
Politecnico di Torino


## POLITECNICO DI TORINO

**Department of**

**Control and Computer Engineering (DAUIN)**

Master Degree in Mechatronic Engineering


# Reinforcement Learning approach for autonomous UAVs path planning and exploration of critical environments


**Supervisor**
Giorgio Guglieri

**Co-supervisor**
Simone Godio

**Candidate**
Riccardo Urban



Academic Year 2020/2021

# 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 autonomous 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, as will be illustrated below.

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 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 in an efficient way i.e. avoiding already explored areas and crossing path. At the same time it has to carry on 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. 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 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 efficiency of the "classical" ones, enhancing their capacities beyond those achieved so far.

# Contents

# 1. Introduction

## 1.1 Introduction

The goal of this thesis is to propose 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 and many potential application fields. This thesis is part of this large research field, and in particular of the area concerning the development of autonomous driving algorithms though Artificial Intelligence (AI) techniques. This research area has grown large in the last decade, following the rapid development of AI algorithms and techniques, in particular for what regards those based on Neural Networks. The proposed solution is based on Reinforcement Learning methods, as it revolves around the training of two Neural Network capable of taking real-time optimal decisions in an unknown environment in order to explore it and accomplish a given task. The exploration algorithm is split in two parts: a *coverage* task, which returns as output the objectives to reach and a *path planning* one, that generates the trajectory to reach them (this is illustrated in detail in the chapter 4). Each part is carried out by a dedicated AI agent. The training of such agents is the core of this thesis work. The rest of the document concerns the development of the training and simulation framework, as well as the development of the additional parts of the algorithm needed for the simulation environment such as the section that manages the environment model, the interactions between UAVs and the dynamic model.

The proposed algorithm has been written mostly in the Python language, with the exception of some parts written in MATLAB. The development and testing has been made mostly in a custom-designed environment, whereas some final simulations have been conducted in the ROS 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, OpenAI Gym 0.17.3 and opencv 4.5. In the simulation environment are used ROS Noetic, Gazebo and ArduPilot. As already mentioned, some sections of the code have been written in MATLAB 2020b. Relevant parts of the final code can be found in the GitHub repositories https://github.com/RiccardoUrb 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 Gianpietro Battocletti, in fact this thesis work was developed in close collaboration with him, whose MSc thesis revolved around the same topics of this one. The entire simulation and training framework was developed together since the algorithms proposed in the two thesis were meant to work in the same type of environment. In both thesis, the goal was to perform the 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.

## 1.2 Thesis outline

The thesis is organised as follows: in Chapter 2 is presented the state of art of autonomous UAVs research, along with some examples of possible applications for the algorithm that are designed. In Chapter 3 the mathematical and engineering methods employed in the design of the algorithm are presented and discussed. In Chapter 4 the structure of the whole algorithm is finally unveiled, going into detail regarding building and employing of the simulation environment. Also the path planning and the coverage algorithm used to coordinate the drone fleet are analysed. In addiction a brief discussion about dynamics and control is introduced, along with the description of the reference UAV used in the dynamic model. In the following chapter 5 the training methodologies and the maps used to obtain the two agents are discussed and illustrated. In chapter 6 simulations performed with the implemented algorithm are analysed and discussed and the performances achieved with this model are compared with other classical algorithms. In Chapter 7 an extension of the algorithm to the 3D case is presented. This chapter contains a first implementation of the code in a 3D environment and related ROS simulations. In the latter section 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 such a scenario. The conclusions of the work, along with the discussion of the algorithm issues and possible future development, are presented in Chapter 8. At the end of the thesis, in Appendix A, there is 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 the training of Neural Network and can be helpful to understand the approach used in the development of the algorithm.

# 2. State of art

Before diving in the algorithm design and development, it is worth dedicating a few words to the scientific and technical environment 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 this topic. In this chapter, a brief review of these works will be performed to picture out 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. In the third section instead (Section 2.3) some significant employment of reinforcement learning algorithm are presented. Finally the last section (Section 2.4) is devoted to the description of some relevant application areas.

## 2.1 Path planning and obstacle avoidance

A considerably large number of algorithms has been developed in the past decades to perform path planning and obstacle avoidance operations [1] [2]. These are crucial tasks in the guidance of UASs and for such reasons, path planning and trajectory planning algorithms assume a very relevant importance. Path planning algorithms solve a computational problem which aims to generate a geometric path passing through valid way-points from the starting position to the assigned goal. In this case the geometric path do not take into account any time law for the determination of the way-points. Instead the trajectory planning algorithms define the times of passage at the way-points, for the given geometric path, influencing not only the kinematic properties of the motion, but also the dynamic ones [1]. In dynamic environment applications, where mobile obstacles are present, some extra features, such as obstacle avoidance, may be added in order to accomplish the task avoiding sudden collisions or hazard situations. In some cases, as representation technique, a model of the environment is used as the input of the path planner, built trough different algorithms and stored in the memory of the device where the code is running. A second option is to give as input only the current sensor measurements and compute the output only taking into account these data. In most data-based algorithms the input is an image obtained from a camera, often enriched by some information about the 3D shape of the environment (for example thanks to a stereo camera or a depth camera) [3].

There are different approaches to facing the path planning problem, the traditional major ones are the roadmap, the cell decomposition and the potential fields methods. The roadmap

[1] from now on, whenever is talk about trajectory, is actually meant to be the geometric path computed by the path planning algorithm.

Approach is dependent on the concepts of configuration space and continuous path. A set of one-dimensional curves which connect two nodes of different polygonal obstacles lie in the free space and represent a roadmap $R$. All line segments that connect a vertex of one obstacle to the vertex of another without entering in the interior of any polygonal obstacles are the viable paths. This set of paths is called the roadmap. If a continuous path can be found in the free space of $R$, the initial and the goal points are then connected to this path creating the final path. Dijkstra, RRT, A* [4] and D* [5] are roadmap approach algorithms. For what regarding cell decomposition, the basic idea behind this approach is that a path between the initial configuration and the goal configuration can be determined by subdividing the free space of the region into smaller regions called cells. After this decomposition, a connectivity graph is constructed according to the adjacency relationships between the cells. From this connectivity graph a continuous path can be determined by simply following adjacent free cells from the initial point to the goal one. Finally the potential field method, such as the Artificial Potential Field (APF) [6], involves modelling the agent as a particle moving under the influence of a potential field that is determined by a set of obstacles with a repulsive field and a target destination with an attractive field. All these algorithms store and use a model to represent the available information about the environment (such as the position of obstacles in the environment) and they are fairly light from the computational point of view.[7]. A common characteristic of the algorithms listed above is that they are deterministic, i.e. given a set of initial conditions they compute always the same path, following a given set of rules. The drawback of this approach is that these rules needs to be explicitly coded in order to obtain a trajectory with some desired characteristics. This is quite simple if the only objective is to find the shortest path to a desired location. However, if the path is required to have some more complex characteristics (e.g. minimum travel time, taking into account the agent dynamics, or maximum possible distance from obstacle), those algorithms become too much limited. In fact, the conditions listed as an example can be extremely difficult to explicitly code, and so deterministic algorithms result insufficient to optimise the path with respect to those characteristics. To overcome this limitations, heuristic path planning algorithms are often used.

Heuristic-search techniques implement an optimisation problem through cost estimation method from the starting position to the goal point. These approaches due to the power and convenience during the optimisation process are considerable interesting in the research field of optimal path planning algorithm for UAVs. The relevant methods include Genetic Algorithm (GA) and Particle Swarm Optimisation (PSO) [8], simulated annealing method and hybrid algorithms (where the optimisation process is made considering multiple cost functions) [9]. These kind of path planning algorithms allow to compute a route optimised with respect to any desired characteristic. In this case, it is much easier to take into account any complex constraint since it is sufficient to include them 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 and robust paths. Often the optimisation process is a stochastic one in order to speed up the computation of the optimal solution. In fact, the main limitation of optimisation-based path planning algorithms is that their computational cost is higher than the one of the algorithms introduced in the section above. Since the microcomputers on which those algorithms usually run often have limited computational power usually they have time constraints, so to keep the computational cost sufficiently low these algorithms end up finding a sub-optimal solution (which is, however, usually acceptable for the path planning purpose).

A class of algorithms that could be revolutionary in the path planning field is that of Artificial-

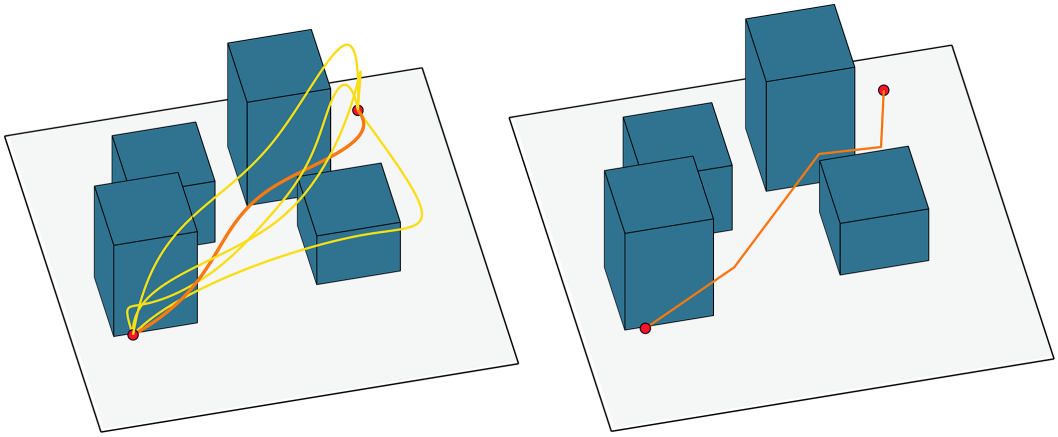

**Figure 2.1:** Comparison between a path generated with an heuristic path planning i.e. PSO algorithm on the left and a trajectory generated with the potential field approach on the right.

Intelligence-based path planning. This class of algorithms allows to compute optimal or nearly-optimal solutions (like in the case of optimisation-based algorithms) but at a very low computational cost. This is possible thanks to a training process, separated from the actual algorithm run phase, that is performed offline, where through an offline optimisation process result a function capable of generating optimal solution with a very low computational cost. In particular the RL algorithms are very promising since, unlike the classical dynamic programming methods they do not assume knowledge of an exact mathematical model of the problem and this became relevant when exact methods become infeasible. The employment of these class of algorithm is very recent and is still subject of research in many areas. Especially with regard to RL ones which are still in an embryonic state from an application point of view due to a lack of robustness. Some of work related with the navigation of autonomous UAV with RL algorithm are: [10, 11, 12].

## 2.2 Fleet coordination

One particular area on which part of the research has focused is that of autonomous UAV fleets. The idea is to create fleets of UAVs capable of coordinating and collaborating in the pursue of a common objective. This task comes with several challenges. A number of different solutions have been proposed or are still being researched to solve this problem [13]. The techniques involved in the design of fleet coordination algorithms are various, and range from the use of mechanical models to the implementation of bio-inspired methods, up to the use of Deep Learning (DL) approaches [14]. A particular application of autonomous UAV fleets is their use in the exploration of unknown or critical environments (i.e., where it could be dangerous for a human operator to go). This particular topic constitutes the main focus of this thesis.

A first approach used for fleet coordination is that of *flocking algorithms* [15]. In this case, fleets are usually organised to have a leader which guides the action of all the components. The algorithm is distributed, i.e., each component of the fleet takes its own decisions, instead, the leader has a prominent role in the decision-making centralising part of the fleet control. Often, flocking algorithms aim at maintaining a desired formation, that can be a relative

position between the fleet leader and the other components, in some cases also between the components themselves. Flocking algorithms are well suited to control large fleets that need to maintain a formation following a leader unit, however, they are not efficient for exploration and coverage tasks with a small number of UAVs, since the single components lack the independence necessary to quickly cover all the area in the environment [16, 17].

A second group of methods developed to assess this problem is that of *swarming algorithms* [18, 19]. Swarming algorithms are based on a decentralised intelligence approach. Each component of the swarm follows the same set of rules, which are usually quite simple and regard only the decisions taken by a single member of the swarm. The application of those rules by all the members of the fleet leads to the emergence of more complex behaviours and strategies which can not be predicted from the basic rules. These kinds of behaviours are called *emerging behaviours* and stem from the concept of *collective intelligence* [20]. These methods are often bio-inspired, as they get inspiration from the observation of the behaviour of swarms of bees or insect colonies. Fleets designed following this approach are usually leaderless, so they have no hierarchical organisation and every component of the swarm takes decisions on its own on the basis of its current state and of the information it has about the other members. This information can be obtained by sensing the surrounding environment or through direct communication with the other components of the swarm. Communication, that can be direct or indirect (where the communication is obtained by leaving a trail or marks in the environment), is one crucial feature of swarming algorithms, as the swarm components rely on information about the other members to take their decisions. A particular type of swarming methods based on indirect communication is that of *stigmergy*-based algorithms [21, 17, 22]. In this approach, each member of the fleet releases a virtual substance (usually called "digital pheromone" from its biological inspiration) as it moves in the environment. The pheromone remains where it is released for a certain time, diffusing slowly in the nearby area. While it persists, it can be used to gain information on the past location of the other members of the fleet, and be exploited to give rise to a collective exploration algorithm. If every member of the fleet tends to go toward the lowest concentration of pheromone, it is possible to obtain an emerging coverage algorithm suitable for unknown environments. The main issue in a practical implementation of this method for a fleet of UAVs is in the simulation of the release and diffusion dynamics of the pheromone. In fact, all these operations must be performed inside the algorithm and can result in time-consuming. Attempts have been made to use a *real* substance released in the air by the UAVs, equipping them with a sensor to detect its presence. However, this kind of approach brings with it a series of complications about the management of this substance, resulting sub-optimal results.

A different and not widely explored approach to swarming algorithms is obtained by exploiting models through a Deep Learning process. In this case, the rules determining the individual behaviour of each component of the fleet are learned through a dedicated training process. The goal is to generate the best set of individual rules to obtain the desired collective behaviour. This kind of method fits in the domain of Multi-Agent Reinforcement Learning (MARL) models, which are a very recent field of study in Artificial Intelligence [23, 24]. MARL models are usually built using special architectures to design the training process, optimised to drive the agent to learn the desired policy. The application of the same policy by all the members of the fleet is what leads to the desired emerging behaviour. This thesis explores this subject by implementing a RL-based swarming algorithm for exploration and coverage of unknown environments. Some works with the same goal of this thesis that deserve to be mentioned are: [25, 26]

## 2.3 Reinforcement learning

Reinforcement learning is an area of machine learning concerned with how software agents ought to take specific actions in an environment so as to maximise some notion of cumulative reward. Due to its generality, the field is studied in many other disciplines, such as game theory, control theory, operations research, information theory, simulation-based optimisation, multi-agent systems, swarm intelligence, statistics and genetic algorithms. Reinforcement learning algorithms are used in that areas where it is nearly impossible to code an explicit algorithm that manages and process a very large number of possible input states. Autonomous vehicles and robots fall into these fields. A possible future field of these algorithms is the human assistance, where the RL agent works with a human operator and it is able to take human action into its considerations in order to perform operations achieving a common goal [27].

## 2.4 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 [14]. However, it is worth pointing out some applications where the usage of fleets of UAVs, autonomous coverage and exploration algorithms could be particularly valuable.

- An application discussed in [28] is to use a UAVs swarm 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 sort 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;

- UAVs can be employed in forest environments also for surveillance tasks. In particular, an application of high 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 [29]. More powerful techniques, such as forest monitoring with satellites, can be used for a more effective 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 [30], [31]. 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 [19]. 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;

- photogrammetry for infrastructure monitoring is a valid application for UAVs that are a low-cost alternatives to the classical manned aerial vehicles. They are capable of performing the photogrammetric data acquisition in a fully autonomous way. Automation

is nowadays necessary in order to obtain the accuracy needed, during the acquisition, to perform the creation of 3D models of infrastructures or terrains starting from a collection of images [32];

- a field in which UAVs can significantly help is in emergency response. After a disaster, like earthquakes, floodings or avalanches swarms of UAVs can be extremely efficient for reconnaissance tasks in order to map the extent of the damages or to look for survivors. The autonomy and high manoeuvrability of UAVs allows them to reach virtually any location, even in critical environments where it would be dangerous for human operators to go [33];

- UAVs can be also employed in the precision agricultural sector. Thanks to lower and lower costs and the approach of increasingly robust autonomous systems, precision farming is becoming a very interesting application for UAV swarm [34]. Their usage in agriculture in fact allows farmers to optimise agriculture operations and to monitor it, increasing crop yield and farm efficiency [35]. UAVs are particularly useful when the farmland area to monitor is very large. The aerial view provided by a drone combined with its sensors, can give to the farmers or industries a richer picture of their fields ensuring rapid access to all data needed to monitor fertility and health state of crops. This can help to inspect the soil detecting pest and fungal infestations and to be more accurate in the application of fertilisers reducing wastage and use of pesticides. The swarm can also monitor the water management inside the farm revealing possible issues in the irrigation system and communicate them to the farmers immediately. UAVs could even be employed to assist autonomous ground robots in the harvesting operations, by supplying images and data to support their tasks during the movement between different crops;

- 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. 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 [36]. 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 Ingenuity 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 of the most powerful tools to rapidly explore large martian regions, where the currently used ground rovers would take many years to fulfil the same task. On $19^{th}$ of April 2021 Ingenuity performed its first flight in the martian atmosphere [37], demonstrating the technical possibility of flying a completely autonomous UAV on another planet in order to perform exploration and scientific tasks;

- An even more exciting example of employment of autonomous UAVs for planetary exploration comes from the Dragonfly mission [38]. 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, collect and study soil samples with dedicated instruments

**Figure 2.2:** Frames from the Perseverance video of Ingenuity first flight (from [37]).

in search of clues of life presence [39]. Radio signals take an average of 1.5 hours to reach Titan from Earth, rendering even more crucial the autonomy of each operations, 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 algorithm will be briefly illustrated. The focus will be on the theoretical background of each method, with some attention on the assumptions that lie at their base. In section 3.1 the 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 and of some learning algorithms will be introduced.

## 3.1 Artificial Potential Field

The Artificial Potential Field (APF) is an algorithm presented in 1985 as a path planning method. 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 to the agent to reach the goal. The obstacle avoidance part of the path planning algorithm is instead implemented by the repulsive potential. A high potential value is associated to each obstacle following an hyperbolic or gaussian distribution (depending by the definition - the former is more common). In this way, the closer a point is to the obstacle, the higher is its potential. When computing the negative gradient of this potential distribution, the resulting

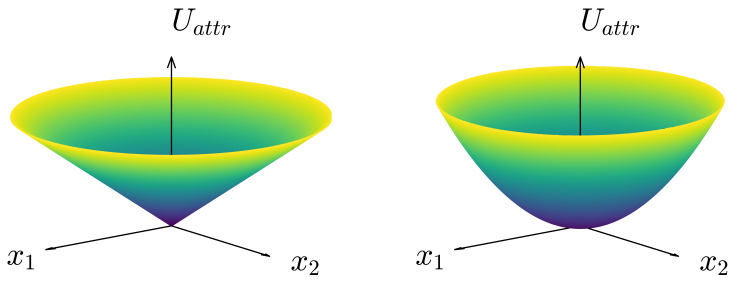


**Figure 3.1:** Attractive potential distribution over a 2D space. On the left, the shape resulting from parabolic definition; on the right, the one obtained from conic definition (as in equation 3.2).

vector points away from the obstacle itself, allowing to the agent to avoid any collision. The equation implementing the repulsive potential distribution when defined as a gaussian curve[2] 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 $d(x) = |x - x_g|$ is the distance from the goal. 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.

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

The traditional 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 the versor of eq. (3.4) does not exist and so the agent does not have a direction to move to. Since this problem has been known for a long time, different solutions have been developed to deal with it, as for example in [40, 41, 42, 43]. The actual solution implemented in the algorithm will be discussed in chapter 4.

[2] 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, as mentioned above, the majority of APF studies use a hyperbolic repulsive potential distribution, like in the original proposal of the APF algorithm [6]. 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, the general shape of the repulsive potential is the same - growing quickly approaching to the obstacle and decaying to zero after a certain distance from it.

## 3.2 Reinforcement Learning

Machine learning (ML) is the study of computer algorithms that improve automatically through experience and by the use of data. This type of algorithms build a model based on "training data", in order to make predictions or decisions without being explicitly programmed to do so. They are used in a wide variety of applications, in particular where the developing of conventional algorithms to perform the needed tasks it is unfeasible. 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. Reinforcement learning differs from supervised learning in not needing labelled input/output data, and in not needing sub-optimal actions to be explicitly corrected. Instead the focus is on finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge). 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 and according to it 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* that associates the optimal action to every situation the agent is faced with.

### 3.2.1 Markov Decision Process (MDP)

The classical mathematical framework through which most Reinforcement Learning problems are formalised is the Markov Decision Process (MDP) [44, 45]. 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 $s$ 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. $\mathcal{P}_s$ is a probability transition function, which determines the probability to reach state $s'$ from state $s$ when action $a$ is taken, namely $\mathcal{P}_a(s, s') = \mathrm{P}(s_{t+1} = s' | s_t = s, a_t = a)$. The probability transition function implies that, in general, the MDP is a stochastic process, since there is no way to determine the state at time $t + 1$ other than 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 $\mathcal{P}_a(s, s') = 1$ only for the 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 $R = \mathcal{R}_a(s, s')$. The block scheme in Figure 3.2 represents the relationship between all the variables involved in a 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 $r_{t+1}$.

From this block scheme representation the equations governing the time evolution of a MDP

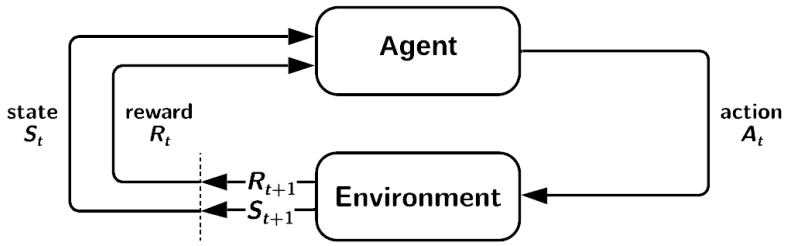


**Figure 3.2:** Simple block-scheme representation of MDP (image taken from [45]).

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 a 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 $a_t$.

