment model and in the exploration map. The function has been developed by using some commands from the OpenCV library. To simplify the implementation of this function, only 1-dimensional (linear) and rectangular obstacles have been considered in the maps used for the simulations. The obstacle-prediction function could be extended to be able to manage more complex obstacle geometries, like "C", "L" or, "T"-shaped obstacles. However, this modification would have been time-consuming in the development of the thesis and therefore was postponed to future developments. The simplification in the possible obstacle geometries allowed to push the shape prediction further. In fact, once two sides of a rectangle have been observed, the location of the other two can be inferred and so the obstacle can be filled even starting from the partial information about just two of its sides. An example of the working of the shape prediction algorithm can be observed in figure 4.9.

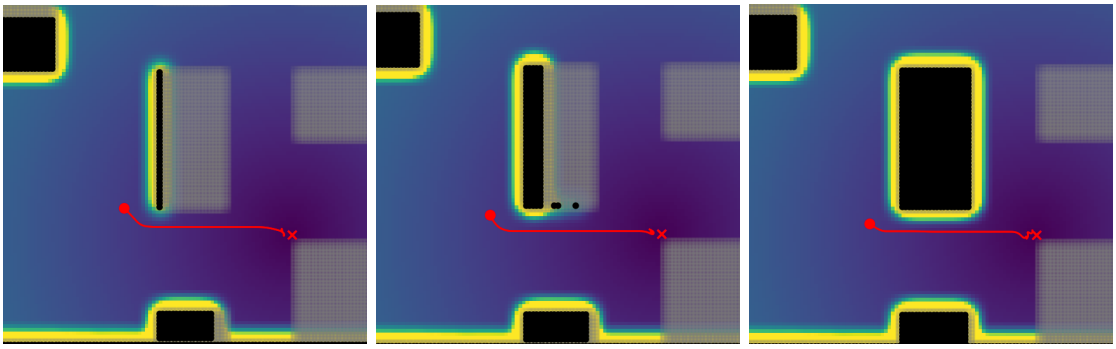

**Figure 4.9:** Example of the working of the *obstacle shape prediction algorithm*. It is worth noting how the obstacle is progressively filled on the base of the available information. In the central image, only a portion of the bottom side of the obstacle is detected; therefore, the obstacle is represented as a smaller one. Once the whole bottom side of the obstacle is discovered (third frame), the obstacle is completely filled.

## 4.4 Path planning

In this section, the path planning algorithm is presented. First, a path planning algorithm based on the original APF is illustrated (section 4.4.1). This will be used mostly as reference. Then, the proposed RL path planning agent is presented (section 4.4.2). The design of the path planning agent is one of the core parts of this chapter. Some relevant aspects of the RL agent, which are the final trajectory computation strategy and the local minima management, are discussed in detail in sections 4.4.3 and 4.4.4.

### 4.4.1 APF path planning

A version of the APF path planning algorithm has been implemented to test the environment models and as reference to verify the trajectories generated by the RL agent. The implementation of this path planning method is quite straightforward and follows the original APF algorithm [7], so it will not be discussed in detail. An example of trajectory between two points obtained with this algorithm is shown in figure 4.10.

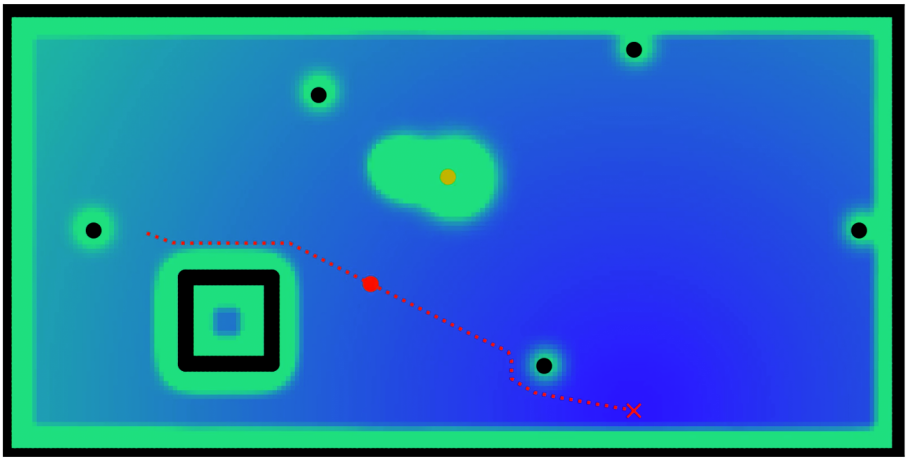

**Figure 4.10:** Example of a trajectory obtained with the numerical implementation of the APF algorithm. The black elements represent obstacles. The red dot is the UAV whose point of view is being shown, whereas the orange one is another UAV of the fleet. The blue/green background represents the potential field.

As shown in the figure, the APF finds an effective trajectory between the two points avoiding all the obstacles. However, the trajectory is not smooth, and the algorithm is not able to "anticipate" the presence of obstacles, but instead it creates sharp last-moment turns (this is particularly evident near the circular obstacles at the bottom of the image). Moreover, this algorithm is affected by the presence of local minima, which can lead the APF algorithm to not being able to find a trajectory reaching the goal. In fact, as introduced in the theoretical discussion of APF done in section 3.1, local minima are common in APF models and are critical since they "trap" the agent. The problem is that the algorithm computes the field gradient to decide in which direction to move, i.e., it moves in direction $v = -\nabla U(x)$. However, in local minima the gradient is null and so the agent is not able to move further.

### 4.4.2 Reinforcement Learning path planning

The path planning agent is the first of the two RL agents trained to build the exploration algorithm. It has the task of choosing the optimal trajectory to follow to reach a given target point. The path planning agent is designed as a Neural Network which receives in input a part of the APF field model and returns as output the trajectory to follow. The reason for the use of a Neural Network to build the RL agents is explained in chapter 3, where the theoretical background of NN-based RL agents is discussed. The NN designed to build the path planning agent is shown in figure 4.11. All its main parts and features are discussed in detail below.

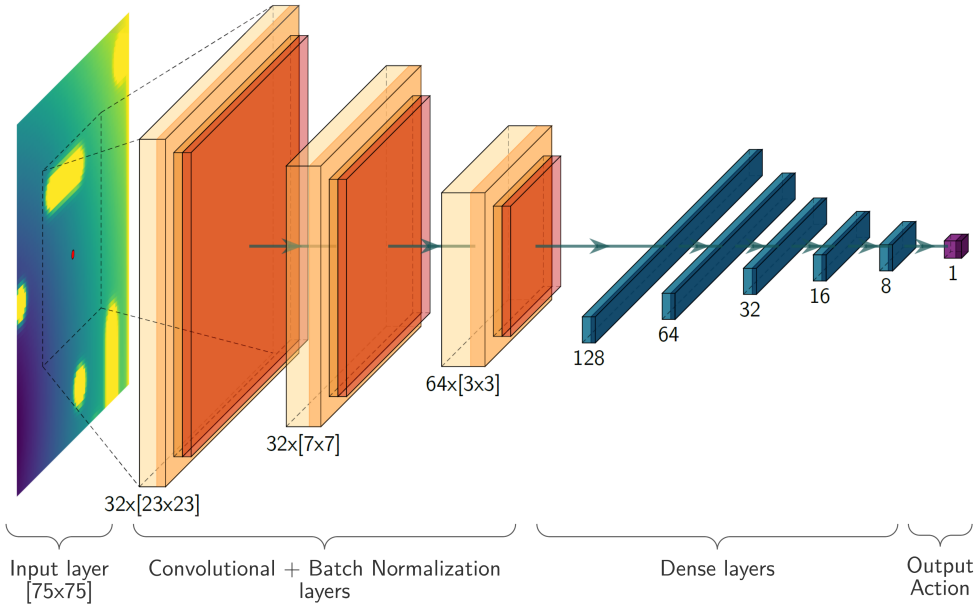


**Figure 4.11:** Graphical representation of the path planning NN. The input layer is on the left, while the output one is on the right. The yellow layers represent the convolutional layers of the network.

First of all, it is necessary to discuss the NN input. It was decided to feed the agent directly a part of the matrix representing the environment model. This allows to supply, in a very compact way, all the necessary information to perform the path planning and obstacle avoidance to the RL agent. Since each cell of the matrix corresponds to an area of space in the environment, it is possible to modify the agent "sense radius" by changing the size of the input matrix. A good sense radius was considered to be between 2 and 4 meters in each direction. For a generic resolution of 0.1m, this corresponds to an input observation having a size between [41x41] and [81x81]. Each time the path planning agent is executed, a portion of the environment model having this size and centred in the UAV position is extracted and fed to the NN. A couple of pre-processing operations are performed on the input of the NN. First of all, an offset is added to the input matrix in order to make the potential of the central cell (corresponding to the current agent position) equal to 0. After this, the status values are mapped from the interval [-1000, 1000] to the interval [-1, 1]. This is done to compress the input dynamics and keep the overall value of the weights and biases lower. An example of three different inputs of the network is shown in figure 4.12.

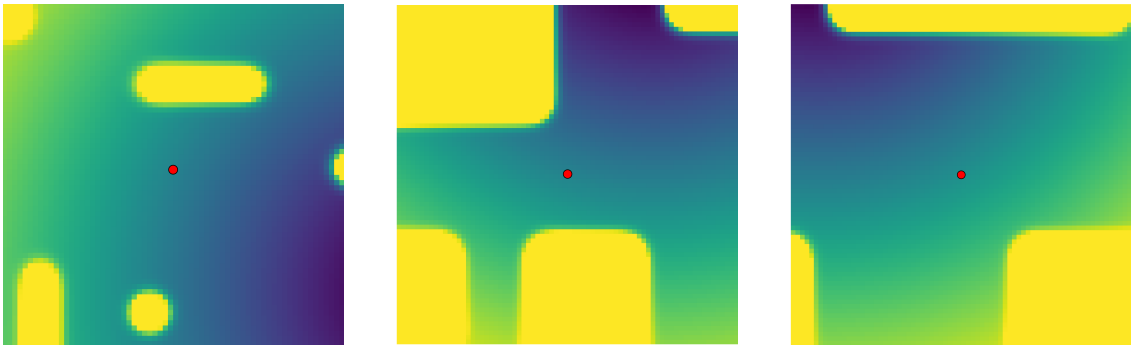

**Figure 4.12:** Example inputs of the path planning agent. The APF portion represented in each image has size [75x75]. Yellow regions correspond to a high value (near $+1$) whereas blue one are near -1 (values are normalised in the interval [-1, $+1$]).

For what regards the internal Neural Network structure, a graphical representation of actor $\mu_\theta(s)$ was introduced in figure 4.11. It is worth mentioning that the structure of critic $Q_\phi(s, a)$ is slightly different. In fact, the critic NN has an additional input channel through which it receives the value of action $a$ in order to be able to evaluate it. Apart from this, the structure of actor and critic is identical. The initial part of the NN is composed of a series of convolutional layers; those layers are particularly adapt to work with image-like inputs. Convolutional layers can perform complex operations on images, extracting relevant features, using only a reduced number of parameters. This is useful in the detection of obstacles from the input state of the network. Even if the APF model is not an image, it can be treated like one to exploit the power of convolutional layers. After the convolutional layers, a series of dense layers (or *fully connected* layers) is placed. This section contains a large number of connections (and therefore of learnable parameters). Here, the logical operations to perform on the "abstract" features extracted by the convolutional layers are learned.

To conclude the actor network analysis it is necessary to discuss the output layer. Here lies another difference between the actor NN and critic NN structure. In fact, the actor must return a value between 0 and 1 representing an angle in the interval $[0, 2\pi]$. On the contrary, the critic must return any real number to represent the goodness (or badness) of action $a$ chosen in state $s$. The output of the actor network is a scalar value between 0 and 1, obtained through a single node performing a sigmoid operation, i.e.:

$$a^{(o)} = \sigma\left(\boldsymbol{W}^{(o)}\boldsymbol{a}^{(o-1)} + \boldsymbol{b}^{(o)}\right) \tag{4.3}$$

where the activation function is a sigmoid:

$$\sigma(x) = \frac{e^x}{1 + e^x} \tag{4.4}$$

The value obtained is then re-scaled between 0 and $2\pi$ and interpreted as an angle in the plane. This angle represents the optimal direction of movement to follow from state $s$ to reach the goal location $x_g$. The direction obtained can be used in two ways. The first is to follow it until the next call of the path planning agent, which will compute a new motion direction. The other, more efficient, is to call the path planning algorithm multiple times in succession in order to compute a series of points, interpolate them and use the resulting trajectory instead of a simple direction to follow. This second strategy was the one used in the thesis. The routine used to obtain the trajectory is presented in section 4.4.3.

### 4.4.3 Trajectory computation

As introduced above, a routine has been designed to translate the output of the path planning agent into a trajectory suitable to be followed by a flight control algorithm. The trajectory computation is obtained by calling iteratively the path planning agent to produce a series of points. A continuous path is then obtained by fitting the points. The fitting operation is performed using a b-spline fitting algorithm. This kind of algorithm guarantees the continuity of the obtained function up to the second derivative, making it suitable to work also with velocity/acceleration constraints if wanted. The complete path planning routine, obtained by using the fitting algorithm in addition to the RL agent, is the one presented in algorithm 4.2. A graphical example of the working of the path planning routine is displayed in figure 4.13.

**Algorithm 4.2** Path planning routine

1: compute the starting state $s_0$ in the current position $x_0$
2: **for** i $= 1...$n **do**
3: compute the motion direction as $\psi = 2\pi \cdot \mu(s_{i-1})$
4: move from position $x_{i-1}$ of a distance $\delta$ in direction $\psi$ to obtain position $x_i$
5: compute the new state $s_i$ in position $x_i$
6: **end for**
7: interpolate the $n$ points $(x_0, x_1 ... x_n)$
8: pass the trajectory to the controller to follow it

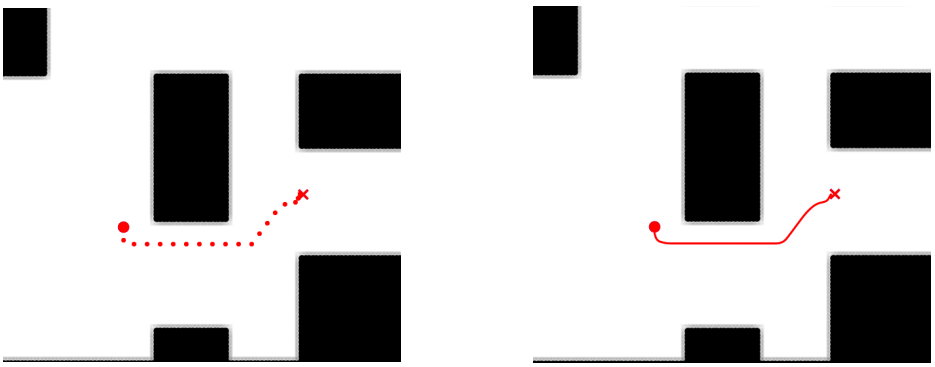

**Figure 4.13:** Trajectory fitting example. The points in the left image are obtained through the path planning agent, and are then fitted to obtain the trajectory on the right.

One issue in the path planning algorithm is that the spacing between the points to interpolate is always constant and equal to the parameter $\delta$. A nice improvement to the routine of algorithm 4.2 would be to add a second output to the RL path planning agent, corresponding to the optimal value of $\delta$ in each position. This second value could be viewed as the *optimal speed* the UAV should have in position $x_i$ and would translate in the possibility to better optimise the trajectory from a dynamical standpoint. The main reason why this modification to the path planning agent was not implemented is that it is not easy to write a good reward function to use in the training process to have the agent learn to predict the optimal velocity to follow. Moreover, constraining the agent learning to a specific dynamic model would significantly lower its generality, since it would work efficiently only on vehicles having the same dynamical properties of the model used during training.

The only exception to the use of algorithm 4.2 is represented by some unwanted situations in which the UAV finds itself closer to an obstacle than a given "safe distance". This kind of situation can be generated by a bad interpolation (e.g. when the interpolating curve "cuts" a turn and get too close to the corner of an obstacle), but more frequently it is generated by a bad prediction by the RL agent. This is one big limit of the use of RL algorithms, and even if the policy should be robust with respect to unexpected input statuses, sometimes the NN prediction fails and the result is a bad trajectory.[13] To tackle this issue a good solution is to pair the RL agent with a deterministic algorithm that comes into operation only when a failure on the part of the main algorithm is detected. In the proposed algorithm, the numeric implementation of the APF algorithm described in section 4.4.1 was used as an emergency path planning algorithm. This measure significantly reduced the risk of a critical malfunctioning of the RL path planning agent.

### 4.4.4 Management of local minima

As introduced in the APF discussion in chapter 3, one of the main problems of the APF algorithm is represented by local minima. Local minima are points in the potential field where, due to the obstacle configuration and the goal location, the potential value is minimum with respect to the surrounding area. The original implementation struggles to manage the presence of local minima, and dedicated pieces of the algorithm are usually added to overcome this limitation. In the proposed algorithm, local minima are present in the APF model and need to be taken into account. In fact, the main strategy that the RL agent is expected to use to find its way to the goal is to go toward the lowest possible potential value. This would mean that the RL agent would get trapped by local minima, like the deterministic APF path planning algorithm. However, with a proper design of the training process, it is possible for the agent to learn to manage local minima effectively, recognising their presence and avoiding them. For this reason, the training environments used for the path planning agent training have been designed to contain a sufficient number of local minima, so that the agent - during the training process - would be able to face this scenario many times and learn how to tackle it. In addition, the reward function with which the agent action choices are prised or punished has also been designed to encourage the agent to find effective ways to avoid local minima. The design of the training maps and reward functions are discussed in chapter 5. The results in terms of local minima avoidance are presented in chapter 6.

One alternative approach that has been tested - which represents a quite interesting usage of the APF environment model - is to exploit the *experience layer* of the environment model to avoid local minima. The idea is to locally (and temporarily) modify the APF shape in order to allow the agent to avoid the minima by "hiding" them. Once a local minimum is detected near the agent, the experience layer is modified to "fill" the minimum so that the agent is not attracted by it. The modification of the experience layer lasts only while the agent is in the same area, and is deleted later. Some attempts have been made to implement this strategy in the proposed algorithm. However, results were poor since this technique requires some testing and refinement to be well-tuned. Therefore, for the moment this solution has been abandoned.

[13] This happens easily in cases - like the one of this thesis - where the training and testing time are limited.

## 4.5 Coverage

In the exploration algorithm, the task of computing the target goal that each UAV has to reach to keep exploring the environment is taken in charge by the coverage algorithm. In this section, the coverage agent design process is described. First of all, the high-level decisions that have been made prior to the design process are discussed (section 4.5.1). After this, the RL agent design is presented (section 4.5.2). Finally, an explicitly-programmed coverage algorithm, used to address the limitations of the RL agent, is presented (section 4.5.3).

### 4.5.1 Preliminary analysis

Some preliminary decisions have been made before starting the coverage agent design process. The first regards the hierarchical structure of the algorithm. The proposed algorithm is intended to produce a *swarming behaviour*, i.e., to obtain a coordinated collective behaviour by a fleet of independent units. As most of the swarming algorithms described in section 2.2 of chapter 2, the exploration algorithm designed in this thesis is leaderless, meaning that there is not a "leader" UAV in charge of coordinating the exploration task. On the contrary, each UAV is in charge of managing its own exploration process, but to do so it takes into account all the information supplied by the other members of the fleet. By sharing relevant information with each other the members of the fleet are able to coordinate the exploration task. This approach leads to the implementation of a *coordinated* behaviour, but not necessarily a *cooperative* one.[14] The fact that there is not a leader means that the same coverage algorithm runs on each UAV of the fleet. The input state must contain sufficient information for the agent to understand its own position as well as the position of the other UAVs. The agent must be able to effectively use the information contained in the input state to produce different temporary goals for each UAV. This approach adds a considerable difficult behaviour for the coverage agent to learn; in fact, the agent does not know where the other UAVs will place their waypoints, so it will have to be able to predict their placements and position its own waypoint accordingly.[15]

A second decision that has been made is to keep the coverage algorithm as scalable as possible. This means that the algorithm has to work for fleets containing any number of UAVs (e.g., between 2 and 10 members). This requirement translates into the fact that there cannot be an agent input specifically dedicated to each UAV. On the contrary, the input must be able to contain all the necessary information about any number of UAVs. The best way to manage this requirement is to use an image input in which the UAV positions are represented. This requirement has no effect on the output shape, since the agent has to deliver only two scalar values, corresponding to the $x$ and $y$ coordinates of the UAV waypoint. In fact, since each UAV computes its own waypoint, the NN only needs to have 2 outputs.

---

[14] The difference is subtle and not necessarily a relevant one, but it is worth pointing it out. It just means that the UAVs are *selfish* in the execution of the exploration task, i.e. each of them tries to explore as much of the environment as it can. In this sense, the fact that the UAVs are *coordinated* just means that they try not to overlap their exploration paths and they avoid collisions with each other by keeping at a distance. However, they give no importance to favouring other UAVs exploration.

[15] It is worth noting that the other UAVs will place their waypoints using the same exact agent, just with a different input.

### 4.5.2 RL coverage

The design process of the coverage agent is similar to the one performed for the path planning one. First, the input of the Neural Network (i.e. the input *state*) is defined, along with its properties. Then, the output of the NN is decided. Finally, the NN itself is designed with a suitable architecture (i.e., type and properties of each layer). The design process of the coverage agent has been performed keeping in mind the preliminary analysis discussed in the previous section. For the coverage problem, the input needs to contain more information than the one used by the path planning agent. In this case, the algorithm has to know data about:

- the UAV current position;
- other UAVs position;
- environment dimension and shape;
- obstacles position and shape;
- which part of the environment has already been explored and which not.

