POLITECNICO DI TORINO

Master's Degree in Aerospace Engineering

Master's Degree Thesis

Neural Network Based Algorithm for Multi-UAV Coverage Path Planning

Supervisors

Giorgio GUGLIERI

Simone GODIO

Candidate

Giovanni SANNA

April 2021

Summary

Unmanned Aerial Vehicles, better known as drones, have become an eye in the sky to aid men with a perspective from above. They provide real-time, high-resolution imagery at low cost. Originally thought for military applications, UAVs have found their way into mainstream usage thanks to the enhanced levels of safety and efficiency they bring. Robotic UAVs operate without an onboard pilot; the trend is of continuous evolution with an ever increasing level of automation. On the other side, Artificial Neural Networks lead the cutting-edge machine learning techniques, whose purpose is to render machines more and more intelligent by means of bio-inspired models.

This work combines both the potential of artificial intelligence with drone-based surveillance capabilities - a fleet of AI-driven UAVs executes the Coverage Path Planning of a complex-shaped urban areas. The outcome is a strategic selection and planning of the trajectories over a map - while accounting for congestion, collisions, and image overlapping issues. The decision-making process is delivered by a balanced "explicit vs implicit" programming. The algorithm relies on a mixed-use of decentralized Artificial Neural Networks which confers elementary cognitive skills to each UAV, and a modified version of the famous A* pathfinder. Moreover, the training session of the Neural Network completely bypasses common drawbacks such as the need of large labeled databases or high computational resources.

Particular attention is given to the scenario developed in the occasion of the Leonardo Drone Contest, proposed by the homonymous company. Further case studies focus on real urban areas, for which the grid resolution of the traditional Coverage Path Planning approaches can't model the problem with sufficient accuracy.

Keywords: Neural Network, Artificial Intelligence, UAV, Supervised Imitation Learning, Urban Coverage Path Planning.

Table of Contents

Chapter 1

Introduction

Multi-robot Coverage Path Planning (mCPP) is a well-known problem that interests researchers all over the world. The growing popularity of Unmanned Aerial Systems (UAS) sheds light on new fields of application: among them, the aerial coverage path planning in urban environments has very unique features for which the use of traditional coverage algorithms could prove problematic. Coverage path planning is defined as:

"[1]Coverage Path Planning is the task of determining a path that passes overall points of an area or volume of interest while avoiding obstacles."

This task is integral to many robotic applications, such as vacuum cleaning robots, painter robots, demining robots, lawnmowers, automated harvesters, window cleaners, and inspection of complex structures, just to name a few[1]. For UAV applications, a slight deviation from the traditional CPP definition leads to a complete change of paradigm. In fact, because of their very nature, UAVs do not need to physically pass over through all the points in the area: to explore the scene, it is sufficient for the point to be within the frame of the camera. The CPP problem, especially harsh in its multi-agent version, has been tackled with a multitude of different proposals, collected in several documents in line with the established scientific research field. In 1998, H. Choset reported a first analysis of CPP state of the art in *Coverage for Robotics - A survey of recent results*[1], a reference document for future development that explored heuristic methods, squared grid-based algorithm, as well as approximate, partial-approximate, and exact cellular decomposition. It was the early '00s and the robotic target of the document was shaded with a kind of vagueness, enough that the author himself declares that the document was focused on "mobile robots constrained to operate in the plane". Later on, in 2013, the survey [2] *"A Survey on Coverage Path Planning for Robotics"* carried out an analysis on the state of the art CPP developed in the previous ten years. Compared to Choset's document[1], it is possible to assist to a drastic increase in the number of explored

solutions, from hexagonal grids to Morse function based decomposition, from to spanning tree to wave-front algorithm, with an introduction to a non-learning version of a neural network. Moreover, the coverage path planning began to involve landmark-based and 3D coverage, to witness the explosion of UAV applications in the mainstream and the gained interest of the coverage path planning from the world of scientific research. More recent surveys include [3] *Survey on path and view planning for UAVs* (2019) and [4] *Survey on Coverage Path Planning with Unmanned Aerial Vehicles* (2018), where a wide set of solutions is deeply analyzed, highlighting the pros and cons of each method. In order to plan a path, most approaches use space discretization over an occupancy grid with squared cells, which allows the creation of known optimal trajectories e.g. the *"back and forth"* or *"spiral"*, while avoiding image overlapping between consequential scans. Those traditional approaches align the camera Field Of View (FOV) with the cells of the occupancy grid, where the cell is the size of the robot footprint: this makes it possible to treat UAV applications just like any other terrestrial robot. This approach is useful in surroundings with low obstacle density and in applications such e.g. surveillance, smart farming, photogrammetry, disaster management, and wildfire tracking. The offline multi-robot coverage path planning literature contains plenty of elegant solutions: e.g. in [5] *"A Bioinspired Neural Network-Based Approach for Cooperative Coverage Planning of UAVs"* (2021), a non-trained version of a neural network is explored; the multi-agent decision making is delegated to a dynamic cost-map where each agent moves following the negative gradient of the cost function over the discrete network. In [6] *"Near-optimal coverage trajectories for image mosaicing using a mini quad-rotor over irregular-shaped fields"* (2013), Valente et al present a wavefront approach, a flooding algorithm that marks the neighborhood adjacency of cells. This method decomposes the target area and converts it into a regular graph numerically labeled by the wavefront algorithm. By using a Deep-Limited Search (DLS) among adjacent cells, the flooding mechanism is capable to determine a start to end trajectory without visiting previously explored nodes for a single UAV case. In [7] *Fractal Trajectories for Online Non-Uniform Aerial Coverage* (2015), uniform coverage path planning is explored with the Hilbert curves and their properties, resulting in fractal trajectories which evenly covers the map underneath. Moreover, this approach contemplates also planning for a discrete fixed height by changing the order of the Hilbert curve, although it has been implemented only in the single-agent case. In [8] *A*-Based Solution to the Coverage Path Planning Problem* (2017), the problem is faced with the A* pathfinder, adapted in order to find a trajectory that minimizes the number of turns while accounting for the complete coverage of non-convex environments. Other architectures include [9] harmony search, [10] genetic algorithm, and [11] chaotic ant colony optimization. Those 2D methods are not fruitful in urban applications with a high density of complex-shaped obstacles, since the grid resolution is not dense enough to allow for optimal planning. Too

low resolution would miss urban complexity, too high resolution would require low flight height from terrain and non-optimal trajectories in urban environments with large, obstacle-free, explorable areas. This work presents a Multi-UAV Coverage Path Planning algorithm that uses a compatible space resolution with real urban maps while keeping the exploration bi-dimensional. The tight grid allows planning smoother and precise trajectories that can follow urban shapes. The core of the algorithm uses a pre-trained Artificial Neural Network (ANN) to decide the next action to take based on the UAV's current local state. The use of artificial neural networks has been tested both in the trained and non-trained versions, but only a few used Artificial Intelligence (AI) to complete the multi-robot coverage path planning task. For those applications, the training stage is usually faced with Multi-Agent Reinforcement Learning (MARL). Several MARL architectures can be implemented, each one with its strengths and weaknesses. The basic idea behind decentralized architectures is: each agent (UAV) senses its state from the environment (e.g. presence of obstacles). Based on its state, it decides what action to take (e.g. move right) and interacts with the environment (the map): the agent receives positive rewards if the action had positive consequences or vice-versa. After a long training, the agent learns a policy to map states to actions to maximize the expected discounted return. This process is not trivial at all, and requires an expensive amount of computational time as the exploration of the environment leads to a huge set of possible states. In 2019, Google DeepMind showed MARL incredible capabilities by creating AlphaStar, an AI-driven player of the game StarCraft II, which defeated several professional players. The action space was so huge that each agent would need an equivalent experience of up to 200 years of real-time plays to reach useful performances. The training session of the neural networks was designed in several precise steps. In one step of the training, the AI was trained on Human priors, collecting data from an expert player and using them as a training basis. Brand new neural networks that still need training are likely to select actions without any meaning or strategy.; the first strategic choices manifest themselves after thousands, if not millions of training iterations. By using expert priors, AlphaStar skipped this "dumb agent" phase and the randomic exploration, restricting drastically the learning time and starting with an advanced infant stage. This technique is called Imitation Learning as the solution is guided and the learning neural networks have a solid foundation to start from. Among the learning paradigms, Imitation Learning is usually implemented with Reinforcement Learning (RL), since the tuple $\{State, Action, Probability, Reward\}$ of the Markov Decision Process (MPD) creates a strong framework for decision-making applications. This work of thesis stands out the Imitation Learning capabilities used in conjunction with Supervised Learning for the multi-UAV coverage path planning problem. Classical taxonomy divides learning paradigms according to the practical uses and potential of the model. Problems related to decision-making

are usually delegated to Reinforcement Learning (RL) due to its nature, while Supervised Learning is applied in problems of classification, image recognition, or data regression. However, unlike RL, training times of supervised networks are reduced by several orders of magnitude compared with RL: this feature has moved the target of this work from reinforcement machine learning to supervised. The first months of work focused on simulations in which the goal was to maximize the expected reward of the problem using a Deep-Q-Network (DQN), in which the reward was proportional to the number of cells explored by the UAV. Regardless of the complexity of the map to be explored, computational times were such as to require the support of High-Performance Computing (HPC) clusters of the Politecnico di Torino. Reducing the planning problem to an image classification problem, it was possible to deal with a decision-making problem with supervised techniques. The migration from the Deep-Q-Network algorithm to Supervised Learning made training times collapse from days to minutes, with equivalent performance to more than 10 days of RL training. On the other hand, supervised learning relies on labeled datasets for the training session, where each state and action pair (in general input feature and class) must be available. The need for a consistent base of data to learn from is the main drawback of Supervised Learning, as it requires thousands of samples to have a performing network able to generalized what it has learned. To overcome this issue the dataset used in this work has been created by collecting experiences from an expert system, that can be both a human or an expert algorithm. Each UAV uses an artificial neural network to sense the nearby environment information and then decide what action or movement to take. The network learns to imitate those data, and it is able not only to reproduce the expert behaviour with outstanding accuracy but can to find common patterns among the training data and predict the next action to take also from never seen states. This ability is called generalized learning and lets the network transfer decision-making policies from previous situations to new ones. Moreover, the imitation framework used to collect the experiences from the expert system records entries for a single-agent case in simplified environments. This inductive approach shows further generalized learning capabilities since the network is trained for a single-agent case while the final applications involve multi-agent scenarios. The whole project has been coded in Python v3.7, since it is the leading Object Oriented Programming (OOP) language in the machine learning field. The entire design of the artificial neural network passed through the Keras Python libraries, included in TensorFlow. Google Maps was used to aid the creation of the real urban maps, modeled as occupancy grid matrices in MS Excel.

In Chapter 2 - Neural Machine Learning, the artificial neural network model is introduced, from the first bio-inspired perceptron of Rosenblatt (1958) to complex state-of-art networks. This overview introduces the basics features of an ANN

as structural architecture, hyper-parameters, design issues, and machine learning potential. Particular attention is paid to the two flows of information that govern the behavior of the network, namely the forward propagation, which is responsible for the network prediction, and backward propagation, which implements the gradient descent algorithm to tune networks' parameters leading to an "Artificial Intelligent" network.

In Chapter 3 - Inside the Neural Network, the artificial neural network used for the simulation is presented in detail, from the network design to the training process. Particular attention is paid to the input vector, whose choice was crucial for the desired outcomes. Moreover, the whole data pipeline is introduced, from the imitation framework to the data partition over the three main data-subsets: training, validation, and testing. Finally, an overview of data augmentation is given, describing the process that led to the achievement of the 54,000 entries present in the current database.

In Chapter 4 - Algorithm, it is presented a modified version of the A* pathfinder, and its synergistic use with the neural network. The whole logical flowchart is presented, highlighting both the potential and the criticalities of the algorithm, presenting an innovative approach that aims not only at the cooperation of UAVs but also at their collaboration. Moreover, the combined use of the Lloyd Algorithm and Voronoi Diagrams is presented to divide the target explorable space and enhance strategic coverage.

In Chapter 5 - Results & Simulation, a base of evaluation metrics is used to evaluate the quality of the results over real urban maps, accounting exploration rates, congestion, strategy, and energetic constraints. Particular attention is given to the case study presented in the Leonardo Drone Contest, whose performances are simulated in a 3D environment using Robotic Operating System (ROS) and Gazebo.

In the final Chapter 6 - Conclusion, the main points of the whole work are resumed, analyzing critically both strengths and weaknesses of the proposed algorithm, laying the basis of further development and future works.

Chapter 2

Neural Machine Learning

The strategic coverage requires a coordinated sequence of actions from each UAV: the limited storage capacity of energetic resources requires efficient trajectories that minimize the amount of in-flight time while avoiding collisions. At each time-step each UAV must select the most convenient action: in this context, the decision-making process is entrusted to a balanced use between Artificial Neural Networks and Path-finding Algorithms. The network confers elementary cognitive skills to each agent, which decides the next action based on the sensed surrounding state. This chapter describes the Artificial Neural Network model: from the basic architecture concept to the information propagation, the mathematical model, practical issues, and finally, the training process that leads to Artificial Intelligence.