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

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.[3] 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 policy that always points to the action that returns the highest reward is called *optimal policy* and is denoted as $\pi^*$. The goal of Reinforcement Learning is to find such an 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 type of policy that is used, the maximisation of the expected return is obtained following different strategies. In the next sections, the RL training process through which this goal is achieved will be introduced.

[3] 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.

### 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 a RL problem is to obtain a policy $\pi$ that is as close as possible to the optimal policy $\pi^*$. 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:

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

where $\mathbb{E}$ is the *expected value* function. The state-value function gives back 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 also be rewritten in 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 that which 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. The idea that emerges is that, knowing the $q$ function, it is possible to 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 approximate the policy function. This means that the agent Neural Network 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.[4] A Neural Network that approximates the action-value function is called a *critic*,[5]

[4] 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 then the use of the $q$ function approximation is a bit different, as will be shown in the DDPG algorithm in section 3.2.5.

[5] on the contrary, a Neural Network that directly approximates $\pi$ is called an *actor*.

and is at the base of different learning algorithms like DDQN, where in those algorithms, the NN is trained to approximate the $q^*$ function and use it to choose the best action. Again, 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).

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

### 3.2.3 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[6] 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 the leading ones in the RL world, is to approximate the 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 [46], 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 by 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. The basic working principle of a NN is that each neuron performs a specific mathematical operation and stores its result. The result is then passed to the neurons in the following layers through the connections. Each connection is associated to a weight, which multiplies the neuron value before passing it to the following layer. The operation performed by each neuron includes a usually very simple nonlinear function called *activation function*. The usage of the layered structure and the nonlinear functions allows to approximate any desired function by just 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 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*

[6] a *stationary agent* is an agent that chooses the best action only on the base of the current state [44, 45]. Every agent that will be considered in this discussion and in the algorithm development fits in this category.

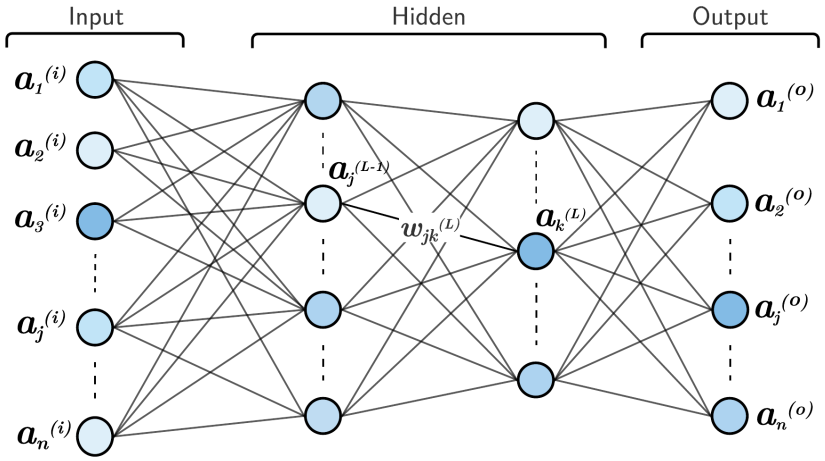


**Figure 3.3:** Simple Neural Net scheme. The input, output and hidden layers are put in evidence. Each neuron holds a value $a_j^{(L)}$ and a bias $b_j^{(L)}$, while each connection holds a weight $w_{jk}^{(L)}$. The colours represent fictitiously the higher or lower value of the neurons.

*NN that approximates the desired optimal function as well as possible.*[7]

### 3.2.4 Policy training in the Reinforcement Learning framework

In the previous section it was defined that the goal of a RL training process is to obtain a set of parameters $\theta$ so that the policy NN function $\pi_\theta(s)$ matches the optimal policy $\pi^*$ as close as possible.[8] 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 and at this point 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 choose an action, performing it in the environment and updating the policy on the base of the reward obtained. The basic idea is that, given a state $s_i$, if the reward obtained for action $a_i$ is positive, the policy will be updated to encourage the choice of the same action when state $s_i$ or a state very 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 principle is the same:

1. given the current environment state $s_t$, choose action $a_t$ through policy $\pi$ or randomly, depending on the exploration/exploitation strategy adopted;

[7] 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 given NN. The NN architecture choice is important to reduce the training time and computational complexity of the net.

[8] for simplicity, in this chapter the 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.

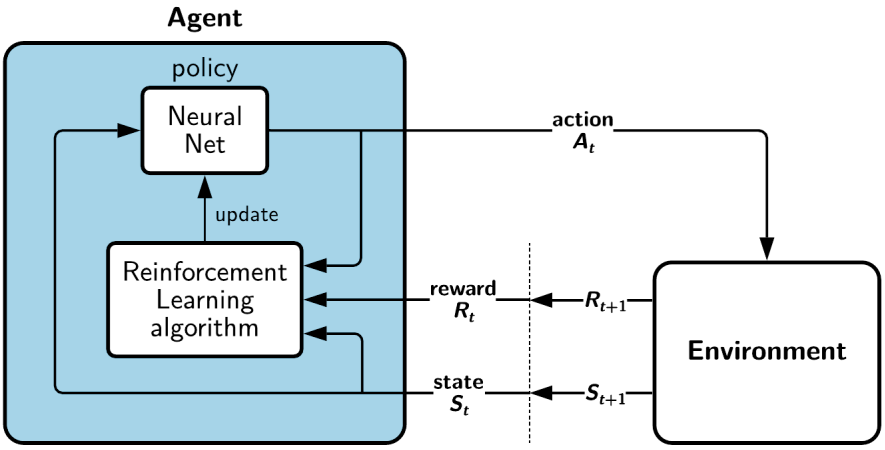


**Figure 3.4:** Block scheme representation of a RL training process. The basic structure is the same of the MDP represented in figure 3.2. However, in this case the agent internal structure and the presence of the RL training algorithm are put in evidence.

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 $a_t$, $s_{t+1} = s'$ can be exactly determined. This is also the reason why $r_{t+1}$ does not depend on $s_{t+1}$;

3. compute the loss value ($\sim$ estimation error) through an arbitrarily defined *loss function*. The loss function commonly used is:

$$L = \sum_{i=1}^{N_o} \left( a_i^{(o)} - y_i \right)^2 \tag{3.14}$$

   where $a_i^{(o)}$ is the value of the output layer neurons, $N_o$ is the number of neurons in the output layer and $y_i$ is the desired ("correct") value of the output;

4. use the loss value to update the NN parameters (following the 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 only issue of this training algorithm 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 value $r$. One of these methods will be presented in section 3.2.5 along with the corresponding training algorithm.[9]

[9] this is also the method used for the training of path planning and coverage model in the chapter 5.

A last consideration before getting to the learning algorithms regards step 1. In fact, one key aspect of RL training is to try as much different strategy 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 strengthen good strategies already learned. There are several ways to promote exploration (it is not necessary to promote exploitation since it is obtained simply by choosing the actions though the NN). Two of them are $\varepsilon$-greedy strategy and *Ornstein-Uhlenbeck* random noise process, the latter will be presented in the DDPG learning algorithms section below.

### 3.2.5 Deep Deterministic Policy Gradient (DDPG)

The learning algorithm that has been used during all the thesis development is the Deep Deterministic Policy Gradient (DDPG) algorithm.[10] As can be seen in Figure 3.5, DDPG is a model-free and off-policy learning method, it is suitable for *continuous action spaces* and is an actor-critic method.

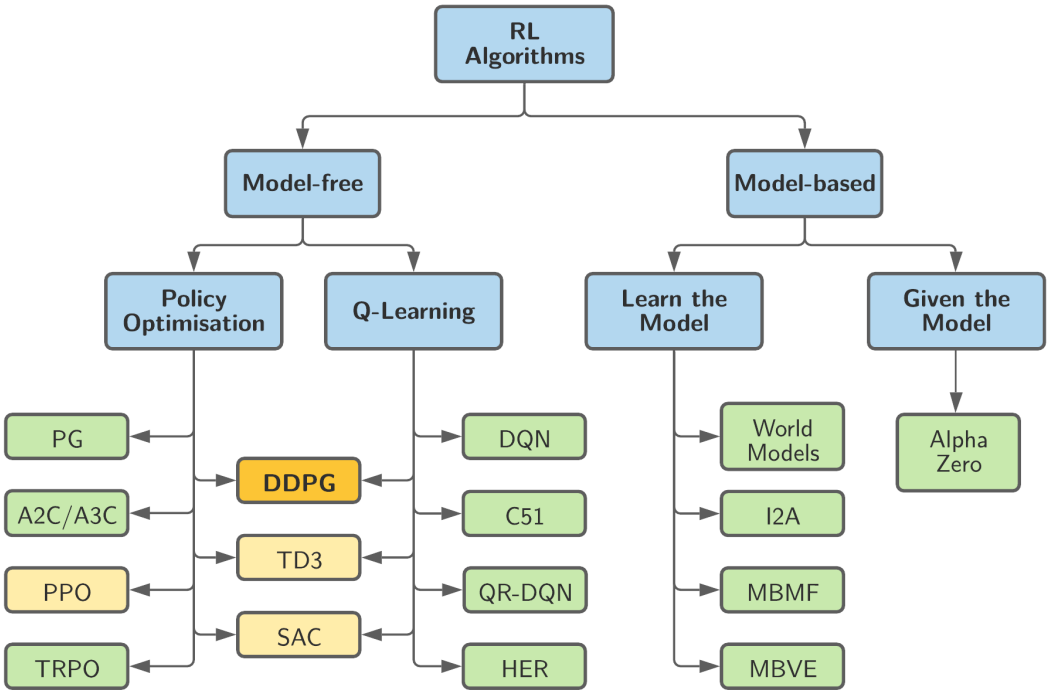


**Figure 3.5:** Scheme of the categorisation of some of the most popular RL algorithms. DDPG is the one chosen for the development of the path planning and coverage algorithms and the other three algorithms evidenced in yellow (PPO, SAC, T3D) are also suitable to work with a continuous action space, but are more advanced and more complex to implement. Image is taken from [47].

The fact that it is an actor-critic method means that, in the end, this algorithm produces a function $\mu_\theta$ that approximates the optimal policy $\pi^*(s)$ (where $s$ is the input state). This function can be directly used to find the optimal action for state $s$ by computing action $a = \mu_\theta(s)$. To train the actor, DDPG also 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

[10] 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.

actor. In fact, in this method the goal of the actor is to find the action $a$ that maximises the Q-value, so the data provided by the Q-value function are sufficient to train it. The complete learning routine of DDPG is illustrated in Algorithm 3.1 [48, 49].

**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 $\phi' = \phi$ [11]
initialise the experience buffer $B$ with size $N_B$
**for** $e$ in $N_{episodes}$ **do**
    initialise a random Ornstein-Uhlenbeck noise process $\mathcal{N}$ for action exploration[12]
    reset episode initial condition (i.e. obtain state $s_0$)
    **for** $t$ in $N_{steps}$ **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 $a$ in the environment and collect the new state $s'$ and the reward $r$
        store the experience sample $(s,a,r,s')$ 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:

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

        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 is defined as:

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

        update the actor policy performing one step of gradient ascent using:

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

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

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

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

    **end for**
**end for**

Before proceeding with the algorithm description and the actual implementation of this training algorithm, it is worth dedicating them a few words to the main features of the learning process.

- as already introduced, 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 $Q_\theta$ 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 an average of the loss computed for each element of the minibatch). The value of $y_i$, which is, in general, unknown, is approximated by the Bellman equation 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, in fact, the target networks output will be different from the one of the "base" networks since the former ones are updated through a smoothed update law and so the value of their parameters is "delayed" in time with respect the actor and critic ones. The smoothed update law, as already seen in the algorithm, is:

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

  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 networks are updated to match their actor/critic counterparts;

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

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

---

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

[12] in DDPG a random noise is used to promote action exploration instead of the $\varepsilon$-greedy approach seen in DDQN.

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 *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:

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

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 thus $Q$ is not derived with respect to them;

- *off-policy* learning 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 NNs in their current state of training, but they could have been produced by an "older" and less trained version of them. 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, this 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 off-policy learning;

- 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 state which could lead to the discovery of new strategies to get more reward in the environment. In the case of DDPG, the exploration strategy is to sum, to the action computed by the agent, a noise value produced through a Ornstein-Uhlenbeck process. 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 exploration that specific direction of the action space.[13] Over time, the trend inverts so that each direction is explored after a sufficient amount of steps.

[13] some critics have been made against the use of an Ornstein-Uhlenbeck noise process instead of a random Gaussian one. In fact, some paper discussing more advanced versions of DDPG state that the Ornstein-Uhlenbeck noise process does not improve training performances and is actually just an over-complication of the algorithm [50], [51]. During the training of this thesis agents, an Ornstein-Uhlenbeck process was used nonetheless.

## 3.3 k-means clustering

K-means is one the most widespread and efficient clustering method, which exploits an unsupervised learning algorithm in order to assign a set of points to different clusters according to the nearest mean from the cluster centroid (i.e. the geometrical centre of the cluster). The result is a subdivision of the points space into Voronoi cells where points are assigned minimising within-cluster variances (i.e. squared Euclidean distances). The problem is computationally difficult (NP-hard), however, efficient heuristic algorithms converge quickly to a local optimum. A brief description of the mathematical fundamental and how the standard algorithm work are explained below.

Given a set of points $\boldsymbol{X} = (x_1, x_2, ..., x_N)$, where each point is a n-dimensional real vector, k-means clustering aims to partition the $N$ points into $k \leqslant N$ sets $\boldsymbol{S} = \{S_1, S_2, ..., S_k\}$ so as to minimise the within-cluster sum of squares i.e. variance. Formally, the objective is to find:

$$\arg\min_{\boldsymbol{S}} \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2 = \arg\min_{\boldsymbol{S}} \sum_{i=1}^{k} |S_i| \operatorname{Var} S_i \tag{3.27}$$

where $\mu_i$ is the mean of points in $S_i$. This is equivalent to minimising the pairwise squared deviations of points in the same cluster:

$$\arg\min_{\boldsymbol{S}} \sum_{i=1}^{k} \frac{1}{2|S_i|} \sum_{\mathbf{x},\mathbf{y} \in S_i} \|\mathbf{x} - \mathbf{y}\|^2 \tag{3.28}$$

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

$$\arg\max_{\boldsymbol{S}} \frac{1}{n} \sum \|x - y\|^2 \qquad \forall \begin{cases} x \in S_i \\ y \in S_j \end{cases} \qquad i \neq j \tag{3.29}$$

The most common algorithm uses an iterative refinement technique.
Given an initial set of k means $m_1(1), ..., m_k(1)$ randomly generated, the algorithm proceeds by alternating between two steps:

1. *Assignment step:* Assign each points to the cluster with the least squared Euclidean distance.
$$S_i^{(t)} = \left\{ x_p : \left\| x_p - m_i^{(t)} \right\|^2 \leq \left\| x_p - m_j^{(t)} \right\|^2 \ \forall j, 1 \leq j \leq k \right\}, \tag{3.30}$$
where each $x_p$ is assigned to exactly one $S^{(t)}$.

2. *Update step:* Recalculate means (centroids) for points assigned to each cluster.
$$m_i^{(t+1)} = \frac{1}{\left| S_i^{(t)} \right|} \sum_{x_j \in S_i^{(t)}} x_j \tag{3.31}$$

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.

# 4. Algorithm design

In this chapter, the whole algorithm developed during the thesis is presented and described. The chapter starts with a section about the assumptions done prior to the design process of the algorithm (Section 4.1). The chapter continues with the description of the algorithm, composed of the training and the simulation environments (Section 4.2.1), where the training process and the environment used during the simulations are described, then the logic tree is presented and all the high-level choices are described. After this section, the model with which the environment is represented and updated dynamically is explained (Section 4.3). After this, the path planning algorithm is discussed (Section 4.5), accompanied by the coverage algorithm (Section 4.4). Follow the description of the dynamic model of a quad-rotor UAV reported along with a suitable control system (Section 4.6). Finally, the reference UAV used for the real simulations is revealed (Section 4.7).

## 4.1 Assumptions and simplifications

A few assumptions have been made in order to start the design of the exploration algorithm. it is worth mentioning them to clearly point out the limits of applicability of the proposed method within real cases, along with possible issues and implementation solutions.
The assumptions and simplifications done in the algorithm design are:

1. **2D environment:** to simplify the design of the algorithm and the RL agent training the method implemented works in a 2D environment where it assumes the UAVs always fly at the same altitude. Two solutions can be considered to extend the algorithm to the 3D case: the first is to add the third coordinate to the output of the RL agent, however this implies a new agent training to take into account the third dimension and to return the additional action. The second solution, simpler but less efficient, is to slice the 3D environment in "layers" with some arbitrary spacing between them, and navigate each 2D layer separately from the whole environment. 3D extension will be discuss more in detail in section 7;

2. **the environment dimensions are known:** each drone stores an internal model of the environment, which is built starting from the environment dimensions. Therefore, this data have to be known in order to initialise the environment. An upper bound on the dimensions is sufficient to identify the operating area and initialise all the required matrices. An additional issue linked to the environment dimension is that, for environment that are too large, the map stored in the drone memory would be quite large and potentially slow to be accessed and manipulated. Therefore, map dimensions should not be too large (approximately 100m x 100m could be a good upper limit for what regarding indoor maps), however a possible solution to manage larger environments would be to store

two maps, one at lower resolution that considers the whole environment and one at high resolution that represents only a neighbourhood of the drone;

3. **the exact position of the drones is always known:** in real applications this is not possible since any method employed to estimate the drones position would be affected by some error. For instance, if the position is computed through a GPS sensor in an outdoor application, this would be affected by some uncertainty, especially due to the fact that the drone is constantly moving. If, as another possibility, the drone position was computed through integration of the acceleration measured by an on-board sensor, an integration error would be added, and potentially grow indefinitely. A possible solution to this problem would be to use an IMU combined with a tracking camera which through a sensor fusion approach it is possible to get a very accurate estimation of the position. However, this kind of problem is not considered in this work and thus in the algorithm but is left to a possible future implementation;

4. **the initial absolute position of the drones is known:** in addition to the exact position of the drones being known, the initial position of each drone with respect to the environment is known. This is equivalent to knowing where the $(0, 0)$ point is. The reason for this assumption is that the environment model is built from the position of the drones expressed in absolute coordinates and not in relative ones;

5. **only static obstacles are stored:** obstacles position is stored in the local model of the environment of each drone. At the moment, there is no way to understand if an obstacle is moving or not, and therefore all obstacles are assumed to be static (except for the other drones, which are the only mobile obstacles considered since their position over time is assumed to be known);

6. **map resolution is limited:** the drone environment model is stored with a given resolution, which means the space is discretised and information are saved for areas of space of size $r$ x $r$, where $r$ is the spatial resolution with which the map is represented. Resolution is approximately in the range 5cm-20cm. Any obstacle smaller than this size (e.g. a rope) must be represented as a bigger obstacle as a conservative solution;

7. **communications are instantaneous and have infinite bandwidth:** the communication between drones is not assessed 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 could be implemented to obtain a more realistic algorithm, but it is outside the scope of the project.

Any other information about the environment is unknown. This means the number, shape, and disposition of any obstacle is not known, i.e. at the beginning the environment is represented with an empty map. The only additional available datum is the mission objective. The goal could be simply to explore the map, discovering as many information as possible about its geometry, or to find a specific point/object etc. The algorithm is flexible in terms of objective and can easily be adapted for different kind of tasks.

## 4.2 Algorithm description

The algorithm is divided into two distinct parts, the first one is the training environment, where the agents dedicated to the exploration part and to the generation of the trajectories are trained. In this part the agents are dipped in a simulation created in order to simulate movements and interactions between drones. In this environment the agents receive a reward according to the action taken inside the simulation. Whereas the second part of the algorithm is the simulation environment where the agents, previously trained, are tested in a simulation of the real world.

### 4.2.1 Training & simulation environment

#### Training process

In the training process of Figure 4.1 the two agents will be inserted in a virtual simulation where they are rewarded or punished according to the action taken while the Model Builder receives some sensor data from the environment and uses them to update the APF model. The model is used by the Coverage and Path Planning agents to choose which actions to apply and to communicate information about interactions. The learning algorithm works alongside this process. At each time step, the action $a = a(t)$ chosen by the agent, the current state $s = s(t)$, the next state $s' = s(t+1)$ (i.e., the state of the environment after the action $a$ has been performed) and the reward $r = r(t)$ are collected. The tuple $(s, a, r, s')$ is stored in a memory buffer and used to perform the DDPG learning operations and at each time step, the state $s'$ obtained in the previous iteration is used as the new input state. The training process continues iteratively in this fashion.