The state designed to supply the NN all the information above is divided in two parts, each one entering the agent through a dedicated input layer. The two inputs are first processed by some separate NN sections and then mixed together to be elaborated by a series of dense layer. The first input is composed of two neurons and contains the agent current position represented by two values in the interval [0, 1]. The two numbers represent the $x$ and $y$ coordinates of the UAV, normalised with respect to the total map dimension. The second input, which contains the other four information of the list above, is a grey-scale image (i.e. a matrix $n \times m \times 1$) built using the following criteria:

- the image is initialised as a $n \times m \times 1$ matrix of zeros;
- obstacle cells are represented as ones;
- the already-explored points are assigned a constant value $\lambda \in (0, 1)$
- a "repulsive" region is placed in correspondence with every other UAV to represent their "area of influence", i.e., the region of space they are more likely to explore soon. This repulsive region is obtained by adding a 2D gaussian shape on top of each UAV position. The size of this shape is a hyperparameter and has to be tuned.

This input state has some similarities with the environment model used by the path planning agent (e.g., the fact that the environment is represented as a "grid world"), but contains different information. The input state of the coverage agent is managed by the environment model builder function, which updates it along with the APF environment model. Some examples of the coverage input state are shown in figure 4.14. As already explained, two scalar values representing the normalised position of the agent in the environment have to be added to the image to get the complete input state received by the NN. The image input shown in figure 4.14 could be viewed as a repulsive field for the placement of the waypoints: they should be placed in a region having the lower possible value, i.e., in points which are unexplored and far from obstacles and other UAVs. The output of the coverage agent NN is designed as a two-node layer where each node holds a sigmoid activation function. The values obtained from the two nodes are interpreted as a couple of normalised $x$ and $y$ coordinates, representing the location of the waypoint to be reached to optimally continue the exploration. The two output values are multiplied for the corresponding environment dimension in order to translate the point into an actual location inside the map.

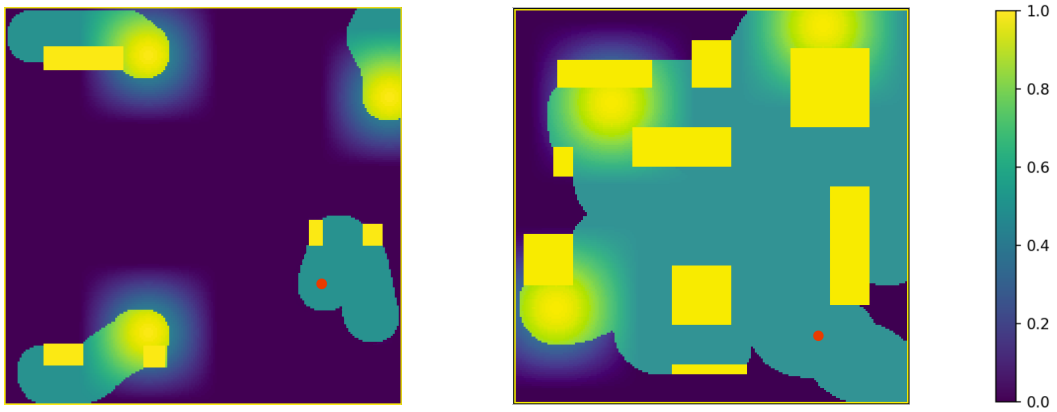


**Figure 4.14:** Examples of the coverage agent input state. Images represent the point of view (i.e., the input state) of the UAV represented by the red dot.

The structure of the NN designed to perform the waypoint computation operation (i.e. the *actor* NN of the coverage agent) is shown in figure 4.15.

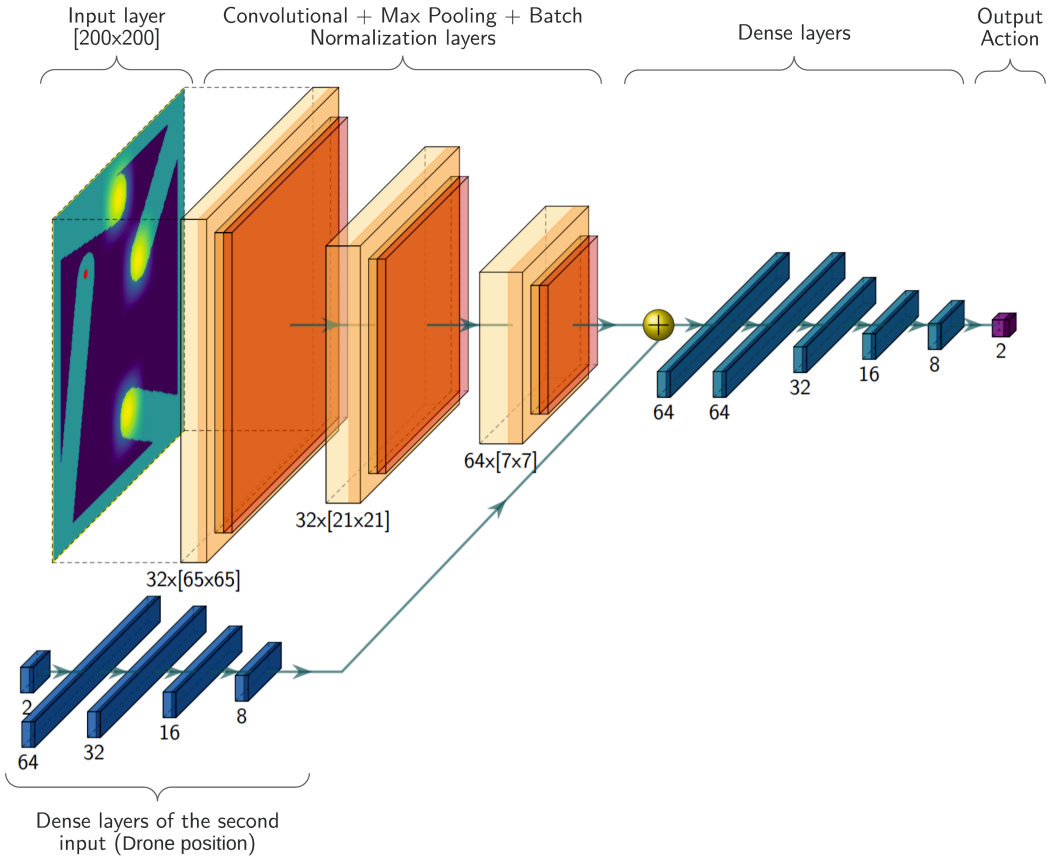


**Figure 4.15:** Graphical representation of the coverage NN. The input layers are on the left, whereas the output one is on the right.

It is worth noticing that, as discussed in the preliminary analysis of the coverage problem in section 4.5.1, the input and output of the NN are independent of the number of UAVs in the fleet. The only thing that changes with the fleet size is the number of repulsive Gaussian shapes placed in the input image. The dimension of the image itself, however, remains constant. It is also worth mentioning that, as in the case of the path planning agent representation, the NN displayed is the *actor* one, i.e., the one that learns to estimate the best action starting from the input state. The *critic* NN is built with some differences. First of all, a third input channel is added. This channel takes in input a couple of values, corresponding to the $[x, y]$ action vector computed by the actor. The critic has the task to evaluate the goodness of this action choice with respect to the corresponding input status. The action input is elaborated by some dedicated dense layers before being merged with the other inputs in proximity of the sum node (before the series of dense layers, which are the ones coloured in blue in figure 4.15). In addition, the critic network has only one output node with no activation function associated to it. The critic output is, in fact, a single real value corresponding to the evaluation of the action choice. This output is the one that the actor aims to maximise.

For what regards the actor NN structure, as can be observed in figure 4.15 the inputs are elaborated in two different ways. The drone position input is processed by a short series of dense layers before getting merged to the other input. The exploration state input, instead, is elaborated by a series of convolutional and pooling layers that aim to extract relevant features from the image. A set of *filters* is used to detect and extract those features and pass them over to the dense layers. After the two input channels are merged, the complete set of features is processed by a series of dense layers containing a large number of learnable parameters. The dense layers, in the end, return the two output values.

### 4.5.3 Explicitly-programmed coverage algorithms

An explicitly-coded coverage algorithm has been implemented to be used as reference during the agent performance evaluation. The explicitly-programmed coverage algorithm is based on the k-means clustering algorithm. This algorithm aims to divide the unexplored part of the environment in $k$ clusters defined so that each point belongs to the partition that has the nearest mean to that point. This results in a partition of the unexplored space into Voronoi cells. More details about the working of the k-means clustering and Voronoi partition of the environment can be found in appendix B. To proceed with the exploration process, the centroid (or mean) of each cluster is assigned to one UAV as temporary objective. This approach guarantees an optimal balance between the fleet spreading (i.e., the distance between UAVs) and the size of the region to explore associated to each UAV. This approach does not necessarily result in an optimal exploration strategy, especially for the fact that the generation of the goals does not take into account the current UAV positions. However, this approach results quite efficient during the last 20% of the environment exploration, where the unexplored areas are only a few and far one from the other. In this instance, the k-mean algorithm can consistently locate the unexplored spots in the environment and generate goals inside them. For this reason the k-mean clustering has been used, in addition to reference and benchmark purposes, to support the RL coverage algorithm during the last part of the exploration. In fact, as will be explained below, the training time has not been sufficient to build a coverage algorithm able to reach 100% of exploration, and therefore the last exploration goals are sometimes placed with this explicit algorithm.

## 4.6 Other parts of the algorithm

It is worth dedicating a few words to some of the parts of the algorithm that were shown in figure 4.3 but were not discussed in detail since they are out of the scope of the thesis. In this section Dynamics, Control and Communications are discussed. These parts of the algorithm have been implemented only partly in the development framework. A better implementation of them is one of the goals for the future development of the proposed algorithm as this would allow to obtain a more realistic simulation of the exploration process.

### 4.6.1 Benchmark UAV

The algorithm described above and all its sections have been developed to be as generic as possible. This means that the algorithm, in theory, can be employed on any kind of UAV (quadrotor, hexacopter, fixed-wing...), with any set of sensors mounted on it. A possible exception is constituted by the necessary presence of a LIDAR camera. A different type of camera could be used to detect the presence and position of obstacles, but this would require a bit of extra work since the Environment Model Builder must receive a list of obstacles coordinates as input. The most relevant difference from one UAV to the other is the dynamic model. Up to now, this model has not been required. However, to optimise the trajectory interpolation, as well as to design a suitable flight controller, the dynamic model of the drone is required. To get the dynamic model it is necessary first to select a *benchmark UAV* to use as reference. The UAV that has been selected is the one developed by the DRAFT team[16] (a student team working within the PIC4SeR research group[17]). The UAV, shown in figure 4.16, is an X8 octacopter (the structure is that of a X4 quadcopter but each arm holds two coaxial propellers) equipped with a set of cameras and sensors that allow it to collect information about the surrounding environment. The technical specifications of the UAV and its sensors are listed in table 4.2.

**Figure 4.16:** The DRAFT team quadcopter that has been used as reference.

[16] https://www.draftpolito.it/

[17] https://pic4ser.polito.it/

**Table 4.2:** DRAFT team quadcopter technical specifications.

| Characteristic | Value |
|---|---|
| Physical dimensions | 500 mm x 500 mm x 350 mm |
| Weight (max weight at take-off) | 3300 g |
| Max payload weight | 500 g |
| Max flight time | 15 minutes |
| Architecture | OctaQuad X8 (8 coaxial propellers) |
| Thrust-to-weight ratio | 2.4 |
| Electronic Power System | LI-PO battery 1200mAh @ 14.8V |
| On-board Computer | NVIDIA®Jetson Xavier NX™ |
| Sensor suite board | Raspberry Pi 4 Model B |
| Flight controller | Pixhawk 2.4.8 |
| Navigation cameras | Intel®Realsense™ D435 + T265 |
| Precision landing camera | Raspberry Pi Camera Module v2 |
| Proximity sensors | Ultrasonic sensors (Adafruit®HC-SR04) |

### 4.6.2 Dynamic model

A dynamic model can be used to take into account the UAV dynamics and optimise the trajectory planning according to it. The integration of the dynamic model in the path planning algorithm is an open point; the easiest place to include it in the trajectory generation process is probably in the phase of trajectory interpolation. A more advanced approach would be to include the dynamic model in the RL agent reward function, as introduced in section 4.4.2. This would allow the path planning agent to optimise the trajectory waypoint position in order to produce a dynamically optimal path. The dynamic model used for the simulations is almost entirely based on chapter 16 of the *Handbook of Unmanned Aerial Vehicles* [4]. The model proposed by Powers, Mellinger and Kumar [4] starts from the definition of a motor model to represent the UAV rotors. The vertical force generated by each rotor is equal to:

$$F_i = k_F \, \omega_i^2 \tag{4.5}$$

Each motor also produces a torque (draft) equal to:

$$M_i = k_M \, \omega_i^2 \tag{4.6}$$

Finally, the internal dynamic of each motor can be represented as a first order differential equation:

$$\dot{\omega_i} = k_m(\hat{\omega}_i - \omega_i) \tag{4.7}$$

where $\hat{\omega}_i$ is the desired speed of each rotor (as computed by the controller) and $\omega_i$ is the instantaneous rotor speed. Equation (4.7), as well as the parameters $k_F$, $k_M$, $k_m$, are obtained from experimental data. The motor model and equations are necessary to build the dynamic equations. Before getting to them, it is necessary to define some reference frames, as shown in figure 4.17. Frame $F$ is a fixed inertial reference frame, while frame $M$ is a mobile reference frame attached to the drone. The origin of the mobile reference frame is pointed by the vector $\vec{r}$, whose origin is in $O_F$. A rotation matrix $[R]_M^F$ can be constructed to translate the UAV attitude between the two reference frames. The shape of the matrix depends on the choice of

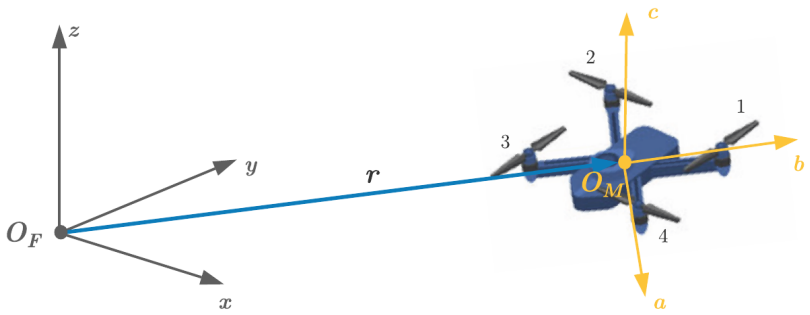


**Figure 4.17:** Reference frames used to build the UAV dynamic model

the angles used to describe the rotation between the two reference frames. A common choice for aerial vehicles is to use the Roll-Pitch-Yaw angles (or Z-X-Y Euler angles), defined here respectively as $\phi$, $\theta$, $\psi$. At this point, it is possible to write a system of Newton-Euler equations to describe the UAV dynamics with respect to the fixed reference frame. The set of Newton equations describing the relationship between the UAV linear accelerations and the rotor forces is:

$$m \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + R_M^F \begin{bmatrix} 0 \\ 0 \\ F_1 + F_2 + F_3 + F_4 \end{bmatrix} \tag{4.8}$$

whereas for what regards the attitude and angular accelerations, the Euler equation describing the UAV rotational dynamics are:

$$\begin{aligned} I \begin{bmatrix} \ddot{\alpha} \\ \ddot{\beta} \\ \ddot{\gamma} \end{bmatrix} &= \begin{bmatrix} l(F_1 - F_3) \\ l(F_2 - F_4) \\ M_1 - M_2 + M_3 - M_4 \end{bmatrix} - \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} \times I \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} \\ &= \begin{bmatrix} l & 0 & -l & 0 \\ 0 & l & 0 & -l \\ \rho & -\rho & \rho & -\rho \end{bmatrix} \begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \end{bmatrix} - \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} \times I \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} \end{aligned} \tag{4.9}$$

where $l$ is the distance between the centre of mass and each propeller and $\rho = \frac{k_M}{k_F}$ is the ratio between lift $F_i$ and drag $M_i$ (as can be derived from the motor model of equations (4.5) and (4.6)). The dynamic equations (4.8) and (4.9) can be used both for simulation purposes and to design a flight controller. It is worth noting that in equation (4.8) the vector of external forces contains only gravity along the direction $z_F$. An interesting consequence of this is that, of the degrees of freedom of the UAV, only 4 are linked to an input force ($\ddot{x}$ and $\ddot{y}$ have no link to the rotor forces). This results in an under-actuated system, and has to be taken into account during the controller design. In [4] there is a complete analysis of the system *differential flatness*, which gets to the results that, despite its under-actuation, the system is able to follow any desired trajectory $\sigma(t)$ in space. The inertial properties used in the dynamic model are the ones of the benchmark UAV described in section 4.6.1. The total mass $m$ is 2.8kg, whereas the inertia tensor $I$ (computed in the centre of mass, i.e. in the origin of the mobile reference frame) is shown in equation (4.10).

$$I = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix} = \begin{bmatrix} 0.0275 & 0.0005 & 0.0002 \\ 0.0005 & 0.0330 & -0.0002 \\ 0.0002 & -0.0002 & 0.0307 \end{bmatrix} \text{kg} \cdot \text{m}^2 \tag{4.10}$$

### 4.6.3 Control

The control algorithm is in charge of computing the actuation forces that allow to follow the desired trajectory. For UAVs, actuation forces are usually the rotor speeds, which are translated into voltages and then fed to the Electronic Speed Controllers (ESCs), which in turn translate the voltages in power values and supply them to the motors. The controller has not been designed while writing the algorithm. In fact, any closed-loop controller can be used for this application. If a new controller is wanted, it is sufficient to use the dynamic model described in the previous section and design a controller starting from it. Otherwise, an already existing flight controller can be used. Many efficient and well-studied UAV flight controllers are available and can be implemented in this algorithm with few modifications. Once a controller is selected or designed, it is sufficient to put it in the control loop of the Control Algorithm section. The control algorithm structure is the one represented in figure 4.18.

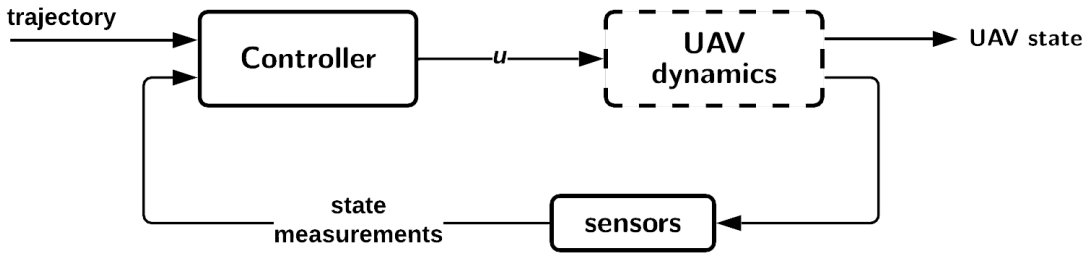


**Figure 4.18:** Block scheme of the control loop algorithm.

The control loop of figure 4.18 takes the trajectory computed by the path planning algorithm as input, and computes the required rotor speeds (collected in the vector $\boldsymbol{u}$). The UAV turns the rotors at the desired speeds and is able to move in space. The output of the UAV dynamics is the UAV state, i.e. the variables $\boldsymbol{x}, \dot{\boldsymbol{x}}, \ddot{\boldsymbol{x}}$, $\boldsymbol{\phi}, \dot{\boldsymbol{\phi}}, \ddot{\boldsymbol{\phi}}$ (respectively: UAV position, velocity, acceleration, attitude, angular speed and angular acceleration). Some of those variables are measured by the UAV sensors and fed back to the controller to close the control loop.

### 4.6.4 Communications

As already explained in section 4.1 at the beginning of this chapter, communications between UAVs have been highly idealised in the development of the algorithm. In fact, it has been assumed that the UAVs are always in communication with each other. Moreover, the bandwidth of the communication systems has been considered to be unlimited, as well as the speed of the communications (i.e., communications are always instantaneous and can transfer any amount of data). These assumptions are clearly highly idealised, and communications should be studied and managed better for a real-world application of the algorithm. Nonetheless, the algorithm has been designed to withstand possible losses of communication, starting from the fact that the fleet is leaderless and so there is no need for the UAVs to keep in communication at every time. In fact, the only crucial information that UAVs need to share is their current position in order to be able to avoid collisions between them. In case of any loss of communication, this issue can be solved by using some "emergency" proximity sensors. The communication of all the other data is not crucial and can be delayed in time until a communication channel is available again. In this sense, the only drawback of a loss of communication is that of a less-optimised exploration process.

# 5. Agent training

In this chapter, the RL agents training process is explained in detail. The first part of the chapter regards the generation of the training data (section 5.1). After this, the path planning agent training is illustrated, starting from the training routine up to the training results (section 5.2). An analogous discussion is made for the coverage agent in the last part of the chapter (section 5.3).

## 5.1 Training set & validation set

A set of maps has been generated to perform training operations. After performing some iterations to find an efficient way to represent the environment and its obstacles, a training set of 200 maps has been generated. A separated set of maps has been created to be used as validation data. Three of the validation maps will be used to show most of the results. The parameters used to generate the training and validation maps are shown in table 5.1. The only difference between training and validation maps is that the latter has bigger size.

**Table 5.1:** Parameters used for the generation of training and validation maps.

| | | **x** | **y** |
|---|---|---|---|
| **training** | dimension [m] | 10 | 10 |
| | resolution [m] | 0.1 | 0.1 |
| | cell number [ ] | 100 | 100 |
| **validation** | dimension [m] | 20 | 20 |
| | resolution [m] | 0.1 | 0.1 |
| | cell number [ ] | 200 | 200 |

The maps have been generated with a medium level of complexity. An environment complexity is determined by the number, shape and positioning of the obstacles. For what regards the shape of obstacles, only linear and rectangular ones have been considered. This decision has been made first of all to simplify a bit the learning task for the agent (more complex obstacles, like "C"-shaped ones, require more complex policies to be managed). Moreover, the variety of the obstacle shapes was kept low to allow an easier implementation of the obstacle shape prediction algorithm that has been implemented in the environment model builder. In future, when a more powerful version of this algorithm is implemented, the variety of obstacles in the training and validation maps will be increased.