2.1 General features

2.1.1 Relation between AI and Neural Network

Our society is driven by Artificial Intelligence. The growing development and fame of this field, in conjunction with Machine Learning, leads often to improper use of the term. A definition of AI is given by Bill Brock, VP of engineering at Very: ".AI, simply stated, is the concept of machines being able to perform tasks that seemingly require human intelligence ...". There are mainly two ways to create an AI machine. The first one is by using explicit programming, where each action is coded and the output is the result of complex but clear cause and effect relations. The machine follows an exact complex logical structure to accomplish the task. In explicit programming, the code is written to reflect the programmer's will through a fully planned flow of events. The second one is by using implicit programming. The program is still coded by a human, but the logical process that leads to the result is created by the machine itself, usually learning from a trial and error

process with some random influences. In this sense, Machine Learning covers only a subset of Artificial Intelligence. This latter approach has unique features whose potentialities interested researchers all over the world. On the other side, the usage of those self-taught machines in real-life applications raises security issues for which the model can not be treated as a black-box, especially in critical operations such as (e.g.) surgery operations or economic predictions. This paradigm opens up a wide range of studies, including Machine Learning Explainability, whose goal is to untangle machines' decision process. The Artificial Neural Network is one of the most promising models in the Machine Learning universe. It follows that Machine Learning can be implemented into several ways: one of them is by means of Artificial Neural Networks, although other models exist, such as (e.g.) the tabular Q-Learning. The set relation between AI, ML, and ANN can be summarized as follows:

$$Artificial\ Neural\ Network \subset Machine\ Learning \subset Artificial\ Intelligence$$

For the sake of completeness, it is necessary to specify that Artificial Neural Networks exist also in the non-learning version, for which Machine Learning is not involved. However, in the greatest majority of use cases, it refers to the trained version. Neural networks are mathematical models that allow the resolution of problems in which explicit programming is difficult. The range of application is particularly broad, ranging from decision making to computer vision. An example of pattern recognition based on neural networks is shown in Figure 2.1, in which a UAV is engaged to aid a search and rescue operation (SAR). The image shows how it can recognize the injured in real time using only the on-board instrumentation.

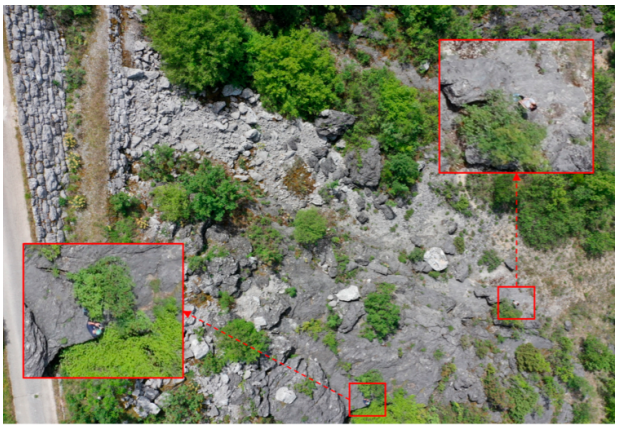


Figure 2.1: [12] UAV engaged in a SAR operation - neural networks aid the computer vision to recognize injured. Thanks to convolutional layers, pattern recognition can occur on multiple objects (classes) in the same image.

2.1.2 Perceptron Model

Perceptron is the elementary constituent unit of an artificial neural network and it itself constitutes the simplest network, with only one layer and one output unit. It was developed in the 1950s and 1960s by the scientist Frank Rosenblatt in the field of psychology, to achieve a simplistic mathematical model of how the human brain works.[13] The Perceptron aims to emulate the behaviour single a neuron. With reference to Figure 2.2, where the isolated perceptron is shown, and Figure 2.3, where the perceptron is immersed in a generic network, it can be thought as a computational unit which performs a two-step calculation, showed in Equation 2.1, (a) (b):

$$\begin{cases} z_1^1 = \sum_{i=1}^{n} w_{i1}^1 a_i^0 + b_1^1 & \text{(a)} \\ a_1^1 = \sigma^1(z_1^1) & \text{(b)} \end{cases} \tag{2.1}$$

$$\vec{w_1^1} = \begin{bmatrix} w_{11}^1 \\ w_{21}^1 \\ \vdots \\ w_{n1}^1 \end{bmatrix} \qquad \vec{a^0} = \begin{bmatrix} a_1^0 \\ a_2^0 \\ \vdots \\ a_n^0 \end{bmatrix}$$

The input of the network is the feature vector $\vec{a^0} \in \mathbb{R}^n$, broken down into its components $a_1^0, a_2^0, ..., a_n^0$. The superscript number is not to be confused with the pow operation, but it indicates the layer of the activation number, while the subscript number indicates the progressive enumeration of the neurons of the layer indicated in the superscript, Figure 2.3. This feature vector is usually an external input that represents the state of the problem. The output of the perceptron is the activation $a_1^1 \in \mathbb{R}$, and it is the result of the two-step calculation of Equation 2.1 that is fired in the outbound link. The relation many-to-one suggests the traveling direction in which the information propagates. When the information propagates from the input layer to the output layer, left to right as in this case, the propagation is called forward propagation. The first step of the calculation is a linear weighted sum of all elements of the input vector $\vec{a^0}$ weighed according to the weight of the corresponding connection, plus a bias term that is internal to the perceptron that produces the intermediate quantity $z_1^1 \in \mathbb{R}$. The second step of calculation applies a generic scalar function to z_1^1, $\sigma^1 : \mathbb{R} \to \mathbb{R}$, producing the output a_1^1. This function is called activation function, and its role is fundamental to achieve the non-linear desired behaviour, the details of which are described in the Subsection 2.1.4. Notice that the perceptron is fully connected with all other neurons contained in the input layer and for that reason, those types of networks are called "Dense".

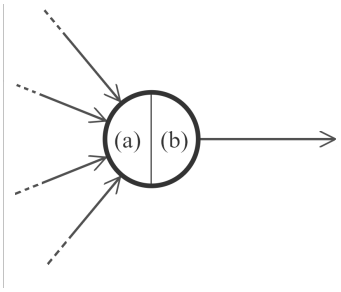


Figure 2.2: The Perceptron - visualization of the computational unit. The perceptron, represented through the circle, is connected with the outside through several incoming edges and one outgoing connection edge. Inbound links provide inputs to the computational unit, i.e. the activation of the previous level, which the perceptron uses to produce the output through Equations 2.1 (a) and (b). The output is then propagated toward the next unit thanks to the outbound connection. Figure 2.3 shows an example of a perceptron immersed in a network.

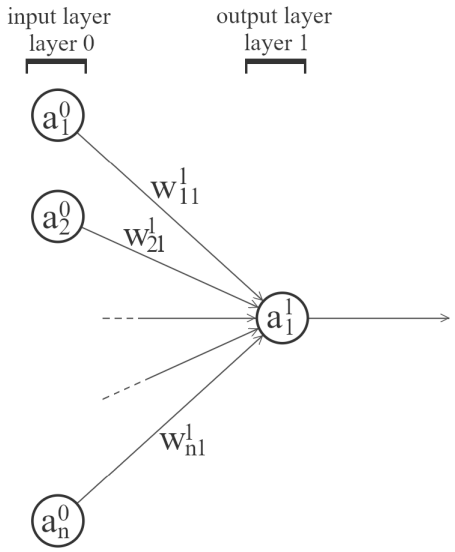
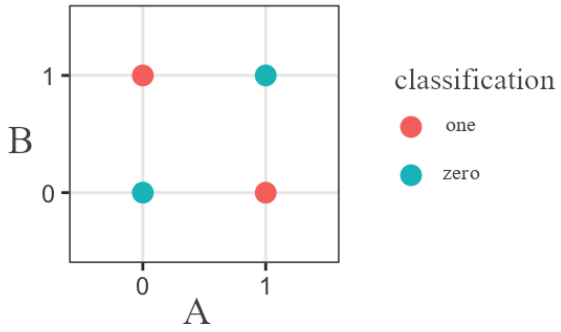


Figure 2.3: A perceptron immersed in a network - it is shown its connection with the input layer. The input layer has n several units which are activated from an external source: $a_1^0, a_2^0, ..., a_n^0$. The input propagates toward the perceptron using the edges, where the Equation 2.1 computes the output a_1^1. Noticed that while the weight values are proper of each edge, the bias term is contained in the perceptron and not visible, as well as the activation function.

The perceptron was critically discussed by Marvin Minsky and Seymour Papert in the 1969 book *"Perceptrons: an introduction to Computational Geometry"*

showing that the class of functions it was able to discriminate was limited to linearly separable forms. The proposed problem was to divide the solutions of the Exclusive-OR XOR function. Placing the truth table data in a Cartesian system it is clear that the perceptron would not be able to learn this problem simply because there is no decision surface to separate the inputs from the outputs with a straight line, as shown in Figure 2.4.



A	B	Z=A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

Figure 2.4: On the right, the XOR Boolean truth table. On the right, the function represented on the Cartesian plane.

The perceptron's reduced computational capacity led to a loss of interest in the subject for more than 10 years, ending what the mathematician Douglas Hofstadter, author of *"Godel, Escher and Bach"*, had defined as the golden age of dreams boolean of artificial intelligence. It was only through the development of multilayer networks and the error back-propagation algorithm, introduced in 1986, that interest in artificial neural networks was reignited. By connecting multiple perceptrons together, it is possible to create more complex neural networks with multiple intermediate layers as well as multiple output networks, as the Deep Neural Networks, and overcome the limits of the single-perceptron model.

2.1.3 Neural Nomenclature - Multilayered Perceptron

Neural networks are involved in most cutting-edge machine learning use cases. As a consequence, research is constantly focused on this field and the literature continuously evolves, creating plenty of variations from the standard models. However, all variations present some common key elements that render the following nomenclature particularly useful for the understanding of this document. The reference architecture is a generic Standard Fully Connected Artificial Neural Network with multiple outputs - Fig 2.5. Each circle represents a Neuron, also called a Unit or Node. Neurons are stacked in vertical piles called layers. There are three types of layers. The Input Layer is the starting point of the forward propagation; its neurons are initialized with an external source of information, and

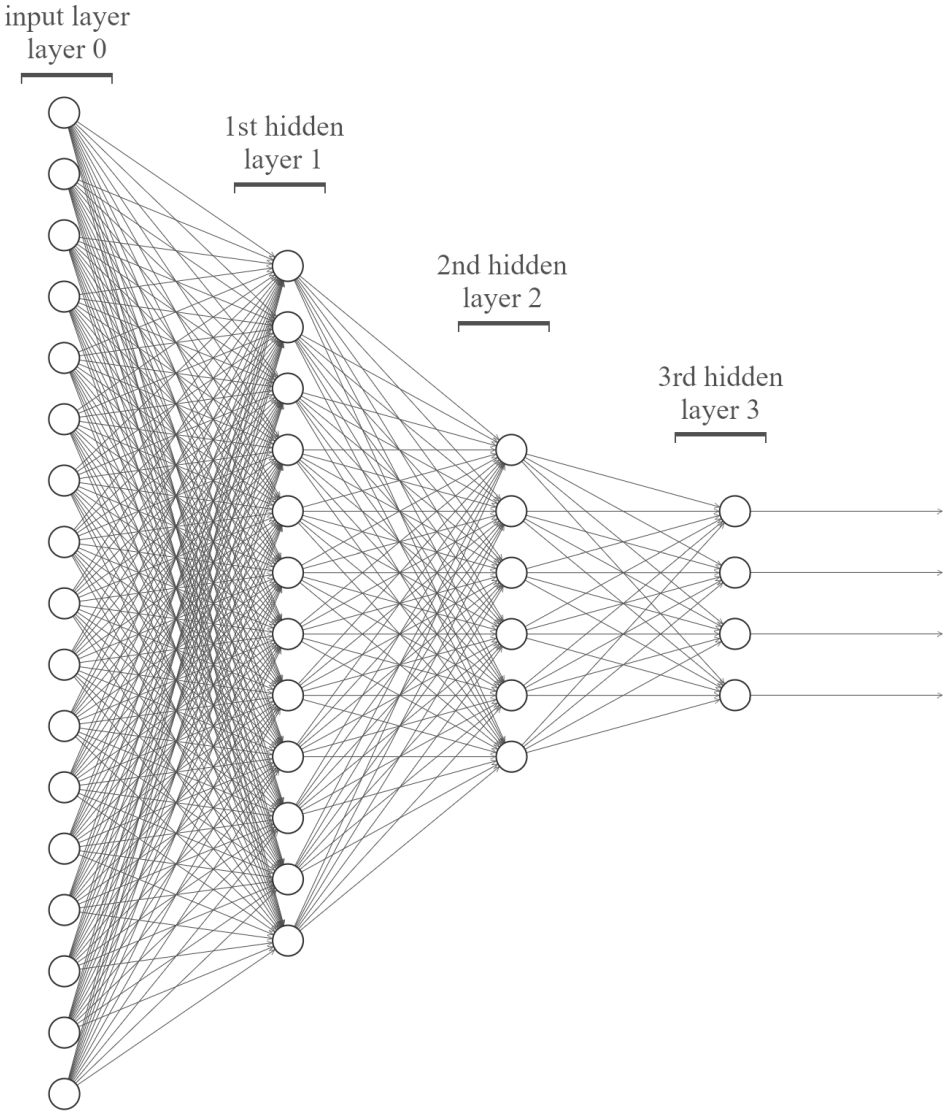


Figure 2.5: A Standard Fully Connected Artificial Neural Network with four output. This network has an architecture with 3 computational layers, two of which are hidden and the last one is the output. The input vector contains 17 feature information, each of which constitutes the activation of a neuron.