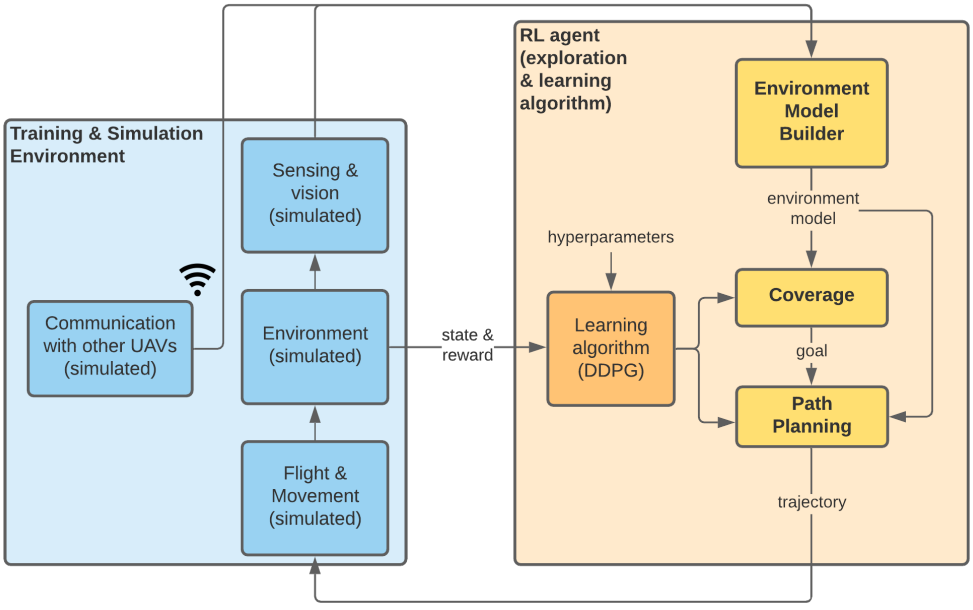


**Figure 4.1:** Block diagram of the learning process. The yellow blocks are the two RL agents and the learning algorithm, whereas the blue ones are the parts used for the environment simulation in the training process.

**Simulation environment**

The algorithm in charge of the simulation procedure is composed by four main parts, as shown in the figure 4.2. These four parts run continuously in parallel, constantly exchanging data between them. Here a brief description of each of them will be made in order to outline their role. In particular, the data exchanged between the parts will be put in evidence. The four parts are:

1. the **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 each UAV and the positions of all the other drones. Then, it uses them to update in real-time the environment model, modifying the potential field representation considering newly discovered obstacles and mobile obstacles and changes in the goal position. The world model is stored in each UAV internal memory to be used by the other parts of the algorithm. In addition, the model is transmitted to the other UAVs to share all possible information. In particular, this allows to each UAV to know a larger portion of the map that the one it explored by itself;

2. the **Coverage Planning algorithm**: this part of the algorithm is used to coordinate the fleet during its operations assigning different goals to each drones. It takes into account the world model and the position of all the UAVs then uses these information to compute a target location for every drone. This part of code aims to generate different positions to be reached in order to optimise the task execution as much as possible spreading the UAVs avoid crossing areas already explored and minimise non-useful travels;

3. the **Path Planning algorithm**: Once the goal computed by the single drone is set the Path Planning Algorithm uses the APF world model to compute an optimal trajectory in order to reach it. The trajectory is constantly updated since the world model is continuously updated by the environment model builder. The trajectory is computed by a properly trained RL agent that is trained to optimise the trajectory in function of smoothness, collision avoidance and travel time;

4. the **Control algorithm**: this last part of the algorithm has the task of translating the trajectory generated by the Path Planning algorithm in actuation signals. Those signals are applied by the UAV to fly and reach the goal position. This part of code consists simply in a control loop, designed on the base of the UAV dynamic model.

After the training process, the yellow blocks of Figure 4.1, are extracted from the training environment. They constitute the exploration algorithm, and are available to be deployed in a real-world application. This passage requires some code adaptation: in fact, the exploration algorithm will run side by side with other applications in the UAV on-board computer. An high-level block scheme representation of the algorithm deployed for a real-world application is shown in the figure 4.2 where is shown how the UAV operating system interacts with the real world. It is worth noting that the latter can be the real world or a simulation of it. In fact the first tests have been carried out in a python simulation without implementing dynamics and control of the UAV, then the algorithm was moved in a ROS-Gazebo simulation where the dynamic of the UAV is taken into account. This part will be discussed in more detail in 7. The green block of the figure 4.2 shows the sensors section which interacts with the real world in order to compute the positions of the obstacle inside the vision range and the UAV position. Whereas the blue block is the UAV plant which receives the actuation input from the control

algorithm. The control algorithm is the one which translates the trajectory computed by the Path Planning algorithm in actuation signals for the rotors and 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. This part will be deepened in section 4.6.

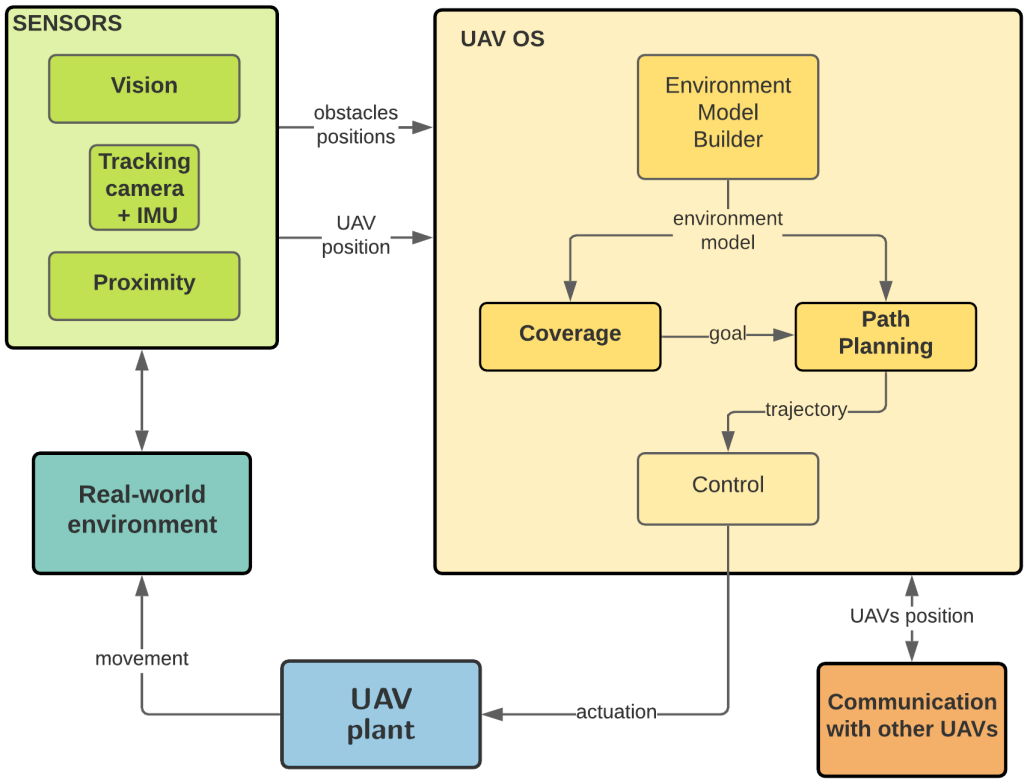


**Figure 4.2:** Block scheme of the simulation environment at an high-level structure.

The main part of the algorithm in charge of manage the inputs received from the real world and to update the trajectory of the UAV is illustrated in the algorithm 4.2. This algorithm is the core of the UAV. It receives the new obstacle positions from the vision sensors, manages the communication with other UAVs or with a ground station avoiding collisions between them and exploits the two RL agents for update the goal position and compute suitable trajectories. All these applications are managed by this dedicated OS, that performs all the scheduling operations. Follows the main routine of the action cycle.

**Algorithm 4.2** UAV OS action routine

```
1: while the operating condition of the UAV is set to action do
2:     detect obstacles with the sensor vision
3:     share the exploration matrix updated during the vision step
4:     if new obstacle position has been found then
5:         update the potential map with the new obstacles positions
6:     end if
7:     receive the communication with the position of the others drones
8:     update the potential map with the mobile obstacles in the positions received
9:     if the proximity sensor has detect obstacles too close then
10:        change the emergency path planning flag state to true
11:    end if
12:    if the goal position is not updated then
13:        call the coverage algorithm which returns as output the position of the new goal
14:        recall the environment model builder in order to update the attractive layer with
           the new goal
15:        check if the goal is a suitable one and in case of wrong position it changes the state
           of goal position as not updated
16:    else
17:        if the drone is in a local minimum then
18:            switch to the local minima path planning algorithm
19:        else if the emergency path planning flag is set to true then
20:            switch to the emergency path planning algorithm
21:        else
22:            switch to the main RL path planning algorithm
23:        end if
24:    end if
25: end while
```

## 4.3 Environment model builder

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 environment measurements in a numerical model that can be used by the other parts of the algorithm to navigate the environment. The model used is based on a numerical implementation of the Artificial Potential Field or APF (see chapter 3, section 3.1 for the theoretical explanation of this method). The basic idea is to associate to each point of space a potential value that is higher near obstacles and lower near the goal. This potential field can be exploited to reach the goal by moving toward the direction of minimum potential. To create the APF model, the environment is discretized in volumes of space all of the same size that depends on a parameter: the model resolution. A potential value is then associated to each of these volumes, in this way, the APF model resulting in a matrix, where each cell corresponds to a small volume and holds the potential value associated to it. In each position of the matrix, 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 various terms of equation (4.1) correspond to different elements in the environment. They are computed and managed separately and summed only when all parts have been updated. So, each UAV computes the different "layers" and sums them to have the final environment model. The different parts of which the APF environment model is composed are shown in figure 4.3, where 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, as expressed by the equation:

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

The attractive layer $U_{attr}$ is updated every time the Coverage Planning algorithm computes a new goal for the UAV.

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 an obstacle is 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 some temporary information about the environment, mainly linked to the management of local minima in the APF (this point will be discussed in more detail in section 4.5.4). In the image, local minima are evidenced in red and a temporary repulsive field can be observed.

The sum of the contributions of the layers above returns the final environment model. Each point in space has a potential value corresponding to the linear sum of the contributions. The matrix with all the potential values (one for each volume of space) is the one that will be passed along to the other parts of the algorithm.

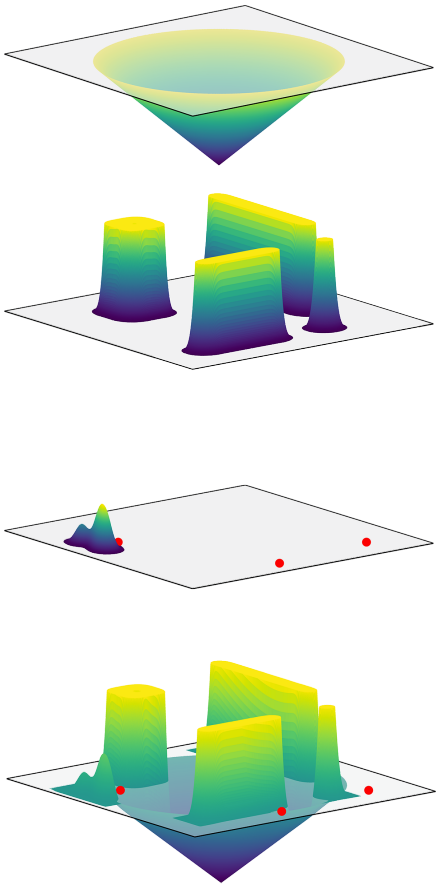

**Figure 4.3:** Layers forming the environment model used in the APF algorithm. The bottom one is the final environment model, corresponding to the sum of all the ones above.

The main advantage of using this 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 thanks to the superposition principle. For example, when the goal position changes, it is sufficient to subtract the current attractive layer, re-compute its value and sum it back into the model.

In fact, since the environment is not known a-priori, all the layers are continuously changing to incorporate new information about the map in real time. For the attractive layer, the update happens every time the Coverage Planning Algorithm computes a new goal for the UAV. When this happens, the attractive layer is removed and completely recomputed as a cone having vertex in the new goal position. For the obstacle layer the procedure is a bit different, since this layer is never deleted (the UAV needs to keep track of all the obstacles positions). In this

case, the potential field is built iteratively. In fact, at the beginning of the exploration there is no available information about the obstacle (e.g. their number, position, shape...). For this reason, the obstacle layer is initialised as empty and filled as the exploration proceeds.

### 4.3.1 Vision

Every time the position of a new obstacle is detected through sensors or communicated by the other UAVs, the obstacle layer is locally updated around the obstacle position. In fact, obstacles are assumed to have only a local effect: after a certain distance from them (*safe distance*, which is another model parameter) their presence can be neglected. To achieve this behaviour in the environment model, a small matrix is placed in the obstacle location. This matrix raises the potential value in the exact location of the obstacle and around it, helping with the collision avoidance task. Figure 4.4 shows the procedure of obstacle detection and placing in the environment model. An interesting feature of the Environment Model Builder

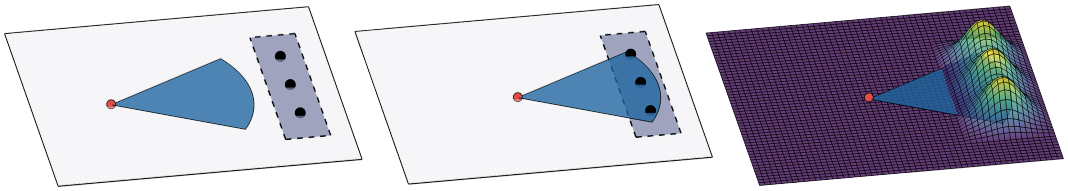

**Figure 4.4:** Obstacle detection and creation procedure in the the environment model builder. In the leftmost image, the obstacles (black dots) are out of vision range. In the image in the middle obstacles are detected. In the rightmost image the potential is raised around the newly discovered obstacles.

that can be observed in Figure 4.4 is the way the environment is discretized. The small wall in the leftmost figure is divided in three point-like obstacles, each one located in the centre of one elementary volume of space. As already introduced above, this happens since the environment is discretized in small portions of space in order to be represented as a matrix.

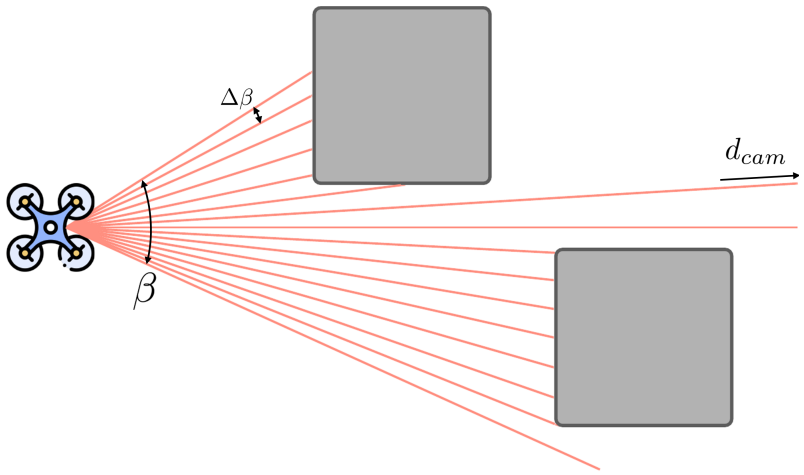


**Figure 4.5:** Simulation of the vision part of the algorithm. The parameters used to set the vision simulation are shown. All the parameters used in the environment model builder are explained in section 4.3.3.

### 4.3.2 Obstacles prediction

As can be observed in 4.6, as the agent moves it discovers new obstacles but, for simplicity, in all the training and validation environments only rectangular obstacles have been considered. This limitation has been introduced to allow the implementation of a shape-prediction algorithm (based on OpenCV) to estimate the shape and dimension of the obstacles without knowing all their contour points locations. This allowed to speed up the exploration process and to manage more easily obstacles. An extension of the shape-prediction algorithm in order to taking into account every possible obstacle shape (such as "L", "T" or "C" shape) is possible, but has not been implemented in the work yet. The prediction algorithm exploits the function "$find\_contours$" of the OpenCV library. This function takes as input the obstacle map, that is a matrix where obstacles are represented as one, whereas the other points are zeros. This matrix is rescaled in the interval $[0, 255]$ in order to increment the contrast between obstacles and free areas. In order to help this function find the contours of closed figures recognising them without the perimeter of the obstacle being continuous, a blur function is used. In this way, close points are seen as continuous walls. After the points that belong to the edges of the identified figures have been collected, the figure is closed assuming the presence of obstacles within the area enclosed by the most extreme edges. In this way, even if the figure has not been entirely discovered, it is possible to predict and anticipate the discovery of a part of the obstacle as can be seen in figure 4.6.

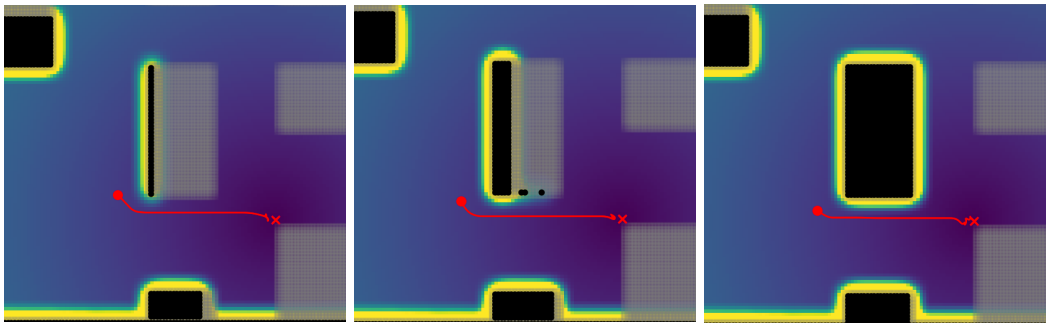

**Figure 4.6:** Detection and prediction of the full shape of an obstacle in the environment model builder. The grey areas are the unexplored obstacles whereas the black ones are the already explored obstacles. In the leftmost image the obstacle is identified as a wall whereas in the rightmost image the full shape of the obstacle is revealed.

When the UAV approaches an obstacle the first part seen is identify as a wall, then as the exploration progresses and new parts of the obstacle are discovered, the whole shape is revealed. Once two sides of the same obstacle are completely seen, the whole shape is predicted and added as a rectangular obstacle in the drone's memory also considering this area explored.

### 4.3.3 Parameters and routine of the environment model builder

Having explained the role and update strategy of each layer, the working of the Environment Model Builder routine is quite straightforward to illustrate. The model builder routine is called at a frequency $f_{EMB}$ and every time it runs it follows the operations listed in Algorithm 4.3.

**Algorithm 4.3** Environment Model Builder routine

**if** goal position $x_g$ has changed **then**
    subtract $U_{attr}$ from the model
    recompute $U'_{attr}$ for the new $x_g$
    subtract $U_{exp}$ from the model and reset it
**end if**
**if** new obstacles are detected or communicated by other UAVs **then**
    check if obstacles are already known
    **for** new obstacle $o$ **do**
        sum the local potential matrix in the obstacle position $x_o$ as in Figure 4.4
    **end for**
**end if**
**if** local minima is detected **then**
    sum the local potential matrix into the $U_{exp}$ in the local minima position $x_{lm}$
**end if**
subtract the mobile obstacle layer from the model
recompute the mobile obstacle layer with the new positions communicated by other UAVs
sum the mobile obstacle layer back into the model
**end routine**

The parameters used in the environment model builder are shown in the table 4.1. Where safe distances, peak values and vision parameters are defined. The value of the safe distance must be intended as the dimension in number of cells of the Gaussian matrix (that is a matrix with a Gaussian distribution) added in the potential matrix in order to add the repulsive zone, whereas the peak value is the maximum potential value that can be assumed. In figure 4.7 an example of a matrix with a Gaussian distribution used for the obstacle is shown and in figure 4.8 various peak value with their distribution are revealed.

**Figure 4.7:** Matrix with a Gaussian distribution used for obstacle avoidance. The higher values of the matrix are visualised with the yellowest colour and they are the value associated with the highest risk (in this case the centre is the obstacle position.)

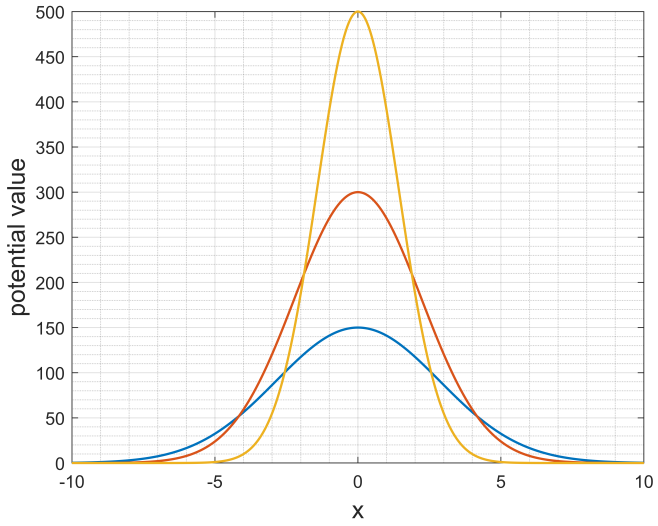


**Figure 4.8:** Distribution of the potential value within the obstacle matrix seen from a one dimensional point of view. Different distributions with disparate peak values are indicated.

The other parameters are the map resolution, the speed of the drone, the attractive constant, that indicates how steep is the cone centred in the goal, the vision parameters and the $N_{lm-steps}$. The last one is the number of steps that the drones have to perform with the assistant algorithm in case of detection of local minima. Instead the vision parameters include the maximum range of vision $d_{cam}$, the angle describing the cone of vision $\Delta\beta$ and how densely filled the cone proportional to the minimum angle between consecutive vision ray $\beta$. Follows the table with the parameters of the environment model builder used in this thesis work.

**Table 4.1:** Parameters of the environment model builder.

| Parameter Name | Variable | Value |
|---|---|---|
| map resolution | $r_{map}$ | 0.10 $[m]$ |
| drone speed | $\delta_{drone}$ | 0.12 $[m/step]$ |
| attractive constant | $k_a$ | 35 |
| max steps in local minima algorithm | $N_{lm-steps}$ | 50 |
| mobile obstacle safe distance | $\Delta_m$ | 150 |
| fixed obstacle safe distance | $\Delta_f$ | 150 |
| local minima obstacle safe distance | $\Delta_{lm}$ | 150 |
| peak value of mobile obstacle | $\rho_m$ | 500 |
| peak value of fixed obstacle | $\rho_f$ | 300 |
| peak value of local minima obstacle | $\rho_{lm}$ | 50 |
| vision range | $d_{cam}$ | 3 $[m]$ |
| vision resolution | $\beta$ | 0.25° |
| vision angle | $\Delta\beta$ | 50° |

## 4.4 Coverage

This section is dedicated to the algorithms in charge of managing the position of the temporary goals in order to coordinate the drones in a efficient way. The first approach is the RL one, where a specific trained agent computes the goal of each UAV considering their relatives positions and the unexplored areas. The second approach, used only for reference purpose, is a explicit algorithm called *K-means clustering*, that it is very useful when the total exploration is greater than $85\%$.