Using only linear and rectangular obstacles, the only tools to modify an environment complexity are the obstacles number and positioning. In particular, positioning can lead to an higher or lower number of local minima that can be encountered during the exploration. This can lead to a much difficult path-planning process by the path planning agent. Some examples of training maps are shown in figure 5.1 plotted as 2D obstacle maps.

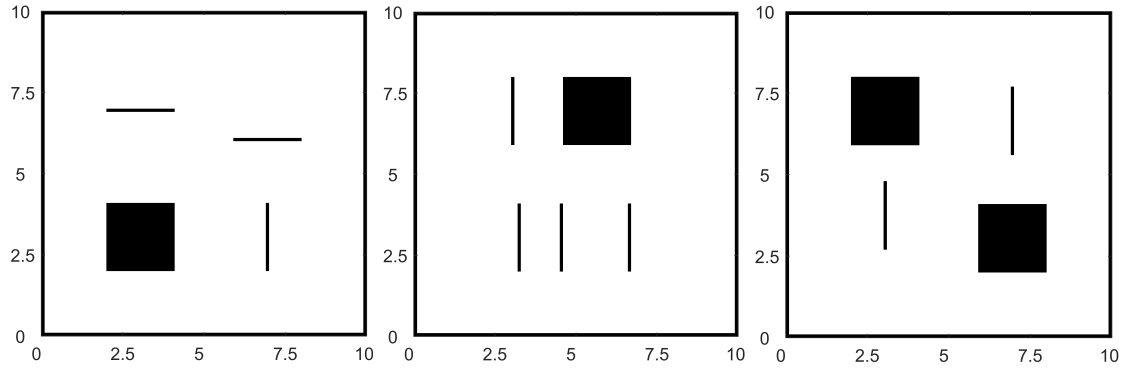


**Figure 5.1:** Examples of training maps. The starting point and the goal are randomly generated for each map (the same map is used multiple times during the training process, so the starting and ending point are re-generated each time). Dimensions are in meters. On average, training maps are covered by obstacles for 7% - 8% of their surface area.

It is worth noting that only a single set of training maps has been generated. This approach poses a problem in the definition of the maps complexity. In fact, the agent uses the same maps to learn all the aspects of its policy, i.e., path planning, obstacle avoidance, local minima management etc. For this reason, the maps have been carefully designed to contain a sufficient number of different scenarios for the agent to manage, while not being to difficult for the agent to learn anything. In fact, if the maps are too complex the agent ends up getting negative rewards too often and does not learn any consistent policy. Validation maps have been designed to be more challenging in order to test all of the agent's capabilities. The three validation maps on which most of the results will be displayed are shown in figure 5.2.

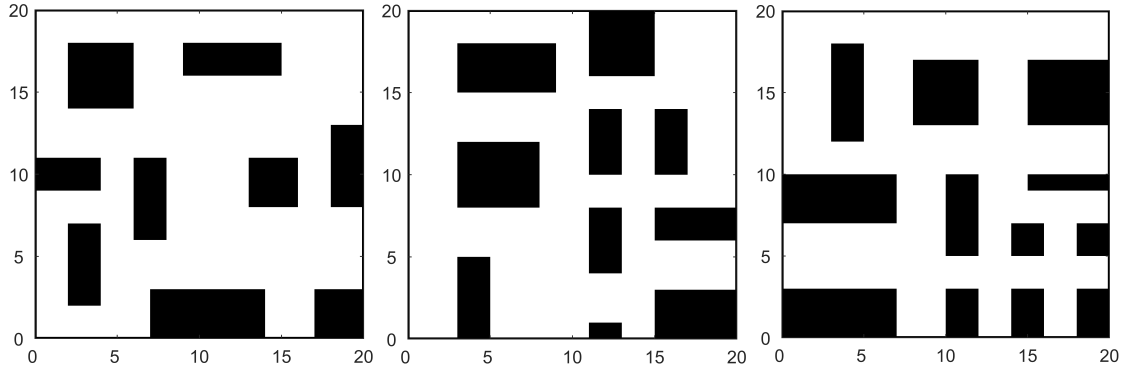


**Figure 5.2:** Validation maps. Dimensions are expressed in meters. Map A is covered by obstacles for 27.8% of its surface area, map B for 30.3% and map C for 34.0%.

## 5.2 Path planning agent training

In this section, the path planning agent training process is presented. The core of the training process has been already introduced as an high-level procedure in section 3.2.4 and in more detail in section 3.2.5. Some adaptations are made to algorithm 3.1 in order to adapt it to the path planning training. The resulting training routine is illustrated in section 5.2.1. After the training routine, the reward function used for the path planning agent is discussed (section 5.2.2). Finally, the path planning training results are presented (section 5.2.3).

### 5.2.1 Path planning training routine

The path planning agent training sequence is quite similar to the generic DDPG learning process previously illustrated in algorithm 3.1. However, it is worth pointing out the parts in which the path planning training process differs from it (e.g. where there are interactions with the simulated environment, as well as some minor other differences). The complete training routine for the Path Planning agent is illustrated in algorithm 5.1. The sections in which the routine is different from algorithm 3.1 are highlighted.

**Algorithm 5.1** Training routine of the path planning agent

1: initialise the actor and critic $\mu(s)$ and $Q(s,a)$ with random parameters $\theta$ and $\phi$
2: initialise the target networks $\mu'(s)$ and $Q'(s,a)$ with parameters $\theta' = \theta$ and $\phi' = \phi$
3: initialise the experience buffer $B$ with size $N_B$
4: define hyperparameters $\mathcal{H}$
5: # start main training loop
6: **for** $e$ in $0...N_{episodes}$ **do**
7: initialise a random Ornstein-Uhlenbeck noise process $\mathcal{N}$ for action exploration
8: change map, randomly selecting one from the training data
9: reset the episode initial condition (i.e. obtain state $s_0$)
10: randomly set the goal position $x_g$
11: **for** $t$ in $0...N_{steps}$ **do**
12: update the environment model using the Environment Model Builder routine presented in Algorithm 4.1
13: select action according to the current actor policy and exploration noise as:

$$a_t = \mu(s_t) + \mathcal{N}_t \qquad \text{where } \mu(s_t) \in [0,1] \tag{5.1}$$

14: multiply the action by $2\pi$ to get an angle in radiants, then clip the result to keep the action inside the interval $[a_{\text{min}},\, a_{\text{max}}] = [0,\, 2\pi]$
15: execute action $a_t$ in the environment. To do so, use the motion simulation functions implemented in the simulated environment. The result is a movement of a fixed distance $\delta$ in direction $a_t \in [0, 2\pi]$
16: use the simulated vision function to detect obstacles inside the field of view of the agent. Store in memory the positions of the obstacles for future use (during the next step they will be integrated in the environment model call on line 12)
17: compute the new state $s_{t+1}$
18: compute the reward $r_t$

**Algorithm 5.2** Training routine of the path planning agent (Part 2)

19: store the experience sample $(s_t, a_t, r_t, s_{t+1})$ in the experience buffer $B$
20: update $s_t \longleftarrow s_{t+1}$ for the next step
21: randomly pick $N_m$ samples from $B$ and store them in the minibatch $M$
22: compute the critic target $\forall i$ in $1...N_m$ as:

$$y_i = r_i + \gamma\, Q'\Big(s', \mu'(s')\Big) \tag{5.2}$$

23: compute the minibatch *cost function* and use it to update the critic parameters $\phi$ by performing one step of Stochastic Gradient Descent (i.e., $\phi = \phi - \alpha_\phi \nabla C$) using the cost function $C$ defined as:

$$C = \frac{1}{N_m} \sum_{i=1}^{N_m} \Big(y_i - Q(s_i, a_i)\Big)^2 \tag{5.3}$$

24: update the actor policy parameters by performing one step of gradient ascent ($\theta = \theta + \alpha_\theta \nabla_\theta J$) where the gradient is computed as:

$$\nabla_\theta J = \frac{1}{N_m} \sum_{i=1}^{N_m} \nabla_\theta Q\left(s, \mu(s_i)\right) \tag{5.4}$$

25: update the target networks parameters following the smoothed-upgrade law:

$$\theta' = \tau\theta + (1-\tau)\theta' \tag{5.5}$$
$$\phi' = \tau\phi + (1-\tau)\phi' \tag{5.6}$$

26: # verify if the current state $s_{t+1}$ is a termination state
27: **if** termination $\mathcal{T}(s_{t+1}) = True$ **then**
28: **end** episode $e$
29: **end if**
30: **end for**
31: # save data for learning progress analysis
32: save episode cumulative reward
33: **end for**

The parts added with respect to algorithm 3.1 regard the simulation operations (in particular, obstacle detection and motion simulation) and the management of the environment model. The routine used to update the environment model is the one described in algorithm 4.1 (section 4.3), and is called each time line 12 is executed. At each time step the environment model is updated by taking into account the positions of all now obstacles detected by the UAV depth camera. In general, the environment update involves also the obstacles detected by other UAVs and shared during the communication operations. However, during the path planning training only a single agent is considered so in algorithm 5.1 this step is skipped. At line 4 the training algorithm hyperparameters are set. The set of hyperparameters $\mathcal{H}$ used during the training, along with their value, are listed in table 5.2.

**Table 5.2:** Hyperparameters used for the path planning agent training. Since multiple agents have been trained, the hyperparameter values reported in the third column correspond only to one specific training (the one that produced the "best" path planning agent, which is the one that will be used for the simulations in the next chapter).

| Parameter Name | Variable | Value |
|---|---|---|
| number of episodes | $N_{episodes}$ | 250 000 |
| number of steps | $N_{steps}$ | 150 |
| agent input state size | $N \times N$ | 75×75 |
| smoothing factor | $\tau$ | 0.1 |
| discount factor | $\gamma$ | 0.95 |
| actor learning rate | $\alpha_\theta$ | 0.0002 |
| critic learning rate | $\alpha_\phi$ | 0.001 |
| action lower bound | $a_{\text{min}}$ | 0 |
| action upper bound | $a_{\text{max}}$ | $2\pi$ |
| step movement distance | $\delta$ | 0.12m |

### 5.2.2 Path planning reward

The design of the reward function for the path planning training is one of the parts of the work that required the most time. In fact, the reward must be well tuned to give the right importance to all the trajectory properties. Moreover, it has revealed to be quite tricky to define a function capable of distinguishing between good behaviours, like going toward the goal, and seemingly-good ones, like going toward a local minimum. The issue is that, given the input state defined in section 4.4.2, both those behaviours are obtained by going toward a low potential. A first way to reduce the incidence of behaviours like this one has been to widen the input state of the agent, increasing its "sense radius". However, a large part of the management of this kind of problem was still linked to the reward function. The reward function with which most of the agents have been trained has been defined as:

$$r = \begin{cases} -10 & \text{if an obstacle is hit} \\ 10 & \text{if goal } x_g \text{ is reached} \\ -w_1 \Delta U - w_2 \Delta \psi - w3\tau & \text{else,} \end{cases} \tag{5.7}$$

The first piece of the reward function is intended to punish unwanted and potentially dangerous behaviours. The second one assigns a large prize to the reaching of the goal, which is the final objective of the path planning agent. The third line is the one that helps the agent learn how to reach the goal and teaches it to compute efficient trajectories (in fact, the objective of the path planning agent is not just to reach the goal, but to reach it *through an efficient trajectory*). This piece of reward is composed of three terms. The first is a term proportional to a variable performance index, which in this case is the difference between the current potential value and the potential at time $t-1$, that is, $\Delta U = U(x(t)) - U(x(t-1))$. If the difference is negative (i.e., a lower potential has been reached), the agent is positively rewarded. If not, the reward is negative. The weight $w_1$ is defined to vary with the sign of $\Delta U$: when $\Delta U > 0$ the weight $w_1$ becomes bigger, punishing more the agent if it goes "against" the potential (but not too much, since in some situations going against the potential is the correct thing to do). The second term of the third line, $\Delta\psi$, punishes the agent with a negative reward if the variation

of direction $\psi$ between two subsequent steps is too large. This term is meant to incentive long-term trajectory planning to produce smooth trajectories. This term could be modified to take into account the dynamical properties of the agent and produce optimal paths from the dynamical point of view. The last term, $\tau$, is constant and represents a time penalty. It urges the agent to find the goal quickly to avoid getting this negative reward. This is the term should also help the agent learn to get out from APF local minima, since once inside them this small negative reward would be continuously obtained for not doing anything. The constant pieces of the reward, as well as the weights in the third line, are tuned through a trial-and-error process.

### 5.2.3 Path Planning training results

Several training processes have been performed to obtain an agent capable of performing all the required path planning and obstacle avoidance operations. After each training, results were analysed to understand how to improve the training performances. The reward function and hyperparameters were updated accordingly, and a new training process was started. The agent that resulted in having the best performances has been obtained using the hyperparameters listed in table 5.2 and the reward function of equation (5.7). Some of the data collected during this agent training are shown in figure 5.3.

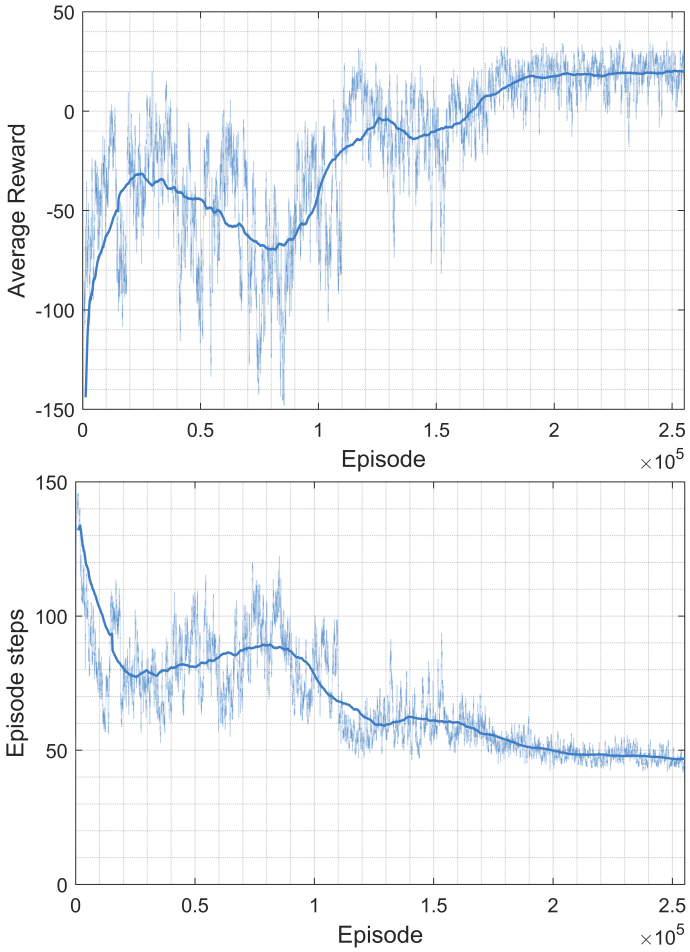


**Figure 5.3:** Training metrics from the path planning agent training.

The top plot of figure 5.3 shows the value of the average episode reward over the episode number. The thin line represents the singular episode rewards, whereas the thick one represents the average reward (computed through a moving average). It can be observed that after about $2 \times 10^5$ episodes the NN parameters go to convergence. The average reward stabilises around the value $+20$ while its variance diminishes significantly. This indicates that the agent has learned an efficient policy to deal with the training environments and is able to consistently obtain positive rewards for reaching the goal. Another metric that confirms the parameter convergence is the average episode length, shown in the bottom image of figure 5.3. As can be observed, after $2 \times 10^5$ episodes become shorter and have almost the same length, as confirmed by the small variance of the thin line around the average. Since the agent has learned how to act efficiently in the environment, it takes less time for it to reach the goal (or, the agent spends less time trying to find an effective way to reach it). The performances of the trained path planning agent in the validation environments are discussed in chapter 6.

## 5.3 Coverage agent training

The coverage agent training has been executed following a procedure similar to the one used for the path planning agent. A different NN has been used, as discussed in section 4.5, and in particular a different input state has been designed, as well as a different output layer. In the following sections, the coverage agent training routine will be presented (section 5.3.1) along with the reward function used (section 5.3.2) and the training results (section 5.3.3).

### 5.3.1 Coverage agent training routine

As done for the path planning agent, the learning routine used to train the coverage agent will be illustrated in this section. Again, the general structure of the routine is similar to the one of the generic DDPG learning algorithm illustrated in algorithm 3.1. The learning routine version used for the coverage agent is shown in algorithm 5.3. The modifications performed to adapt the learning procedure to the coverage agent case are highlighted. As can be observed in the pseudo-code, the structure of the coverage training routine is more complex than the one of the path planning agent. The reason for this is that in this case it is necessary to simulate the actions and movement of the entire fleet $\mathcal{F}$. In fact, as can be observed at line 13, at each training step a waypoint is computed by each UAV. The computation is performed always by the same NN, which is the actor $\mu(s)$ that is being trained. However, for each UAV the input state $s$ is different, and so the computed waypoint is also different. After the computation of the waypoint, each UAV performs a number of steps to move toward it (lines 18-21). During this movement, any new area of the environment that the UAV sensors observe is measured and summed in a variable $A^u$ (for UAV "$u$") that is used to compute the reward. The larger $A^u$, the higher the reward, since a large value of $A^u$ indicates a good waypoint placement and a large area explored. At each training step every UAV of the fleet adds an experience sample to buffer $B$. On the contrary, the training sequence (i.e., the NN parameter update) is executed only once per step. This choice is *not* mandatory, and to speed up the learning process the gradient computation and the parameter update could be performed multiple times per step (this has been done in some of the training sessions). Also, this strategy is not in conflict with the fact that multiple experience samples are added to the buffer, since a minibatch of size $N_m$ is used and the learning is performed off-policy (so the data used to compute the gradient have no relationship with the ones added to the buffer in the same step).

**Algorithm 5.3** Training routine of the coverage agent

initialise the actor and critic $\mu(s)$ and $Q(s, a)$ with random parameters $\theta$ and $\phi$
initialise the target networks $\mu'(s)$ and $Q'(s, a)$ with parameters $\theta' = \theta$ and $\phi' = \phi$
initialise the experience buffer $B$ with size $N_B$
define training hyperparameters $\mathcal{H}$
\# start main training loop
**for** $e$ in $0...N_{episodes}$ **do**
    initialise a random Ornstein-Uhlenbeck noise process $\mathcal{N}$ for action exploration
    change map, randomly selecting one from the training data
    reset the episode initial condition (i.e. obtain state $s_0$)
    \# begin step $t$ inside current episode $e$
    **for** $t$ in $0...N_{steps}$ **do**
        \# repeat the *exploration sequence* for each UAV of the fleet
        **for** UAV $u$ in fleet $\mathcal{F}$ **do**
            build state $s$ as described in section 4.5 taking into account obstacle positions, other UAV positions and the already explored regions
            select action according to the current actor policy and exploration noise as:

$$\begin{bmatrix} \bar{x}_t \\ \bar{y}_t \end{bmatrix} = \mu(s_t) + \mathcal{N}_t \tag{5.8}$$

            where the bar over the outputs of the actor $\mu(s_t)$ indicates that they are in the interval [0, 1] and have to be re-scaled to find the waypoint position in space
            multiply each action by the corresponding environment side size to get the waypoint position [$x_t$, $y_t$] in meters, expressed with respect to one of the corners of the map. Due to noise presence, the resulting value may need to be clipped to stay in the interval [0, *map side length*]
            \# simulate movement during exploration. Some movement steps toward the waypoint are simulated along with the vision functions to observe the environment while moving
            **for** $i$ in $0...N_{substeps}$ **do**
                use the numerical APF path planning algorithm[18]to make one step of length $\delta$ toward the current waypoint position
                use the simulated vision function to "explore" the area in front of the UAV. This includes obstacle detection. The size $\Delta^u$ of the "new" area explored (i.e. the portion of space observed during substep $i$ that had not been visited before, even by other agents), is memorised to compute the reward for UAV $u$ at the end of step $t$, as in:

$$A_i^u = A_{i-1}^u + \Delta_i^u \tag{5.9}$$

            **end for**
            \# exploration step ended, proceed to store experience sample. Then, repeat the same operation for the next UAV of the simulated fleet
            share information with other UAVs through communication channels

**Algorithm 5.4** Training routine of the coverage agent (Part 2)

24: # ...(continues) store experience collected by UAV $u$ in the experience buffer
25: compute the new state $s_{t+1}$
26: compute the reward $r_t = \mathcal{R}(A^u)$
27: store the experience sample $(s_t, a_t, r_t, s_{t+1})$ in the experience buffer $B$
28: update $s_t \longleftarrow s_{t+1}$ for the next step
29: **end for**
30: randomly pick $N_m$ samples from $B$ and store them in the minibatch $M$
31: # action simulation ended, proceed with NN learning. Then, move to the next step
32: compute the critic target $\forall i$ in $1...N_m$ as:

$$y_i = r_i + \gamma\, Q'\Big(s', \mu'(s')\Big) \tag{5.10}$$

33: compute the minibatch *cost function* and use it to update the critic parameters $\phi$ by performing one step of Stochastic Gradient Descent (i.e., $\phi = \phi - \alpha_\phi \nabla C$) using the cost function $C$ defined as:

$$C = \frac{1}{N_m} \sum_{i=1}^{N_m} \Big(y_i - Q(s_i, a_i)\Big)^2 \tag{5.11}$$

34: update the actor policy parameters by performing one step of gradient ascent ($\theta = \theta + \alpha_\theta \nabla_\theta J$) where:

$$\nabla_\theta J = \frac{1}{N_m} \sum_{i=1}^{N_m} \nabla_\theta Q\left(s, \mu(s_i)\right) \tag{5.12}$$

35: update the target networks parameters following the smoothed-upgrade law:

$$\theta' = \tau\theta + (1-\tau)\theta' \tag{5.13}$$
$$\phi' = \tau\phi + (1-\tau)\phi' \tag{5.14}$$

36: # verify if the current state $s_{t+1}$ is a termination state
37: **if** termination $\mathcal{T}(s_{t+1}) = True$ **then**
38: **end** episode $e$
39: **end if**
40: **end for**
41: # save data for learning progress analysis
42: save episode cumulative reward
43: **end for**

The hyperparameters of the Coverage agent training process, which are set on line 4 of the training routine, are listed in Table 5.3.