only propagates towards the next layer. On the other side, the Output Layer is the final computation layer; it only receives information from the previous layer

and constitutes the arrival point. All the other layers that are Input nor Output Layers are called Hidden Layers. The number of total layers, indicated with L, is the sum of hidden layers plus the output layer: the input layer is not counted as the input neurons are filled with external information and does not compute Equation 2.1: in the network in Figure 2.5 there are $L = 3$ three layers. Each unit fires in output a quantity that is called activation and is generally indicated with a_i^l, indicating the activation of the *i-th* neuron of the layer l. Note that the same activation is fired in all the outbound links indistinctly. The last graphical element is the edge. Edges, also called Synapses or Connections or Links, joint together two neurons and represent how information is propagated through the network. Each connection has an associated value called weight, as anticipated in the perceptron model. When an activation value propagates through an edge, its value is weighted with the correspondent edge value. The notation to represent a generic weights is w_{ij}^l: this indicates the connection between the *i-th* neuron of the layer $l-1$ with the *j-th* neuron of the layer l. The activation of a generic unit a_i^l is a function of the input weights, of each activation of the previous level, of the activation function and finally of the bias term b_i^l. There is a bias term for each neuron, and the set of all biases and all connection weights constitute the set of trainable parameters, which are tuned during the training session to make the network learn. Each layer l has an associated activation function σ^l that is unique for all the units of the correspondent layer. Note that each Unit of each Layer is connected to each other Unit of the previous and the subsequent layer (if any): the traditional taxonomy calls this type of connection between layers Fully Connected or Dense. Finally, regardless of the verse of propagation, those type of architecture where the output does not influence the input are called Feed-forward networks, in contrast with Feed-back networks which have feedback edges that link the Output Layer with the Input Layer, and that characterize another type of networks called Recurrent Neural Networks. For standard networks, as reference one showed in Figure 2.5, the forward propagation can be summarized in the following four steps:

Step 1: initialization – the input layer is filled with an external state.

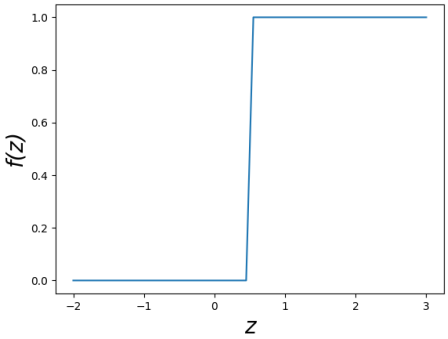
Step 2: propagation – each input neuron is fired toward the following layer and weighted with the weight of the correspondent link in which the information is traveling.

Step 3: computation – each neuron of the following layer receives the propagated information, computes Equation 2.1, and fires again the information toward the following layer.

Step 4: iteration – *Step 2* and *Step 3* are iterated until the output layer is filled and results can be read.

2.1.4 Activation functions

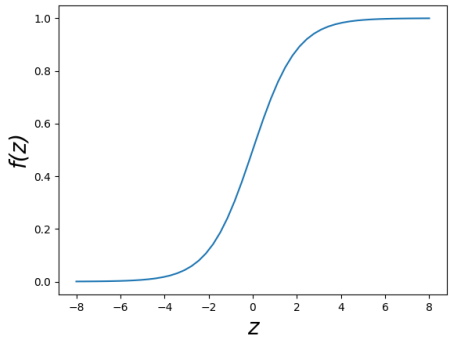
The activation function plays a fundamental role in both computations for forward-propagation and the training mechanism. If each unit performed only the first step described in Equation 2.1, the output level would contain only linear combinations of the initial vector $\vec{a^0}$. In its original publication, referring to human neurons, Rosenblatt stated:[13] *"If the algebraic sum of excitatory and inhibitory impulse intensities is equal to or greater than the threshold (b) of the A-unit, then the A-unit fires, again on an all-or-nothing basis"*, where A-unit is the equivalent to a Node, Unit or Neuron. In the same way, the biologically inspired perceptron emulates this behaviour by using the activation function σ^l. The information is fired to the next layer only when the excitation level is greater than a threshold. There are various types of activation functions, each one with pros and cons. The first perceptron model used a unit step activation function with a threshold, following the "all-or-nothing" rule. Let $b \in R$ be a generic bias term, the threshold value. The step function is described as:



$$f(z) = \begin{cases} 0 & \text{if } z < b \\ 1 & \text{if } z \geq b \end{cases} \qquad (2.2)$$

Figure 2.6: Step function with a bias example of 0.5 - this was the first activation function used in the pioneering perceptron model, although the non-differentiable point in $z = b$.

Although Rosenblatt understood that the perceptron model needed an activation function that was able to discriminate the active from the passive state, the mathematical implementation with the step function presented the problem of the non-differentiability in $z = b$, as shown in Figure 2.6. Logistic activation function, also known as Sigmoid activation function $\sigma(x)$ overcome this problem with a formulation that is monotonic and has a first derivative which is bell-shaped and easy to use, as shown in Equation 2.4. Its output is contained in $(0, 1)$ and is commonly used for classification:

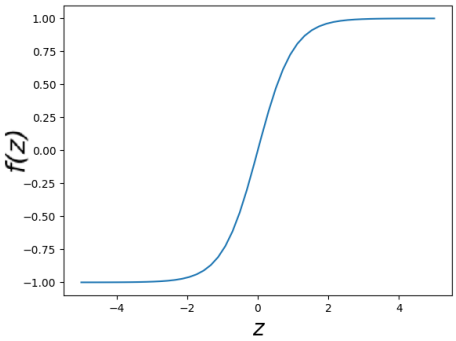


$$f(z) = \sigma(z) = \frac{1}{1 + e^{-z}} \qquad (2.3)$$

$$f'(z) = \sigma(z)(1 - \sigma(z)) \qquad (2.4)$$

Figure 2.7: Logistic or Sigmoid activation function. Note that the output is contained in (0, 1), with horizontal tangent to plus and minus infinity.

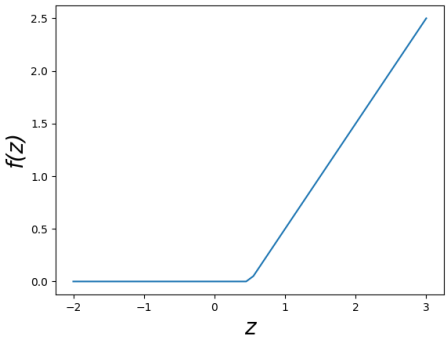
The main limitation of the Sigmoid function is its codomain, forced to be between $(0, 1)$ for each input value. Hyperbolic tangent activation function, in comparison with logistic function, allows the output to have also negative values, expanding the range a $(-1, +1)$ as shown in Figure 2.8.



$$f(z) = \tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \qquad (2.5)$$

Figure 2.8: Hyperbolic tangent activation function over the range $[6, 6]$. Note the admissibility of negative values when compared with Figure 2.7.

The ReLU, acronym of Rectified Linear Unit, was first introduced in 2000 with strong biological motivations and mathematical justifications; later on, in 2011 it has shown an excellent behaviour with Deep Artificial Neural Networks, thanks to its semi-linear behaviour and practicality,



$$f(z) = \max(0, z + b) \qquad (2.6)$$

Figure 2.9: Rectifier Linear Unit - ReLU function with a bias example of 0.5 - note the linear behaviour the follows the bias/activation point. This activation function is particularly important since the network used for the coverage path planning uses exploits ReLU in most of the layers.

Although there are many other activation functions, the above listing is enough for the purpose of this document. Noticed that all the activation functions have a remarkable non-linear behaviour, a fundamental property which springs the potentiality of Feed-Forward artificial neural network models, enough to earn it the name of Universal Approximation.

2.1.5 The Universal Approximation Theorem

Let $n \in \mathbb{N}$ be the number of input neurons and $m \in \mathbb{N}$ be the number of neurons in the output layer. The Neural Network creates a functional relation between two set: the domain of dimension n and the co-domain of dimension m , $NN : \mathbb{R}^n \to \mathbb{R}^m$, by means of Equation 2.1 applied in each computational unit. If on the one hand, the non-linearity introduced by each activation function raises up the complexity of this function, on the other hand, this represents the key-element that renders Neural Network a Universal Approximator. In 1989 Hornik, Stinchcombe and White published the article:[14] *Multilayer feedforward networks are universal approximators.* The article proof the fact that for any continuous function g on a compact set K, there exists a Feed-forward Neural Network, having only a single hidden layer, which uniformly approximates g to within an arbitrary $\epsilon > 0$ on K. This amazing outcome renders Artificial Neural Networks not only a powerful instrument to access multidimensional spaces in an elegant manner but also a key element for function approximation of any space dimension. Regardless of the final use of the neural network, it is then clear that the mathematical purpose beyond the training process is to approximate a complex function in spaces that are usually multidimensional. The learning mechanism behind the training comes down into a minimum cost problem, as explained in Subsection 2.2.4. Potentially, the more

complex network architecture, the more the approximate function will minimize the cost function. The main drawback is the limitation of computational resources, so the addition of a single neuron in an articulated network would cause a surge in processing time. To have an idea about the order of magnitude, the neural network described in Figure 2.11 has 3 trainable parameters, the neural network described in Figure 2.5 has 322 trainable parameters, while the neural network used for the coverage path planning problem has 106˙684 trainable parameters. An example could clarify the ANNs purpose. Since the perceptron is the elementary unit, step-back to the perceptron model. Its simple architecture allows immediate visualization of the approximation process. Consider a perceptron with an input layer of two units a_1^1, a_1^1 and a ReLU activation function, as shown in Figure 2.10: this kind of perceptron is usually used to classify the data into two parts, therefore, it is also known as a Linear Binary Classifier. The linear name should not be misleading as the ReLU function is and remains non-linear. In the example of Figure 2.11, two classes, represented by the white dots and black dots must be grouped. The purpose of the binary classifier is to train the model in order to have the output y in a high state when the combination of x_1, x_2 correspond to a black dot, or the output in a low state when the combination of the inputs corresponds to a white dot. The functional relation of this particular architecture of the perceptron can be written in the extended form by combining Equation 2.1.(a) and Equation 2.6:

$$y = a_1^1 = f(a_1^0, a_2^0) = \max(0, w_{11}^1 a_1^0 + w_{21}^1 a_2^0 + b_1^1) \tag{2.7}$$

This relation contains three trainable parameters: the two weights w_{11}^1, w_{21}^1,

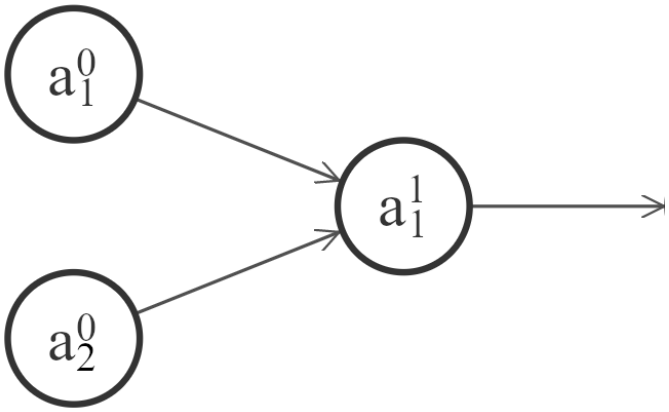


Figure 2.10: The purpose of this simple neural network is to divide the data shown in Figure 2.11. The equation that governs its behaviour has been made explicit in Equation 2.7.

corresponding to the two connection edges, and the bias term b_1^1 of the ReLU

function. By adjusting the trainable parameters it is possible to move the threshold value that allows the perceptron to fire (high state, black dots) or not (low state, white dots). This process is known as neural network training. The network starts with random trainable parameters that do not group the two classes; after an iterative process, the classification converges and the network is able to distinguish one class from another. In the Figure 2.11 there are shown three moments of the training process, then three ways of splitting white and black sets: the line H_3 belongs to a training step in which the network was not tuned to classify the dots and so its classification is wrong because it does not divide the two sets; H_1 and H_2, instead, correctly divide the two sets - however, H_2 should belong to the last training steps, when the net has been trained since its division is preferred because it minimizes a reference value called Loss Function that will be explained in Section 2.2. In the example shown in Figure 2.11, a single perceptron with a

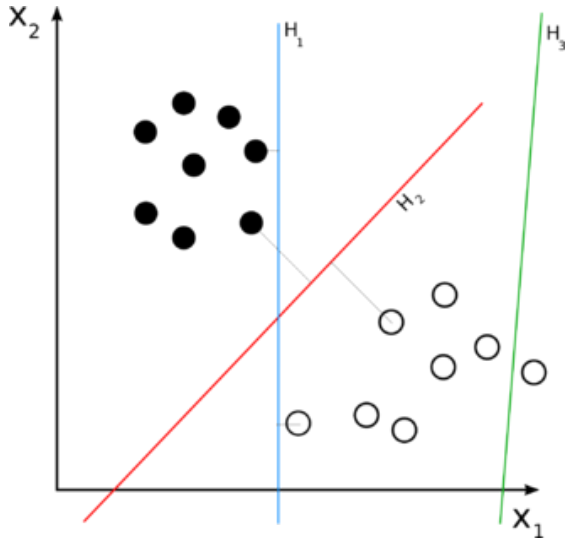


Figure 2.11: In this case, the solid and empty dots can be correctly classified by any number of linear classifiers. H_1 (blue) classifies them correctly, as does H_2 (red). H_2 could be considered "better" in the sense that it is also furthest from both groups. H_3 (green) fails to correctly classify the dots.