### 4.4.1 Reinforcement Learning coverage approach

As explained previously, the task of the coverage agent is to provide a temporary objective for each UAV. The motion of each UAV toward its designed temporary goal leads to the observation of new regions of the environment, contributing to the exploration process. The simultaneous use of the coverage algorithm by each member of the fleet leads to the collective exploration. A first requirement on the Coverage agent is that the UAV fleet must be leaderless. This means that the algorithm is distributed, and each drone computes its own goal position. This increases the flexibility of the Coverage agent, since it is not required that the UAVs can communicate with each other. It is worth noting that each drone use the same agent to compute the goal position, but the only difference is the input state that varies according to the coordinate of the UAV itself and the relative position with the other ones. This approach leads to the implementation of a coordinated behaviour, where each UAV try to cover as many areas as possible taking into account the position of other drones and the unexplored areas thus avoiding coverage of previously explored areas or trajectory entanglements. A second requirement regards the scalability of the Coverage algorithm. The Coverage agent is trained to work for any number of UAVs in the fleet (at least in a reasonable range, e.g. between 2 and 10 UAVs). This requirement directly translates into a constraint on the input of the agent NN. In fact, the input must be independent from the number of UAVs, but at the same time, it must contain all the necessary information for the agent to take into account all the other UAV positions.

The requirements discussed above lead to the design of a state composed of two parts. The first is a 2-node input layer containing the normalised current position of the UAV in the range $[0, 1]$. The second is a $n \times m$ matrix containing all the necessary information about the coverage state of the map and the other UAV position. The matrix is initialised as a matrix of zeros with the same dimension of the environment and is build in the Environment Model Builder summing the following components:

- obstacles found during exploration are added as ones;
- covered areas represented by ones are summed;
- the areas of influence of the other UAVs are represented by a matrix with a Gaussian distribution (whose dimensions depend on the map size) summed to the input matrix.

This matrix is finally normalised into the interval $[0, 1]$ obtaining the input state of the NN. The resulting state contains all the required information about the environment status and highlights the best allowed position for the future goals, i.e. the regions of the matrix with lower values. The Gaussian matrix associated to other UAVs represents their "area of influence",

i.e. the region they are more likely to explore and to not overlap. This way, exploration paths should be more efficient and the areas to explore are well divided between UAVs. An example of the resulting input state is displayed in Figure 4.9.

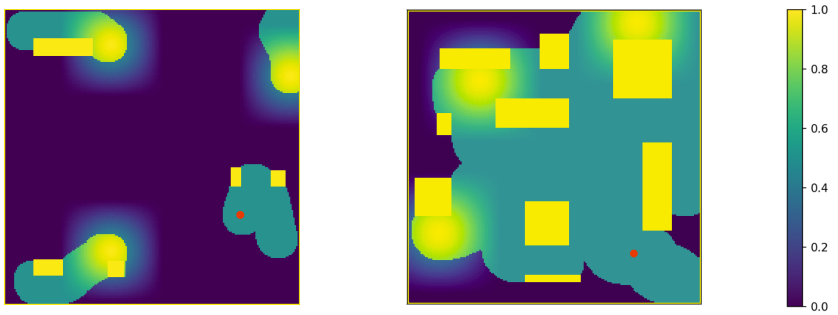


**Figure 4.9:** Example of two Coverage agent input states. The matrices displayed have dimension $[200 \times 200]$, representing an environment of size $[20\text{m} \times 20\text{m}]$ with a resolution of $0.1\text{m}$. The colour bar shows the value of each point.

The NN designed to build the Coverage agent is shown in Figure 4.10. In the image, the main sections composing the NN can be observed. The input state, divided into the two channels described above, is processed by initial dense layers. Here, the most relevant part is represented by the convolutional part that elaborates the matrix input, applying some "filters" to the input in order to extract relevant features that are then passed to the fully connected section, where the two inputs are merged and elaborated together. All the neurons in the dense and convolutional layers are associated with a ReLU activation function. Finally the output layer of the agent is composed of 2 neurons holding a sigmoid activation function $\sigma(x) = \frac{e^x}{1+e^x}$. The two outputs correspond to the computed position of the UAV temporary objective normalised with respect to the environment size. It is sufficient to multiply the outputs for the environment dimension to obtain the position of the designated goal.

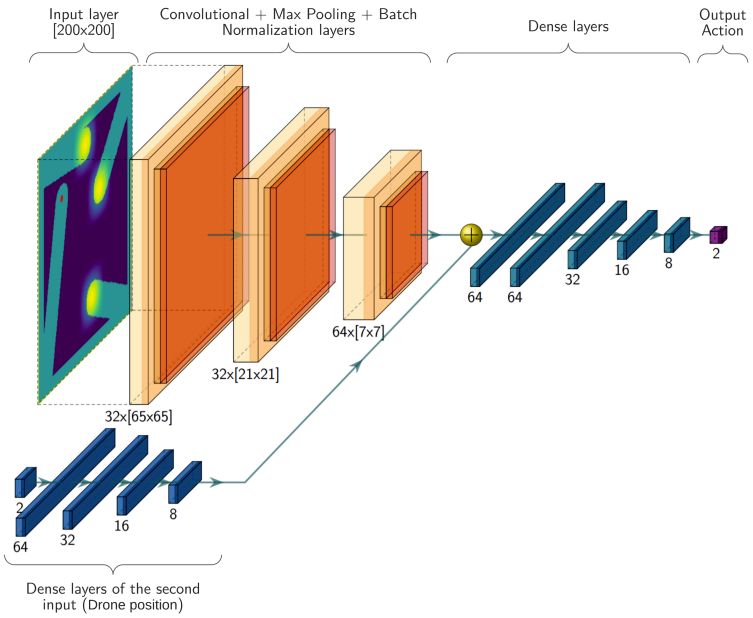


**Figure 4.10:** Representation of the neural network structure of the Coverage actor.[14]

### 4.4.2 K-means clustering coverage approach

A different approach to the coverage problem is through explicit programming. The goal is to obtain $n$ points evenly spaced over an area $A$ of interest. The points are assigned to the drones, that acquire them as their new objectives. The focus of this algorithm is to keep the drones well spaced over the map, avoiding overlapping in their paths. The implemented method consists of a function that first defines an area of interest $A$ where the exploration objectives has to be placed, in this case is the unexplored parts of the map. Then, this area is divided in parts (called Voronoi cells) following the Lloyd's Algorithm, see section 3.3. The centroids of these cells (i.e. the geometric barycentre of each area) are the drones goals. Several modification have been made in order to increase the algorithm performance. These improvements are obtained following the routine presented in Algorithm 4.4.

**Algorithm 4.4** Routine for the K-means clustering coverage approach

```
compute the list of points that represent the location of the unexplored areas
find the best number of cluster in order to minimise the variance of the points from the centroid
compute the n_c centroids with the Lloyd's algorithm
for centroids from 1 to n_c do
    if i_th centroid is not already present in the goals list and in the memory list then
        the centroid is inserted in the goals list
    else
        the centroid is skipped
    end if
end for
if a goal in the goals list is inside an already explored area then
    the goal is deleted from the list
end if
the closest goal is assigned to the drone that call the algorithm
goal assigned is deleted from the goals list and added to the memory list
```

This code, dedicated to compute the goals for the drones, is executed by a ground station that communicates with them. In this way when an UAV reaches a goal it recall this algorithm inside the ground station returning a new objective. This approach guarantees that the average distance between the goals is maximum, however This does not necessarily result in an optimal exploration strategy, especially for the fact that the UAV positions and their paths are not taken into account. This approach results quite efficient during the last 15% of the environment exploration, where the unexplored areas are only few fragmented part of the map. 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 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, therefore the last goals are placed with this explicit algorithm.

[14] The figure shows only the actor Neural Network, that is the NN used for choose the correct action, whereas the critic one (not shown in figure) it is composed of an additional input, that is the action of the actor.

## 4.5 Path Planning

In this section, the path planning algorithm will be presented. Two types of path planning algorithm will be developed. The first one is based on the traditional APF algorithm [6] and is mostly used as reference. The second one is based on an innovative Reinforcement Learning approach and constitutes the core of the Path Planning algorithm.

### 4.5.1 Artificial potential field path planning

The traditional artificial potential field path planning is implemented in this work only for a reference purpose. In order to adapt it to the environment model builder some changes has been made. The new algorithm exploits the pre-computed potential map making it faster and relieving the computational cost of calculating the repulsion vector from the obstacles step by step. This simple approach to the path planning problem is quite robust and reliable, but has some drawbacks. The main issue of this algorithm is that the drone is subject to stalling in the local minima of the potential map (this issue will be well explained in the section 4.5.4). Another issue is the quality of the output trajectories that are non-smooth and dynamically inefficient. This is due to the fact that it is not able to foresee the obstacles in advance. This improvement of the traditional APF path planning method, still based on the numerical computation of the gradient starting from the APF environment model, is obtained following the routine presented in Algorithm 4.5.

**Algorithm 4.5** Routine for advanced numerical computation of the APF negative gradient

initialise sweep angle $\alpha = 0$, $a = +1$ and a small $\delta$ (e.g. $\delta = 0.1$)
compute the initial vector $\vec{v}$ that goes from $x$ to $x_g$ (i.e. with $\alpha = 0$)
**while** $\beta <= 180$ **do**
    **if** $U(x + \delta \cdot \vec{v}) < U(x)$ **then**
        move in direction $\vec{v}$
        break from the loop
    **else**
        $a = a \cdot -1$
        $\alpha \leftarrow \alpha + a \cdot \beta$
        compute a new vector $\vec{v}'$ equal to $\vec{v}$ but rotated of $\alpha$ with respect to point $x$
        repeat cycle with $\vec{v} = \vec{v}'$
    **end if**
**end while**

The basic idea in 4.5 is to go straight toward the goal, following vector $\vec{v}$ which goes from $x$ to $x_g$. If the potential in the direction of the goal is lower than the one in $x$, the UAV will follow direction $\vec{v}$ to move. Otherwise, a new vector $\vec{v}'$ is computed rotated of an angle $\alpha = 1°$ from $\vec{v}$. If the potential in direction $\vec{v}'$ is lower than the one in $x$, $\vec{v}'$ will become the direction of motion. Otherwise, the procedure is repeated with $\alpha = -1°$. The procedure is repeated until a suitable motion direction is found (i.e. a direction leading to a lower potential, even if it is not the lowest possible in the neighbourhood of $x$). The strategy followed here privileges directing toward the goal over going in the direction of the lowest possible potential. This second approach performs quite well and present a consistent and efficient way to implement a path planning algorithm based on an APF model.

### 4.5.2 Reinforcement Learning APF path planning

The reinforcement learning artificial potential field path planning agents is trained in order to contribute to the realisation of the exploration algorithm. Inside each UAV the temporary objective computed by the coverage agent is passed to the Path Planning agent. The task of this agent is to compute a suitable trajectory to reach this goal. As already introduced in the Section 4.2, the Path Planning agent exploits the APF environment model to get all the information necessary to perform its computations. The neural net designed for the path planning agent will be illustrated below and it is shown in figure 4.12.

At each time step, a portion $s(t)$ of the APF model is extracted and passed as input state to the Path Planning agent $\mu_\theta(s(t))$. The state has fixed dimension ($[75 \times 75]$ cells, representing a $[7.5m \times 7.5m]$ area with a resolution of $0.1m$) and represents the environment in the neighbourhood of the UAV. A wider portion of the potential map, taken as a matrix, means that the agent is able to use more information to calculate its trajectory and therefore has a wider range of perception. The state is then manipulated in order to promote the learning process of the agent. Firstly the state is clipped to a maximum value of potential, in order to restrict the possible states. Then a differential state is set, i.e the value of the potential of the cell where the agent is located is subtract to all other cells, in this way the value of this cell is set to 0, whereas the other cells of the state will acquire a value related to it. After this the state will assumes values in the interval $[-1000, 1000]$ that are mapped into the range $[-1, 1]$ further reducing the variability of the state and compressing the input to the NN in order to better train the agent. The state is finally ready to be given as input to the NN.

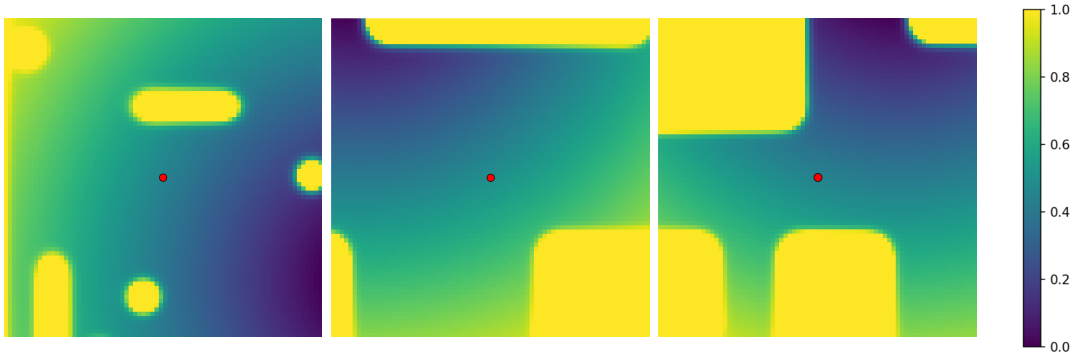


**Figure 4.11:** Three example of input state. An higher value of risk is associated with an higher value in the state matrix.

The NN designed to build the Path Planning agent is shown in Figure 4.12. It is composed of three convolutional layers, with ReLU activation functions, that processes the input matrix (a portion of the potential map) seen as an image, extracting features by mean of some trained filters. Convolutional layers are fundamentals when operating with images, because they are able to extract information from them in a very efficient way. In this case one of the main features to identify is the obstacles conformation of the map, in this way a consistent obstacle avoidance algorithm will be produced. Afterwards, these "abstracted" features are passed to six fully connected layers where are elaborated to produce the output value. The output of the last dense layer is mapped between $0$ and $1$ through the sigmoid activation function $\sigma(x) = \frac{e^x}{1+e^x}$. Then this value is rescaled from $0$ to $2\pi$ in order to convert it into an angle $\psi$, corresponding to an angle in the plane that is the best direction to assume in order to reach the goal.

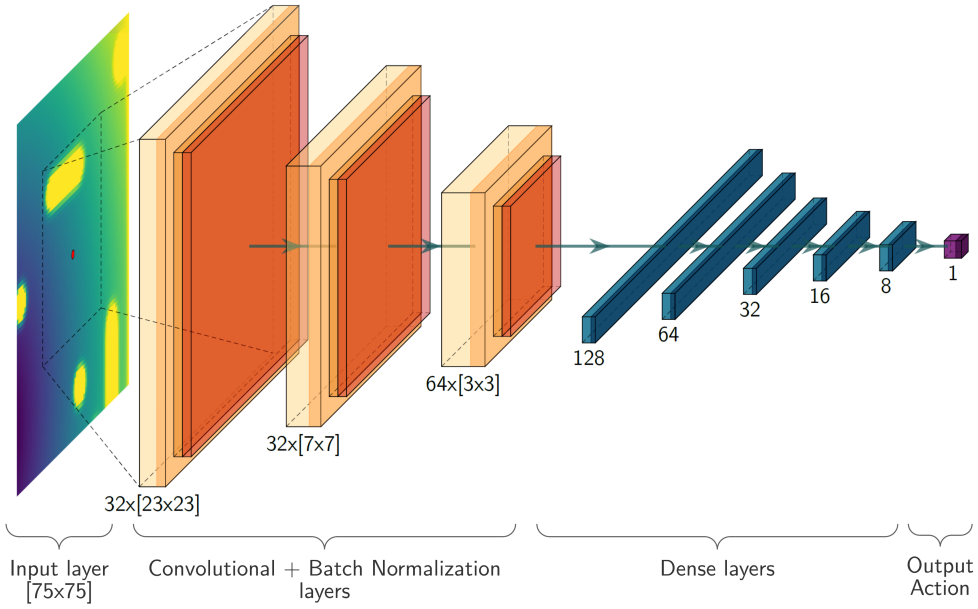


**Figure 4.12:** Neural net of the actor used in the path planning. The input of the neural net is a portion of the potential map around the drone and after the convolutional and dense part the output is obtained as the direction to follow.

### 4.5.3 Trajectory interpolation

The direction obtained is called by the path planning algorithm $n$ times in order to compute a series of points, interpolate them and use the resulting trajectory instead of a single direction to follow. The points are interpolated with a *B-spline* curve, that is a composite curve, build with joining adjacent segments of polynomial curves with appropriate continuity. The $C_2$ type conjunction is used. In this type of conjunction the last point of the first curve coincides with the first point of the second curve and at these two points there is the same tangent (first derivative) and curvature (second derivative). The routine of Algorithm 4.6, with the interpolation process, is performed to compute the desired path. The trajectory resulting from the interpolation of points computed by the agent can be seen in the figure 4.13.

**Algorithm 4.6** 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: compute the trajectory by applying a fitting function to the $n$ points $(x_0, x_1 ... x_n)$
8: pass the trajectory to the controller to follow it

The main limit of the Path Planning routine is the fact that the distance between each step is fixed ($\delta$). A significant improvement could be obtained by adding a second output node to the agent, representing the desired speed. This way, one of the agent outputs would be a velocity vector, and it would be possible to compute more dynamically efficient trajectories.

However, this improvement in the NN structure has not been introduced due to the difficulty to define an effective reward function for the second node output. Therefore, this enhancement is postponed to future developments.

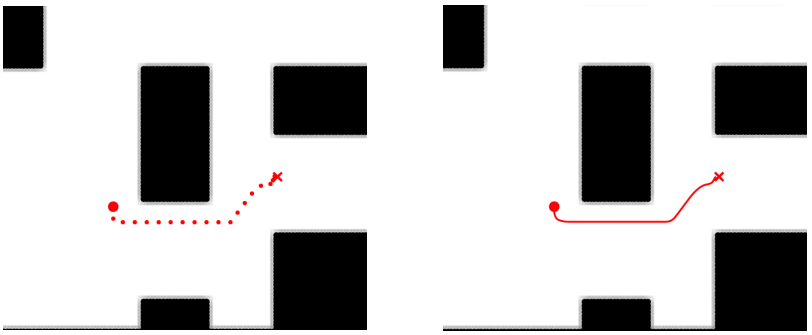

**Figure 4.13:** Interpolation of a series of points computed by the path planning agent.

The only exception to the use of the RL path planning is represented by some unwanted situations in which the UAV is led to approach an obstacle closer than the safe distance. This 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 unexpected input statuses, sometimes the NN prediction fails and the result is a bad trajectory. To tackle this issue a good solution is to pair the RL agent with a "classical" deterministic algorithm that comes into operation only when a failure on the part of the main algorithm is detected. In our case a specific agent trained during risky situations is exploited. This security implementation significantly reduced the risk of a critical malfunctioning of the RL agent.

### 4.5.4 Management of local minima

An issue encountered in the model validation has been its management of local minima. 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 problems ($< 10\%$ 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 which the penalty for going against the potential was lowered and the time penalty was increased). This second agent produces, in general, sub-optimal trajectories, but is able to effectively manage all local minima in which the "main" agent would get trapped. If the agent falls into a local minima even during the use of the assisting agent a series of operations are performed. These operations involve placing a Gaussian matrix directly under the drone centred in the matrix cell with the higher potential with the aim of compensating the potential pit and creating a "slide" to follow in order to exit the local minimum. At this point there are two algorithms that can be used, the RL assisting agent and the local minima APF algorithm, which is a modified version of the classic implementation of the APF. The performances of the local-minima-avoidance agent are shown in the simulations and results section (section 6).

### 4.5.5 Emergency Collision Avoidance

The path planning agent is in charge of computing the trajectory towards the objective. However, as any RL agent it can fail since during the training process it is nearly impossible to feed it all possible states. Therefore, even if NN theoretical study guarantees the stability of the output with respect to new states, it can happen that the agent fails to find the best trajectory and returns a sub-optimal one[15]. This is one of the most critical drawbacks in the use of RL agents, and can lead to dire consequences. In particular, the most critical failure happens when the RL agent takes the trajectory too close to an obstacle leading to collisions and resulting in a loss of control of the UAV or even to the loss of the UAV itself. To tackle this problem, an Emergency Collision Avoidance system (ECA) is implemented. For the purposes of this thesis, a quite simple but effective ECA algorithm has been implemented. The ECA is a sensor-triggered system. Whenever one of the proximity sensors on board of the UAV[16] detects an obstacle closer than a given safe distance, the ECA is activated. The ECA overwrites the path planning algorithms and computes a new trajectory that has the objective to keep the UAV at a safe distance from the obstacle. For the ECA operations the "*assisting*" agent is used, that during the training has developed the capability of maintain a safe distance between the obstacles. in the event of failure of the algorithm and therefore in the event of extreme approach of the drone to an obstacle even using the ECA, another algorithm will be activated. In this specific case will be adopted the advanced APF path planning described in Section 4.5.1.

The interactions between the various algorithms and callbacks during the local minima detection and the activation of the ECA are described in the following figure.