[18] In alternative, an already-trained RL path planning agent can be used. This would make the path planning simulation more realistic but a bit slower.

**Table 5.3:** Hyperparameters used for the coverage agent training. Since multiple agents have been trained, the hyperparameter values reported in the third column correspond only to one specific training (the one that produced the "best" Coverage agent, which is the one that will be used for the simulations in the next chapter).

| Parameter Name | Variable | Value |
|---|---|---|
| number of episodes | $N_{episodes}$ | 250 000 |
| number of steps | $N_{steps}$ | 10 |
| number of substeps | $N_{substeps}$ | 50 |
| number of UAVs | $N_{\mathcal{F}}$ | 3 |
| UAV area of influence | $d_{inf}$ | 7.5m |
| step movement distance | $\delta$ | 0.12 |
| agent input state size | $N \times N$ | 200×200 |
| smoothing factor | $\tau$ | 0.1 |
| discount factor | $\gamma$ | 0.95 |
| actor learning rate | $\alpha_\theta$ | 0.0002 |
| critic learning rate | $\alpha_\phi$ | 0.001 |
| action lower bound | $a_{\text{min}}$ | [0, 0] |
| action upper bound | $a_{\text{max}}$ | [1, 1] |

### 5.3.2 Coverage reward

It is worth discussing the reward function used for the training of the coverage agent. The reward function used to train the coverage agent is:

$$r = \mathcal{R}(s,a) = \begin{cases} -5 & \text{if } x_g \text{ is in an illegal location} \\ -1 & \text{if } x_g \text{ is in an unwanted location} \\ w_1 \Delta^u & \text{else,} \end{cases} \tag{5.15}$$

where $x_g$ is the waypoint position and $\Delta^u$ is the size of the region explored by a single UAV $u$ while moving toward a valid waypoint. A temporary goal $x_g$ is considered to be in an *illegal* location if it is over a known obstacle. An *unwanted* location, instead, is defined as an already explored point, or a point under the area of influence of another UAV. While the first two lines of the reward function are intended to prevent unwanted behaviours, the third is the one that prizes the agent for exploring new regions of the environment. The reward is proportional to the size of the explored region, encouraging the agent to find smart ways to explore as much space as possible. The area is computed by simulating the movement of the agent toward the waypoint. This is done by performing a certain number of steps using an already-trained path planning agent (as seen in algorithm 5.3). A property of the coverage algorithm that is worth noticing is that the agent is trained to be *selfish*. In fact, it receives no reward for the other UAVs exploration. This results in a fleet where each UAV just wants to explore as much of the environment as possible, with no regard for other UAVs. The coordination of the fleet is guaranteed only by the fact that this greedy behaviour has been trained to take into account the area of influence of other UAVs and by the fact that, since all the UAVs use the same policy for the coverage, they can implicitly predict the behaviour of other UAVs.

### 5.3.3 Coverage training results

A number of training processes have been performed to obtain an effective coverage agent. As already mentioned above, the training of the coverage agent has resulted to be more difficult than the one of the path planning agent. The need to simulate the movement of all the members of the fleet results in a significant slowdown in the training process. This leads to a much smaller number of training episodes per hour, which in turn means a smaller number of learning steps per hour. A solution introduced to mitigate this slowdown has been to perform a double learning sequence each step (i.e., to perform the lines from 30 to 35 of algorithm 5.3 twice per step $t$), but the overall training speed has improved only a little. The time constraints that come with the development of a thesis work did not allow to perform a complete training process of the coverage agent. A full training process with the most computationally demanding simulation setup is estimated to require about 500 CPU-hours to complete. This kind of training, taking into account the trial-and-error process required to optimally tune the learning routine hyperparameters and the reward function, resulted unfeasible. Some data from the training process of the "best" coverage agent obtained are shown in figure 5.4.

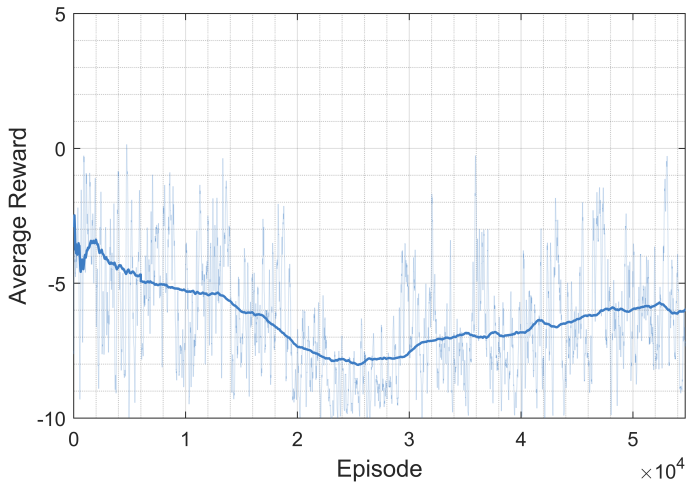


**Figure 5.4:** Average reward over episodes during the coverage agent training. Each episode corresponds to two learning steps (i.e., two updates of the NNs parameters).

Figure 5.4 shows the average reward obtained by the selected coverage agent during its training. As can be observed on the $x$ axis, the final number of episodes is quite small ($5 \times 10^4$) with respect to the number of steps reached during the path planning agent training (about $2.5 \times 10^5$). Considering 10 steps per episode, and 2 learning routines performed per step, the total number of coverage learning steps is about $1 \times 10^6$. This result is significantly smaller than the number of learning step performed during the path planning agent training, which is approximately $25 \times 10^6$. The difference lies mainly in the aforementioned higher complexity of the simulation process required by the coverage agent, which slows down the coverage training. It is very likely that better result could be obtained by optimising more the coverage simulations in order to speed the training process up. In any case, some simulations and results obtained from the trained coverage agent are shown in chapter 6. The results are not optimal, as can be predicted by observing the non-convergence of the average reward in figure 5.4, but some interesting result are obtained and discussed nonetheless.

# 6. Simulations & results

In this chapter, the results obtained from the simulation of the trained agents are presented. First, the path planning agent is evaluated as a standalone algorithm (section 6.2). Then, all the parts of the algorithm are integrated to build a complete version of the proposed exploration model, which is then simulated and evaluated (section 6.3).

## 6.1 Simulation routine

Before getting to the discussion of the simulations and results obtained from the trained agents, it is worth illustrating the simulation routine used to produce all the results. The routine is illustrated as pseudo-code in algorithm 6.1.

**Algorithm 6.1** UAV action routine

```
1: while True do
2:     # update the environment model
3:     detect obstacles with the sensor vision
4:     share information with other UAVs (send and receive current position, obstacle position
       and exploration data)
5:     if new obstacles have been found then
6:         update the potential map with the new obstacles positions
7:     end if
8:     update the potential map with the mobile obstacles in the positions received
9:     # perform coverage operations
10:    if the goal position is not updated then
11:        call the coverage algorithm to compute the position of the new goal
12:        call the environment model builder in order to update the attractive layer with the
           new goal position
13:    end if
14:    # perform path planning operations
15:    if the drone is in a local minimum then
16:        switch to the local minima path planning algorithm (more details in section 6.2.)
17:    else
18:        switch to the main RL path planning algorithm
19:    end if
20:    compute trajectory using the right path planning algorithm
21:    follow the trajectory for a distance δ
22: end while
```

The routine illustrated in algorithm 6.1 is the one used in all the simulations, both for those of the only path planning agent (with the exception of lines 9-13) and those of the complete fleet for the exploration simulation. When used to simulate the entire fleet, the routine is applied in the same way for all the UAVs composing it.

## 6.2 Path planning simulation and results

In this section, the path planning agent is evaluated. In section 6.2.1 the evaluation metrics used to assess the algorithm performances are listed. Simulations are shown in section 6.2.2 and results are discussed in section 6.2.3.

### 6.2.1 Path planning evaluation metrics

To assess the performances of the path planning agent a series of evaluation metrics have been defined. The metrics have been selected on the basis of the points of interest of the path planning agent. In particular, computational cost, safety and energetic efficiency have been considered. The metrics through which the agent performances have been evaluated are:

- $goal\ reached$: a boolean value indicating whether or not a trajectory reaching the goal has been found;
- $\varepsilon$: the minimum measured distance between the agent and any obstacle point;
- $t$: the time required by the algorithm to compute the trajectory;
- $\bar{\Delta}_{IO}$: the average value of $\Delta_{IO}$, which is the difference between the "input angle" and "output angle" computed at each point of the trajectory.[19] $\bar{\Delta}_{IO}$ quantifies how sharp are - on average - the turns in a trajectory. A high value of $\bar{\Delta}_{IO}$ is associated to a less optimal trajectory from the dynamical and energetic point of view (that is, once the UAV dynamics is taken into account the trajectory is more difficult to follow, requiring lower speed or a higher actuation force).

### 6.2.2 Path planning simulations

Several simulations have been performed to assess the performances of the path planning agent. The simulations that will be discussed below are performed in the validation maps of figure 5.2. The RL-trained path planning agent has been compared with two other path planning algorithms to have a reference during the evaluation of the performance metrics. The reference path planning algorithms chosen are A* and APF.[20] The code used for A* was adapted from https://github.com/AtsushiSakai. First of all, the qualitative results of the RL path planning agent in terms of trajectory computation are shown in figure 6.1.

[19] For each discrete point $x_i$ of the trajectory (obtained by ignoring the *fitting* step in Algorithm 4.2), the input-output angle difference is computed as $\Delta_{IO} = |\psi(x_i, x_{i+1}) - \psi(x_{i-1}, x_i)|$ where $\psi$ is the direction of a given vector. The vectors considered are the one "exiting" from $x_i$ (i.e., going from $x_i$ to $x_{i+1}$) and the one "entering" in it (i.e., going from $x_{i-1}$ to $x_i$).

[20] In this case, Artificial Potential Field is intended as a *path planning* algorithm, differently than before where APF was used as a tool to build the environment model.

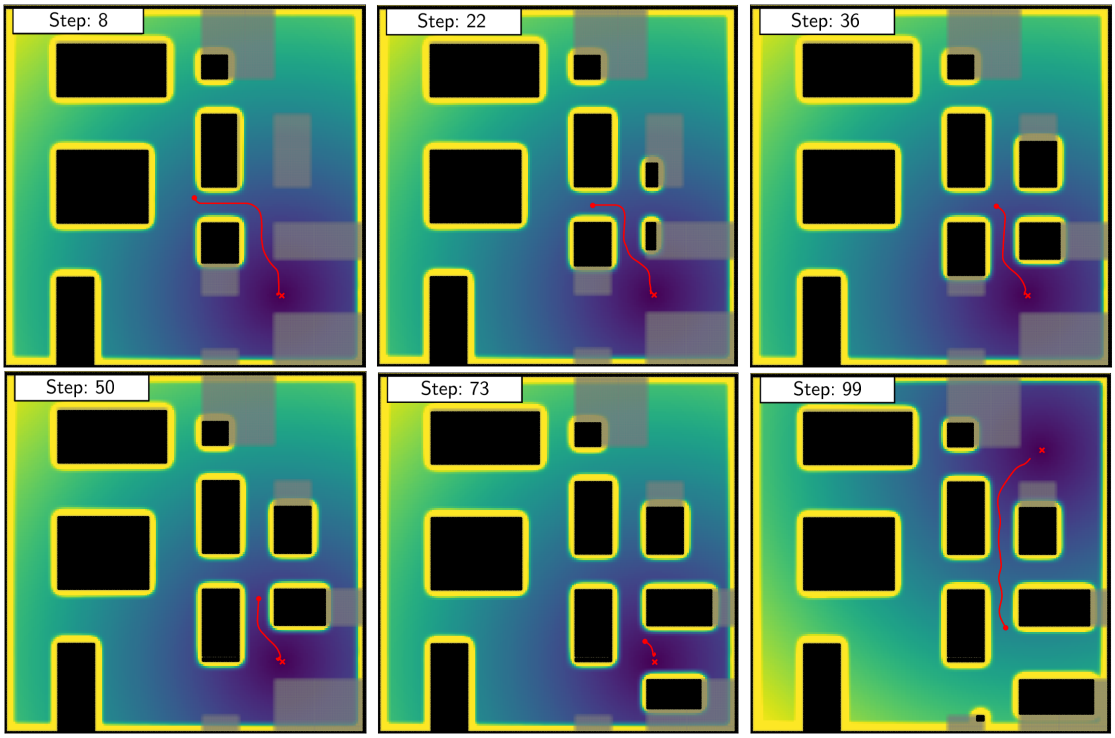


**Figure 6.1:** Simulation of trajectory planning using the trained path planning agent (in validation map B).

Each frame of figure 6.1 represents an instantaneous configuration of the simulation environment. The red dot represents the UAV, the red cross is its objective (randomly generated, since there is not a trained coverage agent yet) and the red line connecting the two is the computed trajectory. Black regions represent obstacles known by the agent, whereas the grey ones indicate the locations of other obstacles which are still unknown. The coloured background represents the APF model value in each point. Blue tones indicate a low potential (i.e., the goal neighbourhood) whereas yellow ones represent obstacles and the areas near them. The images are taken at different moments of the exploration of a validation environment. Time is represented by the *step* counter located in the upper left corner of each frame. The counter starts from 0 and increases by 1 at each step performed by the agent in the simulated environment. At each time step, the agent re-computes the trajectory from its current status using algorithm 4.2 and performs a single simulated movement step. A movement step corresponds to a movement of $\delta$=0.12m along the trajectory. As can be observed in the images, as the agent moves it discovers new obstacles,[21] whose influence is added to the APF model, and updates its trajectory accordingly. At each time step, the path planning agent is able to compute an efficient trajectory to move toward the goal, always keeping a safe distance from obstacles. In

[21] In the simulation the aforementioned *obstacle shape prediction algorithm* can be observed in action. In fact, not all the points of the obstacles are directly observed. The position of many of them is simply estimated from the location of the other known obstacle points and from the fact that in the validation maps only rectangular obstacles are present.

the last image (step = 99) the APF background is different from that of the other images; in fact, between step 73 (5th frame) and 99 (6th frame), the goal has been reached. For this reason, a new objective has been computed in the upper region of the environment, and the agent computes a new trajectory to reach it. The fact that the trajectory does not reach exactly the goal is due to the fact that the trajectory planning of algorithm 4.2 computes a finite number $n$ of trajectory waypoints to fit to obtain the final path. In this case, the maximum number of steps has been reached without getting to the goal. This does not represent a problem, since the agent is still able to go toward the objective safely, and will update the trajectory to reach it in the future. A second, longer simulation in which the results discussed up to now can be observed in a different scenario is displayed in figure 6.2.

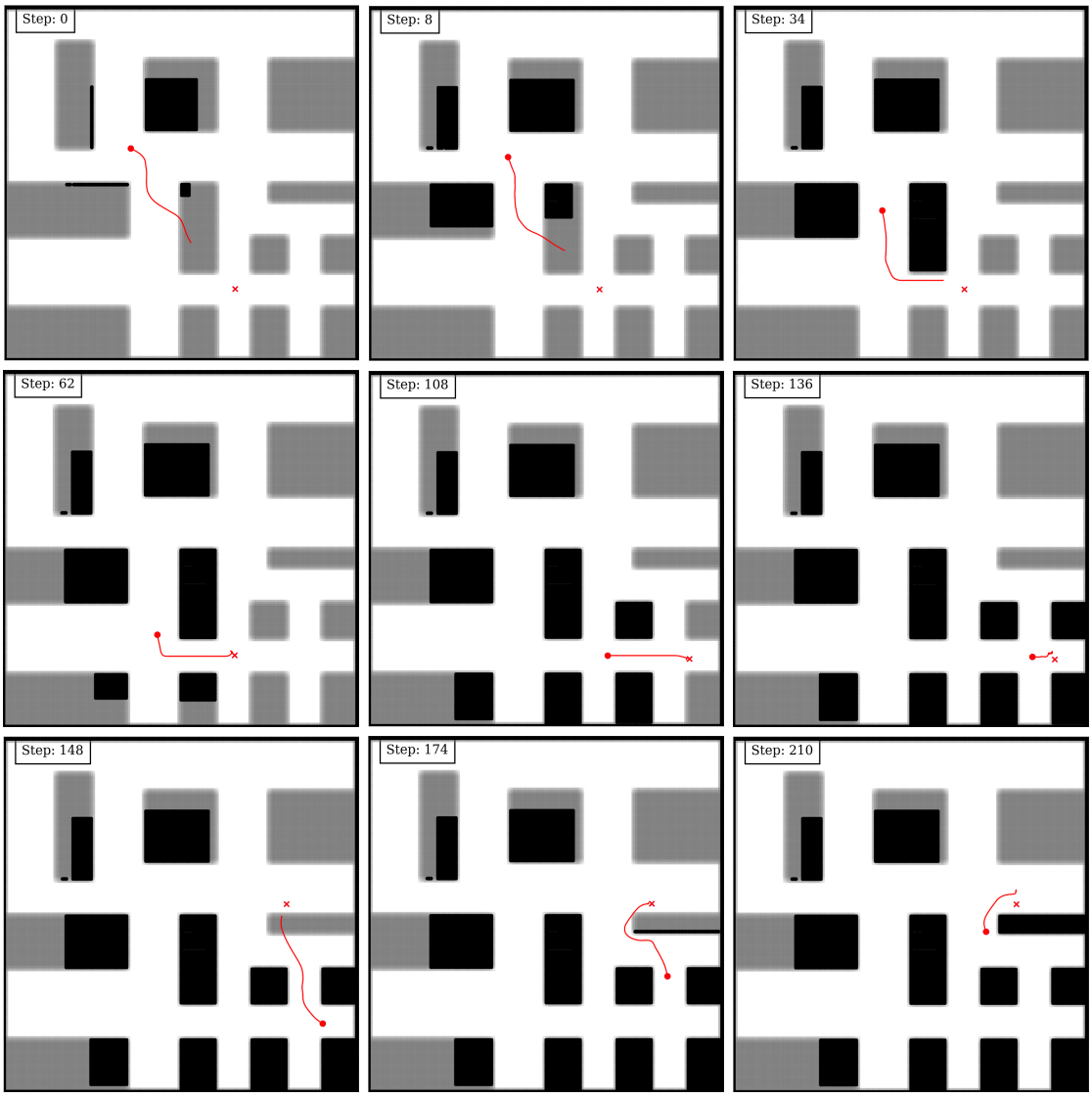


**Figure 6.2:** Simulation of trajectory planning using the trained path planning agent (in validation map C). The background representation of the APF is not shown for simplicity.

An issue encountered in the agent validation has been its management of local minima. As introduced in section 3.1, local minima are a common problem in APF-based algorithms. The model presented above, in a situation where it found itself near some particularly difficult local

minima ($< 10\%$ of the local minima encountered), has not been able to consistently reach the goal. For this reason, in the final implementation it has been paired with a second agent (named "*assisting*" agent), trained on different environments and with a slightly different reward function. In particular, the training environments were designed to present a high number of complex local minima, whereas the reward function was modified to punish less the agent when going toward a high potential (which is necessary to get out from local minima). The performances of the "assisting" agent are displayed in figure 6.3.

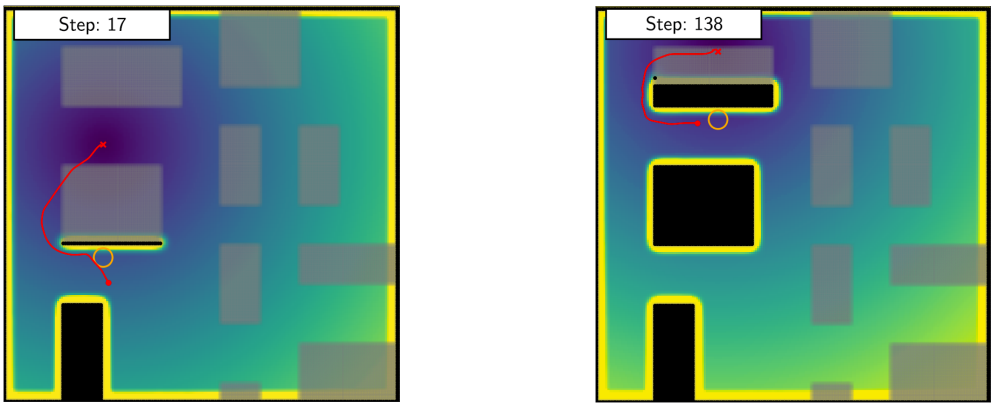


**Figure 6.3:** Examples of trajectories computed by the "assisting" agent, i.e., the one devoted to local minima avoidance. Two difficult local minima in map B are considered.

As can be observed, the assisting agent is able to compute effective trajectories even in presence of very difficult local minima (in both the images the local minimum location is indicated by the orange circle). In the occurrences where the main agent is not able to manage a local minima in the APF model, the assisting agent is activated to compute a suitable trajectory to continue going toward the desired goal. After a sufficient amount of time, during which the UAV has moved correctly toward the objective and far from the local minimum, the trajectory planning task is given back to the main agent. The logical flow with which the right agent is selected to compute the trajectory every time the Path Planning function is called is illustrated in figure 6.4. As can be observed in figure 6.4, the switch between the two agents is performed only when a local minimum near the current UAV position is detected. It is worth mentioning that, for both the path planning agents used in this architecture, the routine of algorithm 4.2 is applied to obtain the path.