ReLU activation function was sufficient to linearly divide the two sets. For more complex distributions it is likely that it doesn't exist any efficient solution with a simple perceptron or classes are not linearly separable. In those cases more complex neural networks are mandatory. The more complex the architecture is, the more accurate the classification. In Figure 2.12, it is shown artificial neural networks with increasing complexity. For the same data, or column of the table, it is shown how the correspondent network would solve the classification problem. The first example shows the X-OR problem, highlighting the evident limitations of

the single-layer perceptron, since a point "A" is contained in the "B" partition. For an efficient clustering on the X-OR function, a network with at least two layers is required. For more complex data distribution as the second double-spiral case, the network with two layers is not sufficient anymore and a three-layered network is required. For more complex data, it is the task of the designer to choose an appropriate architecture, without exceeding in complexity nor a too minimalist structure.

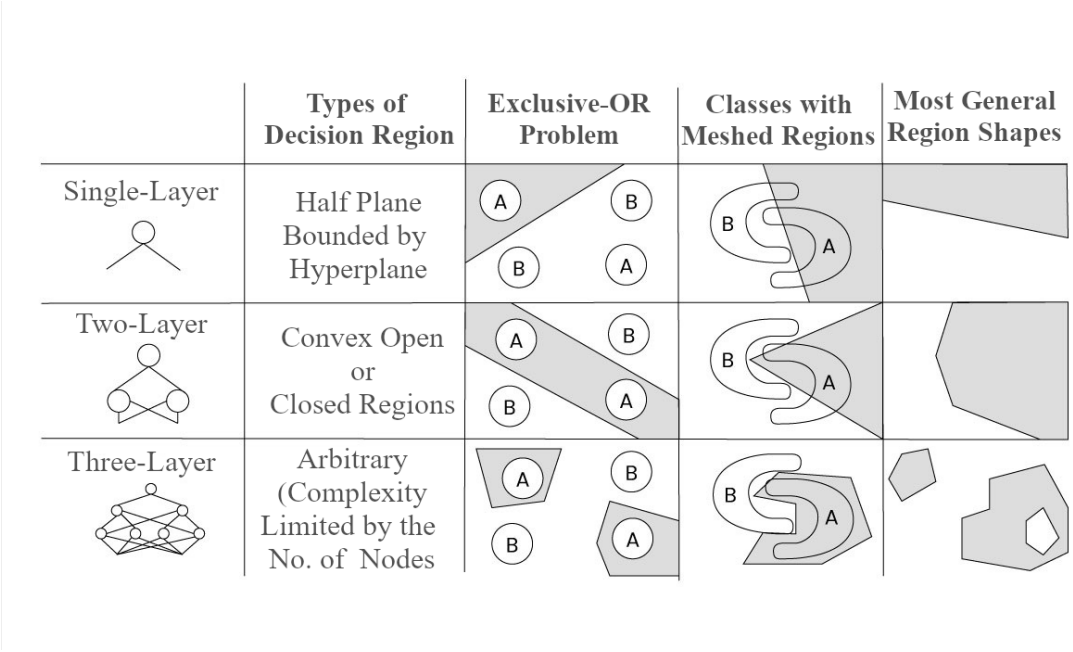


Figure 2.12: Different levels of approximation and different datasets. The single-layer network is able to classify only linear separable classes. Networks with increasing complexity are able to group data with more articulated distribution.

2.2 Supervised Learning Process

The following discussion specializes in the case of Supervised Learning, object of use in the development of the algorithm, although many concepts can also be generalized to other learning paradigms.

2.2.1 Types of Learning

There are mainly three types of learning:

- Supervised Learning: the machine is trained by using a labeled database where the data is already tagged with the correct answer. Each entry of the database contains two coupled information: an input state vector, also called feature vector, and its label that contains information about how the output should be. The purpose of the neural network is to emulate the input-output association by finding their inner relations. As a result of the training, the network should be able not only to predict the correct label of the training set but also of unforeseen data. Supervised Learning deals mainly with two types of problems - classification problems and regression problems. An example of application can be image classification: for a fixed number of classes (airplane, car, train) and given a labeled database of vehicles, the neural network should learn which class is represented in the picture.

- Unsupervised Learning: only input states are known and the machine explores the underlying patterns among them and predicts the output. The purpose is to finds relationships or correlations between the available data when those are too complex to guess. In this sense, the learning process is opposite to Supervised Learning since the correct outputs are not known: the learning process is self-organized. Unsupervised Learning deals with problems such as K-means clustering or Principal Component Analysis. An example of application can be recognition of cancer risk factors by analyzing the clinical history of patients and finding common patterns that may have led to the disease.

- Reinforcement Learning: these algorithms are useful in the field of robotics and gaming. One or more agents learn to explore an environment by interacting with it. On a reward/punishment basis, some behavioral policies are encouraged in order to reach a goal. This learning mechanism is an extremely simplified version of how humans learn.

2.2.2 What does learning means?

Behind the learning process of the neural networks, there is a mathematical framework that exploits differential algebra. The following description is focused on Supervised Learning for Classification problems since the exploration algorithm relies on a supervised method. In the Perceptron Model, Subsection 2.1.2, and the Multi-layer Perceptron Model, Subsection 2.1.3, the forward propagation has been introduced. The forward propagation is used when the neural model is asked to predict a solution. The input layer is filled with an external initial state called the feature vector. This state is then weighted and propagated through all the hidden layers toward the output layer, thanks to the edges by using Equation 2.1 in each node. Once the output layer is reached, the final prediction can be read and used

for its purpose. In contrast to forward-propagation, backward-propagation is used in the model training. This process, instead, starts from the predicted results of the output layer and propagates backward toward the input layer for each training example, tuning the trainable parameters following some criteria. Given a set of $N_{trng} \in \mathbb{N}$ training examples in the form $\{(\vec{a}_1^0, \vec{a}_1^L), ..., (\vec{a}_{N_{trng}}^0, \vec{a}_{N_{trng}}^L)\}$ such that $\vec{a}_i^0$ is the feature vector of the i-th example and $\vec{a}_i^L$ is its label, with $i = 1, ..., N_{trng}$, a learning algorithm seeks a function $f : X \to Y$ where X is the input set of possible states and Y is the output set. The function f is an element of some space of possible functions F, usually called the hypothesis space. This hypothesis space is determined by the network's architecture: the number of layers, the number of units for each layer, the activation functions, and the connections, determine a specific family of functions where the solution is searched by tuning trainable parameters. The design of the neural network architecture is hence a delicate task: a wrong choice could lead the network to poor optimization, therefore low learning capabilities. For each training input sample of the training dataset $\vec{a}_i^0$, the neural network is asked to try to predict an output. Predicted results are compared to the true labels: a global loss function evaluates the quality of the results. The loss function, described in Subsection 2.2.3, produces a scalar value called loss: minimizing the loss allows an alignment between the network predictions and the attended results. This process represents the training part: the neural network tunes its trainable parameters in order to minimize the loss value. It is important to underline that the loss function is not calculated punctually for each sample but considers the overall loss over all the N_{trng} examples. When the entire dataset is passed forward and backward through the neural network once, literature calls it an epoch. For large datasets one epoch could be too big to load in the computer's RAM at once, hence it is usually divided into several smaller batches for logistic reasons. The tuning criteria by which parameters are updated follows the Gradient Descent algorithm, described in Subsection 2.2.4. The training part and the application use of the network occur in two different stages: in this sense, the supervised learning performs offline training. The trainable parameters are changed exclusively during the training session; once the network has achieved a sufficient accuracy, all the parameters are frozen at their value and the network is ready to perform, so the supervised training is an offline training since the learning stage of the network is previous of its application.

2.2.3 Loss function

Behind the main purpose of the neural network, the loss function represents the mathematical mean by which the learning is possible. All the parameters are tuned with the only aim to minimize the loss function. For a fixed architecture of the network, the class of functions is fixed, and the training process seeks for a function

among the possible choices, following the criteria explained in the Subsection 2.2.4. The loss is calculated by comparing the predicted results of the network with the desired results contained in the training samples. Let Y be the output set. The loss function computes a scalar value that is representative of the quality of the predictions: $L : Y \times Y \rightarrow \mathbb{R}$. For the same output set, the choice of the loss function opens up to a wide range of possibilities, although some of them are preferred with respect to others in machine learning applications. One fundamental property useful is the convexity of the function, for reasons that will be clear in the following subsection. First of all, it is important to point out that the output set is a convex set. Let $\hat{y} \in \mathbb{R}^4$ be an output vector, such that $\hat{y_i} \in [0, 1]$ is its *i-th* component. The set Y is convex if, for each pair $y_1, y_2 \in Y$, their affine combination described in Equation 2.8 is contained in Y:

$$(1 - t) \cdot \hat{y_1} + t \cdot \hat{y}_2 \in Y, \tag{2.8}$$

$$\forall \hat{y_1}, \hat{y_2} \in Y, t \in (0,\, 1)$$

The set Y is a convex set since there is no way such that any component $\hat{y_i}$ of any vector $\hat{y} \in Y$ lies outside the interval [0, 1] for the affine combination. The loss function $L(t, \hat{y}) : Y \times Y \rightarrow \mathbb{R}$ is convex if it satisfies the inequality:

$$f(t\hat{y_1} + (1 - t)\hat{y_2}) \leq tf(\hat{y_1}) + (1 - t)(\hat{y_2}), \tag{2.9}$$

$$t \in [0, 1],$$

$$\forall \hat{y_1}, \hat{y_2} \in Y$$

In order to reach the minimum value of the loss function, it would be sufficient to follow the direction of the maximum negative gradient of a convex function. The gradient descent algorithm described in Subsection 2.2.4 aims to tune the trainable parameters of the network in order to minimize the loss function. Unfortunately, only a small portion of neural network applications relies on convex loss functions. When supervised learning is applied to a binary classification, where the network is asked to predict a true or false statement, the logistic loss function ensures function convexity. In this work, where the neural network is asked to predict over four different classes, the logistic loss function can't be applied. This implies that the solution could be stuck in a local minimum instead of a global minimum. However, some corrective methods aim to overcome these problems as explained in Subsection 2.2.4. The loss function is also known as the cost function since the problem can be treated as a cost optimization problem, and it is usually indicated with C. The selected cost function is the categorical cross-entropy, which compares the predicted probability distribution with the target probability distribution:

$$C = -\frac{1}{E} \sum_{e=1}^{E} \sum_{c=1}^{N_c} \hat{a}_{c,e}^{L} \log a_{c,e}^{L} \tag{2.10}$$

, where E is the number of sample experiences, N_C the number of output classes, $\hat{a}^L_{c,e} \in [0,1]$ is the target prediction and $a^L_{c,e} \in [0,1]$ is the real neural network's prediction. The training session revises all the network's hyper-parameters, such as weight and biases, in order to minimize the loss function. Moreover, the trained network is also able to predict the best actions to take in a never-seen situation. Apparently, it is possible to demonstrate that the categorical cross-entropy loss function is convex with respect to the output prediction $a^L_{c,e}$. However, the output is the result of a forward propagation from the input layer through the hidden layers and the whole chain of calculus must be accounted for. If only linear activation functions were used then the statement would be correct and the convexity property of the loss function would be satisfied, however, for the reason explained for the universal approximation theorem described in Subsection 2.1.5, non-linear activation functions are needed to solve complex classification and regression problems and increase the accuracy on predictions.