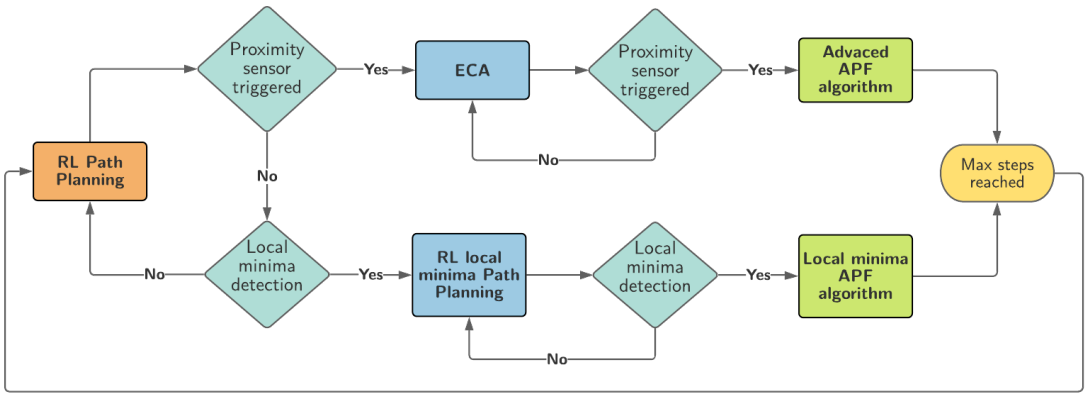


**Figure 4.14:** Logic diagram of the path planning algorithm.

[15] NN agent robustness is one of the most critical fields of study in the AI area.

[16] we assume that the UAV used as test has some form of proximity sensing tool, as for example the ultrasonic sensors of the reference UAV presented in Section 4.7.

## 4.6 Dynamics and Control

### 4.6.1 Dynamics

A dynamic model can be used to take in account the UAV dynamics and optimise the trajectory planning according to it. In practice, during this thesis the UAV dynamic model has been used only in the ROS simulation phase. The dynamic model used for the simulations is almost entirely based on Chapter 16 of the *Handbook of Unmanned Aerial Vehicles* [14]. The model proposed by Powers, Mellinger and Kumar [14] starts from the definition of a motor model to represent the UAV rotors. The vertical force generated by each rotor is equal to:

$$F_i = k_F \, \omega_i^2 \tag{4.3}$$

Each motor also produces a moment equal to:

$$M_i = k_M \, \omega_i^2 \tag{4.4}$$

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

$$\dot{\omega}_i = k_m(\hat{\omega}_i - \omega_i) \tag{4.5}$$

where $\hat{\omega}_i$ is the desired speed of each rotor (as computed by the controller). Equation (4.5), as well as the parameters $k_F$, $k_M$, $k_m$, are obtained from experimental tests. The motor model and equations are necessary to build the dynamic equations. Before analysing them, it is necessary to define a couple of reference frames.
The 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 $O_M$. A rotation matrix $[R]_M^F$ can be constructed to translate the UAV attitude between the two reference frame. The shape of the matrix depends on the choice of 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 Euler angles), which will be 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:

$$m \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + R_M^F \begin{bmatrix} 0 \\ 0 \\ F_1 + F_2 + F_3 + F_4 \end{bmatrix} \tag{4.6}$$

For what regards the attitude and angular accelerations, the Euler equation for the UAV dynamic system are:

$$\begin{aligned} I \begin{bmatrix} \ddot{\alpha} \\ \ddot{\beta} \\ \ddot{\gamma} \end{bmatrix} &= \begin{bmatrix} l(F_1 - F_3) \\ l(F_2 - F_4) \\ M_1 - M_2 + M_3 - M_4 \end{bmatrix} - \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} \times I \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} \\ &= \begin{bmatrix} l & 0 & -l & 0 \\ 0 & l & 0 & -l \\ \rho & -\rho & \rho & -\rho \end{bmatrix} \begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \end{bmatrix} - \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} \times I \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} \end{aligned} \tag{4.7}$$

where $l$ is the distance between the centre of mass and each propeller, while $\rho = \frac{k_M}{k_F}$ is the ratio between lift $F_i$ and drag $M_i$ (as can be derived from the motor model of equations (4.3) and (4.4)). In equation (4.6), the vector of external forces contains only gravity along the direction $z_F$. An interesting aspect that emerges from the dynamic equations above is that, of the degrees of freedom of the UAV, only four of them are linked to an input force, because $\ddot{x}$ 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 [14] there is a complete analysis of the system *differential flatness*, which results in the fact that, despite the under-actuation of the system, it is able to follow any desired trajectory in space.

### 4.6.2 Control

The control algorithm is in charge of computing the motor inputs that allow to follow the desired trajectory. Motor inputs are usually rotor speeds, which are translated into voltages and then fed to the Electronic Speed Controller (ESC), which in turns translates the voltages in power values and supplies them to the motors. The controller has not been specifically 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 derive a dynamic model of the UAV 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 4.15. The control loop of Figure 4.15 takes the trajectory

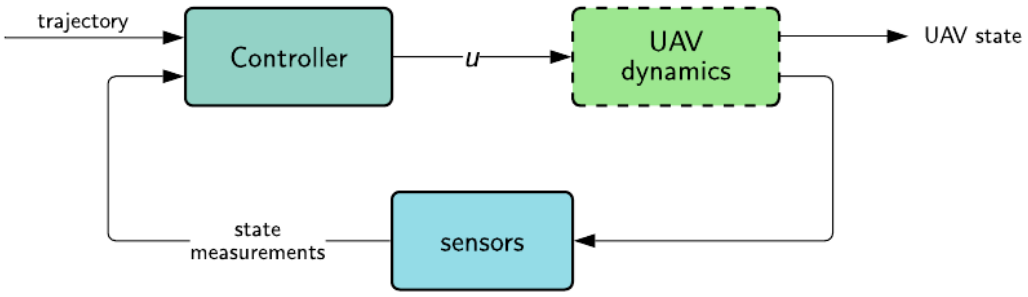


**Figure 4.15:** Block scheme of the control loop algorithm.

computed by the Path Planning Algorithm as input, and computes the rotor speeds (collected in the vector $\boldsymbol{u}$). The UAV turns the rotors at the desired speeds and, following its dynamics, it moves 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 sensors on board of the UAV and fed back to the controller to compute the next rotor speed values.

## 4.7 Reference UAV

The algorithm described above and all its subsections have been developed to be as generic as possible. This means that the algorithm 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), different types of camera could be used to detect the presence and position of obstacles, since the algorithm must receive a list of obstacles coordinates as input in order to build the environment model. The only part that would differ from one UAV to the other is the dynamic model. Up to now, this model has not been required. However, to optimise the trajectory as well as to design a suitable flight controller, the dynamic model of the drone is required. However, to get the dynamic model it is necessary first to select a *reference UAV*. The UAV that has been selected is the one developed by the DRAFT team[17] (a student team working within the PIC4SeR research group[18]) in order to compete in the Leonardo Drone Contest.[19]

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

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

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

[19] https://www.leonardocompany.com/it/innovation/open-innovation/drone-contest

The UAV, shown in Figure 4.16, is an octacopter (the structure is that of a X4 quadcopter but each arm holds two coaxial propellers) equipped with a set of cameras and sensor that allow it to sense the surrounding environment and collect information about it. The inertial properties used for the dynamic model are the ones of the reference UAV. The total mass $m$ is 2.8kg, while the inertia tensor $I$ (computed in the centre of mass, i.e. in the origin of the mobile reference frame) is shown in Equation (4.8). The technical specifications of the UAV and its sensors are listed in Table 4.2.

$$I = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix} = \begin{bmatrix} 0.0275 & 0.0005 & 0.0002 \\ 0.0005 & 0.0330 & -0.0002 \\ 0.0002 & -0.0002 & 0.0307 \end{bmatrix} [kg \cdot m^2] \tag{4.8}$$

**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) |

# 5. Training process

In this chapter the training process of the two agents is discuss. Firstly the training and the validation set along with the general settings for the training procedure are explained in section 5.1. After this section the training of the path planning agent is analysed in every detail (section 5.2.1). Finally the training of the coverage agent is examined in the same way of the path planning agent (section 5.2.2).

## 5.1 Training set & validation set

First of all the training and validation data are presented. This mostly consists on the creation of simulation environments used to train the RL policies in order to test and compare the different agents. The training set has been generated using a generator function of random maps, whereas the validation set has been generated with the manual positioning of the obstacles in the maps.[20] The map parameters that have been used to generate the training and validation environments are collected in Table 5.1:

**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 |
| | average complexity | 7.15% | |
| **validation** | dimension [m] | 20 | 20 |
| | resolution [m] | 0.1 | 0.1 |
| | cell number | 200 | 200 |
| | average complexity | 30.68% | |

The training set composed of 200 maps with a medium-low complexity has been generated in order to show to the agent as many scenarios as possible. The map complexity measure the density of obstacles in the map. This metric is very helpful in order to understand how complex is the exploration process in a specific map, defined as $\mathcal{C} = \frac{A_{tot}}{A_{obs}}\%$. where $A_{tot}$ is the total environment explorable area while $A_{obs}$ is the area occupied by obstacles. In these sets only

[20] Both functions have been written in Matlab language.

linear and rectangular obstacles are considered, thus the only way to modify the environment complexity is to change the obstacles number, shape and positioning. In particular, positioning and shape 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 agent. The possible shapes for the obstacles are points, wall, rectangles, "T", "L" and "C" shape. The points obstacles have been used in the first training set with poor performance result for the agent in the training process, whereas T and C shape obstacles creates very complex environments, which makes policy learning difficult. Moreover the prediction shape function implemented in the algorithm (as discuss in chapter 4) works only with rectangular obstacles since it recognises edges and corners starting from an image (in this case is the obstacles matrix, where obstacles are represented by ones and the free areas are represented by zeros). Some example of training maps are shown in Figure 5.1 plotted as 2D obstacle maps (viewed from above).

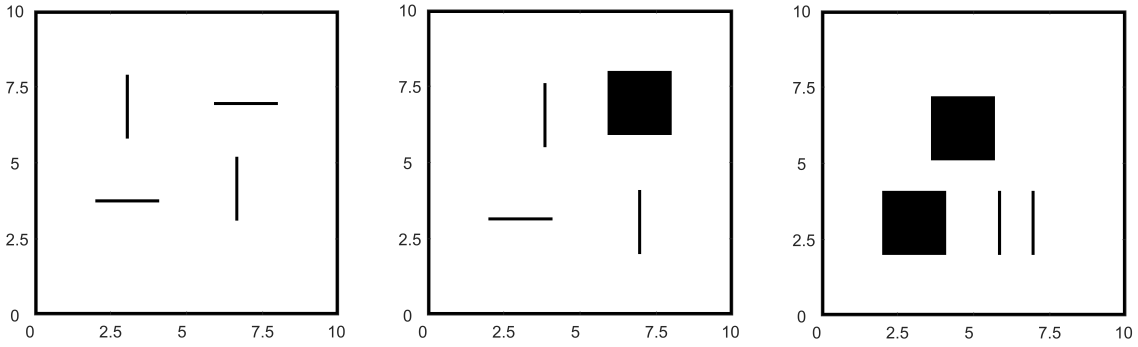


**Figure 5.1:** Examples of training maps of medium complexity. Dimensions are in meters.

The agent uses the same maps to learn all the aspects of its policy that are path planning, obstacle avoidance and local minima management, 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 in the training process. Validation maps have been designed to be more complex in order to test the agent's performances. The three validation maps on which the simulations have been performed will be displayed in Figure 5.2.

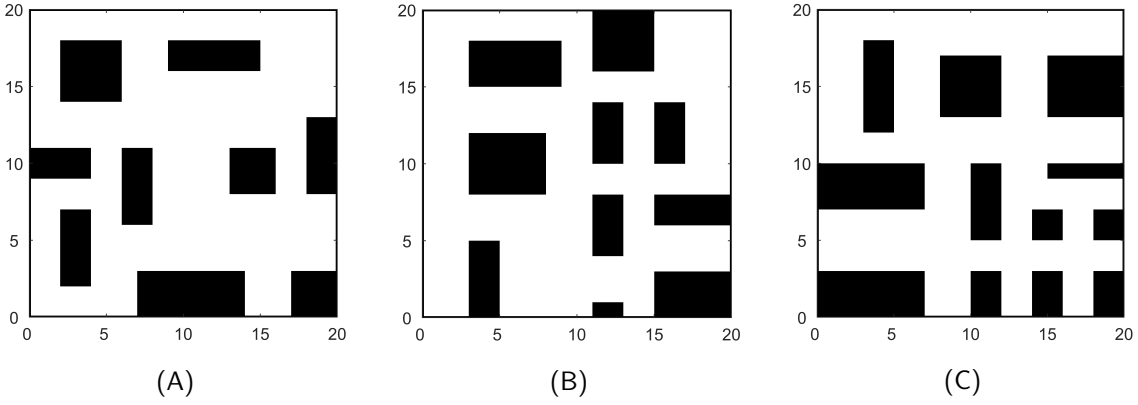


**Figure 5.2:** Validation maps used for simulations. Dimensions are in meters.

## 5.2 Agents training

In this section the focus will move to the details of the training process of the two different agents.

### 5.2.1 Path Planning agent training

The Path Planning agent training sequence is quite similar to the generic DDPG learning process previously illustrated in section 3.2.5. However, it is worth described the training process of the path planning agent and how interacts with the simulation environments. At each time step the potential matrix that represents the environment is updated by taking into account the positions of all the newly-discovered obstacles sensed by the UAV simulated vision. Once an episode is terminated the simulation is reset and a new map with a predetermined goal is picked. Training is performed only for the single agent case, in fact in this process it is not need any kind of coordination between multiple UAVs. The routine, in which each detail of the path planning training process is explained, is show in the algorithm 5.7.

**Algorithm 5.7** Learning process routine of the path planning 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 hyperparameters $\mathcal{H}$
$\#$ start main training loop
**for** $e$ in $N_{episodes}$ **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 and obtain state $s_0$
    set goal position $x_g$
    **for** $t$ in $N_{steps}$ **do**
        update the Environment Model using the Environment Model Builder routine presented in section 4
        select the action according to the current actor policy and exploration noise
        multiply the action by $2\pi$ to get an angle in radiants, then clip the result to keep the action inside the interval $[a_{\text{min}}, a_{\text{max}}]$
        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 $a_t \in [0, 2\pi]$
        use the simulated vision function to detect obstacles inside the field of view of the agent. Store in memory the positions of the obstacles.
        compute the new state $s'$
        compute the reward $r$
        store the experience sample $(s, a, r, s')$ in the experience buffer $B$
        update $s \longleftarrow s'$ for the next step
        randomly pick $N_m$ samples from $B$ and store them in the minibatch $M$
        compute the critic target $\forall i$ in $1...N_m$

**Algorithm 5.8** Learning process routine of the path planning agent (Part 2)

compute the minibatch *cost function* and use it to update the critic parameters $\phi$ by performing one step of Stochastic Gradient Descent
update the actor policy parameters by performing one step of gradient ascent
update the target networks parameters
**if** termination $\mathcal{T}(s') =$ **True then**
**end** episode $e$
**end if**
**end for**
# save data for learning progress analysis
save episode cumulative reward
**end for**

The set of hyperparameters $\mathcal{H}$ used during the training, along with their value, are listed in Table 5.2 Since different agents have been trained, the hyperparameter values reported in the third column correspond only to one specific training, which is the one that will be used for all the simulations.

**Table 5.2:** Hyperparameters used for the Path Planning agent training.

| Parameter Name | Variable | Value |
|---|---|---|
| number of episodes | $N_{episodes}$ | 250 000 |
| number of steps | $N_{steps}$ | 150 |
| step movement distance | $\delta$ | 0.12 |
| agent input state size | $N \times N$ | 75×75 |
| smoothing factor | $\tau$ | 0.005 |
| discount factor | $\gamma$ | 0.95 |
| actor learning rate | $\alpha_\theta$ | 0.0001 |
| critic learning rate | $\alpha_\phi$ | 0.001 |
| action lower bound | $a_{\text{min}}$ | 0 |
| action upper bound | $a_{\text{max}}$ | $2\pi$ |

The design of a reward function for the path planning training is one of the more difficult parts of the training work, in fact, the RL approach is extremely reward-sensitive, i.e. trainings passed from convergence to failure with a small variation in the reward parameters. Thus is quite difficult to define a function capable of distinguishing between good behaviours, like going toward the goal, and seemingly-good ones, like being trapped in a local minimum. The issue is that, given the input state defined in Section 4.5.2, both those behaviours are obtained by going towards a low potential. The chosen reward allowed the drone to not choose in every situation the lowest possible potential, that would promote, in some cases, to stuck in local minima. Another important parameter which if changed varies the agent's performance considerably during the training is the input size. An higher input size means that the agent is able to early detect obstacles and in this way taking into account of it earlier since the input size is linked with the sense radius of the UAV. The reward function with which the agents chosen for the simulation has been trained is shown above:

$$r = \begin{cases} -10 & \text{if an obstacle is hit} \\ 20 & \text{if goal } x_g \text{ is reached} \\ -w_1 \cdot \Delta U + w_2\psi + w_3\tau & \text{else,} \end{cases} \tag{5.1}$$

The first piece of the reward function is intended to punish the agent whenever it collides with an obstacle. The second one assigns a large prize to the reaching of the goal, which is the final objective of the path planning agent. Whereas the last part is composed of three components:

- $w_1\Delta U$ - this is a proportional term which 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 the term will assume a positive value proportional to the $\Delta U$ thanks to the negative sign. Whereas if the delta is higher than the potential value $100$, the term will assume a negative value always proportional to the difference between the potential value. Instead if the difference is between the potential value $0$ and $100$, the term will became flat and equal to $w_1\Delta U = -1$.
  In the first two case $w_1 = 0.1$

- $w_2\psi$ - this term punishes the agent with a negative reward if the variation of direction $\Delta\psi$ between two subsequent steps is too large. This term is meant to incentive long-term trajectory planning to produce smooth trajectories.[21]
  In this case $w_2\psi = -2$

- $w_3\tau$ - this last term is constant and represents a time penalty. It brings the agent to find the goal quickly avoiding this negative reward. This is the term that helps to reduce the number of steps in a single episode during the training.
  In the training of this agent it is assigned the value $w_3 = -0.2$

The parameters of the reward are tuned through a trial-and-error procedure. A lot of trainings have been performed in order to obtain an agent able to perform obstacle avoidance and to reach the designated goal in a efficient way. The agent with the best performances has been obtained using the hyperparameters listed in Table 5.2 and the reward function of equation 5.1. The training data collected during the training process of the agent $22$ (which is the one used in all the simulations) are shown in Figures 5.3 and 5.4. In the following figure the overall trend of the training process is shown.

[21] This term could be modified to take in account dynamical properties, such as the actuation control or the energy consumption during manoeuvres, in order to produce dynamically-optimal path

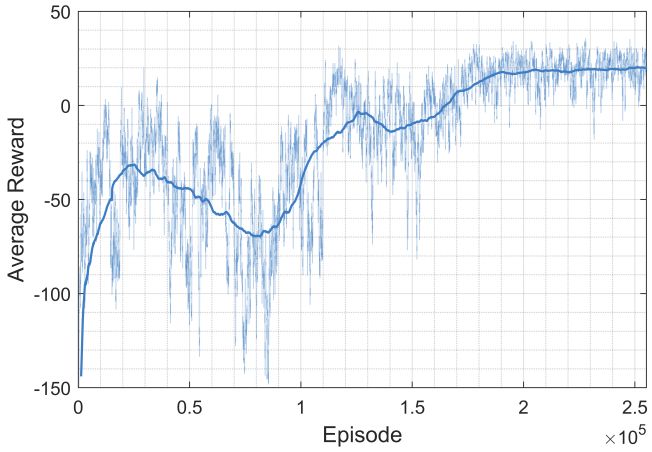


**Figure 5.3:** Average reward over the episodes during the Path Planning agent training.

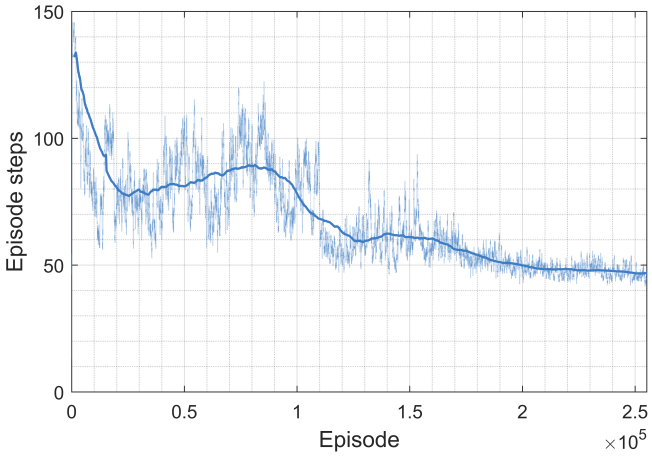


**Figure 5.4:** Number of steps per episode during the Path Planning agent training process.

Figure 5.3 shows the value of the average reward among episodes. The thin line represents every episode rewards, whereas the thick one represents the average reward computed through a moving average. It can be observed that after about $2 \cdot 10^5$ episodes the NN parameters go to convergence an 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 move towards the goal avoiding obstacles and local minima. Another metric that confirms the parameter convergence is the average episode length, shown in Figure 5.4, where after about $2 \cdot 10^5$ episodes, it becomes shorter and constant as confirmed by the small oscillations of the thin line around the average. Since the agent has learned how to act efficiently in the environment, it takes less time to reach the goal. The performances of the trained Path Planning agent in the validation environments are illustrated and discussed in Chapter 6.

### 5.2.2 Coverage agent training

In this training procedure it is necessary to simulate the actions and movement of each UAV, in fact, at each training step after the goal computation performed by the coverage actor, which is the actor NN $\mu(s)$ that is being trained, the UAVs compute their trajectory with the path planning algorithm in order to reach the goal. This steps (called substeps in order to differentiate them from normal steps) necessary to move toward it are computed using the action cycle described in chapter 4. During this movement, any new part of the environment discovered by the UAV sensors is stored and used to compute the reward. When the substeps of an UAV are terminated (and so one step is concluded), the next UAV computes its goal e and starts moving towards it. In the action cycle, communication between drones is active, thus the explored area and the information stored are shared in the swarm. The routine followed for the coverage training process is illustrated in the algorithm 5.9.