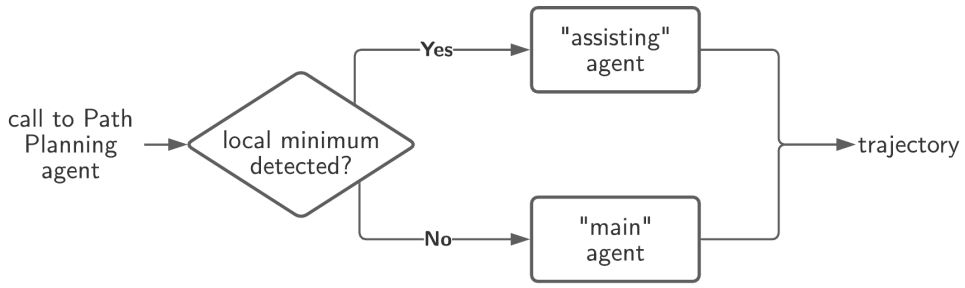


**Figure 6.4:** Logical flow through which the right path planning agent is selected to compute the trajectory every time the path planning routine is called. Both the agents use the routine of algorithm 4.2 to compute the trajectory.

A goal for the future development of the algorithm is to obtain a single path planning agent capable of merging the capacities of the "main" and "assisting" agent in a single function. This would streamline the path planning workflow and avoid having to perform two separate training processes. The main difficult of obtaining a single path planning agent is that local minima management probably requires a dedicated part of the training process (i.e., after a "generic" path planning policy has been learned, a series of local minima are presented to the agent during the training to get it to learn how to manage them). This part of the training process has to be designed carefully, to avoid for the agent to unlearn some behaviour.

### 6.2.3 Path planning results

The RL-based path planning algorithm has been compared with other path planning algorithms in order to have a reference while evaluating its performances. A series of tests has been performed in the three validation maps of figure 5.2. For each test, a starting point and an ending point are arbitrarily selected in one of the maps. Then the three algorithms considered (the proposed RL agent, A* and APF) are executed and the trajectories obtained are analysed. In particular, for each trajectory the data about the four evaluation metrics listed in section 6.2.1 are collected. The results of some of the tests are listed in table 6.1. For each test, the map is indicated (as A, B or C, with reference to the maps of figure 5.2) along with the starting and ending point locations. The approximate length of the trajectory is also listed, as it is the most important feature correlated with the computational time (as will be discussed in more detail later). For each test, the measured value of all the four metrics is reported for all the three path planning algorithms.

**Table 6.1:** Numerical results obtained from the simulation of the path planning algorithms in the test environments.

| Test | Map | Start/End Point [m] | $l$ [m] | Algorithm | Goal Y/N | $\varepsilon$ [m] | $\bar{\Delta}_{IO}$ [deg] | t [s] |
|---|---|---|---|---|---|---|---|---|
| 1 | A | $x_0 : (17, 4)$<br>$x_g : (14.5, 13)$ | 9.80 | RL | yes | 0.53 | 10.01 | 0.49 |
| | | | | A* | yes | 0.36 | 9.84 | 0.42 |
| | | | | APF | no | - | - | - |
| 2 | B | $x_0 : (7, 3)$<br>$x_g : (16, 18)$ | 20.00 | RL | yes | 0.72 | 6.06 | 1.06 |
| | | | | A* | yes | 0.45 | 10.43 | 1.66 |
| | | | | APF | yes | 0.76 | 6.81 | - |
| 3 | C | $x_0 : (17, 4)$<br>$x_g : (17, 18)$ | 19.00 | RL | yes | 0.67 | 14.17 | 0.92 |
| | | | | A* | yes | 0.42 | 8.64 | 0.69 |
| | | | | APF | no | - | - | - |
| 4 | A | $x_0 : (10, 10)$<br>$x_g : (5, 5)$ | 8.00 | RL | yes | 0.61 | 15.59 | 0.399 |
| | | | | A* | yes | 0.57 | 10.52 | 0.218 |
| | | | | APF | yes | 0.60 | 10.89 | - |
| 5 | B | $x_0 : (2, 13)$<br>$x_g : (14, 9)$ | 14.00 | RL | yes | 0.61 | 15.82 | 0.70 |
| | | | | A* | yes | 0.36 | 20.61 | 0.52 |
| | | | | APF | yes | 0.60 | 17.50 | - |
| 6 | C | $x_0 : (2, 18.8)$<br>$x_g : (6, 17.2)$ | 4.75 | RL | yes | 0.70 | 8.70 | 0.258 |
| | | | | A* | yes | 0.40 | 20.54 | 0.05 |
| | | | | APF | yes | 0.58 | 16.58 | - |

Some interesting results can be observed in table 6.1. The RL path planning agent is capable of always finding the trajectory, even when the original implementation of the APF path planning algorithm fails to find a suitable path to the goal (as in tests 1 and 3). From the tests performed on the RL path planning agent it emerges that the agent is always able to compute a trajectory (i.e., a path is always computable using the RL agent) even if, in some very difficult scenarios such as hard local minima, the computed path does not converge to the solution (i.e., the trajectory does not reach the goal). The minimum distance from obstacles is always the most conservative one, meaning that the agent can consistently find the goal while keeping a safe distance from obstacles. This behaviour does not lead to sub-optimal trajectories, as can be observed in the column reporting the values of $\bar{\Delta}_{IO}$. The average curvature of the trajectory is always similar to the one of the path computed by A*. In some of the tests (e.g. in test 2) the average angle difference is lower than the one of the other algorithms because the RL agent managed to find a more dynamically-efficient trajectory. This can be observed in the left image of figure 6.5, where the three trajectories computed to solve test 2 are displayed. The RL agent, whose trajectory is represented by the orange line, finds a different path from those of the A* and APF algorithms (represented respectively by the blue line and yellow line). The RL trajectory is optimised to minimise the turns, i.e., to optimise the energetic cost of travelling along the trajectory (which means that less actuation force is needed to follow the trajectory, or a higher speed can be maintained along with it). A similar result is obtained also in test 5, represented on the right image of figure 6.5. In this case, as in the previous one, turns are smoother in the trajectory computed by the RL agent, which also keeps the highest distance from obstacles. However, in this case the other two trajectories which "cut" the curves are probably a bit more efficient energetically since they are shorter.

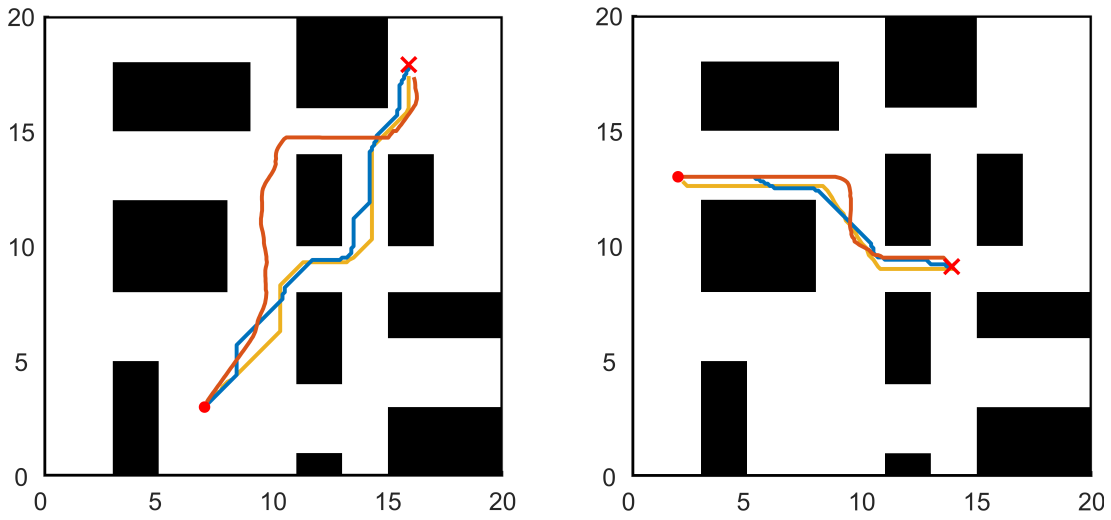


**Figure 6.5:** Comparison of the trajectories computed by the path planning algorithms in test 2 (on the left) and test 5 (on the right). Orange line represents RL trajectory, blue is A*, and yellow is APF.

Some interesting results can also be observed about the computational time. First of all, it is worth mentioning that the computational time of APF has not been measured but was - in general - very low (<0.1s) thanks to the fact that the algorithm implementation used exploited the already-computed APF environment model and so was very fast at finding the direction

of the lowest potential.[22] The issue with the APF algorithm is that, following the "classical" implementation of it, local minima cannot be avoided and when one of them is reached the test ends without having computed a suitable trajectory (as in tests 1 and 3). For what regards the RL agent and A* computational times, it is interesting to observe their trend when they are plotted against the approximate trajectory length. In figure 6.6 time $t$ is plotted against length $l$ for both the proposed RL agent and the A* algorithm. As can be observed from the plot of figure 6.6, the computational cost of the RL agent grows linearly with the length of the trajectory, independently from the number of obstacles or the complexity of the environment. On the contrary, A* is far more sensitive to those parameters. The computational cost of A* grows exponentially with the trajectory length and is very sensitive to the presence of obstacles between the UAV position and the goal. This can be observed, for example, in the results of test 2. The corresponding point on the plot of figure 6.6 can be observed approximately at the coordinates $x = 20, y = 1.6$. The high computational cost is due to the presence of obstacles directly between the starting and ending position. While A* is able to effectively avoid them and compute a suitable trajectory, this configuration of the obstacles introduces a significant increase in the computation time with respect to other runs. These significative variations in the computational time are due to the routine followed by A* to compute the path (A* contains a circular propagation of the points to consider for the trajectory, and the length of the propagation increases when an obstacle is found directly between the UAV and the goal).

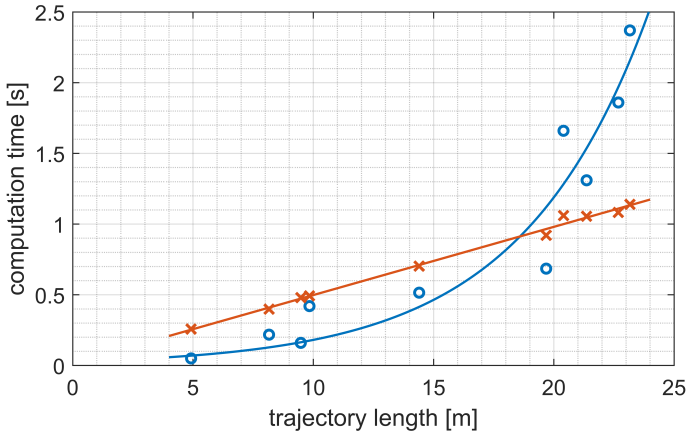


**Figure 6.6:** Comparison of the computational time required by the RL agent (orange line) and A* (blue line) to obtain trajectories of various lengths.

[22] The APF implementation used simply selects as direction of motion the one where the lowest potential is located (i.e., it chooses amongst the surrounding cells of the environment model and goes toward the one with the lowest potential value).

## 6.3 Exploration simulations and results

In this section, simulation and results of the complete exploration algorithm will be presented and discussed. The focus is on the performances of the coverage agent, since the path planning one was already analysed in the previous part of this chapter. As for the path planning agent, some evaluation metrics are defined to quantitatively measure the performances of the coverage agent (section 6.3.1). Then, simulation and results obtained evaluating the exploration process are presented (sections 6.3.2 and 6.3.3).

### 6.3.1 Exploration evaluation metrics

Some evaluation metrics have been defined to assess the exploration algorithm performances in the validation environments. The metrics are selected in particular to measure the efficiency of the coverage algorithms in exploring the environment in the fastest and most optimal way possible. The environment is considered "explored" when at least $90\%$ of its surface has been observed. A piece of environment is considered *observed* when it has been scanned by the depth camera of one of the UAVs. This means that it is not necessary for the UAVs to cross every single point of the environment, but it is sufficient to observe them. The metrics defined assess the coverage agent performances in the exploration process are:

1. $\hat{d}(t)$: average distance between the UAVs over time. This metric is computed by averaging the distance between each couple of UAVs. For example, in the case of a fleet composed of three UAVs, the metric is computed as:

$$\hat{d}(t) = \frac{1}{3}\sum \|d_{ij}\| \qquad \text{where } \begin{cases} i = 1,2,3 \\ j = 1,2,3 \\ i \neq j \end{cases} \tag{6.1}$$

   This metric determines how to spread the fleet is over the map. It is interesting also to analyse the standard deviation of $\hat{d}(t)$, since it gives additional information about the fleet formation over time;

2. $d_{min}(t)$: minimum distance registered between two UAVs. This parameter is important to evaluate both the effectiveness of the inter-UAV collision avoidance system and how well the UAVs were spread in the environment.

$$d_{min} = \min \|d_{ij}\| \tag{6.2}$$

   This metric is evaluated over each couple of UAVs (indicated by the $i$ and $j$ subscripts, where $i \neq j$) and the minimum value is taken at each time instant;

3. $t_{N\%}$: the number of simulation steps after which the fleet has explored at least $N\%$ of the environment surface area. This value is measured for different values of $N$;

4. $A_{\%}(t)$: the explored area percentage over time, computed as:

$$A_{\%}(t) = \frac{A_{explored}}{A_{explorable}}\% \tag{6.3}$$

   This metric is evaluated both for the whole fleet and for each individual UAV, i.e., the percentage explored area of each UAV is measured over time.

### 6.3.2 Exploration simulations

Multiple simulations have been performed to test the performances of the complete exploration algorithm. In particular, the focus has been put on the performances of the coverage agent, since the path planning one had already been tested in the previous part of this chapter. Different configurations of the exploration algorithm have been tested. First, the coverage agent has been evaluated singularly. However, due to the limitations in the training process, the agent resulted able to effectively drive the exploration process only up to ∼70% of the exploration. After that, the temporary goal placement resulted inefficient and repetitive. For this reason, the completion of the exploration process has been assigned to the explicitly-coded algorithm based on the k-means partitioning of the environment (introduced in section 4.5.3 and explained in detail in appendix B). All the simulations displayed in this chapter have been obtained by combined use of the RL agent and the k-means algorithm. Some additional simulations have been obtained by using only the k-means agent. These simulations are used as reference. Two simulations, obtained using a fleet of four UAVs in validation map A, are shown in the next pages. The first is a simulation obtained through the combined use of the RL coverage agent and the k-means algorithm, and is shown in figures 6.7 and 6.8. The second one, obtained using only the k-means algorithm, is displayed in figures 6.9 and 6.10. Each frame of the simulations displays the location of the four UAVs with different colours, as well as the four temporary goals. The explored regions are marked in blue, whereas known obstacles are coloured in black. The total percentage explored area is reported in the top-left corner of each frame. Before each simulation, the environment obstacles configuration is displayed (corresponding, in both cases, to validation map A).

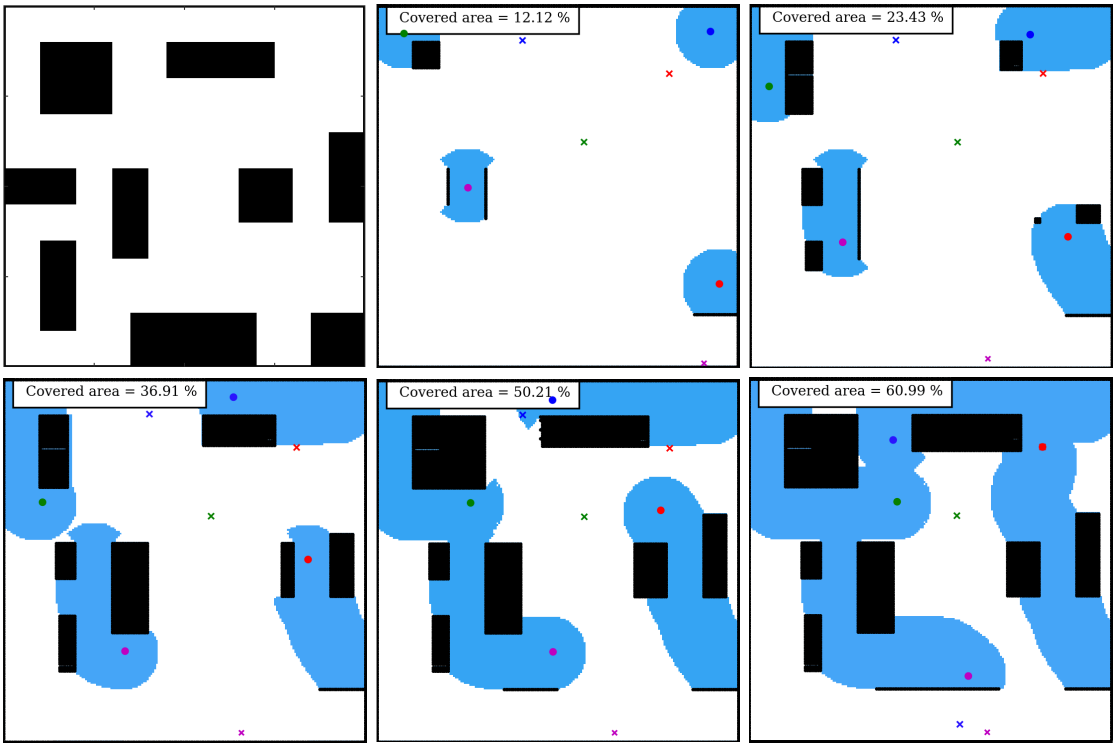


**Figure 6.7:** Exploration simulation using the RL agent + k-means coverage algorithm.

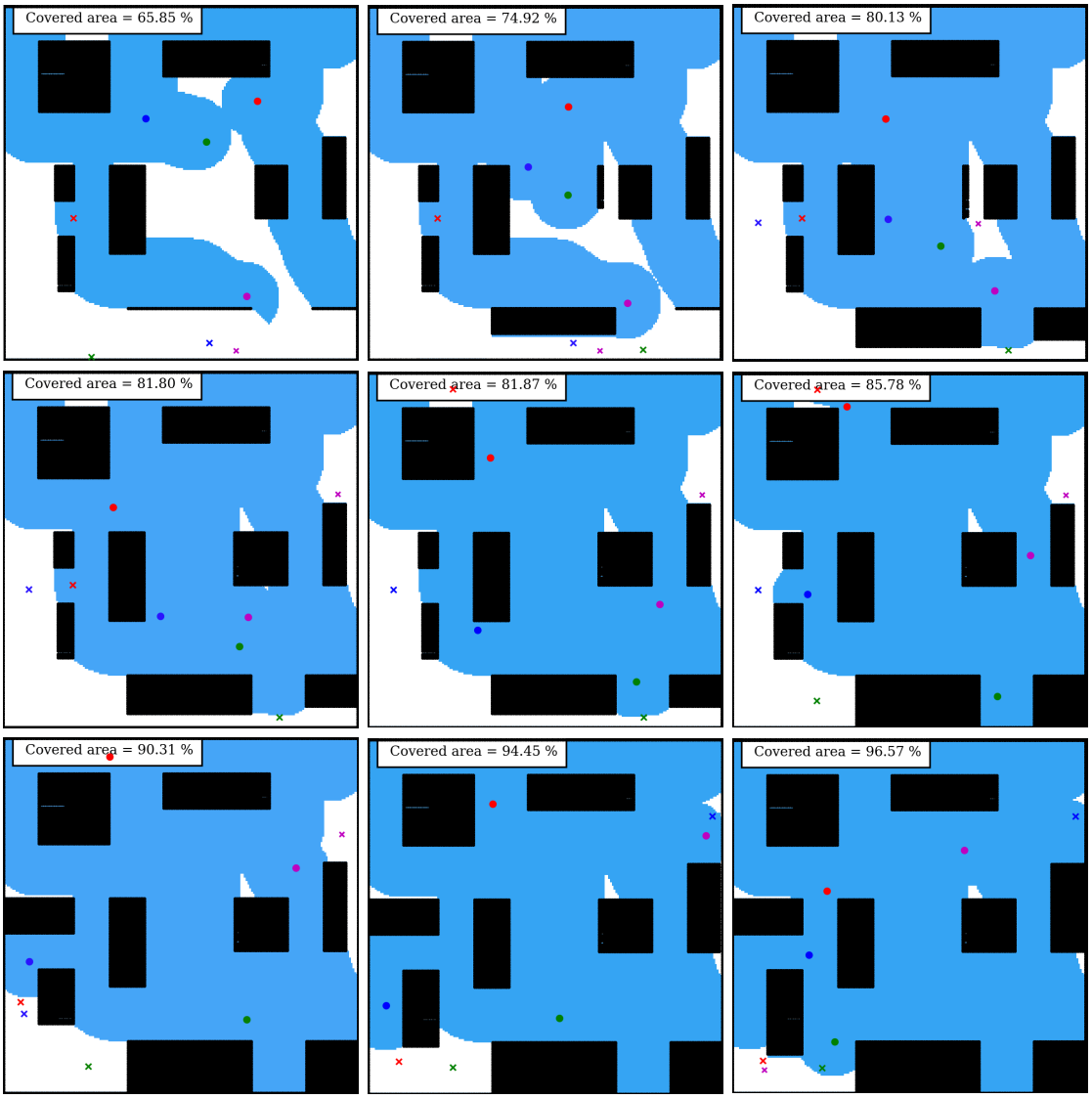


**Figure 6.8:** Exploration simulation using the RL agent + k-means algorithm (part 2).

For what regards the RL + k-means coverage algorithm displayed in the simulation of figures 6.7 and 6.8, the switch between the two coverage methods happens after 70% of the total environment surface has been explored (which happens between the sixth and seventh frames of the simulation). In the first six frames, the goal placing of the coverage algorithm can be observed. Some interesting behaviours can be observed. First of all, the agent is able to differentiate the goal location from one UAV to the other, which is a first good result. As can be observed, the temporary goal placing is not always optimal. For example, in the first frame the red goal is too close to the blue UAV. However, the goal placement is effective to initiate the exploration process. It must be said that in this first section of the training it is difficult for the UAV to learn a consistent policy. This has a simple reason: since the environment is completely unexplored, almost any goal positioning leads to a positive reward in the training process. In this case, the objective for the agent is to learn an efficient long-term strategy so that the placement of the initial goals positively influences the future exploration, by positioning the UAV in a way where it is able to continuously observe new areas, without having to go

back over an already-explored region. Unluckily, behaviours like this are learned through many repetitions of the same scenarios, which means they require many training episodes. As already discussed, this was not the case in the coverage agent training, and so the initial goal placement is not completely optimised. In frames 5 and 6 the waypoints are moved since the previous ones are reached. Here it can be observed that the blue, purple and green agents decide to move toward the bottom side of the map, which is the most unexplored one. This, in general, is a smart choice. However, the fact that all three UAVs start moving toward the same region is inefficient (especially since the purple UAV is much closer to the objectives than the other two). The reason for this behaviour lies in the choice to make the agent a *greedy* and *selfish* one (so each of the UAVs just wants to reach the biggest unexplored area). All these observations can be very useful to optimise a future training process that could allow to obtain a much more efficient coverage agent. In any case, despite the flaws detected, the agent policy is able to quickly reach an exploration percentage equal to 70% of the total environment surface area.