2.2.4 Gradient Descent

The loss function described in Subsection 2.2.3 creates a continuous multi-dimensional surface that constitutes the means in which the gradient descent finds application. For a given input vector, the neural network is always able to predict an output. This output is combined with the target output by using Equation 2.10 and identifies a precise point value in the multidimensional-surface: the loss. The smaller the loss value, the more capable the network is of predicting an output approaching the target. By revising its trainable parameters, the neural network tends to align its prediction with the target for that particular input vector, in order to minimize the loss function. By following the loss function's gradient, it is possible to reduce the loss value. In order to follow the gradient, it is necessary to identify the set of adjustable parameters on which the loss value depends. Since fully connected neural networks are organized in layers, it is useful to analyze the gradient descent algorithm per each layer. The final result will link the reduction of the loss value with an adjustment of all the trainable parameters, from the output layer to the input layer, passing through all the hidden layers. Consider a simplified case with the network configuration with a number of layers $L = 2$ with one unit per layer, as shown in Figure 2.13. An input is fed in the first layer x^0 in the first (and only) neuron x_1, and it represents the activation $a^0_1 \equiv x^0_1$. Let $\hat{a}^2_1 \in \mathbb{R}$ be the label of the input, the known target prediction. In order to compute the cost C_0, it is necessary to forward propagate the input. For this simplified case, it is possible to explicit all the equations starting from the input:

$$\text{known input} = a_1^0$$

$$\text{1-st neuron computation} = \begin{cases} z_1^1 = a_1^0 w_{11}^1 + b_1^1 \\ a_1^1 = \sigma(z_1^1) \end{cases} \tag{2.11}$$

$$\text{2-nd neuron computation} = \begin{cases} z_1^2 = a_1^1 w_{11}^2 + b_1^2 \\ a_1^2 = \sigma(z_1^2) \end{cases} \tag{2.12}$$

$$C_0 = (a_1^2 - \hat{a}_1^2)^2 \tag{2.13}$$

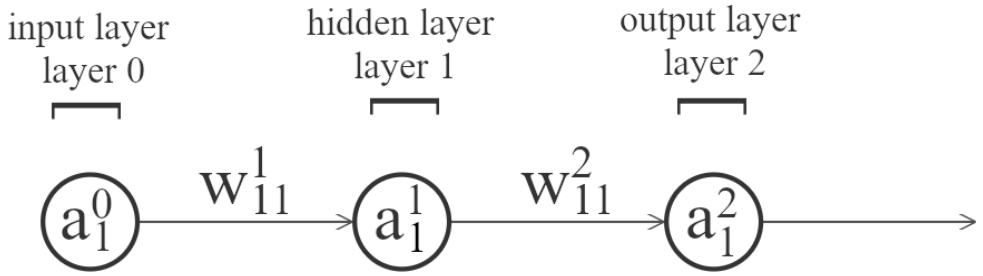


Figure 2.13: Different levels of approximation and different datasets. The single-layer network is able to classify only linear separable classes. Networks with increasing complexity are able to group data with more articulated distribution.

It is important to keep in mind that only the external brackets' 2 is a pow; when associated to a trainable parameter it denotes the correspondent layer, as explained in Subsection 2.1.3. So, for instance, w_{11}^2 is the weight associated with layer 2 and connects neuron 1 of the 2-nd layer with neuron 1 of the 1-st layer. For this example, the Mean Squared Error (MSE) has been used to compare the target prediction $\hat{a}_1^2$ with the network prediction a_1^2. If those two values are equal, then the network is predicting as it should behave, and the cost C_0 is zero. For a non-trained network, it is likely that the predicted result diverges from the target; the gradient descent offers a base to understand the influence that each trainable parameter has on the cost. In this example, four parameters $w_{11}^2, b_1^2, w_{11}^1, b_1^1$ can be tuned to reduce the cost, but generalized concepts can be applied in case of studies with further increased complexity, as the 106˙684 parameters neural network used in this

simulation and presented in Chapter 3 - Inside the Neural Network. The gradient of the cost function produces a four-dimensional vector field:

$$\nabla C|_{C=C_0} = \frac{\partial C_0}{\partial w_{11}^2}\vec{\boldsymbol{v_1}} + \frac{\partial C_0}{\partial b_1^2}\vec{\boldsymbol{v_2}} + \frac{\partial C_0}{\partial w_{11}^1}\vec{\boldsymbol{v_3}} + \frac{\partial C_0}{\partial b_1^1}\vec{\boldsymbol{v_4}} \tag{2.14}$$

, where nabla, ∇, denotes the vector differential operator while $\vec{\boldsymbol{v_z}}$ denotes the *z-th* versor. The gradient offers a tool to measure the sensitivity to change of the cost function in C_0 with respect to a change in one of its arguments. Note that C_0 identifies a specific point on the cost surface since it is the result of the cost of one training sample. To tune the trainable parameter it is useful to start from the last layer L and then go back toward the input layer: for this reason, this operation is called backpropagation. Starting from the weight w_{11}^2, the partial derivative can be adjusted by using the rule chain:

$$\frac{\partial C_0}{\partial w_{11}^2} = \frac{\partial C_0}{\partial w_{11}^L} = \frac{\partial C_0}{\partial a_1^2} \cdot \frac{\partial a_1^2}{\partial z_1^2} \cdot \frac{\partial z_1^2}{\partial w_{11}^2} \tag{2.15}$$

Lets define the quantity δ_i^l for the l-th layer and for the i-th neuron as:

$$\boxed{\delta_i^l = \frac{\partial C_0}{\partial z_i^l} = \frac{\partial C_0}{\partial a_i^l} \cdot \frac{\partial a_i^l}{\partial z_i^l}} \tag{2.16}$$

By using the equations 2.13 and 2.12, for a generic activation function σ, and for the specific case of MSE cost function:

$$\frac{\partial C}{\partial a_1^2} = 2(a_1^2 - t)$$

$$\frac{\partial a_1^2}{\partial z_1^2} = \frac{\partial \sigma(z_1^2)}{\partial z_1^2} = \sigma'(z_1^2)$$

$$\delta_1^2 = 2(a_1^2 - t) \cdot \sigma'(z_1^2)$$

$$\frac{\partial z_1^2}{\partial w_{11}^2} = a_1^1$$

, resulting in:

$$\frac{\partial C_0}{\partial w_{11}^2} = \frac{\partial C_0}{\partial w_{11}^L} = \delta_1^2 \cdot a_1^1 = 2(a_1^2 - t) \cdot \sigma'(z_1^2) \cdot a_1^1 \tag{2.17}$$

For the same layer, it is possible to use similar differential relations to obtain the partial derivative of the cost respect to b_1^2, taking into account that the partial derivative of z_1^2 in relation with b_1^2 is unitary :

$$\frac{\partial C_0}{\partial b_1^2} = \frac{\partial C}{\partial a_1^2} \cdot \frac{\partial a_1^2}{\partial z_1^2} \cdot \cancelto{= 1}{\frac{\partial z_1^2}{\partial b_1^2}} = \delta_1^2 = 2(a_1^2 - t) \cdot \sigma'(z_1^2) \tag{2.18}$$

To find the derivative of parameters that result in deeper layers than the last one, as the first one, it is possible to lean Equation 2.16 and Equation 2.18, using the chain rule to extend the differential calculus. Consider the parameter w_{11}^1, that belongs to the layer $l = 1$:

$$\frac{\partial C_0}{\partial w_{11}^1} = \frac{\partial C_0}{\partial z_1^1} \cdot \frac{\partial z_1^1}{\partial w_{11}^1} = \delta_1^1 \cdot \frac{\partial z_1^1}{\partial w_{11}^1} \tag{2.19}$$

The differential term z_1^1/w_{11}^1 can be easily calculated by using Equation 2.11:

$$\frac{\partial z_1^1}{\partial w_{11}^1} = a_1^0$$

The δ_1^1 term can be written in a form that exploits the δ_1^2 term, already obtained:

$$\delta_1^1 = \frac{\partial C_0}{\partial z_1^1}$$

$$\delta_1^1 = \frac{\partial C_0}{\partial z_1^2} \cdot \frac{\partial z_1^2}{\partial z_1^1} = \delta_1^2 \cdot \frac{\partial z_1^2}{\partial z_1^1}$$

The ration between the two intermediate activations z_1^2 and z_1^1 can be obtained combining Equation 2.11 and Equation 2.12:

$$\frac{\partial z_1^2}{\partial z_1^1} = \frac{\partial[\sigma(z_1^1)w_{11}^2 + b_1^2]}{\partial z_1^1} = w_{11}^2 \cdot \sigma'(z_1^1)$$

And the final expression of Equation 2.19 became:

$$\frac{\partial C_0}{\partial w_{11}^1} = \delta_1^2 \cdot w_{11}^2 \cdot \sigma'(z_1^1) \cdot a_1^0 \tag{2.20}$$

With a similar procedure it is possible to obtain the variational expression for the last trainable parameter b_1^1:

$$\frac{\partial C_0}{\partial b_1^1} = \delta_1^1 = \delta_1^2 \cdot w_{11}^2 \cdot \sigma'(z_1^1) \tag{2.21}$$

The Equations 2.25, 2.18, 2.20, and 2.21 form the basis with which to calculate the 4 components of the gradient for the considered example. Starting from the trainable parameters of the last layer L, Equations 2.25 and 2.18 contain the differential of the cost function respect to the two trainable parameters. As far as it concerns the parameters of previous layer $L-1$, the first layer, Equations 2.20 and 2.21 rely on the relations calculate for the L-th layer. The backpropagation exploits a cascade effect for which the influence of a parameter upon the cost function relies on the following layer. Once the gradient is calculated in all its component, it is possible to guess how parameters can be tuned in order to reduce the loss function, much that sometimes it is referred as the negative gradient $-\nabla C$. A generic edge is updated by following the instruction:

$$w_{ij}^{l} := w_{ij}^{l} - \alpha \frac{\partial C_0}{\partial w_{ij}^{l}} \tag{2.22}$$

, while biases are updated with

$$b_{i}^{l} := b_{i}^{l} - \alpha \frac{\partial C_0}{\partial b_{i}^{l}} \tag{2.23}$$

The term α is the learning rate, a configurable hyper-parameter fundamental in the training process. It dictates the magnitude of the step by which the gradient descent is followed. The training process aims to reduce the cost function by moving across the multidimensional surface, but the extent to which this task can be accomplished can't be with analytical methods. Conversely, if on the one side numerical methods allow the descent of the gradient, to the other the resulting descent is sharp and only approximates the true gradient since each step follows the inline minimum gradient. Too low learning rate will result in precise descent but long computational time; too high learning rate will not converge in the optimal solution and could cause numerical instability issues in the worst cases. The selection of the learning rate is fundamental and can't be chosen a priori. Modern solutions involve adaptive learning rates, resulting from a gross to fine-tuning by progressively reducing a large α during each epoch. Other methods associate a personal adaptive learning rate to each trainable parameter and are able to distinguish among the most influential parameters. Supervised Artificial Neural Networks are used to carry out complex classification or regression problems and seldom have an elementary configuration as the one shown in Figure 2.13. The following discussion aims to build a base to generalize the concepts and equations previously introduced in this subsection. In architecture with a multitude of output neurons, an edge in the hidden layers influences each output through several connections, resulting in more articulated equations. A generic trainable parameter of the last layer still influences directly only one output of the last layer, so the equations do not change:

$$\frac{\partial C_0}{\partial w_{jk}^{L}} = \frac{\partial C}{\partial a_{k}^{L}} \cdot \sigma'(z_{k}^{L}) \cdot a_{j}^{L-1} \tag{2.24}$$

$$\frac{\partial C_0}{\partial b_k^L} = \frac{\partial C}{\partial a_k^L} \cdot \sigma'(z_k^L) \tag{2.25}$$

For a generic weight of the layer $L-1$, the Equations 2.20 and 2.21 get more complicated since in the multiple output case each contribution must be accounted and the k-th generic neuron influences all the output neurons through multiple connections. Let N_c be the number of output classes, each deeper trainable parameters influences all the output unit and the differential expression became:

$$\frac{\partial C_0}{\partial w_{jk}^l} = \underbrace{\frac{\partial C_0}{\partial a_1^L}\frac{\partial a_1^L}{\partial w_{jk}^l}} + \underbrace{\frac{\partial C_0}{\partial a_2^L}\frac{\partial a_2^L}{\partial w_{jk}^l}} + ... + \underbrace{\frac{\partial C_0}{\partial a_1^L}\frac{\partial a_{N_c}^L}{\partial w_{jk}^l}} = \sum_i^{N_c} \frac{\partial C_0}{\partial a_i^L}\frac{\partial a_1^i}{\partial w_{jk}^l} \tag{2.26}$$

This equation accounts for the interconnection between a generic hidden edge and influences in each output. In under-bracket, each influencing term of the overall cost function. This multi-output contribution will be involved also for a generic δ expression. The relations obtained in Equations 2.20 and 2.21 can be generalized for any neuron of any layer, output or hidden:

$$\boxed{\frac{\partial C_0}{\partial w_{jk}^l} = \delta_k^l \cdot a_k^{l-1}} \tag{2.27}$$

$$\boxed{\frac{\partial C_0}{\partial b_k^l} = \delta_k^l} \tag{2.28}$$

Equations 2.27 and 2.28 are the fundamental blocks for back-propagation. Notice that while a generic a_k^{l-1} is known, the common unknown faction is the δ_k^l. At this point the problem shifts to determining the value of δ_j^l for a generic neuron in the hidden layers. Particularly, the aim of the following manipulation is to find an expression of δ_j^l which relies on the terms δ_j^{l+1} of the following layer:

$$\delta_j^l = \frac{\partial C_0}{\partial z_j^l}$$

$$\delta_j^l = \sum_k \frac{\partial C_0}{\partial z_k^{l+1}} \cdot \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1} \cdot \frac{\partial z_k^{l+1}}{\partial z_j^l}$$