**Algorithm 5.9** Learning process 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 $N_{episodes}$ **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, obtaining state $s_0$
    **for** $t$ in $N_{steps}$ **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 4.4 taking into account obstacle positions, other UAV positions and the already explored regions
            select action according to the current actor policy and exploration noise 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 rescaled to find the goal position in space
            multiply each action by the corresponding environment side size to get the goal position [$x$, $y$] in meters, expressed with respect to the point $(0, 0)$ 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.
            **for** $i$ in $N_{substeps}$ **do**
                use the numerical APF path planning algorithm to make one step of length $\delta$ toward the current goal position
                use the simulated vision function to "explore" the area in front of the UAV. This includes obstacle detection. The size $\Delta$ of the "new" area explored is memorised to compute the reward at the end of the step
            **end for**
            \# exploration step ended for UAV $u$
            share information with other UAVs through communication channels
            \# store experience collected by UAV $u$ in the experience buffer

**Algorithm 5.10** Learning process routine of the coverage agent - Part 2

compute the new state $s'$
compute the reward $r = \mathcal{R}(A)$
store the experience sample $(s, a, r, s')$ in the experience buffer $B$
update $s \longleftarrow s'$ for the next step
**end for**
randomly pick $N_m$ samples from $B$ and store them in the minibatch $M$
# action simulation ended, proceed with NN learning
perform the Stochastic Gradient Descent and update actor policy and target networks parameters
# verify if the current state $s'$ is a termination state
**if** termination $\mathcal{T}(s') =$ **True then**
**end** episode $e$
**end if**
**end for**
# save data for learning progress analysis
save episode cumulative reward
**end for**

In order to speed up the learning process, the gradient computation and the parameters update can be done multiple times, since at each training step every UAV of the fleet adds an experience sample to the experience buffer. This strategy is not in conflict with the fact that multiple experience samples are added to the buffer, since a minibatch is used and also 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.

The hyperparameters of the Coverage agent training process are listed in Table 5.3.

**Table 5.3:** Hyperparameters used for the Coverage agent training.

| Parameter Name | Variable | Value |
|---|---|---|
| number of episodes | $N_{episodes}$ | 500 000 |
| number of steps | $N_{steps}$ | 7 |
| number of substeps | $N_{substeps}$ | 50 |
| step movement distance | $\delta$ | 0.12 |
| agent input state size | $N \times N$ | 200×200 |
| smoothing factor | $\tau$ | 0.005 |
| discount factor | $\gamma$ | 0.95 |
| actor learning rate | $\alpha_\theta$ | 0.01 |
| critic learning rate | $\alpha_\phi$ | 0.001 |
| action lower bound | $a_{\text{min}}$ | 19.99 |
| action upper bound | $a_{\text{max}}$ | 19.99 |
| influence matrix of the other UAVs | $M \times M$ | 75×75 |

The reward function used to train the Coverage agent is:

$$r = \mathcal{R}(s,a) = \begin{cases} -5 & \text{if } x_g \text{ is in an illegal location} \\ -2 & \text{if } x_g \text{ is in an unwanted location} \\ w_1 \cdot \Delta & \text{else,} \end{cases} \tag{5.2}$$

where $x_g$ is the temporary goal position and $\Delta$ is the size of the region explored while moving toward a valid goal, which is multiplied by a weight $w_1$. 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. The first two terms of the reward function aim to punish the positioning of the goals in unwanted locations, whereas the third term is the one that promotes the exploration task. The explored area is that area that the simulated vision of the drone has captured during the movement towards the goal. This is done by performing a certain number of steps using an already trained Path Planning agent.It is worth noting that the Coverage agent does not receives reward for the other UAVs exploration or goals positioning, in fact, this approach leads to the implementation of a coordinated behaviour, but not a cooperative one. After a considerably large number of episodes the training process should encourage UAVs to explore unexplored areas as individually as possible accumulating positive rewards avoiding approaches or crossing path with the other drones, that lead to a drop of the reward.

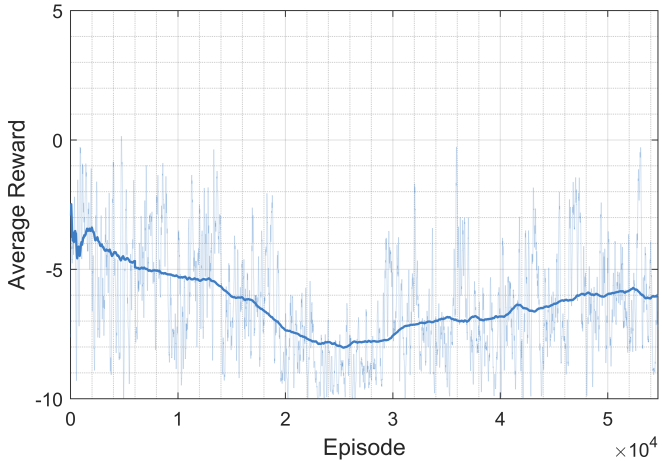


**Figure 5.5:** Average reward over the episodes during the path planning agent training.

Figure 5.5 shows the value of the average reward among episodes. As in the training process of the path planning agent the thin line represents the episode reward, whereas the thick one represents the average reward. The trend of the average reward not converges and it is always negative for the whole duration of the training. At about $2.5 \cdot 10^4$ episodes the slope turns positive and starts rising. This is due to the fact that the number of episodes sufficient to learn an optimal policy is much higher (probably $> 2 \cdot 10^5$). This indicates that the agent has did not have enough time to learn a consistent policy and able to find optimal goal positions in the training environment. The performances of the trained coverage agent in the validation maps are illustrated in the simulation and result (Chapter 6).

# 6. Simulations & results

In this chapter, the practical implementation of the algorithm is finally presented, along with the results of the tests performed. First the path planning agent as a single agent is analysed. Finally all the part that contributes to the exploration algorithm are integrated and merged in order to evaluate the whole exploration algorithm by assessing the performance of the coverage algorithm.

## 6.1 Path planning agent

### 6.1.1 Evaluation metrics

To evaluate the performances of the Path Planning agent a series of metrics needed to be defined. The metrics have been selected on the base of computational cost, safety and energetic efficiency of the path planning algorithm. 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.[22] $\bar{\Delta}_{IO}$ quantifies how sharp are, on average, the turns in a trajectory. A low value of $\bar{\Delta}_{IO}$ is associated with 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).

[22] for each discrete point $x_i$ of the trajectory (obtained by ignoring the fitting step in Algorithm 4.6), 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 angle determining 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 $x_i$).

### 6.1.2 Path planning results

Various simulations have been carry out in order to measure the performance of the path planning agent. The simulations shows in this chapter are performed in the validation maps of Figure 5.2. The RL path planning agent has been compared with two other path planning algorithms to compare its performances with other classical implementations. The other Path Planning algorithms chosen as reference are A*[23] and APF.[24] First of all, the qualitative results of the RL path planning agent in terms of quality of the trajectories are shown in Figure 6.1.

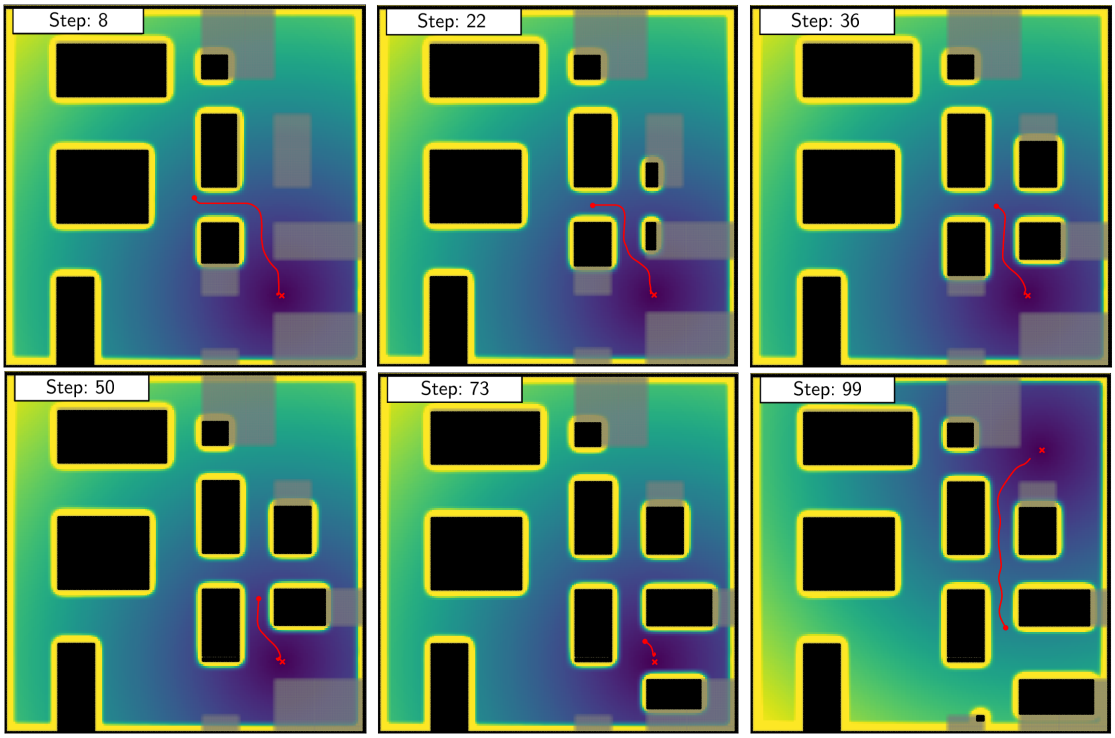


**Figure 6.1:** Simulation of path planning agent in validation map B.

Figure 6.1 shows a simulation of a path planning operation, in each frame is represented an instantaneous configuration of the simulation environment. The UAV i s represented by a red dot, the goal by a red cross and the red line connecting each other is the computed trajectory. Black regions represent obstacles known by the agent, whereas the grey ones indicate the unknown obstacles. The background represents the APF model value in each point. The colour scale starts from blue tones, associated with low potential values (i.e., the goal neighbourhood) and goes up to yellow tones, that indicate high potential values associated to the area close to an obstacle. Time is represented by the *step* counter located in the upper left corner, it starts

[23] The code used for A* was adapted from https://github.com/AtsushiSakai

[24] in this case, the APF is intended as a *path planning* algorithm, differently than in the algorithm design were the APF was used only as a representation of the environment.

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 position using Algorithm 4.6 and performs a single step of $0.12m$ along the trajectory. As can be observed in the images during the movement of the agent in the environment new obstacles are discovered and added to the environment model. The discovered obstacles are the black areas whereas those still to be discovered are the grey areas. Once the goal is reached (as can be seen in frame 99) the attractive layer is subtracted, a new goal is computed and the attractive layer associated with it is added to the APF model. An interesting behaviour of the agent can be seen among frame 8 and frame 22. In the first frame the agent detect only the left obstacle and in order to compute a conservative trajectory (avoiding possible risky situation) maintain a certain distance from it. When the right obstacle is revealed (frame 22), the trajectory is re-computed and the distance from the left obstacle is reduced maintaining the same distance from the two obstacles (as can be seen in frame 36). The trajectory has a constant length because the trajectory planning of Algorithm 4.6 uses a finite number $n$ of intermediate points to obtain the final path, thus if the maximum number of steps has been reached, the trajectory end in that final point and not in the goal.

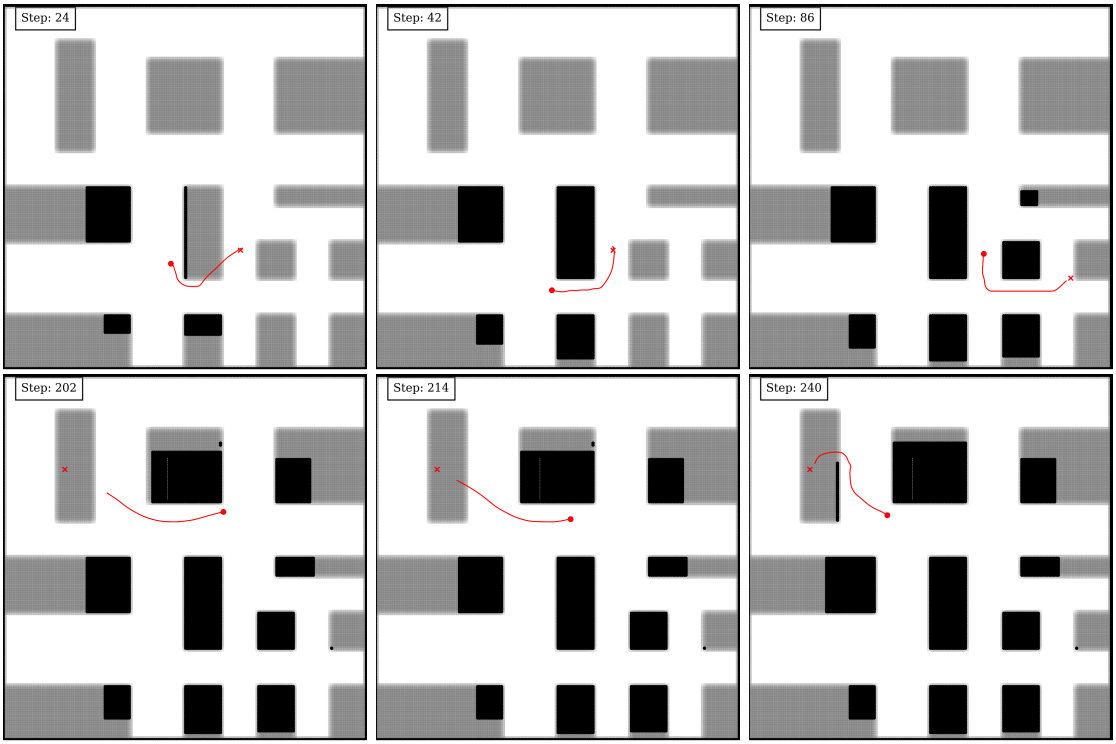


**Figure 6.2:** Simulation of Path Planning agent in validation map C.

Simulations of Figure 6.2 shows effective choices in trajectory generation, as the drone always maintains safe distance from the walls, does not change direction abruptly and always reaches the goal consistently (even in the presence of local minima). In the last figure the attractive layer is not plotted in order to highlight the trajectories. It is worth noting that the trajectories of this RL algorithm are always computable despite the possibility that the result may not converge to the desired solution.

### 6.1.3 Comparison of Path Planning algorithms

As mentioned at the beginning of this section, 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, the starting and the goal position in the map are chosen in order to create specific condition of testing. Then the trajectories of the 3 algorithms under consideration are analysed. These algorithms are the proposed RL agent, A* and APF. The results of some of the simulations performed are listed in Table 6.1. For each test the maps used, the achievement or non-achievement of the goal and the metrics $\bar{\Delta}_{IO}$ and $\epsilon$ are shown. The approximate length of the trajectory and the computational time spent are also listed, but they are discussed in more detail later.

**Table 6.1:** Results of the Path Planning simulations 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 | $x_0 : (17, 4)$ $x_g : (14.5, 13)$ | 9.80 | RL | yes | 0.53 | 10.01 | 0.49 |
| | | | | A* | yes | 0.36 | 9.84 | 0.42 |
| | | | | APF | no | - | - | - |
| 2 | B | $x_0 : (7, 3)$ $x_g : (16, 18)$ | 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 | $x_0 : (17, 4)$ $x_g : (17, 18)$ | 19.00 | RL | yes | 0.67 | 14.17 | 0.92 |
| | | | | A* | yes | 0.42 | 8.64 | 0.69 |
| | | | | APF | no | - | - | - |
| 4 | A | $x_0 : (10, 10)$ $x_g : (5, 5)$ | 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 | $x_0 : (2, 13)$ $x_g : (14, 9)$ | 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 | $x_0 : (2, 18.8)$ $x_g : (6, 17.2)$ | 4.75 | RL | yes | 0.70 | 8.70 | 0.258 |
| | | | | A* | yes | 0.40 | 20.54 | 0.05 |
| | | | | APF | yes | 0.58 | 16.58 | - |

As can be observed in Table 6.1 the RL path planning is always able to find a suitable trajectory in order to reach the assigned goal. conversely the APF algorithm in test 1 and test 3 has not reach the goal getting stuck in a local minimum point. Test 2 shows an interesting behaviour of the RL agent (orange line). It chooses a different path with respect to the others algorithm. The reason behind this behaviour is that the RL agent is optimised to minimise the turns. In this way the energetic cost associated with the path following is optimise. As can be seen in Table 6.1 the proposed algorithm always has the highest value among the other ones in terms of the minimum distance from the obstacles. In fact the trajectory computed is always safe without losing consistency. 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 angle difference of the trajectory of RL agent, as can be seen in Test 2, 5 and 6, is lower than the one of the other algorithms because it aims to find a more dynamically-efficient trajectory. In fact 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 stay close to the corner are more efficient in terms of minimum distance travelled and energy consumption, but less safe and robust. These results can be observed in Figure 6.3, where the three trajectories computed to solve all the tests are displayed.

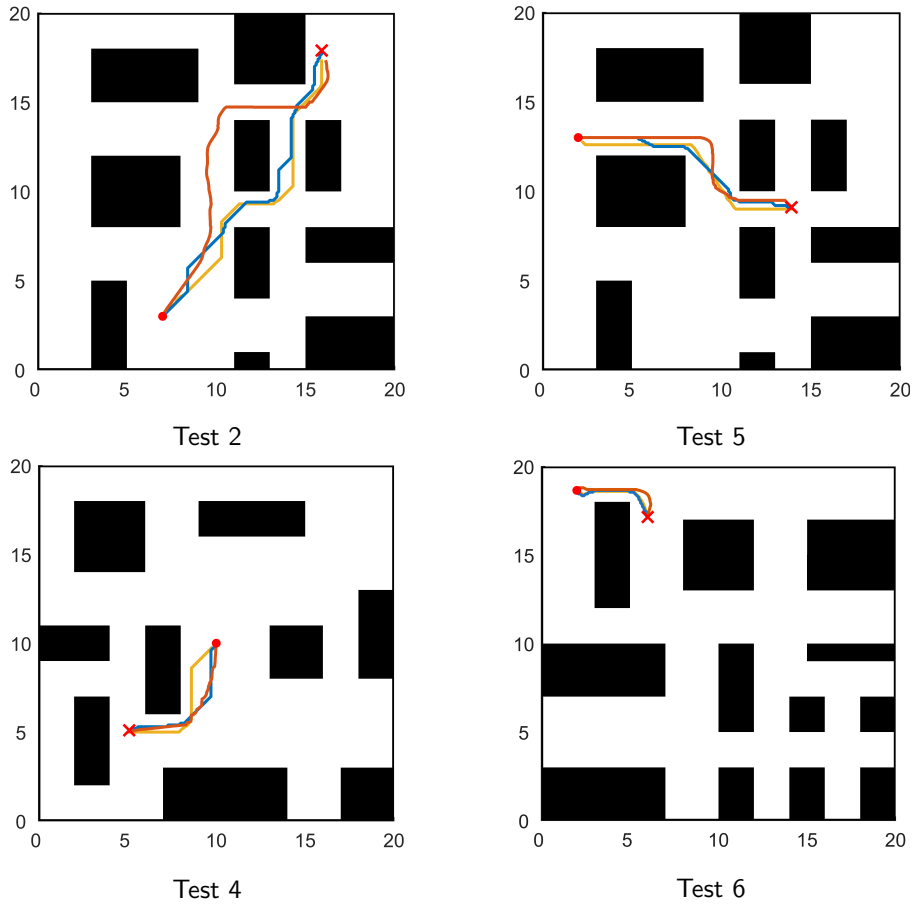


**Figure 6.3:** Comparison of the trajectories computed by the path planning algorithms. Orange lines represents RL trajectories, blue ones are A* trajectories, and yellow ones are APF trajectories.

### 6.1.4 Computational time performances

Another interesting results can also be observed about the computational time. It is worth noting the trend of computational time of RL and $A^*$ algorithms when they are plotted against the approximate trajectory length. In Figure 6.4 time $t$ is plotted against length $l$ for both the proposed RL agent and the A* algorithm. For what regarding the computational time of APF, it has not been measured because the algorithm implementation exploits the already computed APF environment model, in this way the time spent finding the direction to the lowest potential is very low ($< 0.1s$).

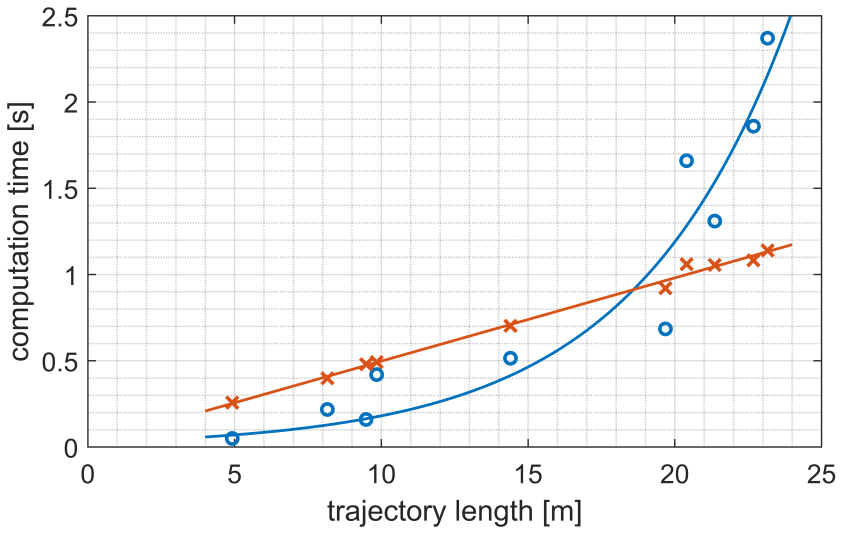


**Figure 6.4:** Comparison of the computational time [25]required by the RL agent (orange line) and A* (blue line) to obtain trajectories of various lengths.