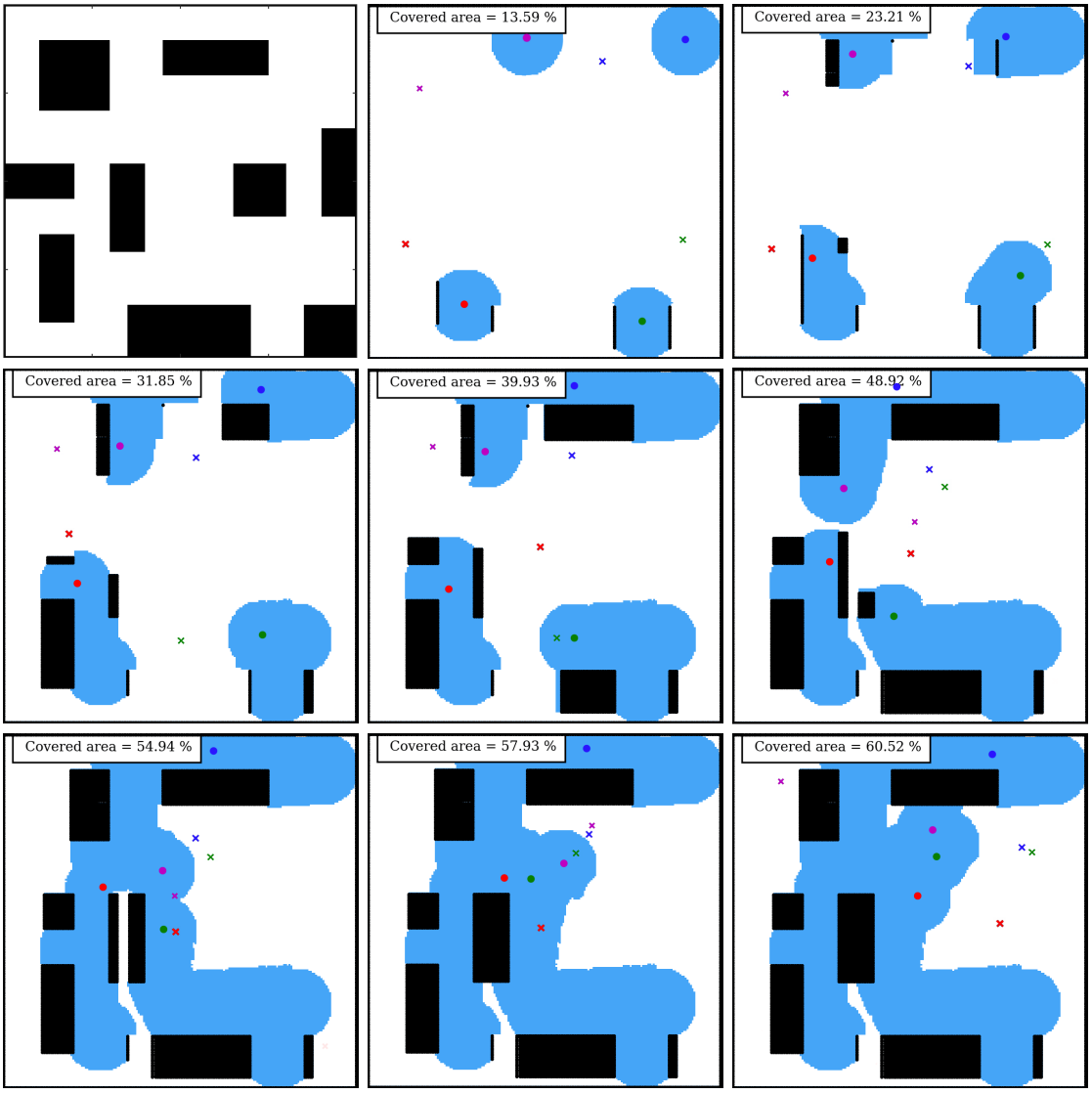


**Figure 6.9:** Exploration simulation using the k-means coverage algorithm.

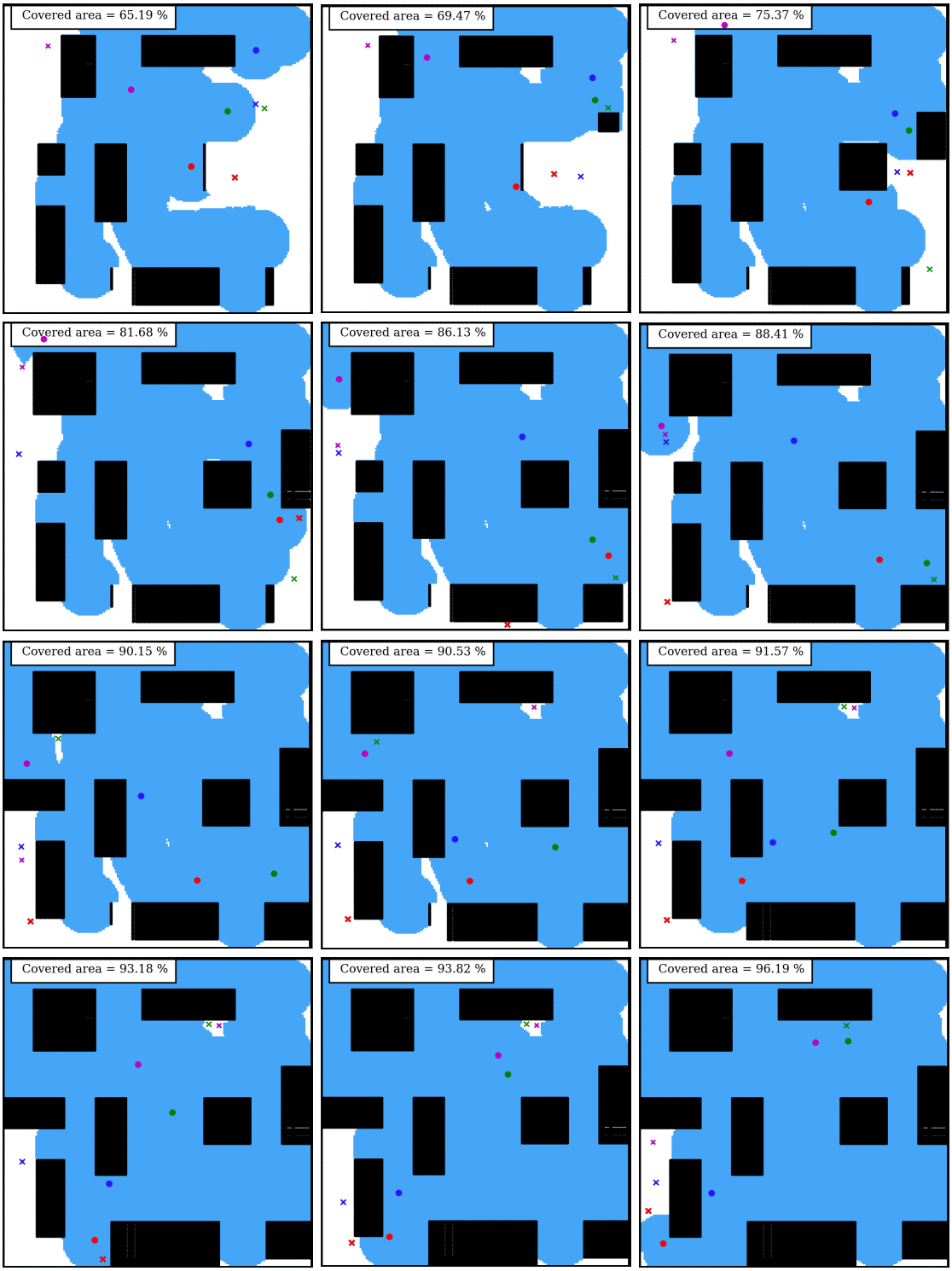


**Figure 6.10:** Exploration simulation using the k-means coverage algorithm (part 2).

Some observations can be made also for what regards the k-means coverage agent, used in the second part of the first simulation (figures 6.7 and 6.8) and in the whole second simulation (figures 6.9 and 6.10). The k-means algorithm computes an adaptive number of clusters, which is computed through a dedicated piece of algorithm that aims to find the best trade off between

the number of clusters and the in-cluster variance (see appendix B for more details). The k-means coverage algorithm returns a list of points, corresponding to the centroids of the Voronoi cells in which the environment is clustered. From the list, each UAV selects the closest point as chooses it as temporary goal. Also in this case, the placing of the points is not optimal. This is particularly true at the beginning of the exploration process, as can be observed in the second simulation. The k-means coverage algorithm is, instead, quite efficient towards the end of the exploration process, where it is effective in localising and isolating the single unexplored regions and assigning goals to them. This can be observed particularly well in the first simulation. It is due to the good results that this algorithm has shown in the completion of the exploration process that it has been decided to pair it with the RL agent to improve the performances of both methods. It is worth mentioning, at this point, that the reason why the RL coverage agent is not efficient after 70% of the environment has been explored is that the final stages of the exploration process can be obtained only if a good policy is used for the first part. To reach this point, it is necessary that the training process is long enough. In fact, the policy learns gradually how to behave in later scenarios of the exploration, and so only a long training process can lead to learning efficient end-simulation behaviours. The shortness of the training process performed on the coverage agent is, once again, a significant limitation.

### 6.3.3 Exploration results

A number of simulations have been performed in the validation maps to evaluate the efficiency of both the RL coverage agent and the k-means one. In particular, both the single k-means coverage algorithm and the combined RL + k-means one have been tested in all three validation maps. The simulation results in map A have been shown in figures 6.7 and 6.8 for the combined agent and in figures 6.9 and 6.10 for the k-means alone. The same simulation process has been repeated in maps B and C. From all the simulations, numeric values have been collected for all the evaluation metrics introduced in section 6.3.1. The numeric results obtained are collected in table 6.2.

**Table 6.2:** Numerical results obtained from the coverage simulation in the test environments. The values of $\hat{d}$ and $d_{min}$ are the average of the correspondent metrics over all the simulation. The values of $t_{N\%}$ are expressed as a number of simulation steps.

| Test | Algorithm | Map | $A\%$ | $\hat{d}$ [m] | $\sigma(\hat{d})$ [m] | $d_{\text{min}}$ [m] | $t_{50\%}$ | $t_{70\%}$ | $t_{80\%}$ |
|---|---|---|---|---|---|---|---|---|---|
| 1 | k-means | A | 95.01 | 9.51 | 3.36 | 1.18 | 96 | 169 | 246 |
| 2 | | B | 95.70 | 12.05 | 3.88 | 1.58 | 114 | 224 | 271 |
| 3 | | C | 93.24 | 9.94 | 3.95 | 1.16 | 96 | 161 | 230 |
| 4 | RL + k-means | A | 93.64 | 10.27 | 3.24 | 1.62 | 82 | 148 | 228 |
| 5 | | B | 95.58 | 8.90 | 2.88 | 1.64 | 93 | 156 | 211 |
| 6 | | C | 93.25 | 9.64 | 3.35 | 2.34 | 84 | 140 | 255 |

For each of the six tests performed the coverage algorithm used and map are reported in table 6.2. In addition, the value of all the metrics is reported in the table. The variable $\sigma(\hat{d})$ is the standard deviation of the average distance between UAVs, and gives a quantitative evaluation of how much the fleet is spread in the environment (a low value is associated to a packed fleet, a high value to a well-spread one). The time after which certain exploration thresholds (50%, 70% and 80% of coverage) are reached is reported, expressed as number of simulation steps after which the desired percentage is reached. From the table it can be observed that in all six tests the coverage algorithm employed is able to reach a coverage percentage value between 93% and 95%. The results collected in the last three columns can be plotted to compare them more easily. The resulting graphic can be observed in figure 6.11.

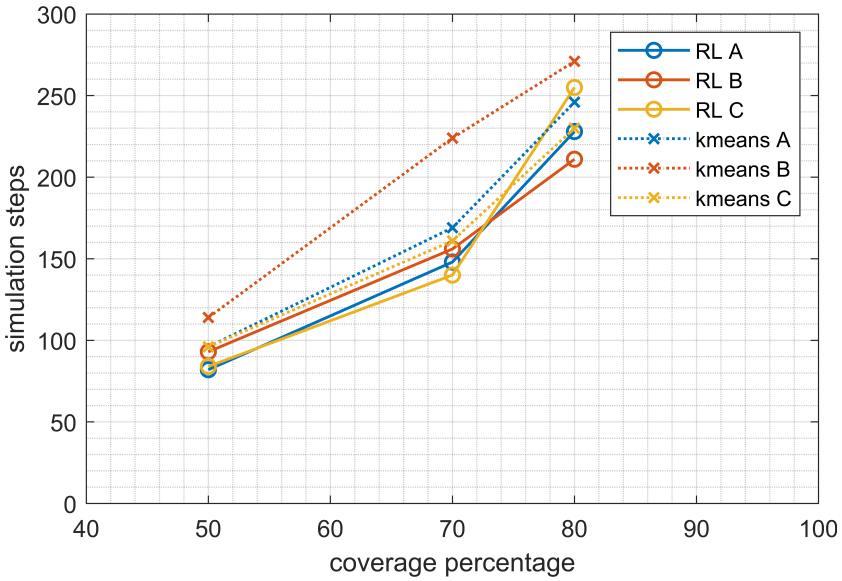


**Figure 6.11:** Comparison of the time employed to reach some coverage percentage thresholds in the six exploration tests.

The three full lines represent the three tests performed using the combined RL+k-means coverage algorithm (indicated in the legend just as "RL"). On the contrary, the dotted lines correspond to the tests where only the k-means algorithm was used. The plot shows on the $x$ axis the coverage percentage and on the $y$ axis the number of simulation steps employed to reach each threshold (50%, 70% and 80%). A lower number of steps corresponds to a faster exploration process. As it can be observed, the combined algorithm performs better than the k-means one in almost every test. In particular, the points corresponding to the 70% threshold are always lower for the RL+k-means algorithm. It is worth noting that, up to 70% of the exploration, the coverage is driven only by the RL coverage agent. It results that the coverage agent is more efficient than the k-means one a the beginning of the exploration, confirming the proposed model to be viable to coordinate the exploration task. Some additional information about the exploration speed can be obtained by plotting, for a single test, the coverage percentage of each UAV as a cumulative plot. Two of these plots are analysed. The first is obtained plotting the coverage percentage of each UAV during test number 1 of table 6.2 (which is the one represented in figures 6.9 and 6.10), whereas the second corresponds to test 4 (obtained using the combined coverage algorithm and displayed in figures 6.7 and 6.8). The two coverage percentage plots are shown in figure 6.12.

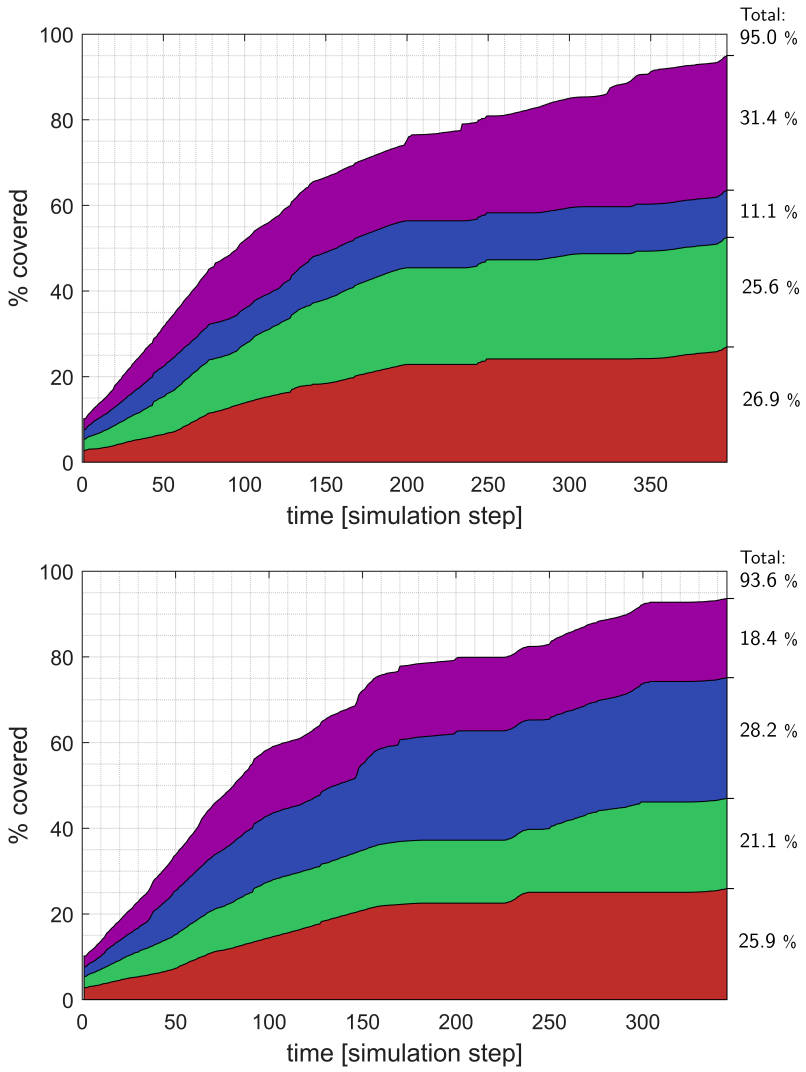


**Figure 6.12:** Coverage percentage of the single UAV contributions in the exploration process for tests 1 and 4. The plot on the top shows the results of test 1, obtained in map A using the only k-means coverage algorithm. The one on the bottom shows results of test 4, which is performed using the combined coverage agent. The colours correspond to those of the UAVs represented in the simulations of figures 6.9, 6.10, 6.7, and 6.8.

As can be observed, in both the exploration process the coverage percentage ends above 90%. In case of the combined algorithm (bottom plot), the exploration is very well distributed amongst the four members of the fleet. In the case where only the k-means was used, instead, one of the UAVs contributes significantly less with respect to the other ones. In fact, in the plot it can be observed that the blue region is much smaller than the other ones. This is due to the fact that the k-means algorithm is not optimised to coordinate the exploration process, and produces simply viable locations for the goals. In the bottom plot, where the coverage process was started with the RL agent, the coordination of the fleet is much more efficient.

Some useful information can be obtained also by observing the column $\hat{d}$ of table 6.2. As can be observed, the average distance between the UAVs of the fleet is around 10m, whereas the standard deviation associated to it is $\sim$3.3m. This is a good result, and means that the fleet is well spread in the environment. The only issue is that, as can also be observed in the simulation figures above, sometimes a couple of the UAVs tend to stick together and move close one to the other for some time. This is a kind of behaviour that should be punished during the training process. However, early results are good in this sense.

# 7. 3D extension

The exploration algorithm has been developed in a 2D environment. This decision has been made mainly to keep all the development stages as simple as possible, as well as to reduce the agent training complexity and time. Since the real world is better represented as a 3D environment, some considerations have been made about a possible 3D extension of the algorithm. Only a small part of the 3D extension code has been actually implemented, mainly due to time constraints in the thesis development. Some ideas for the algorithm 3D extension are collected here as a starting point for any further development of the code.

## 7.1 3D extension strategies

The use of the proposed algorithm in a real-world application requires some adaptations to work in a 3-dimensional space. In this section, some space is dedicated to the different strategies that have been considered for this extension. The most straightforward approach for a 3D extension of the algorithm is to "augment" all 2D matrices in 3D matrices. This way, the matrix representing the environment would become a 3D matrix, with an extra dimension to represent height and each element representing a volume of space instead of an area. The implementation of this strategy is quite easy, starting from the code developed for the 2D case. However, this approach brings with it some issues that need to be solved.

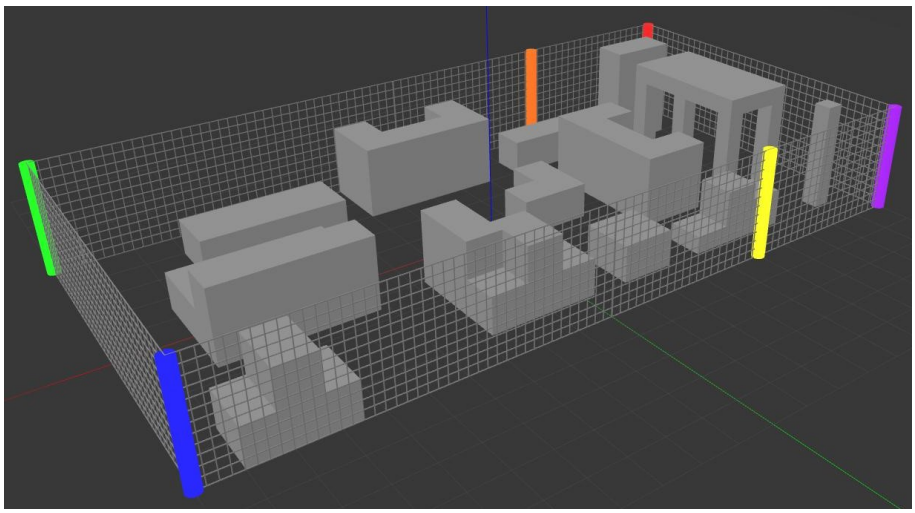

**Figure 7.1:** Example of a 3D environment in the ROS framework, where 3D simulations could be performed.

First of all, memory usage grows quickly. In particular, the dependence of the number of APF values in the environment model on the resolution grows as $n^3$ instead of $n^2$. This can lead to a massive memory usage increase, as well as to a considerable slow-down in the computational times. An increase in the environment model dimension (e.g. 20m of the side instead of 10m) leads to the same exponential growth in memory usage. This needs to be taken into account when initialising the environment model. A possible solution to keep memory usage low could be to create two different environment models, one with high dimensions but low resolution, and the other with small dimension and higher resolution. The former one would be used for global path planning operations, while the latter one would be employed to locally optimise the trajectory. The drawback, in this case, would be that the local environment would need to be continuously updated to match the current UAV position and neighbourhood. The update could require a significant amount of time, so this solution would need some preliminary study and testing before getting implemented.