Remembering the expression of z_k^{l+1}, and differentiating respect to z_j^l:

$$z_k^{l+1} = \sum_j w_{jk}^{l+1} a_j^l + b_k^{l+1} = \sum_j w_{jk}^{l+1} \sigma(z_j^l) + b_k^{l+1}$$

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{jk}^{l+1} \sigma'(z_j^l)$$

Finally the expression for a generic delta is obtained:

$$\boxed{\delta_j^l = \sum_k \delta_k^{l+1} w_{jk}^{l+1} \sigma'(z_j^l)} \tag{2.29}$$

The boxed Equations 2.27, 2.28, and 2.29 are the fundamental equations for back-propagation thanks to which is possible to predict the negative gradient of the cost function and tune each trainable parameter to increase accuracy on predictions. One last remark concerns the number of labeled samples. The whole previous discussion has been developed for one fixed target t_0 from which the cost C_0 was calculated. Reducing the cost value following the gradient descent trains the artificial neural network to predict the correct label for that specific input. But the purpose of the network is to perform correctly among a huge quantity of data and not only for that sample. Moreover, if the training iteration was repeated for a new labeled sample t_1, it is likely that the change of the weight set would mess up results achieved on t_0, nullifying the work already done. For that reason, the gradient descent must be used accounting for a large set of data simultaneously, so much so that this training technique is called batch learning. Each component of the gradient is arithmetically averaged over the whole batch in order not to reduce the overall cost function and resulting in improved global performance. This is possible since the neural networks' architecture is fixed and shared among all the forward and backward propagation. There exist several ways to implement gradient descent: the Batch Gradient Descent, the Mini Batch Gradient Descent, and the Stochastic Gradient Descent, also known as SGD. The aim of each method is to minimize the loss function in order to reach the minimum over the dataset, but the main difference consists in the amount of data they use. The Batch GD gradient descent uses the whole set of training data and updates the weights and biases only after the entire dataset has been evaluated. This method would lead directly to the global minimum only if the convexity property of the cost function exists since it would be the only minimum. For real cases cost functions it is unlikely to obtain a convex function over a complex, non-linear network, as explained in Subsection 2.1.5, so the deterministic gradient descent could lead to a local minimum. Moreover, the computational resources required to compute the gradient for a whole training dataset that could contain million or billion of samples would be computationally burdensome and can not be held in the RAM. The SGD overcome this problem by updating the trainable parameters after each training sample, and not after the whole dataset. In this way the cost function changes after each training iteration since each input sample is unique, and the gradient descent is therefore disturbed by a strong algorithmic noise due to a mobile target cost value. On the other side, computational resources are damped due to the streamlined process of training. Moreover, since the cost function changes as the current training sample changes, this process avoids the trainable parameters being stuck in a local minimum. For

convex cost function surfaces, this method does not reach the global minimum directly but oscillates and the result is a swinging descent. A trade-off method is the Mini Batch Gradient Descent. Instead of updating weight and biases punctually for each training data or updating for the whole dataset, the entire set of data is partitioned in mini-batches. The number of entries contained in each mini-batch is defined by a hyper-parameter called batch size. Usually, this number is equal to a pow of 2, for the technical reason that involves the processing on a Graphical Processing Unit (GPU). If the number of total entries does not match with a multiple of the batch size, then the remaining entries form a separate mini-batch. With this method, the algorithm noise is reduced and computational issues do not arise, since the mini-batches are smaller than the batch. There is plenty of other elegant solutions for gradient descent optimization that includes e.g. data shuffling and momentum, but they are not discussed in this work for compactness.

2.2.5 Issues

The main purpose of the supervised trained network used in this work is to predict the correct class given a particular state. Before using the network in practical applications, the training session exploits the labeled training set to achieve the required performances by tuning all the trainable parameters. Correct predictions are achieved by training the network on the same training data multiple and multiple times. After each training epoch, the accuracy on predictions hopefully increases, making the network always more competitive. If on the one hand, high accuracy is a positive network feature, on the other hand there is the risk of overfitting. More than a high accuracy on labeled and known sample, one fundamental property of the neural network is the ability to generalized the acquired classification abilities also in never-seen data. It exists a number of training epochs following which the network loses the ability to transfer classification capabilities to more general cases, so this number should not be exceeded. Moreover, there is not a rule of thumb to determine this value, but must be determined by using a separate dataset, called validation set. This set is involved for the gradient descent, but is used to monitor the ability to predict on this set of never-seen data. When accuracy on predictions decreases both in the training and validation set, then the network is improving, while when only the accuracy decreases only in the training set, it means that the network is not learning general feature but any further improvements will be tailor suited for that specific training set: this problem is called overfitting. In opposition to the overfitting, there exists the underfitting problem, less common, that consists in not reaching the full potential of the training session, hence not reaching the best accuracy on both training and validation set. Further information on the types of dataset are contained in Subsection 3.1. In the Figure 2.14 it is shown a

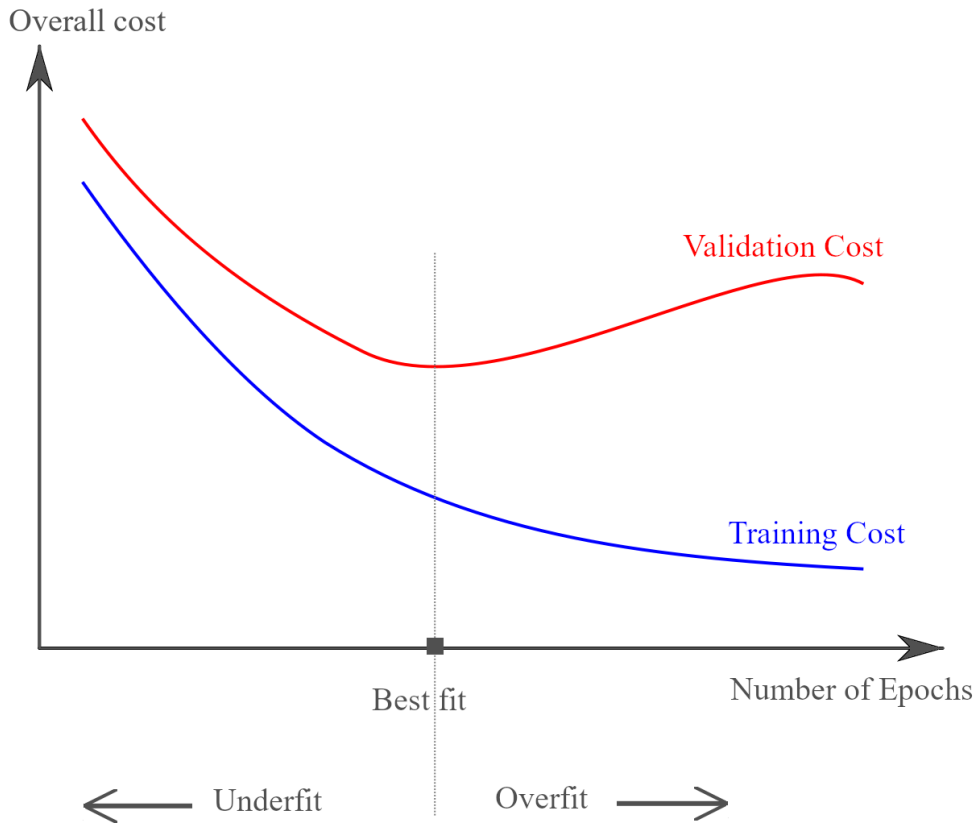


Figure 2.14: An example of a neural network training session. The trend of the cost value for the training set data, typically decreasing, is shown in blue. In red, the cost value for the validation set data. It is noted that from a certain number of epochs onwards the trend of the validation set first begins to become horizontal, then even increases. The corresponding number of epochs is the one to obtain a neural network capable of both performing on training data and on data never seen before. Continuing the training further, we encounter what is called overfitting, and the network gradually loses generalized learning

graphical representation of underfitting and overfitting, while the vertical dashed line indicates the ideal perfect number of epochs. To avoid overfitting, a common efficient technique is to use a dropout layer. The dropout layer is a fictitious layer that is only used in the training session of the neural network. It has an associate deactivation percentage, usually between 30% and 60%, that represents the number of deactivated neurons on the previous layer during a forward propagation. By randomly setting to zero a percentage of neurons equal to the deactivation

percentage, the network is tested for robustness and, in particular, its ability to still predict the correct class also with this functional handicap. This contribution turned out to be a key element to improve accuracy on training data and avoiding overfitting. If on the one side, the expected accuracies with the dropout layer are lower compared to a trained without dropout, the first type of network showed practical advantages compared to the other type. Moreover, the randomic deactivation introduces a further source of unpredictability on the process that leads to network training, contributing to the use of the model as a black-box.

Chapter 3

Inside the Neural Network

In the context of Neural Networks, the Coverage Path Planning problem has always been tackled with Reinforcement Learning techniques applied to the mathematical framework of the Markov Decision Process. If, on the one hand, this approach can model the problem successfully, on the other hand, the training session requires a huge amount of computational resources. This chapter introduces alternative an approach that exploits Supervised Learning techniques on the basis of imitation of an expert system while bypassing common drawbacks such as database availability. A detailed analysis of the architecture of the Artificial Neural Network used in the simulations is described. Particular attention is paid to the input vector, whose choice was crucial for the desired outcomes. Moreover, the whole data pipeline is introduced, from the imitation framework that allows the data collection, to the data partition over the three main data-subsets: training, validation, and testing. Finally, an overview of data augmentation is given, describing the process that led to the achievement of the 54,000 entries present in the current database.

3.1 Dataset

3.1.1 Dataset division

The learning mechanism behind the neural networks relies on the availability of large labeled databases. A label is the desired output associated to an input. Each entry of the database comes in the form (*problem*, *solution*), respectively (*network input*, *network output*): in this sense, databases are usually structured. Given a set of $N_{trng} \in \mathbb{N}$ training examples, labeled data comes in the form $\{(\vec{x}_1, \vec{y}_1), ..., (\vec{x}_N, \vec{y}_N)\}$ such that $\vec{x}_i \equiv \vec{a}^0$ is the input vector of the i-th example and $\vec{y}_i \equiv \vec{\hat{a}}^L$ is its label, with $i = 1, ..., N_{trng}$, and $L =$ number of the last layer. For Supervised Learning applications, the entire dataset should be broken down into

three distinct datasets: test, validation and set test - each one used for a specific development phase of the network, as shown in Table 3.1.

	Update Weights	Labeled Data	Proportion
Training set	Yes	Yes	75%
Validation set	No	Yes	15%
Test set	No	No	10%

Table 3.1: Dataset Division - the first column indicates whether weights, such as biases, are updated after the training phase. The second column indicates whether the data has a known associated/desired output in the database. The last column indicates the percentage of the division with respect to the entire data collection.

The following is a general overview of the three types of datasets:

The training set is the set of data used to train the model: the neural network is cyclically trained over and over again on this same data, and it continues to learn about the features of this data by updating its weights, as explained in Subsection 2.2.4. When the entire dataset is passed forward and backward through the neural network once, an epoch is passed. The neural network aims to increase the accuracy of the prediction on this data after each epoch. However a too high accuracy could turn into overfitting issues and the model could not be able to generalize the prediction on never-seen data: for this purpose, the training set is flanked by a validation set.

The validation set is the set of data used to validate the training process. One of the main advantages of using a neural network is the transposition of the abilities deployed during the learning process in never-seen situations, a biologically inspired ability known as generalized learning. Since the Validation Set is not used to train the neural network, it is as if this set contained never-seen situations. After each epoch, it is possible to cross-check the generalized capabilities of the network by calculating the loss function of the prediction on the validation data. In this way, it is possible to monitor the training session by using two parameters: the first one is the loss calculated on the training set and used for the backpropagation; the second is the loss calculated on the validation set and used to check generalized learning capabilities. When both losses decrease, the accuracy is increased in both seen and never-seen situations. From experience, the training gets to a point such that more training epochs lead to increased accuracy in the training set and a

decreased accuracy in the validation set. This is the optimal number of training epochs: more epochs would lead to reduced generalized learning capabilities while fewer epochs would lead to a non-optimal accuracy.

The Test Set represents the final verification which can decrease the conclusion of the training session. Compared to the training and validation set that are used during the training, the test set is used once the training session is over. Although the learning mechanism is supervised, this set represents an exception since it does not contain any label. The neural network designer runs the neural network on this set and personal checks if the network behaves as desired - in a certain way, the label is implicitly known since the human can discriminate the results. If the predictions of the neural network over the test set meet the requirements, weights and bias are frozen at their value and the network is ready to perform off-design. On the contrary, hyper-parameters are tuned and the training session starts over from the beginning.