As can be observed from the plot of Figure 6.4, the computational cost of the RL agent grows linearly with the length of the trajectory. It has a linear trend because it is independent from the number of obstacles or the complexity of the environment maintaining in this way a constant computational time per step. On the contrary, A* is far more sensitive to those parameters, this is due to the way the algorithm is constructed. In fact in order to reach the objective it operates a sort of "propagation of the trajectories" and the more the distance increases the more the propagation takes longer to be calculated. So the computational cost of A* grows exponentially with the trajectory length and is more sensitive to the presence of obstacles. The variability of the computational time in figure 6.4 is due to the complexity of the map, in fact the presence of obstacles introduces a significant increase in the computation time for A* algorithm.

[25] all computation time tests have been carried out on 64-bit Windows 7 system with Intel i7-4710HQ processor and 16GB memory through Python 3.8.3.

### 6.1.5 Local minima performances

In order to ensure the computation of effective trajectories in a consistent way, an agent has been developed to compensate for the problem of local minima, as described in chapter 4.5.4. A specific training procedure has been made to create an "assisting agent" able to manage efficiently the most common issue of the APF models.
This agent is called by the main RL agent when it detects a consequential repetition of position of the UAV. Results obtained with the "assisting agent" are shown in Figure 6.5. The trajectories

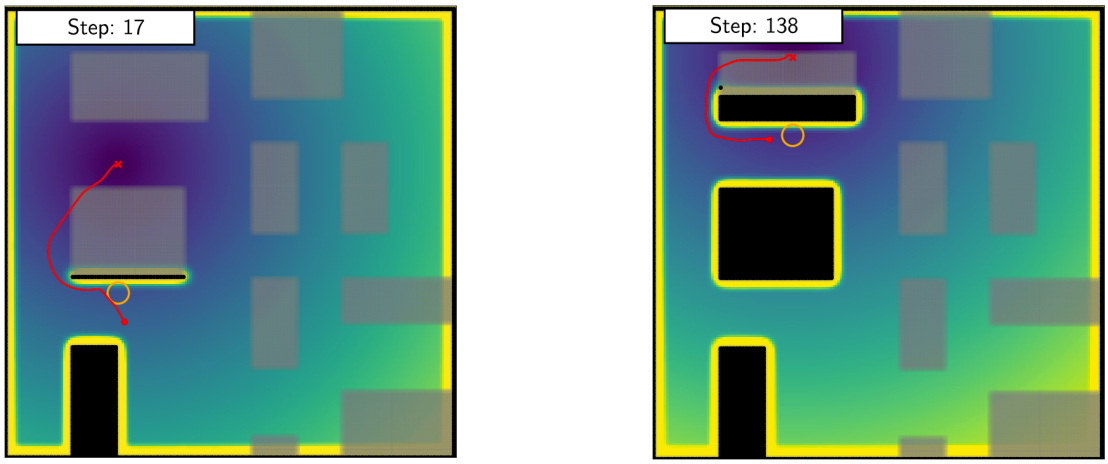


**Figure 6.5:** Examples of trajectories computed by the "assisting" agent.

displayed in Figure 6.5 show an effective avoidance of the local minima by the assisting agent. This task is very relevant because it allows the drone to create trajectories that a classic APF algorithm would not be able to calculate. The location of local minima is indicated by an orange circle. As can be observed, the agent is able to avoid the local minima by computing a suitable trajectory to reach the goal. In both the images the trajectory passes through an unknown obstacle: this is not a problem, since once the simulation proceeds the presence of the obstacle will be discovered and the trajectory will be updated to avoid it, re-computing also the position of the goal to put it outside the obstacle.

## 6.2 Coverage agent

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 6.2.1). Then, simulation and results obtained evaluating the exploration process are presented (sections 6.2.2 and 6.2.3).

### 6.2.1 Exploration evaluation metrics

Some evaluation metrics have been defined in order to measure the efficiency of the coverage algorithms during the exploration task. The environment is considered "explored" when at least $90\%$ of its surface has been covered.[26] The metrics that will be used to evaluate the different agent performances are:

1. $\bar{d}(t)$: average distance between UAVs during the exploration task. This metric highlights how UAVs are spreading in the environment during the exploration.

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

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.

$$d_{min}(t) = \min \|d_{ij}\| \qquad \text{where } i \neq j \tag{6.2}$$

   This metric is measured over all the UAVs (indicated by the $i$ and $j$ subscripts) and during the whole exploration task;

3. $\sigma(\bar{d})$: variance associated to the metric $\bar{d}(t)$ is also significant, since it measures how well the exploration task is divided amongst the UAVs. In fact, if the variance is low then the distances between them is more or less constant resulting in a optimal disposition of the UAVs in the space:

$$\sigma(\bar{d}) = \mathrm{Var}(M_2(t)) \tag{6.3}$$

4. $t_{N\%}$: 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 $N$;

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

$$A_{\%}(t) = \frac{A_{explored}}{A_{explorable}}\% \tag{6.4}$$

   This metric is evaluated both for the whole fleet and for each individual UAV.

[26] a piece of environment is considered *covered* when it has been observed by the UAV camera. This means that it is not necessary for the UAVs to physically cover every single point of the environment, but it is sufficient to observe them with vision sensor.

### 6.2.2 Exploration simulations

Multiple simulations have been performed to test the performances of the complete exploration algorithm. 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 is 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 finalisation of the exploration process has been assigned to the explicit algorithm, based on the k-means partitioning of the environment (introduced in section 3.3, whereas the implementation in the whole exploration algorithm is explained in section 4.4.2). All the simulations presented in this chapter have been obtained by a combined use of the RL agent and the k-means algorithm. Some additional simulations have been obtained by using only the k-means agent in order to use them 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 6.6 and 6.7. The second one, obtained using only the k-means algorithm, is displayed in figures 6.8 and 6.9. Each frame of the simulations displays the location of the four UAVs with different colour, as well as the four temporary goals. The explored regions are marked in blue, whereas known obstacles are coloured in black. The total explored area is reported as a percentage 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).

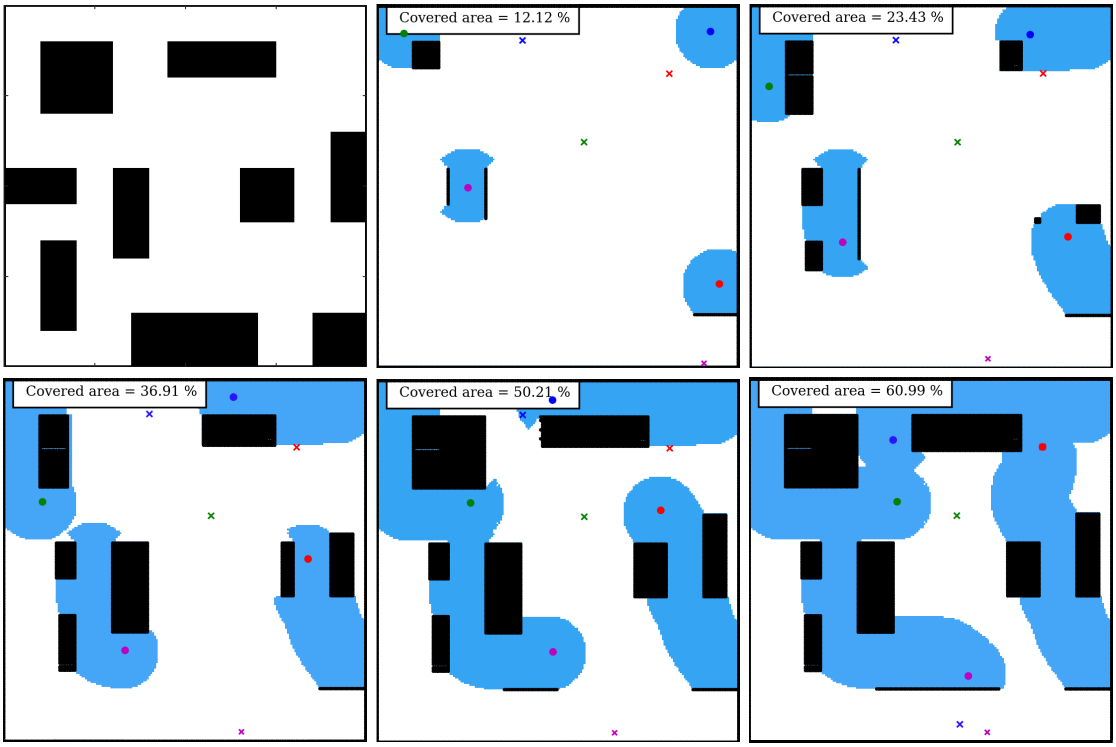


**Figure 6.6:** Simulation of the exploration task using the RL agent + k-means algorithm.

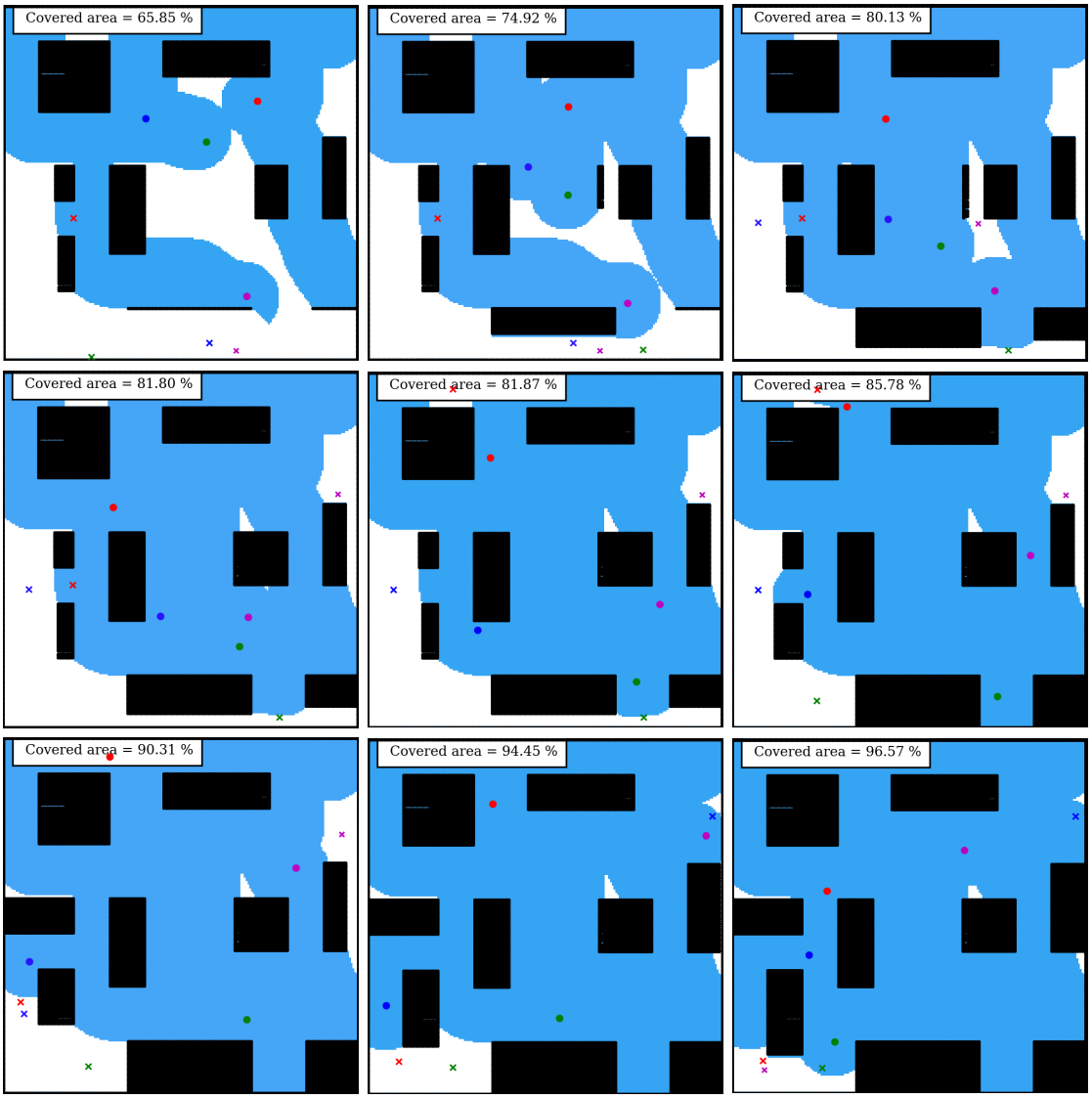


**Figure 6.7:** Simulation of the exploration task using the RL agent + k-means algorithm (pt 2).

For what regards the RL + k-means coverage algorithm displayed in the simulation of figures 6.6 and 6.7, 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. However the temporary goal placing is not always optimal. For example, in the first frame the red goal is too close to the blue UAV, although 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 in the first steps of an episode, 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 require many training episodes. As already discussed, this was not the case in the coverage agent training, and so the goal placement is not completely optimised. In frames $5$ and $6$ the goals 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 this 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 issues detected, the agent policy is able to quickly reach an exploration percentage equal to $70\%$ of the total environment surface area.

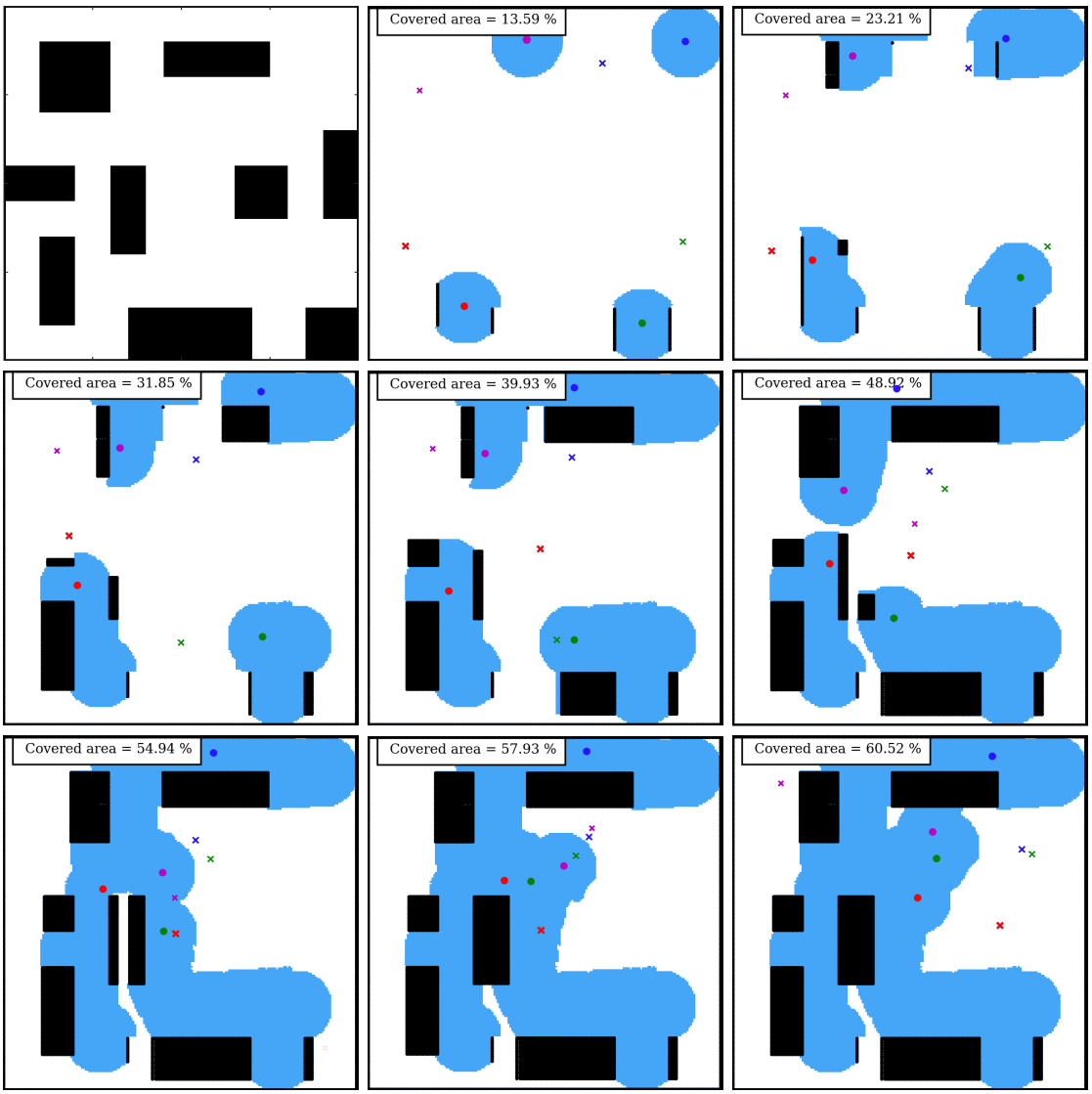


**Figure 6.8:** Simulation of the exploration task using k-means coverage algorithm.

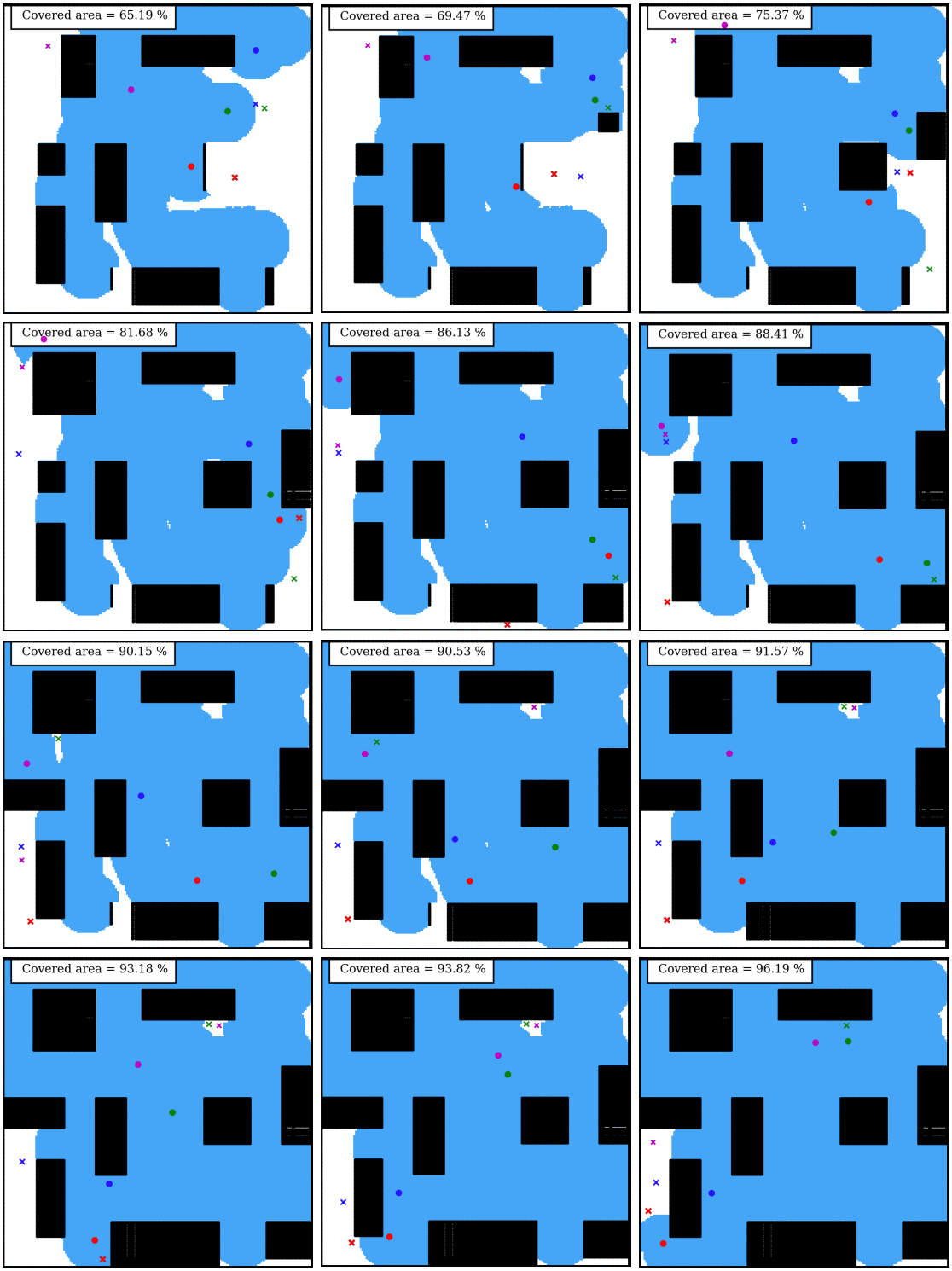


**Figure 6.9:** Simulation of the exploration task using k-means coverage algorithm (pt 2).

Some observations can be made also for what regards the k-means coverage agent, used in the second part of the first simulation (figures 6.6 and 6.7) and in the whole second simulation (figures 6.8 and 6.9). 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 4.4.2 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. Due to the good results that this algorithm has shown in the completion of the exploration process 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 learn efficient end-simulation behaviours. The shortness of the training process performed on the coverage agent is, once again, a significant limitation.

### 6.2.3 Exploration results

Several 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 the three validation maps. The simulation results in map A have been shown in figures 6.6 and 6.7 for the combined agent and in figures 6.8 and 6.9 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 6.2.1. 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 $t_{N\%}$ are expressed as number of simulation steps.

| Test | Algorithm | Map | $A\%$ | $\bar{d}$ [m] | $\sigma(\bar{d})$ [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 and map used are reported in table 6.2. In addition, the value of all the metrics is reported in the table. The variable $\sigma(\hat{d}\,)$ 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). From the table it can be observed that in all the six tests the coverage algorithm employed is able to reach a coverage percentage value between $93\%$ and $95\%$. With only small differences between the different tests, the average distance between the UAVs of the fleet is around $10m$, whereas the standard deviation associated to it is $\sim 3.3m$. As can be observed in the table, the Coverage agent efficiency decreases as the map complexity grows (see Figure 5.2 for details about each map complexity). The average distance between the UAVs is similar in the three cases, whereas in the third one the minimum distance between the agents is smaller than in the other two. The reason for this is that the UAV swarm is still not efficient at moving in an environment rich in obstacles and so the drones end up getting closer, especially in the final steps. The resulting graphic can be observed in figure 6.10.