A second problem is linked to the motion model used for the path planning operations. In the 2D model, the movement direction obtained as output from the RL agent was simply an angle, corresponding to the direction (in absolute coordinates) toward which to move. In the 3D extension, a new motion model has to be introduced. The most immediate extension of the motion model used in the 2D case would be spherical coordinates, adding a second angle to the agent output to represent the angle between the trajectory and the horizontal plane. However, some other possibilities could be considered, both to take into account some dynamic properties of the UAVs and to avoid the numerical problems that spherical coordinates bring with them (in particular, singular configurations of the angles).

A different approach that could be used, especially for an initial implementation of the 3D navigation algorithm, is to "slice" the environment in a number of sections along the $z$ axis and treat each one of them as a separated 2D environment. This strategy allows to use the exact same algorithm proposed for the 2D environment, and requires only a small portion of additional code to switch from one level to the other. This strategy is very easy to implement starting from the 2D algorithm developed in the course of the thesis, at the cost of some efficiency in the exploration process since the coverage and path planning are optimised only with respect to each level and not globally. However, for environments where obstacle shapes are constant along $z$ (like in an urban environment, where building shapes are constant amongst many levels) this simplified 3D algorithm could produce satisfactory results. Some simulations using this approach have been performed. An example is shown in figure 7.2. The simulation shows a fleet of 4 UAVs exploring a 3D environment by slicing it in four different layers along the $z$ axis. Each UAV flies only on one of these layers, i.e., it keeps a constant altitude with respect to the ground. The four layers are located respectively at 1, 2, 3, and 4 meters from the ground. While flying, each UAV observes the environment in front of it to detect obstacles. A slightly modified version of the vision function is used, to be able to observe a conic shape in space (and not just in the flight plane).

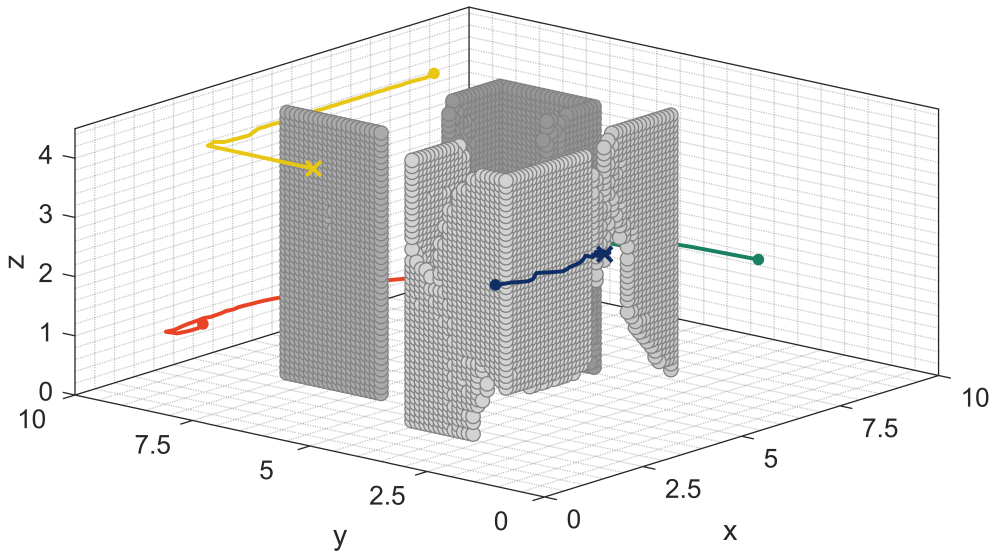


**Figure 7.2:** Example of 3D simulation with "slicing" along the $z$ axis.

### 7.1.1 Neural Networks in the 3D extension

The most crucial elements that need to be adapted for a 3D extension are the Neural Networks from which the path planning and coverage agents are built. In particular, the two sections that need to be modified are the input section and the output one. The latter requires a rather simple modification. In the path planning NN, for example, the number of output nodes just needs to be increased by 1, in order to get two angles as output (i.e., the two angles defining the direction of the vector in spherical coordinates) instead of one. The modification of the input layer, instead, is a bit more complex. The input section of the NN, before the dense layers, is composed of a succession of convolutional layer mixed with pooling and normalisation layers. All these layers need to be adapted to a 3D input. For example, the filters of the convolutional layers have to be defined as 3-dimensional filters, since they have to multiply a portion of input having 3 dimensions. A drawback of this fact is that the number of weights composing the NN grows significantly,[23] which leads to an increase in the training time required for the NN to reach the parameter convergence. Some time must be dedicated to the study of a suitable NN architecture for the 3D scenario in order to avoid ending up with a very slow training process (as well as a slow feed-forward execution of the NN once it is trained, which would imply a long computational time required to obtain the trajectory).

[23] Each filter is associated to a number of weights equal to $n^3$ (in case of a cubic filter, having dimension $n \times n \times n$) instead of $n^2$ as in the 2D case.

# 8. Conclusions

The conclusions of this thesis work are discussed in this chapter. First of all, a brief analysis of the main limits and issues encountered during the development and validation process is made (section 8.1). After this, the main open points on which future work should focus are discussed (section 8.2). In the end, a complete overview of the conclusion is presented (section 8.3).

## 8.1 Issues

The algorithm proposed to present some issues that negatively influence its efficiency. The main issues in the algorithm design and implementation process are listed here. Aside from the limitations posed by the initial assumptions and simplifications (see section 4.1), all the issues are somewhat linked to the use of Neural Network and Reinforcement Learning methods and to the intrinsic limitations they have.

- One of the main issues of the method proposed lies in the use of Reinforcement Learning. In fact, while being a very powerful and versatile tool, RL (and more in general Artificial Neural Networks) presents some critical issues. One of them is the fact that during training it is impossible to provide the model all the possible inputs it could meet, and therefore there is always the possibility that some "strange" new input leads to unwanted or dangerous behaviour. A consistent solution to this problem is to pair the RL algorithm with an explicitly-programmed system that catches any unwanted behaviour and fixes it in a safe way. However, the design of such a system is not always easy and negates some benefits of the RL methods (namely, the fact that it is not necessary to explicitly implement any desired behaviour). In the proposed algorithm these "safety" algorithms have not been implemented, but they would probably be necessary for a real-world application;

- a second problem linked to the intrinsic issues of RL is that when an RL agent is implemented it has to be treated as a black box. In fact, due to how NNs are built, it is nearly impossible to understand their "internal reasoning". It is not easy to correlate weight values and NN architecture to the way the agent processes the inputs. This means that, when a model works correctly, it is very difficult to understand *why* it does so. Unluckily, the same holds also when it does *not* work correctly, and this makes it very difficult to debug and correct flawed models. In particular, this reflects the difficulty to find a suitable reward function in the training process. The training algorithm, in fact, just aims to maximise its reward, so the *wanted* behaviour must match the *encouraged* behaviour (i.e. the behaviour associated to the maximum reward). This problem, overall, results in a difficult trial-and-error design process for what regards the state, reward function and NN architecture;

- a third issue, still linked to NNs, is the fact that NN working is constrained to a specific input typology. This leads to a loss of generality. For instance, each path planning agent trained in section 5.2 has a very specific input size. If, for any reason, the size of such input has to change (e.g. to increase the "sense radius" of the agent), a brand new agent has to be trained, as it would be difficult to adapt the already existing one to the new input shape. This leads to some relevant limitations in terms of algorithm versatility. The piece of algorithm mostly influenced by this factor is the coverage algorithm. In this case, in fact, the input shape is connected to the environment dimension, limiting the usability of an agent to the environments with the same dimension. Some solutions have been implemented to tackle this issue, reducing its influence (e.g., the possibility to "compress" the coverage agent input using a dedicated pooling layer). However, in some aspects the generality of the algorithm proposed had to be reduced due to this limitation of NNs.

## 8.2 Future work

This thesis work can be considered complete, which means that the algorithm performs all the required operations with a satisfying level of efficiency. However, there is room for improvement and further development of the proposed method. The most interesting aspects in which it is worth continuing the research to improve the algorithm performances are listed here:

- there is still much room for improvement in terms of network training. In fact, starting from the design of the networks themselves, different aspects of the learning process can be optimised. The most important aspect to work on is probably the research of a better reward function, capable of driving the agent to learn more complex policies (e.g., for the path planning agent, this would mean to produce a single agent capable of managing all kinds of local minima). Other than this, larger datasets could help the learning process, while dedicated learning sessions with particular environment configurations could help learn specific behaviours (e.g. the aforementioned management of local minima in the case of the path planning agent);

- another aspect to consider to improve the agent performances is to try learning algorithms different from DDPG. In fact, while DDPG is the simplest learning algorithm able to deal with a continuous action space, other algorithms like T3D or PPO (see figure 3.5) represent more advanced and efficient learning tools that could lead to a faster and more stable learning process;

- in general, an important goal of the future work on the algorithm is its extension to a 3D environment. A detailed discussion of this topic was already made in chapter 7;

- both for the 2D and 3D versions of the algorithm, a good final point of the development process would be to perform some real-world tests. This kind of test could start from some simulations performed in the Robotic Operating System (ROS) and Gazebo frameworks. These simulations would allow to take into account many real-world phenomena that have not been modelled in the training environment. After this, real-world tests could allow to further evaluate the goodness of the algorithm. Real-world simulations would also ensure the compatibility of the algorithm with all the other applications running on the UAV Operative System in parallel to it.

## 8.3 Conclusions

The proposed exploration model has been illustrated and tested in all of its parts. The design process of the algorithm and its final structure has been discussed. Particular attention has been dedicated to the design and training of the two RL agents of which the algorithm is composed. In fact, the two RL agent represent the core, as well as the novelty, of the proposed algorithm. It has been shown that very good results have been obtained for the path planning agent, which performed well even when compared with state-of-art algorithms like A$^*$. Some work is still necessary before being able to deploy the path planning algorithm in a real-world application, but the results obtained so far are very promising. For what regards the coverage agent, a longer and more complex design process has been carried out. Some early results have been obtained in the training of the coverage RL agent, which show that the approach followed is viable and can lead to an efficient exploration of an unknown environment. Due to the excessive length of the required training process, the obtained coverage agent is not completely optimised with respect to the exploration task. However, as in the case of the path planning agent, the results obtained so far are consistent and very promising. With some further optimisation and improvements, the algorithm can become a very powerful exploration tool for autonomous UAV fleets.

# Acknowledgements

Some parts of this work required quite a lot of computational resources, mainly to speed up the development of Reinforcement Learning agents. I would like to thank Pietro and Massimiliano for allowing me to access their computers overnight and have them train some initial (and very bad) agents. I would also like to thank HPC@POLITO for providing a significant part of the computational resources used, especially in more advanced development stages. HPC@POLITO is a project of Academic Computing within the Department of Control and Computer Engineering at the Politecnico di Torino (http://hpc.polito.it).

# List of Figures

# List of Tables

# List of Algorithms

# Bibliography


[1] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo, *Robotics*. London: Springer London, 2009.

[2] A. Tsourdos, B. White, and M. Shanmugavel, *Cooperative path planning of unmanned aerial vehicles*. Chichester West Sussex U.K. and Hoboken N.J.: Wiley, 2011.

[3] M. W. Otte, "A survey of machine learning approaches to robotic path-planning," 2009. Department of Computer Science - University of Colorado at Boulder.

[4] K. Valavanis and G. J. Vachtsevanos, eds., *Handbook of unmanned aerial vehicles*. Dordrecht: Springer Reference, 2015.

[5] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[6] A. Stentz, "Optimal and efficient path planning for unknown and dynamic environments," *International Journal Of Robotics and Automation*, vol. 10, pp. 89–100, 1993.

[7] O. Khatib, "Real-time obstacle avoidance for manipulators and mobile robots," in *Proceedings. 1985 IEEE International Conference on Robotics and Automation*, vol. 2, pp. 500–505, 1985.

[8] A. Mirshamsi, S. Godio, A. Nobakhti, S. Primatesta, F. Dovis, and G. Guglieri, "A 3d path planning algorithm based on pso for autonomous uavs navigation," *BIOMA 2020 - The 9th International Conference on Bioinspired Optimisation Methods and Their Applications*, 2020.

[9] P. Vadakkepat, K. C. Tan, and W. Ming-Liang, "Evolutionary artificial potential fields and their application in real time robot path planning," in *Proceedings of the 2000 Congress on Evolutionary Computation. CEC00 (Cat. No.00TH8512)*, pp. 256–263, IEEE, 16-19 July 2000.

[10] B. Zhou, F. Gao, J. Pan, and S. Shen, "Robust real-time uav replanning using guided gradient-based optimization and topological paths."

[11] B. Zhou, F. Gao, L. Wang, C. Liu, and S. Shen, "Robust and efficient quadrotor trajectory generation for fast autonomous flight," *IEEE Robotics and Automation Letters*, vol. 4, no. 4, pp. 3529–3536, 2019.

[12] X. Lei, Z. Zhang, and P. Dong, "Dynamic path planning of unknown environment based on deep reinforcement learning," *Journal of Robotics*, vol. 2018, pp. 1–10, 2018.

[13] Q. Yao, Z. Zheng, L. Qi, H. Yuan, X. Guo, M. Zhao, Z. Liu, and T. Yang, "Path planning method with improved artificial potential field - a reinforcement learning perspective," *IEEE Access*, vol. 8, pp. 135513–135523, 2020.

[14] J. F. Kennedy, R. C. Eberhart, and Y. Shi, *Swarm intelligence*. The Morgan Kaufmann series in evolutionary computation, San Francisco: Morgan Kaufmann Publishers, 2001.

[15] D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, L. Rutkowski, M. Korytkowski, R. Scherer, R. Tadeusiewicz, L. A. Zadeh, and J. M. Zurada, *Swarm and Evolutionary Computation*, vol. 7269. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[16] J. Xiao, G. Wang, Y. Zhang, and L. Cheng, "A distributed multi-agent dynamic area coverage algorithm based on reinforcement learning," *IEEE Access*, vol. 8, pp. 33511–33521, 2020.

[17] H. University, "Programmable self-assembly in a thousand-robot swarm." https://news.harvard.edu/gazette/story/2014/08/the-1000-robot-swarm/. Accessed: 2021-05-29.

[18] M. S. Innocente and P. Grasso, "Self-organising swarms of firefighting drones: Harnessing the power of collective intelligence in decentralised multi-robot systems," *Journal of Computational Science*, vol. 34, pp. 80–101, 2019.

[19] Y. Jia, J. Du, W. Zhang, and L. Wang, "Three-dimensional leaderless flocking control of large-scale small unmanned aerial vehicles," *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 6208–6213, 2017.

[20] T. Kuyucu, I. Tanev, and K. Shimohara, "Superadditive effect of multi-robot coordination in the exploration of unknown environments via stigmergy," *Neurocomputing*, vol. 148, pp. 83–90, 2015.

[21] A. L. Alfeo, M. G. Cimino, and G. Vaglini, "Enhancing biologically inspired swarm behavior: Metaheuristics to foster the optimization of uavs coordination in target search," *Computers & Operations Research*, vol. 110, pp. 34–47, 2019.

[22] K. Chang, Y. Xia, and K. Huang, "Uav formation control design with obstacle avoidance in dynamic three-dimensional environment," *SpringerPlus*, vol. 5, no. 1, p. 1124, 2016.

[23] Q. Chen, Y. Meng, and J. Xing, "Shape control of spacecraft formation using a virtual spring-damper mesh," *Chinese Journal of Aeronautics*, vol. 29, no. 6, pp. 1730–1739, 2016.

[24] J. N. Yasin, S. A. S. Mohamed, M.-H. Haghbayan, J. Heikkonen, H. Tenhunen, M. M. Yasin, and J. Plosila, "Energy-efficient formation morphing for collision avoidance in a swarm of drones," *IEEE Access*, vol. 8, pp. 170681–170695, 2020.

[25] École Polytechnique Fédérale de Lausanne, "Drone swarms avoid obstacles without collision." https://www.youtube.com/watch?v=cAXUKNGpMG4. Accessed: 2021-05-19.

[26] M. R. Brust and B. M. Strimbu, "A networked swarm model for uav deployment in the assessment of forest environments," *aXiv*, pp. 1–6, 2015.

[27] S. Sudhakar, V. Vijayakumar, C. Sathiya Kumar, V. Priya, L. Ravi, and V. Subramaniyaswamy, "Unmanned aerial vehicle (uav) based forest fire detection and monitoring for reducing false alarms in forest-fires," *Computer Communications*, vol. 149, pp. 1–16, 2020.

[28] V. C. Moulianitis, G. Thanellas, N. Xanthopoulos, and N. A. Aspragathos, "Evaluation of uav based schemes for forest fire monitoring," in *Advances in Service and Industrial Robotics* (N. A. Aspragathos, P. N. Koustoumpardis, and V. C. Moulianitis, eds.), vol. 67 of *Mechanisms and Machine Science*, (Cham), pp. 143–150, Springer International Publishing, 2019.

[29] O. Tkachuk, "Detailed design of a forest surveillance uav," 2018.

[30] P. Boccardo, F. Chiabrando, F. Dutto, F. G. Tonolo, and A. Lingua, "Uav deployment exercise for mapping purposes: Evaluation of emergency response applications," *Sensors*, vol. 15, no. 7, pp. 15717–15737, 2015.

[31] j. Zoto, M. A. Musci, A. Khaliq, I. Aicardi, and M. Chiaberge, "Automatic path planning for unmanned ground vehicle using uav imagery," *Advances in Service and Industrial Robotics*, vol. 980, pp. 223–230, 2019.

[32] M. Mammarella, G. Ristorto, E. Capello, N. Bloise, and G. Guglieri, "Waypoint tracking via tube-based robust model predictive control for crop monitoring with fixed-wing uavs," *2019 IEEE International Workshop on Metrology for Agriculture and Forestry*, 2019.

[33] NASA/JPL-Caltech, "Nasa's ingenuity mars helicopter succeeds in historic first flight. first video of nasa's ingenuity mars helicopter in flight." https://mars.nasa.gov/news/8923/nasas-ingenuity-mars-helicopter-succeeds-in-historic-first-flight/. Accessed: 2021-04-25.

[34] B. Balaram, T. Canham, C. Duncan, H. F. Grip, W. Johnson, J. Maki, A. Quon, R. Stern, and D. Zhu, "Mars helicopter technology demonstrator," *2018 AIAA Atmospheric Flight Mechanics Conference*, 2018.

[35] Ralph D. Lorenz, Elizabeth P. Turtle, Jason W. Barnes, and Melissa G. Trainer, "Dragonfly: A rotorcraft lander concept for scientific exploration at titan," *Johns Hopkins APL Technical Digest*, vol. 34, no. 3, 2018.

[36] N. S. Website, "Titan - quick facts." https://solarsystem.nasa.gov/moons/saturn-moons/titan/in-depth/. Accessed: 2021-4-27.

[37] L. Zhou and W. Li, "Adaptive artificial potential field approach for obstacle avoidance path planning," in *2014 Seventh International Symposium on Computational Intelligence and Design*, pp. 429–432, IEEE, 13/12/2014 - 14/12/2014.

[38] M. Gronemeyer and J. Horn, "Collision avoidance for cooperative formation control of a robot group," *IFAC-PapersOnLine*, vol. 52, no. 8, pp. 434–439, 2019.

[39] R. L. Galvez, G. E. U. Faelden, J. M. Z. Maningo, R. C. S. Nakano, E. P. Dadios, A. A. Bandala, R. R. P. Vicerra, and A. H. Fernando, "Obstacle avoidance algorithm for swarm of quadrotor unmanned aerial vehicle using artificial potential fields," in *TENCON 2017 - 2017 IEEE Region 10 Conference*, pp. 2307–2312, IEEE, 05/11/2017 - 08/11/2017.

[40] J. Sun, J. Tang, and S. Lao, "Collision avoidance for cooperative uavs with optimized artificial potential field algorithm," *IEEE Access*, vol. 5, pp. 18382–18390, 2017.

[41] W. Ertel, *Introduction to Artificial Intelligence*. Springer International Publishing, 2nd ed., 2017.

[42] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. Adaptive computation and machine learning series, Cambridge Massachusetts: The MIT Press, second edition ed., 2018.

[43] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals and Systems*, vol. 2, no. 4, pp. 303–314, 1989.

[44] OpenAI, "Kinds of rl algorithms." https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html. Accessed: 2021-05-30.

[45] 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."

[46] OpenAI, "Deep deterministic policy gradient." https://spinningup.openai.com/en/latest/algorithms/ddpg.html. Accessed: 2021-05-30.

[47] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods."

[48] G. Barth-Maron, M. W. Hoffman, D. Budden, W. Dabney, D. Horgan, D. TB, A. Muldal, N. Heess, and T. Lillicrap, "Distributed distributional deterministic policy gradients."