3.1.2 Handwritten Digits Recognition

The Handwritten Digits Recognition problem is the equivalent *Hello World* for computer vision and image recognition. This example aims to introduce a simplified problem that benefits from strong parallelism with the coverage path planning. The reference database is the MNIST Database, a large database that collects more than 160000 samples of handwritten digits, as shown in Figure 3.1. Each sample is a 28x28 pixel image in grey-scale and the correspondent label is the number represented in the image. The output classes vary from 0 to 9, ten classes in total. This is a classification problem: given an input image with a handwritten

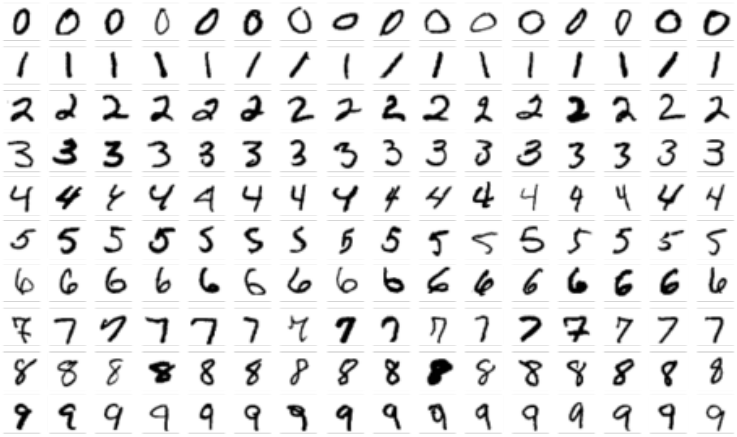


Figure 3.1: MNIST Example

digit, recognize the correspondent class. Each pixel of the image is fed into the input layer of the designed network as its correspondent gray value. The input layer of the networks has $28 \times 28 = 784$ units, while the output layers have 10 units, correspondent to the 10 possible outcomes. The whole dataset can be split using proportion presented in the Table 3.1. Suppose to use 75000 samples as the training set and 15000 samples for the validation set. Each sample is unique and the network will find and learn the pattern that characterizes every single digit. During the training process, all the trainable parameters are tuned by means of the Gradient Descent algorithm described in Subsection 2.2.4, comparing the network prediction to the known and truth label. The network is updated on the same 75000 training sample more and more times, in order to increase the network's accuracy. In the meanwhile, at each iteration, the network is asked to predict also values of the remaining 15000 validation samples for a cross-check. The prediction over the validation samples does not influence the trainable parameters but increases the awareness of the network Designer with respect to the network capability to predict correct labels of "never-seen" handwritten digits, in order to avoid the overfitting of Figure 2.14. In truth, the network has already tried to predict the class of the validation entries, so they are not never-seen data, but since this process did not involve parameters tuning, it is possible to use the same validation test at each training iteration to check the generalization performances of the network. Regardless of the network architecture, not interesting for the purposes of the example, what governs the training is the accuracy over the training dataset and the accuracy over the validation set. If both of them are high, then the network can predict correct labels both on seen and never-seen data. In the coverage path planning problem, the input is related with the state vector, described in Subsection 3.2.1, the output classes are four, correspondent to each possible UAV action - $\{Up, Right, Down, Left\}$, while the database is created with the imitation framework, Subsection 3.3.1.

3.1.3 Occupancy Grids and Maps.

The coverage path planning problem is faced with a 2D approach, with UAVs flying at a fixed height from the terrain below. Thanks to the maps, it is possible both to train the neural network and to test its capabilities in a multitude of different scenarios with non-convex obstacles. In this work, maps are occupancy grids represented by matrices. Each squared element of a matrix, called cell, can be in only one of the following three states: unexplored, explored, obstacle. Unexplored cells turn into explored cells when being visited by at least one agent. Coverage path planning aims to turn all the unexplored cells into visited cells with an overall efficient strategy. As far as the obstacles are concerned, obstacle cells are fixed, cause they are a unique feature of the environment, e.g. buildings, monuments.

There is no unexpected deviation from the original obstacles' shape and position which are known a priori when the simulations start, as the path is planned offline. The only exception stands for the UAV: since each agent senses other agents as obstacle cells to avoid, maps are dynamically updated accounting for the current position of each agent by cyclically setting and unsetting the corresponding cell as an obstacle cell. Since real-life obstacles are of any size shape, a matrix map approach can only approximate their curves, hence, every cell that contains at least a portion of an obstacle is set as an obstacle cell. Moreover, each UAV occupies only one and only one cell per time-step, and its state is set to "obstacle cell". The division into cells also allows treating the matrix of cells as a graph, thus opening up to a multitude of algorithms and properties, as will be seen later. Two macro-families of maps have been developed to train the neural network: the maps of type I have been created to check network capabilities as proof of concepts; the second type is inspired to real urban environments. To demonstrate the feasibility of the application of supervised neural networks in the coverage path planning problem, highly customizable maps were needed. The type I maps, have been created manually. Those maps allowed the creation of ad-hoc scenarios, to deeply understand both the weaknesses and the strengths of the model. Thanks to those manual maps, it has been possible to explore a wide range of neural architectures. This type of maps, of which some examples are shown in Figure 3.2, have been used as Proof of Concept (PoC), the demonstration that neural networks can be used in supervised learning to implement decision making. If, on the one hand,

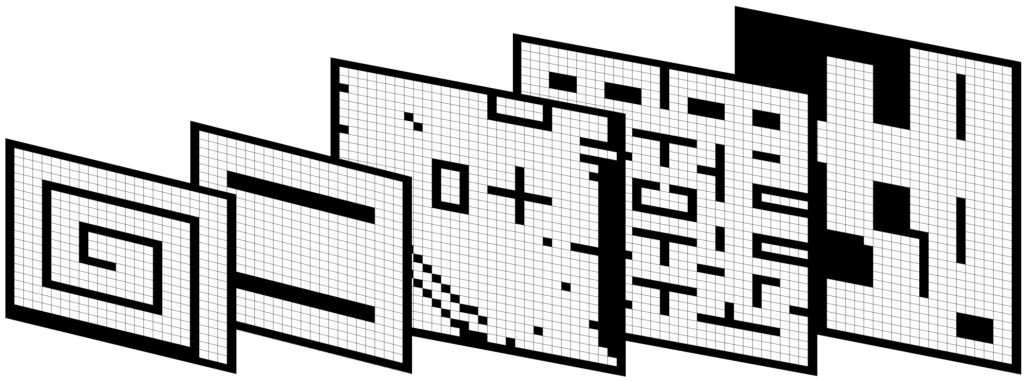


Figure 3.2: Some example maps of type I. The figure shows 5 out of 48 total maps created. The geometry is simple and the obstacles' shape is basic. In black, obstacle cells; in white, unexplored cells. To get an idea about the order of magnitude, the last map from the right is a grid with 38x31 cells.

the maps of type I allow extremely high customization, on the other hand, the effort needed for their creation is too high to create functional maps that can emulate real environments. For this reason, the second generation of maps is created automatically by using real case studies. The target has been the satellite views, thanks to which it has been possible to create occupancy grids. Although UAVs have the ability to fly over buildings and ground obstructions, the maps are created setting the streets as the explorable target region. This leads to a drastic increase in map complexity and a strict focus on urban environments. One real urban map of type II is shown in Figure 3.3. The red rectangle shows a zoomed-in view of the map. Comparing the grid resolution and the obstacle density with the maps of type I, the difference is patently clear. With respect to traditional supervised applications, where the network' prediction is based on a single input (e.g. image recognition: one image, one prediction), the full exploration of the map presupposes that the network guesses the majority of actions that leads to effective exploration, so each map is not to be considered only as a labeled example, but as a resource capable of generating a multitude of different states in various situations that are dynamically updated according to the mechanics of exploration.

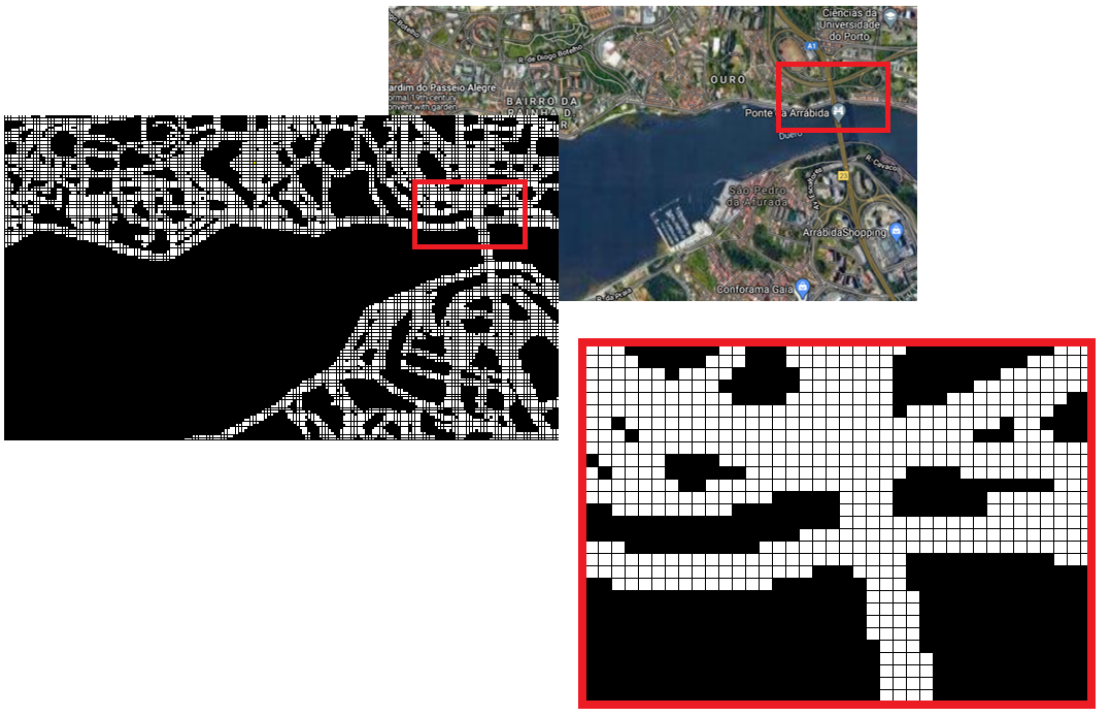


Figure 3.3: This map of type II uses the city of Porto, Portugal, as an example. In black, obstacle cells; in white, unexplored cells. This map is a grid with 133x199 cells, while the red rectangle contains a zoomed view of the grid.

3.2 The Neural Network

3.2.1 Network Input - State and Sensors

On the surface, the handwritten digit example presented in the Subsection 3.1.2 seems to diverge from the coverage path planning. However, the neural network learned how to recognize a digit by using the grayscale of each pixel: the set of pixels can be thought of as the state of the image, hence, the network learned how to map the state to the correspondent digit. In the same way, the state of the UAV can be associated with a determined action to take. Note that the state can contain every kind of information and not only a grayscale value. In this work, the state of the UAV is the set of information used to select the next action to take but does not contain any information about the UAV's dynamics, since it is treated as a point-like object. That information, usually stacked in the feature vector, can be related to the environment or can be embedded in the drone. On the base of its state, each agent selects the best action to take: the entire decision-making process relies then on the state, whose choice became a crucial focal point. The state variables can be chosen arbitrarily, provided that the information is comprehensive with respect to the purpose of the neural network. There is a huge amount of information that can be used as the state: for this reason, the selection of the features is not trivial at all. The state contains the exact number of information useful for the decision-making without redundancies - the contrary would be detrimental to the efficiency of the network. Each state variable (or equally each element of the feature vector) is fed as an external input to the correspondent neuron of the input layer. Once that all information has been provided, the neural network executes a forward propagation and predicts which one of the four is the best choice according to the input state. The final network's architecture uses 376 neurons in the input layer, 376 different features that drive the choice of action. The state variable can be grouped into three distinct categories:

- The first 360 units contain information about its nearby cells, one unit per cell. Each drone has a local view of the map: more specifically, it can sense a squared space of $19 \times 19 = 361$ cells . Since the local view is always centered in the UAV, the central cell does not contain any relevant information. Each unit can assume only one of the following values:
  - unexplored cells, +1;
  - explored cells, 0;
  - obstacle cells, -1.

  The attribution criteria aim to sort in ascending order the attractive contribution of each cell.

- The third group uses 8 units to detect unexplored cells or obstacles out of the field of view. The detectors search these cells inline with the four directions of movement. Four detectors are used to look for unexplored cells while the other four are used to look for obstacle cells. When a detector encounters an unexplored cell, then the correspondent neuron is activated. Each one of the 8 detectors can be in only one of the binary states - on or off.

According to the described input, it is possible to redirect over $3.71 \cdot 10^{175}$ different states with a local limited view. Moreover, each state can be associated with one of the four actions. The state space is so huge that any tabular method, whose purpose is to map state to actions, would fail, according to the limited RAM resources (e.g. Q-learning method). Therefore, the only feasible way is to obtain an approximate solution. Importantly, not all states are of interest since certain situations are not likely to happen in any situation - consider, for example, of all those states in which the physics of simulation would lead to a state with inconsistent information.