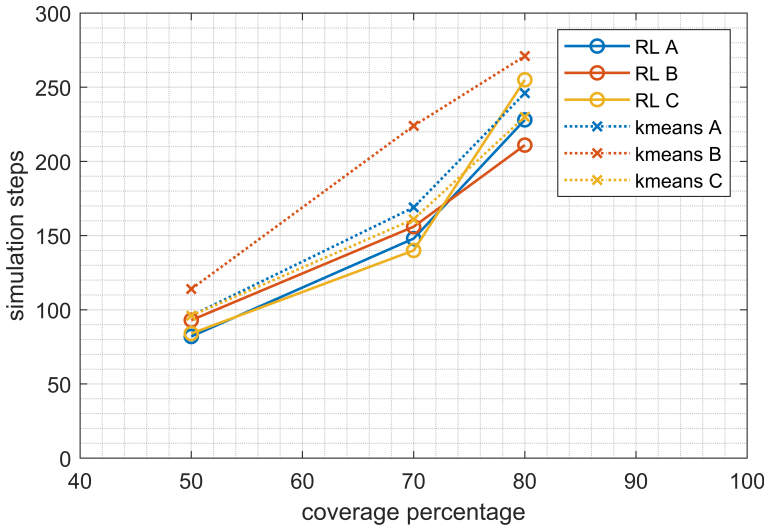


**Figure 6.10:** Comparison of the time employed to reach some coverage percentage thresholds in the six exploration tests.

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 in 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 6.8 and 6.9), whereas the second corresponds to test 4 (obtained using the combined coverage algorithm and displayed in figures 6.6 and 6.7). The two coverage percentage plots are shown in figure 6.11.

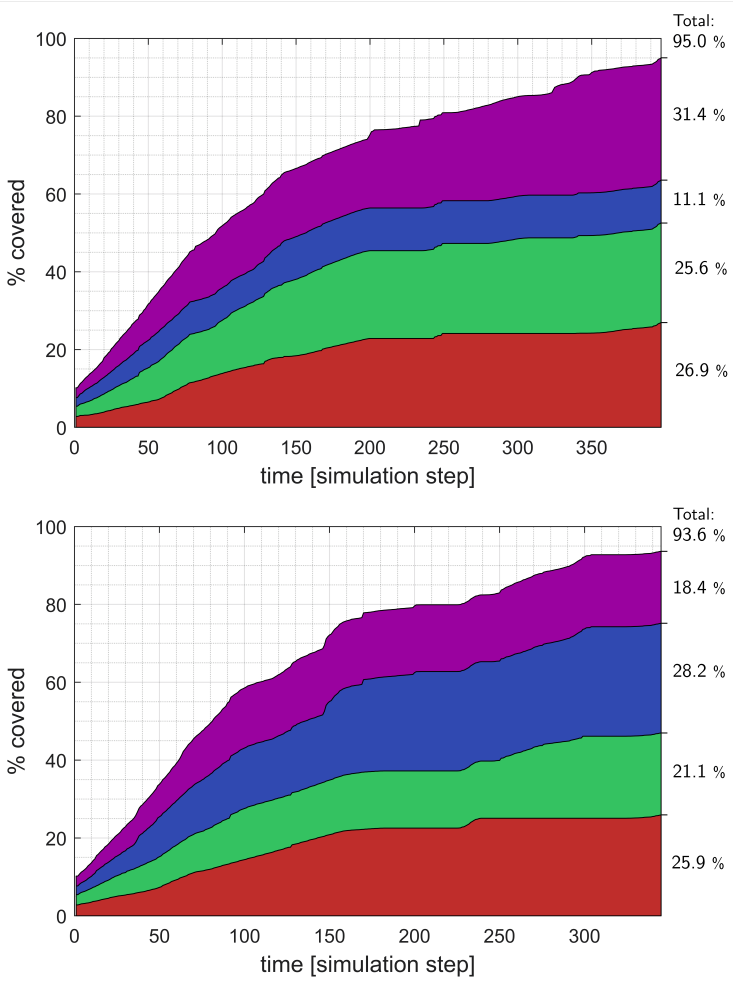


**Figure 6.11:** Coverage percentage of the single UAV contributions in the exploration process for tests 1 and 4. The plot on the top shows the results of test 1, whereas the one on the bottom shows results of test 4. The colours correspond to those of the UAVs represented in the simulations of figures 6.8, 6.9, 6.6, and 6.7.

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. Instead in the case where only the k-means was used, 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. The only issue is that sometimes a couple of 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

## 7.1 3D extension

There are different ways to face the 3D extension, the first one is a simple extension of the 2D approach previously discussed, whereas the second one is a more complicated implementation, where a new training is set in order to taking into account the augmented matrix as input state.

### 7.1.1 Slicing method

The first possibility is the "slicing" of the environment in a number of sections along the z-axis where each of them is treated as a 2D portion of the environment. This strategy allows to use the exact same path planning and coverage algorithms presented for the 2D environment augmenting all the matrices in 3D ones. In this way, the potential matrix representing the environment would become a 3D matrix, i.e. each cell, representing a volume of space instead of an area determined by three coordinates, contains a potential value, which works in the same way as in 2D case. This strategy is very easy to implement starting from the 2D algorithm developed, at the cost of some efficiency in the exploration process since the coverage and path planning are optimised only with respect to 2D level and not in the whole environment. In fact the movement between different layers is not optimised because it has not been taken into account in the training process. However in case of environments where obstacle shapes are constant along the z-axis (like in an urban environment) this simplified 3D algorithm could produce satisfactory results.

This approach brings with it some issues that need to be solved. First of all, the memory usage grows quickly. In particular, the dependence of the number of elements 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 in order to manages high dimensional matrices. An increase in the environment model dimension leads to the same exponential growth in the 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. Furthermore, 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, e.g. the quaternions avoiding in this way the numerical problems that spherical coordinates bring with them (in particular the singular configurations of the angles).

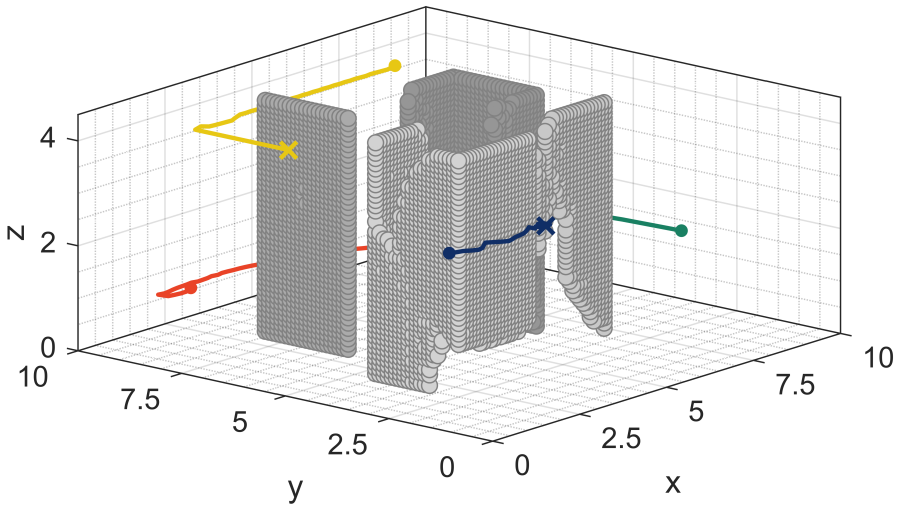


**Figure 7.1:** Simulation performed with the "slicing" approach in the python simulation environment.

### 7.1.2 Neural Network updating method

The second possible implementation can be done modifying the structure of the agents NN. The two sections that need to be modified are the input section and the output one. The latter one requires a rather simple modification, in fact the output nodes will change to having two angles as output for the path planning agent (this number depends on the type of 3D coordinates is chosen, two nodes in case of spherical coordinates, whereas four if quaternions have been chosen) and three for the coverage 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 by a succession of convolutional layer mixed to pooling and batch normalisation layers. All these layers must be changed in order to works with 3D matrices. The filters of the convolutional layers will be changed into 3-dimensional filters, i.e. they will multiply a 3D volume of the input state. A temporary drawback is that the number of weights composing the NN grows significantly and this leads to an increase in the required training time for the convergence towards a good behaviour of the agent.

# 8. Conclusions

## 8.1 Issues

The algorithm proposed to present some issues that negatively influence its efficiency. In this section, they will be illustrated and discussed.

- one of the main problem of the method proposed lies in the use of Reinforcement Learning. In fact, despite being a very powerful and versatile tool, RL (and more in general Artificial Neural Networks) presents some critical points. One of them was already discussed in Section 4.5.5, and is the fact that during training it is impossible to provide the model with all the possible inputs it could encounter, and therefore there is always the possibility that some new input will lead to undesired or dangerous behaviour. A consistent solution to this problem is to pair the RL algorithm with an explicitly-programmed system that captures any unwanted behaviour and safely corrects it. However, designing such a system is not always easy and negates some of the benefits of RL methods;

- a second issue, still linked to the intrinsic issues of RL, is that when a RL agent is implemented, it has to be treated as a black box. In fact, because of the way NNs are constructed, it is almost 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. Unfortunately, the same is also true when it does not work correctly, which makes debugging and fixing faulty models very difficult;

- A third problem, which did not arise during this thesis but could be critical in more complex applications, is the difficulty of defining an effective reward function. The use of the reward function is a powerful tool as it does not require to explicitly compute the optimal output that the NN should return; however, finding the best reward function could be difficult for complex problems where it is unclear which metric is related to the optimal solution (e.g., when computing an optimal trajectory the best metric to reward could be the smoothness of the curve, travel time, or both, etc.). Methods different from the reward-guided training are being proposed, such as [52], but they are still under research;

- a fourth and final 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 example, each path planning agent trained in section 5.2.1 has a very specific input size. If, for any reason, the size of such input has to change (e.g. to increase the "awareness radius" of the agent), a new agent has to be trained, as there is no way 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 dimensions. Some solutions have been implemented in order to manage map dimension different from the input state. However, in some aspects, the generality of the algorithm proposed had to be reduced due to this limitation of NNs.

## 8.2 Future work

- The main open point regards the further training of the Coverage agent. Further investigation of the reward function and NN structure design is programmed, as well as the tuning of the training hyperparameters. It is also possible that the overall architecture of the training process will be modified (mainly to implement some MARL-specific techniques) if it becomes evident that the current one is not suitable to obtain the desired behaviour);
- one goal for future developments is to merge the two path planning agents into a single one. This could be obtained by performing two successive training processes, the first one to learn the generic Path Planning operations (the one carried out by the "main" agent) and the second one to specialise in the management of local minima The addition of a second output node to the Path Planning agent NN, through which the UAV speed in each point of the trajectory can be computed, is also being considered;
- 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, where it can be considering actuation force and any sort of energy optimisation process;
- another future refinement could be the use of more advanced training algorithms, such as the Twin Delayes DDPG (TD3), the Proximal Policy Optimization (PPO) or the Soft Actor Critic (SAC). This could lead to better results for the same training time;
- a topic that has not been covered but that could be very interesting to explore further is the implementation of communications. In fact, it would be interesting to add the management of communications between drones and simulate interactions between them in a more realistic way;
- Once all the missing pieces are added and the 3D extension is completed with the second method described in chapter 7, it will actually be possible to test the performance of the algorithm in a real swarm of drones and field-test its true performance;

## 8.3 Conclusions

An efficient Path Planning agent based on RL was designed and implemented. The results show that it is an efficient algorithm, capable of calculating safe and dynamically efficient trajectories even in obstacle-rich environments. It also achieved remarkable results compared to classical and widely used algorithms such as $A^*$ and APF. The Coverage agent, designed to handle high-level exploration operations, shows some positive results up to $70\%$ in exploration operations. In fact, it is able to position targets efficiently while keeping the drones well spaced out. The agent combined with the k-means algorithm succeeds in overcoming the problems encountered in the final part of the exploration and achieves full coverage of the map. The whole algorithm is also absolutely usable in a real drone, although with some refinements, as the computation times are compatible with the hardware specifications of the micro controllers used. It should be noted that the computation time of the path planning algorithm is linear, whereas the computational time needed to compute $A^*$ scales with the square of the distance. This means that the proposed algorithm can be used efficiently to calculate trajectories in large environments without encountering computation times that could compromise system performances. In conclusion, the RL agent needs to be refined to become fully effective in complete exploration of unknown environments. But the results achieved so far open up a world of possibilities for a new type of innovative and promising algorithms for all kinds of applications.

# 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, expecially 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] X. Yu and Y. Zhang, "Sense and avoid technologies with applications to unmanned aircraft systems: Review and prospects," *Progress in Aerospace Sciences*, vol. 74, pp. 152–166, 2015.

[2] A. Gasparetto, P. Boscariol, A. Lanzutti, and R. Vidoni, "Path planning and trajectory planning algorithms: A general overview," *Mechanisms and Machine Science*, vol. 29, pp. 3–27, 03 2015.

[3] H. Choi, M. Geeves, B. Alsalam, and F. Gonzalez, "Open source computer-vision based guidance system for uavs on-board decision making," in *2016 IEEE Aerospace Conference*, pp. 1–5, 2016.

[4] 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.

[5] A. Stentz, "Optimal and efficient path planning for unknown and dynamic environments," *INTERNATIONAL JOURNAL OF ROBOTICS AND AUTOMATION*, vol. 10, pp. 89–100, 1993.

[6] 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.

[7] S. Aggarwal and N. Kumar, "Path planning techniques for unmanned aerial vehicles: A review, solutions, and challenges," *Computer Communications*, vol. 149, pp. 270–299, 2020.

[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," in *Bioinspired Optimization Methods and Their Applications* (B. Filipič, E. Minisci, and M. Vasile, eds.), (Cham), pp. 268–280, Springer International Publishing, 2020.

[9] V. Roberge, M. Tarbouchi, and G. Labonte, "Comparison of parallel genetic algorithm and particle swarm optimization for real-time uav path planning," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 132–141, 2013.

[10] M. Theile, H. Bayerlein, R. Nai, D. Gesbert, and M. Caccamo, "Uav path planning using global and local map information with deep reinforcement learning," 2021.

[11] C. Wang, J. Wang, X. Zhang, and X. Zhang, "Autonomous navigation of uav in large-scale unknown complex environment with deep reinforcement learning," in *2017 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pp. 858–862, 2017.

[12] D. B. Megherbi and A. T. A. J. Boulenouar, "A Time-varying environment based machine learning technique for autonomous agent shortest path planning," -, vol. 4364, pp. 419–428, 2001.

[13] T. M. Cabreira, L. B. Brisolara, and R. Ferreira Paulo, "Survey on coverage path planning with unmanned aerial vehicles," *Drones*, vol. 3, no. 1, pp. 1–38, 2019.

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

[15] A. Barve and M. Nene, "Survey of Flocking Algorithms in Multi-agent Systems," *International Journal of Computer . . .*, vol. 10, no. 6, pp. 110–117, 2013.

[16] J. N. Yasin, S. A. Mohamed, M. H. Haghbayan, J. Heikkonen, H. Tenhunen, and J. Plosila, "Navigation of Autonomous Swarm of Drones Using Translational Coordinates," *Lecture Notes in Computer Science*, vol. 12092 LNAI, pp. 353–362, 2020.

[17] A. L. Alfeo, M. G. Cimino, N. De Francesco, A. Lazzeri, M. Lega, and G. Vaglini, "Swarm coordination of mini-UAVs for target search using imperfect sensors," *Intelligent Decision Technologies*, vol. 12, no. 2, pp. 149–162, 2018.

[18] J. N. Yasin, S. A. Sayed 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.

[19] 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, may 2019.

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

[21] H. V. Parunak, M. Purcell, and R. O'Connell, "Digital Pheromones for Autonomous Coordination of Swarming UAV's," in *1st UAV Conference*, no. May in -, (Reston, Virigina), pp. 1–9, American Institute of Aeronautics and Astronautics, may 2002.

[22] 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.

[23] L. Buşoniu, R. Babuška, and B. De Schutter, "Multi-agent Reinforcement Learning: An Overview," in *Technology*, vol. 38, pp. 183–221, Springer, Berlin, Heidelberg, 2010.

[24] S. Gronauer and K. Diepold, "Multi-agent deep reinforcement learning: a survey," *Artificial Intelligence Review*, vol. 2, apr 2021.

[25] Y. Huang, S. Wu, Z. Mu, X. Long, S. Chu, and G. Zhao, "A Multi-agent Reinforcement Learning Method for Swarm Robots in Space Collaborative Exploration," in *2020 6th International Conference on Control, Automation and Robotics (ICCAR)*, pp. 139–144, IEEE, apr 2020.

[26] H. X. Pham, H. M. La, D. Feil-Seifer, and A. Nefian, "Cooperative and Distributed Reinforcement Learning of Drones for Field Coverage," *CoRR*, vol. abs/1803.0, mar 2018.

[27] U. Kartoun, H. Stern, and Y. Edan, "A human-robot collaborative reinforcement learning algorithm," *Journal of Intelligent and Robotic Systems*, vol. 60, pp. 217–239, nov 2010.

[28] 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.

[29] 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.

[30] 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.

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

[32] F. Remondino, L. Barazzetti, F. Nex, M. Scaioni, and D. Sarazzi, "UAV PHOTOGRAMMETRY FOR MAPPING AND 3D MODELING – CURRENT STATUS AND FUTURE PERSPECTIVES," *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. XXXVIII-1/, pp. 25–31, sep 2012.

[33] P. Boccardo, F. Chiabrando, F. Dutto, F. Tonolo, and A. Lingua, "UAV Deployment Exercise for Mapping Purposes: Evaluation of Emergency Response Applications," *Sensors*, vol. 15, pp. 15717–15737, jul 2015.

[34] C. Ju and H. Son, "Multiple UAV Systems for Agricultural Applications: Control, Implementation, and Evaluation," *Electronics*, vol. 7, p. 162, aug 2018.

[35] M. Mammarella, G. Ristorto, E. Capello, N. Bloise, G. Guglieri, and F. Dabbene, "Waypoint tracking via tube-based robust model predictive control for crop monitoring with fixed-wing uavs," in *2019 IEEE International Workshop on Metrology for Agriculture and Forestry (MetroAgriFor)*, pp. 19–24, 2019.

[36] 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.

[37] 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.

[38] 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.

[39] N. S. Website, "Saturn's largest moon, titan." https://solarsystem.nasa.gov/moons/saturn-moons/titan/in-depth/. Accessed: 2021-4-27.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

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

[45] 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.

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

[47] OpenAI, "Deep deterministic policy gradient algorithm." https://spinningup.openai.com/en/latest/algorithms/ddpg.html, February 2021.

[48] 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."

[49] OpenAI, "Key concepts of reinforcement learning." https://spinningup.openai.com/en/latest/spinningup/rl_intro.html, February 2021.

[50] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing Function Approximation Error in Actor-Critic Methods," *CoRR*, feb 2018.

[51] 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," *CoRR*, apr 2018.

[52] B. Eysenbach, "Recursive classification: Replacing rewards with examples in rl." https://ai.googleblog.com/2021/03/recursive-classification-replacing.html. Accessed: 2021-03-29.

# Appendix

# A. Neural Networks, Backpropagation & Stochastic Gradient Descent

## A.1 Neural Networks

As introduced in section 3.2.3, Neural Networks are functions composed by 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 of Figure 3.3 already seen in section 3.2.3).

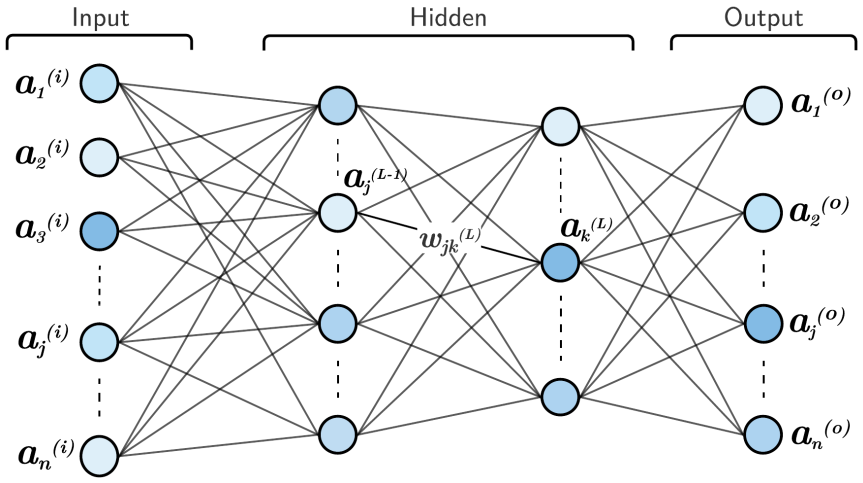


**Figure A.1:** Simple Neural Network scheme. Input, output and hidden layers are put in evidence. In general, any number of hidden layers can be used (copy of Figure 3.3).

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 *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.

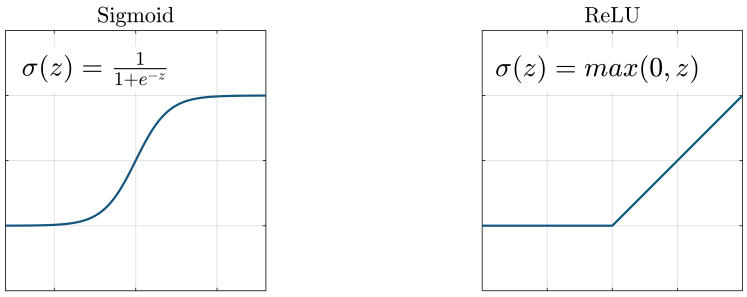


**Figure A.2:** Common choices for the activation functions. On the left the sigmoid activation function, whereas on the right the ReLU.

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 to 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 a 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 $\mathcal{M}$.[27]

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 trough 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*.

[27] 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.

## 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 backward from the output layer to the parameter in question (from which 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:

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

As already mentioned, this last equation differs slightly in 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):

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