# Appendix

# A. Neural Networks, Backpropagation & Stochastic Gradient Descent

## A.1 Neural Networks

As introduced in section 3.2.3, Neural Networks are functions composed of a layered structure of *neurons* connected between them. An example of a simple Neural Network is shown in figure A.1 (the image is the same as figure 3.3 already seen in section 3.2.3).

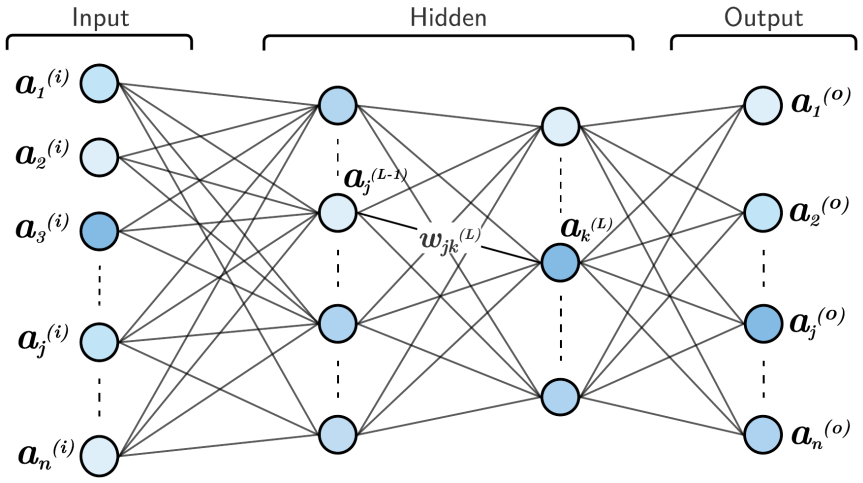


**Figure A.1:** Simple Neural Network scheme. Input, output and hidden layers are put in evidence. In general, any number of hidden layers can be used (copy of figure 3.3).

The NN function takes $N_i$ inputs, assigned to the input neurons, that are elaborated through the layered structure of the NN to return $N_o$ outputs. Each neuron performs a single operation. This operation involves a weighted sum of the values held by the precedent layer neurons, as well as a (usually simple) non-linear function, called the *activation function*. The operation performed by each neuron to compute its value is:

$$a_k^{(L)} = \sigma\left(\sum_{j=1}^{n_{L-1}} w_{jk}^{(L)} \cdot a_j^{(L-1)} + b_k^{(L)}\right) \qquad \forall\, k = 2...N_L \tag{A.1}$$

$a_k^{(L)}$ is the value of the $k$-th neuron of layer $L$; its value is computed through the activation function $\sigma$, whose argument is the weighted sum of the values of the neurons of layer $L-1$. The weights of the sum are denoted as $w_{jk}^{(L)}$ (which represents the weight between neuron $j$

of layer $L-1$ and neuron $k$ of layer $L$), while $b_k^{(L)}$ is a bias associated to each neuron. $N_L$ represents the total number of layers, where the first layer is the input one and the last is the output layer. $k$ starts from 2 since the input layer values are already defined and do not need to be computed through this equation. Equation (A.1) can be rewritten in matrix form, collecting all the neurons of a layer in the column vector $\boldsymbol{a}$ and the weights in the matrix $\boldsymbol{W}$:

$$\boldsymbol{a}^{(L)} = \sigma(\boldsymbol{W}^{(L)}\boldsymbol{a}^{(L-1)} + \boldsymbol{b}^{(L)}) \tag{A.2}$$

Clearly, to compute $\boldsymbol{a}^{(L-1)}$ it is necessary to apply the same equation using $\boldsymbol{a}^{(L-2)}$ as input. More in general, the neuron values of each layer are computed by recursively applying equation (A.2) for all precedent layers. The output of the NN is obtained by applying equation (A.2) $N_L - 1$ times. In this way, the analytical equation of the output vector becomes quite complex, but can be written in a compact way as:

$$\boldsymbol{a}^{(o)} = g(\boldsymbol{a}^{(i)}) \tag{A.3}$$

It is worth mentioning that the activation function $\sigma(z)$ can be chosen arbitrarily; the only important property that it must have is to be nonlinear. In fact, the combination of the non-linear activation function and the layered structure allows a NN to approximate any desired function. Two common choices of activation function are the sigmoid function and the rectifying linear unit (ReLU), represented in Figure A.2.

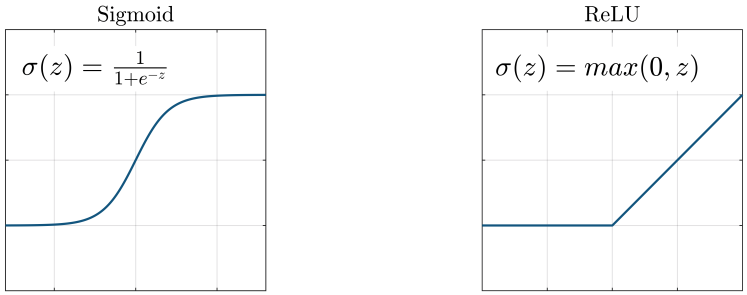


**Figure A.2:** On the left, the sigmoid activation function. On the right, the ReLU.

Weights $w$ and biases $b$ represent all the parameters that determine the behaviour of a NN. In fact, given a NN architecture, the approximation of any desired function is obtained by properly selecting all the parameters. The vector of all the parameters of a NN is indicated as $\theta$. The issue is that the parameters space is, in general, a space with very high dimension (it is quite common to find parameter spaces containing a number of parameters between $10^3$-$10^6$).

## A.2 Stochastic Gradient Descent

Once defined the structure and the working principle of a NN, the only remaining problem is to find the right parameters to put in it. There is no way to perform this operation analytically. First of all, because the analytical expression of the NN is very complex (equation (A.3) hides a lot of this complexity); secondly, the parameter space has very high dimension, which combined with the non-linearity of function $g$ makes it difficult to find a solution to the problem. Therefore, to find the parameters that allow to best approximate a desired function (or *target function*) $\hat{g}(\boldsymbol{x})$, a numerical optimisation method is employed. The optimisation method commonly

used to optimise NN parameters is the Stochastic Gradient Descent (SGD). The goal of this optimisation method is to find a set of parameters $\theta$ so that:

$$g_\theta(\boldsymbol{x}) \approx \hat{g}(\boldsymbol{x}) \tag{A.4}$$

i.e. the NN function must approximate the target function as close as possible. To reach this goal, a minimisation process is performed. The quantity to be minimised is the error between the outputs of the two functions, i.e.:

$$\boldsymbol{e} = \hat{g}(\boldsymbol{x}) - g_\theta(\boldsymbol{x}) \tag{A.5}$$

The quantity $\boldsymbol{e}$ is the difference between the outputs of the target function $\hat{g}$ and the NN function $g$, which are two vectors; therefore, $\boldsymbol{e}$ is a vector of errors $e_i$. Starting from the error vector $\boldsymbol{e}$, the *loss function* $L$ can be defined:

$$L = \sum_{i=1}^{n_o} \Big( g_i(\theta) - y_i \Big)^2 \tag{A.6}$$

The loss is defined as the squared sum of the difference between each output element and its target value. Some relevant manipulations have been performed to obtain equation (A.6). First of all, each element of the target vector $\hat{g}(\boldsymbol{x})$ is indicated as $y_i$, with $i \in 1...n_o$ (being $n_o$ the length of the output vector). Secondly, the dependency of $g$ and $\hat{g}$ from $\boldsymbol{x}$ has been hidden, since the input is known when computing the loss function, and most importantly since the relevant variables in equation (A.6) is the parameter vector $\theta$. In fact, for a given input vector $\boldsymbol{x}$ the loss value is only function of $\theta$:

$$L = L(\theta) \tag{A.7}$$

At this point, the SGD approach aims to find $\theta$ in order to minimise the value of $L$ for any input $\boldsymbol{x}$, which is equivalent to finding the function that better approximates $\hat{g}(\boldsymbol{x})$. To do so, the gradient descent method is implemented. The basic idea of this method is to follow the negative direction of the gradient in order to reach the global minimum of the function $L$. The standard Gradient Descent optimisation method updates the parameters through the following equation:

$$\theta \leftarrow \theta - \alpha \nabla L(\theta) \tag{A.8}$$

The issue in equation (A.8) is that the gradient value cannot be computed analytically, since its expression is too complex to be solved. On the other hand, the numerical computation of the "true" gradient is infeasible too, since it would require to compute the loss gradient for every possible input $\boldsymbol{x}$ and average all the results, requiring a huge (if not infinite) amount of time. Therefore, instead of considering the "true" gradient, an approximation of it is computed. To do so, a small set of $n_m$ inputs is randomly sampled from the input domain, the gradient is computed for each one of them and the average of the results is taken as the gradient to be used in equation (A.8). The set of inputs is called *minibatch* and is indicated as $\mathcal{M}$.[24]

[24] The fact that an approximation of the gradient is used instead of the "true" one is the reason why this method is called *Stochastic* Gradient Descent. In fact, since the minibatch data are sampled randomly from the input domain, the gradient computed at each step differs some times more, other times less from the "true" gradient. This discrepancy between the true and approximated gradient, however, has some advantages. In fact, it allows to escape from local minima encountered in the minimisation process. The minibatch, however, must not be too small, since in that case the gradient error becomes too large and leads to instabilities in the minimisation process.

The approximated gradient is computed as in the following equation:

$$\nabla L(\theta) = \frac{1}{n_m} \sum_{i=1}^{n_m} \nabla L_i(\theta) \tag{A.9}$$

A further step can be made to speed up the gradient computation. In fact, for the linearity of the operators involved, the following equivalence holds:

$$\frac{1}{n_m} \sum_{i=1}^{n_m} \nabla L_i(\theta) = \nabla \left( \frac{1}{n_m} \sum_{i=1}^{n_m} L_i(\theta) \right) \tag{A.10}$$

i.e. the average of the loss gradients is equal to the gradient of the losses average. This equivalence allows to perform the gradient computation only once, cutting a lot of time-expensive computations. The right-hand term of equation (A.10) can be rewritten in a more compact way by defining the *cost function* $C$:

$$C(\theta) = \frac{1}{n_m} \sum_{j=1}^{n_m} L_j(\theta) = \frac{1}{n_m} \sum_{j=1}^{n_m} \left[ \sum_{i=1}^{n_o} \Big( g_i(\theta) - y_i \Big)^2 \right]_j \tag{A.11}$$

where the rightmost term is found by substituting the definition of the loss function seen in equation (A.6). The cost function is the one used in practice to compute the approximated gradient, since it requires to compute the gradient only once. The SGD iterative procedure to update the parameters becomes:

$$\theta \leftarrow \theta - \alpha \nabla C_m(\theta) \tag{A.12}$$

which is just a more efficient version of equation (A.8). The parameter $\alpha$, which appeared also in the previous version of this equation, is the step size of the gradient descent, called *learning rate*, and determines how fast the solution converges to the global minimum ($\alpha$ must be chosen through a trade-off between convergence speed and optimisation stability). The subscript $m$ underlines the fact that the cost function is computed only over the minibatch $\mathcal{M}$.

At this point, the only operation necessary to be able to apply this method is the computation of the gradient. This operation, in the NN framework, is performed through a numerical process called *backpropagation*.

## A.3 Backpropagation

Backpropagation - which stands for *backward propagation of the error* - is a numerical method used to compute the cost function gradient in the NN framework. As previously mentioned, the cost function $C$, for a given input $\boldsymbol{x}$, is a function of the parameter vector $\theta$. Therefore, the computation of the gradient of $C(\theta)$ with respect to the parameter vector $\theta$ is performed by computing the partial derivative of $C(\theta)$ with respect to each one of the parameters $\theta_i$. The gradient vector expression is:

$$\nabla C(\theta) = \begin{bmatrix} \vdots \\ \frac{\partial C(\theta)}{\partial \theta_i} \\ \vdots \end{bmatrix} \tag{A.13}$$

The gradient vector is then computed one term at a time. Each term is computed through the chain rule of derivation, expanding the expression of $C(\theta)$ and decomposing it in simple derivation terms that can be solved analytically. The resulting equations only differ slightly, depending on whether the derivation parameter is a weight $w$ or a bias $b$. In the case of derivation of $C(\theta)$ with respect to a weight, the derivative is computed as:

$$\begin{aligned}
\frac{\partial C(\theta)}{\partial w_{ij}^{(L)}} &= \frac{\partial C(\theta)}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial w_{ij}^{(L)}} = \\
&= \underbrace{\sum_{k=1}^{n_{L+1}} \left( \frac{\partial C(\theta)}{\partial a_k^{(L+1)}} \frac{\partial a_k^{(L+1)}}{\partial a_j^{(L)}} \right)}_{\frac{\partial C(\theta)}{\partial a_j^{(L)}}} \underbrace{\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial w_{ij}^{(L)}}}_{\frac{\partial a_j^{(L)}}{\partial w_{ij}^{(L)}}} = \\
&= \sum_{k=1}^{n_{L+1}} \left( \underbrace{\sum_{l=1}^{n_{L+2}} (\cdots)}_{\frac{\partial C(\theta)}{\partial a_k^{(L+1)}}} \frac{\partial a_k^{L+1}}{\partial z_k^{(L+1)}} \frac{\partial z_k^{(L+1)}}{\partial a_j^{(L)}} \right) \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial z_j^{(L)}}{\partial w_{ij}^{(L)}}
\end{aligned} \tag{A.14}$$

The equation above shows how the derivative of $C$ is expanded in a product of simple derivatives, that keeps expanding until the output layer is reached (i.e. when the sum $\sum_{i=1}^{n_o}$ appears). The derivative of the cost function with respect to a neuron value $a$ expands in a sum of derivatives, as can be seen in the expression in the third row. The equation seems quite complex at this point, but it quickly simplifies once each term of the derivation is explicitly computed. In fact, all the derivative terms can be analytically computed and result in simple equations. The derivative of $z$ with respect to a weight can be computed as:

$$\frac{\partial z_j^{(L)}}{\partial w_{ij}^{(L)}} = a_i^{(L-1)} \tag{A.15}$$

while the derivative of $a$ with respect to $z$ results just in:

$$\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} = \sigma'(z_j^{(L)}) \tag{A.16}$$

where $\sigma'$ (the first derivative of the activation function) is known. As previously mentioned, the "expansion" of the equation goes on recursively until the output layer is reached. At that point, the derivative $\frac{\partial C(\theta)}{\partial a^{(o)}}$ is computed, whose result is simply:

$$\frac{\partial C(\theta)}{\partial a_i^{(o)}} = 2(a_i^{(o)} - y_i) = 2e_i \tag{A.17}$$

i.e. twice the error on neuron $i$ (this result can be easily derived from equation A.6). At this point, each parameter of the NN is directly connected to the output error through a chain of derivation terms which goes backwards from the output layer to the parameter in question (from which is the name of the *backpropagation* method). So for example, the gradient term

corresponding to the weight $w_{jk}^{(o)}$ (the weight associated to a connection between a neuron of the second-to-last layer and one in the last layer) is computed as:

$$\frac{\partial C(\theta)}{\partial w_{ij}^{(o)}} = \frac{\partial C(\theta)}{\partial a_j^{(o)}} \frac{\partial a_j^{(o)}}{\partial z_j^{(o)}} \frac{\partial z_j^{(o)}}{\partial w_{ij}^{(o)}} = \\ = 2(a_j^{(o)} - y_j)\, \sigma'(z_j^{(o)})\, a_i^{(o-1)} \tag{A.18}$$

As already mentioned, this last equation differs slightly in the case of derivation with respect to a bias. The only difference, in this case, is that the derivation term $\frac{\partial z}{\partial b}$ appears instead of $\frac{\partial z}{\partial w}$ at the end of the derivation chain. This new term is computed as shown in the following equation (as can be easily derived from equation A.1):

$$\frac{\partial z_j^{(L)}}{\partial b_j^{(L)}} = 1 \tag{A.19}$$

# B. k-means clustering algorithm

K-means is one the most widespread and efficient clustering method, which exploits an unsupervised learning algorithm in order to divide a set of points in different clusters according to the nearest distance from the cluster centroid (i.e. the geometrical centre of the cluster). In a nutshell, a k-means clustering algorithm divides a set of points in a number of clusters so that the distance of each point from the centroid of the cluster it belongs to is the lowest distance from all the centroids. The result is a subdivision of the points into Voronoi cells, where points are assigned minimising within-cluster variances (i.e. squared Euclidean distances). The problem is a computationally difficult (NP-hard) but efficient heuristic algorithms that converges quickly to a local optimum. A brief description of the mathematical fundamentals and how the standard algorithm works is explained below.

Given a set of points $X = (x_1, x_2, ..., x_n)$, where each point is a n-dimensional real vector (that is, $x_i \in \mathbb{R}^n$), k-means clustering aims to partition the $n$ points into $k \leqslant n$ sets $\boldsymbol{S} = \{S_1, S_2, ..., S_k\}$ so that the cluster variance (i.e., the within-cluster sum of squares) is minimised. Formally, the objective is to find $S_i \; \forall i = 1...k$ by solving:

$$\arg\min_{\boldsymbol{S}} \sum_{i=1}^{k} \sum_{x \in S_i} \|x - \mu_i\|^2 = \arg\min_{\boldsymbol{S}} \sum_{i=1}^{k} |S_i| \operatorname{var}(S_i) \tag{B.1}$$

where $\mu_i$ is the mean of the points in $S_i$ (i.e., the centroid of $S_i$), whereas $|S_i|$ is the number of points in the cluster. The problem of equation (B.1) is equivalent to minimising the pairwise squared deviation of the points in the same cluster, that is:

$$\arg\min_{\boldsymbol{S}} \sum_{i=1}^{k} \frac{1}{2|S_i|} \sum_{x,y \in S_i} \|x - y\|^2 \tag{B.2}$$

Since the total variance is constant, this is equivalent to maximising the sum of squared deviations between points in different clusters, as in:

$$\arg\max_{\boldsymbol{S}} \frac{1}{n} \sum \|x - y\|^2 \qquad \forall \begin{cases} x \in S_i \\ y \in S_j \end{cases} \qquad i \neq j \tag{B.3}$$

The most common algorithm used to obtain a k-mean clustering is based on an iterative refinement technique.[25] Given a set of points $X$, an initial set of $k$ means $m_1^{(1)}, ..., m_k^{(1)}$ is randomly generated (i.e., a set of $k$ random centroids is picked). Then, the algorithm proceeds by alternating between two steps:

1. *assignment step*: assign each point of $x_p \in X$ to a cluster. To do this, associate it to the centroid $m_i^{(t)}$ with the least squared Euclidean distance from $x$. In the end, each cluster is defined as:

$$S_i^{(t)} = \left\{ x_p : \left\| x_p - m_i^{(t)} \right\|^2 \leq \left\| x_p - m_j^{(t)} \right\|^2 \ \forall j, 1 \leq j \leq k \right\} \tag{B.4}$$

2. *update step:* recalculate the position of the $k$ means by computing the centroid of each cluster:

$$m_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{x_j \in S_i^{(t)}} x_j \tag{B.5}$$

   At this point, centroids for step $t+1$ have been computed and the iteration can go back to step 1.

The iterative process stops when algorithm has converged, i.e., when the difference in the centroid position between two subsequent steps is minor than a desired threshold. However, convergence to the optimum clustering is not guaranteed.

A common problem in the implementation of the clustering algorithm is in the selection of the number of clusters to generate. In fact, keeping the number $k$ of clusters constant can lead to sub-optimal results. For example, in the implementation of the clustering algorithm used to complete the exploration process the number of unexplored regions is not always the same. To solve this problem, a piece of algorithm is usually introduced to evaluate the clustering efficiency when the number of clusters is unknown or varying. This allows to estimate the optimal number of clusters to use. Different methods are available to assess the clustering performances, such as the "elbow method", the "silhouette coefficient" or the "adjusted rand index" (ARI).

The *elbow method* has been used in the implementation of the algorithm used for the completion of the exploration process. This method is composed of two steps. First of all, the clustering algorithm is run several times with different number of clusters (e.g., between 2 and 10). For each run, the average variance of the clusters (or their average *inertia*, i.e., the average distance between the centre of the cluster and each point) is computed. Then, the average variance is plotted against the cluster number. The optimal number of clusters (i.e., the point which presents the best trade-off between a number of clusters and average variance) is selected by looking for the point in the plot where the highest relative difference slope variation between two subsequent points is registered. An example of the selection of the optimal cluster number through the elbow method can be observed in figure B.1.

[25] The algorithm presented here is often called simply "k-means algorithm", but is also referred to as "Lloyd's algorithm". Sometimes it is also called "naive k-means" because there exist much faster alternatives.

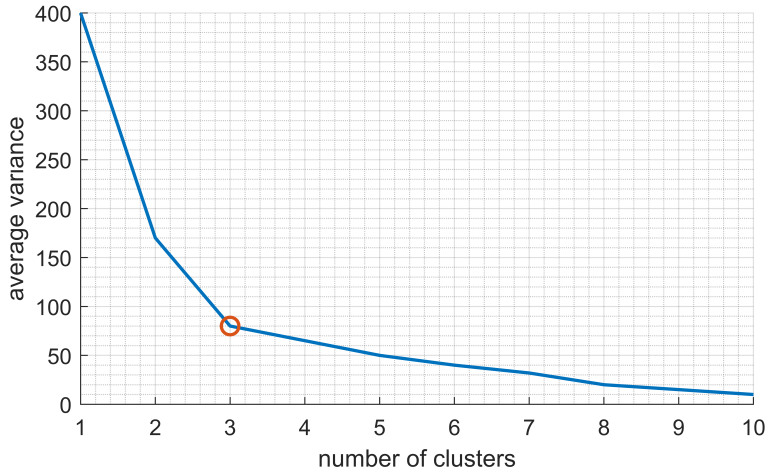


**Figure B.1:** Example of selection of the optimal cluster number through the elbow method. The orange circle indicates the elbow point, corresponding to a number of clusters equal to 3. Higher cluster numbers lead to a smaller average variance, but the relative variance difference is lower.