3.2.2 Neural Network Design

The Neural Network architecture is defined by hyper-parameters. Unlike the trainable parameters, which are updated during the training session with the back-propagation, hyper-parameters define the structure of the neural network and must be chosen by design. The neural network's behaviour will be strongly influenced by hyper-parameters as they determine its intrinsic functioning. Below is a list of some hyper-parameters:

- Number of neurons per layer.
- Number of hidden layers;
- Activation function per layer.
- Loss function.
- Number of epochs.
- Dropout level.
- Batch size.
- Weight initialization.
- Input vector.
- Output vector.
- Optimization algorithm.

- Learning rate.

Some hyper-parameters have already introduced described in Chapter 2. The following overview about design parameters is referred to Standard Dense ANNs.

The number of neurons per layer is one of the most challenging design issues. For the input and the output layer, this problem does not occur; with respect to the number of input neurons, this parameter is completely and uniquely determined once the shape of the training data is known, since each relevant feature is fed as input to the network. On the other side, the number of output neurons is determined by the nature of the problem. The problem arises when it comes to hidden layers. The design is a mixture of experimentation and experience, intuition and inspiration from other well-performing models. Each network has to be tailor suited for the specif problem and it is not possible to generalize the design process. In[15] *"Effect of number of neurons and layers in an artificial neural network"*, the effect of hidden units variation is highlighted by experimentation.

Theoretically, artificial neural networks with an only one-hidden layer with a non-linear activation function are universal function approximators, as shown in Subsection 2.1.5. However, the number of trainable parameters of those networks might be much larger if compared with multiple-hidden layered. By adding more hidden layers or more units per layer, the number of trainable parameters increases anyway, but it is possible from the model to fit more complex functions. It is possible to thinks of each layer as an abstraction level. With an image recognition parallelism, if the input layer contains the set of pixels of the image, the first hidden layer could detect circles, the second hidden layer might detect edges, the third hidden layer might detect human faces, and so on toward more complex patterns recognition.

The activation function, Subsection 2.1.4, is usually defined per layer, hence it is the same for each unit of a layer. It is fundamental as it defines the non-linear behavior of the model, computing the second step of calculation showed in Equation 2.1. Although linear activation functions exist, the greatest majority of models use non-linear activation functions, as otherwise, the final result would be merely a linear combination of the input. Moreover, the universal approximation theorem demonstrated the function approximation capabilities of neural networks for a non-linear activation function.

The loss function is used to evaluate a candidate solution for the target problem. The set of all trainable parameters are iteratively updated during the training session, seeking among the hypothesis space the best function that fits the problem. The loss function used in Supervised Learning compares the target or labeled output with the network's prediction. For that reason, there are infinite functions that can compare these two vectors. Among them, some loss functions are preferred,

as explained in Subsection 2.2.3, since they have better properties in terms of convexity, a fundamental feature useful for the gradient descent algorithm.

The number of training epochs is the discriminating factor between an over-trained network, which would lead to overfitting, and an under-trained network, which would lead to underfitting. This number must be balanced to obtain a network with high prediction accuracy, while maintaining the generalization capabilities on the data, as shown in Figure 2.14.

To avoid overfitting, a common efficient technique is to use a dropout layer. The dropout layer is a fictitious layer that is only used in the training session of the neural network. It has an associate deactivation percentage, usually between 30% and 60%, that represents the number of deactivated neurons on the previous layer during forward propagation. By randomly set to zero a percentage of neurons equal to the deactivation percentage, the network is tested for robustness and, in particular, its ability to still predict the correct class also with this functional handicap. This contribution turned out to be a key element to improve accuracy on training data and avoiding overfitting.

The Batch size determines the number of training samples used in the training process with the Mini Batch Gradient Descent, a trade-off method between the Batch Learning and the Stochastic Gradient Descent described in Subsection 2.2.4. Instead of updating weight and biases punctually for each training data or updating for the whole dataset, the entire set of data is partitioned in mini-batches. The number of entries contained in each mini-batch is defined by a hyper-parameter called batch size. Usually, this number is equal to a pow of 2, for the technical reason when the computation is carried out on a Graphical Processing Unit (GPU). If the number of total entries does not match with a multiple of the batch size, then the remaining entries form a separate mini-batch. With this method, the algorithm noise is reduced and computational issues do not arise, since the mini-batches are smaller than the batch.

Weight and biases initialization aims to prevent layer activation outputs from exploding or vanishing. This occurs especially in deep neural networks, where the loss of gradient might lead to slow or null backward training. To overcome this issue, trainable parameters can be initialized with several techniques that allow smoother training.

The state vector has been described in Subsection 3.2.1. It contains the exact number of information useful for the decision-making without redundancies - the contrary would be detrimental to the efficiency of the network. Each state component is fed as an external input to the correspondent neuron of the input layer. Once that all information has been provided, the neural network executes a forward propagation and predicts which one of the four is the best choice according

to the input state.

The output vector contains the number of classes and depends only on the network's main purpose. This information should be clear from the start of the design process in order to select the best design choices in accordance with a fair estimate of the complexity of the model. Neural networks with only one output unit in the last layer are called also binary classification.

The optimization algorithm selects the best weight-set in the hypothesis space of functions, with regard to some criterion based on the loss function. In the Machine Learning universe, the optimization algorithm is what makes the artificial neural network learn.

Kevin P. Murphy in[16]"Machine Learning: A Probabilistic Perspective" defines the learning rate as a hyper-parameter that determines the step size at each iteration while moving toward a minimum of a loss function in an optimization algorithm, as already discussed in Subsection 2.2.4. Depending on the chosen optimization algorithm, often the learning rate has to be configured explicitly, but it is the algorithm itself that dictates the step .

The idea is to keep the neural network architecture as simple as possible while aiming for high accuracy rates. Complex architectures would weigh down both forward and backward propagation but could have better-generalized performances. There is not a rule of thumb for the design of the neural network and each problem must treat with its singular features. The design process is usually tackled manually with a try-and-error approach. The result is a trade-off architecture designed ad-hoc according to the network's purpose.

3.2.3 Neural Network Architecture

A visual representation of the designed neural network is shown in Figure 3.4. For large and complex networks, as this one, the traditional representation with neurons and edges became cumbersome, and it is replaced with a compact and descriptive notation. The input layer uses 376 neurons according to the received state sensed by the UAV. This state is inclusive of the local view of the map, the two-stage memory, and the cell detector, as described in Subsection 3.2.1. A neural network with only one hidden layer is also called a shallow network, in contrast with deep networks which have more complex structures. For classification applications, shallow networks are sufficient to obtain high performance. The hidden layer uses a ReLU activation function: ReLU is the most commonly used function as it speeds up the training process and because of its simplicity. As anticipated, ReLU is defined as:

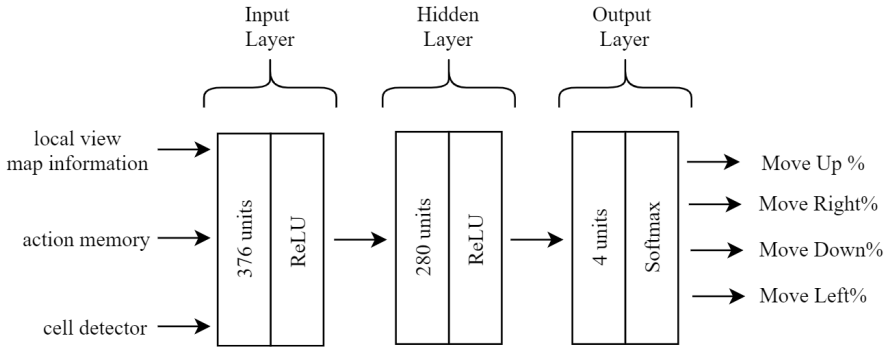
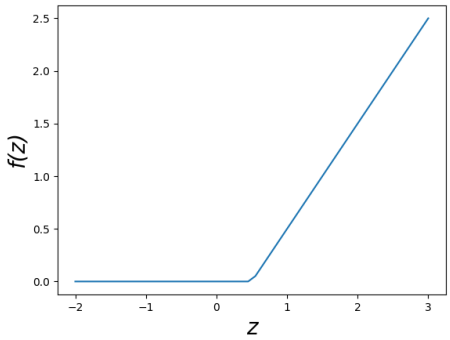


Figure 3.4: Neural Network's architecture. From left to right: the feature vector (state) is fed in the input layer. Then it gets propagated forward in the hidden layer and finally reaches the output layer, where the softmax function normalizes the probability distributions. For each of the four actions, it is possible to read the percentage of prediction of the network and use it to in the algorithm to move the correspondent agent.



$$f(z) = \max(0, z + b)$$

Figure 3.5: Rectifier Linear Unit - ReLU function with a bias example of 0.5 - reference to the Subsection 2.1.4

This function does not saturate for the large positive value of the weighted sum of inputs and creates nonlinear relation between two consecutive layers. The last output layer does not use the ReLU function but uses the softmax. Since the results of the output layer can contain both very high positive values or negative ones, it is useful to normalize the results in some way. The softmax function $\sigma : \mathbb{R}^{N_c} \to \mathbb{R}^{N_c}$, with $N_c \in \mathbb{N}$ the number of output classes, is a normalized exponential function that normalize the output vector in such a way that the four output values are normalized into a probability distribution. In this way, it is possible to read directly the network's percentage prediction over the four actions. Let $\mathbf{v}$ be the output

vector prior to the normalization, and v_i its *i-th* element. The softmax function is defined as:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \quad \text{for } i = 1, ..., K \tag{3.1}$$

The final vector has two important properties. Each element $\sigma(z_i)$ has a value

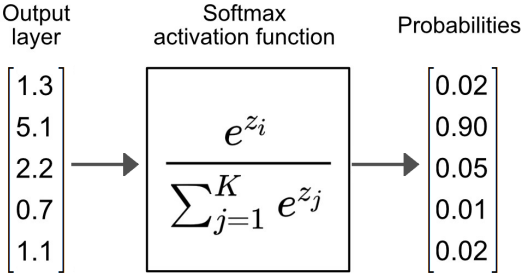


Figure 3.6: Softmax application example - each component of the input vector is transformed into an exponential probability distribution using Equation 3.1.

contained in $[0, 1]$, expressed as a probability. Moreover, the sum of all its components is equal to 1 or 100%, as shown in Figure 3.6. In this way, it is possible to read directly which one of the four actions has been predicted by the network and its relative percentage of choice. Moreover, for problems like the coverage path planning for multi-UAV, there is not only a single solution, but very divergent trajectories could lead both to acceptable results, in accordance with the multitude of possible scenarios. For this reason, a UAV that is using the neural network for decision-making could predict a similar probability distribution on distinct actions. The total number of trainable parameters, including both weight and biases, is 106˙684. The summary of the model given by Keras is shown in Figure 3.5. As for the weight initialization of the model, Keras uses the Glorot Uniform initialization as the default kernel, also known as Xavier Uniform. The state-space presented in the Subsection 3.2.1 showed the enormous set of possible states. The Neural Network searches for patterns between state and action pairs among the labeled training dataset, and tune the neural network to execute a good classification. The evaluation metric for the classification is accuracy, defined as the percentage of correct predictions over the total number of labeled data. In Figure 3.8, it is possible to visualize the increased accuracy over each epoch of training. The accuracy is quite high since from the first epoch, 80.67%, and increases up to 92.01% at the end of the training. The high accuracy achieved since the first epoch is not a coincidence due to the initialization of the weights, but rather it is attributable to learning through mini-batches. Each mini-batch contains 32 elements, equal to 2^5, for a total of 1692 batches, while the last mini-batch contains the remaining

28 elements coming from the division between 54000 and 32. It is noted that the algorithm implements the gradient descent with mini-batches as the learning curve is not monotonous increasing but is subject to the characteristic fluctuations of an imperfect descent of the gradient typical of mini-batch learning. More over, the training process is not deterministic, and a new training session would lead to another accuracy of the network, around 92%. The time required for the training session has been of 122 [seconds] on a commercial laptop with an Intel(R) Core(TM) i5-4210U, 1.70GHz-2.40GHz CPU, and 4GB of RAM. The fast training time is one of the biggest achievements in using Supervised Imitation Learning.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 280)               105560
_________________________________________________________________
dense_1 (Dense)              (None, 4)                 1124
=================================================================
Total params: 106,684
Trainable params: 106,684
Non-trainable params: 0
_________________________________________________________________
```

Figure 3.7: Training results plotted by TensorFlow.Keras by calling the model summary.

3.3 Expert System

3.3.1 Imitation Learning

(POV framework ...) Although the basic technical idea of artificial neural networks has been around for decades, the reason why are the neural networks taking off only just now lies in the availability of Big Data. The increased computing resources, the digitization of industries, the Internet of Things (IoT) paradigm, the availability of large databases, the development of ever smaller and more performing electronic components, are contingent factors that have made the advent of Big Data possible, and, with them, also the need of tools in order to fully exploit these resources. In this work, a neural network searches and learn cross-data patterns in a database containing over 54000 entries. What the network learns is a planning strategy in order to carry out the coverage path planning in never-seen urban maps. To achieve this result the dataset contains past experiences collected from an expert system, which can be both a human or a computer algorithm. The experiences'

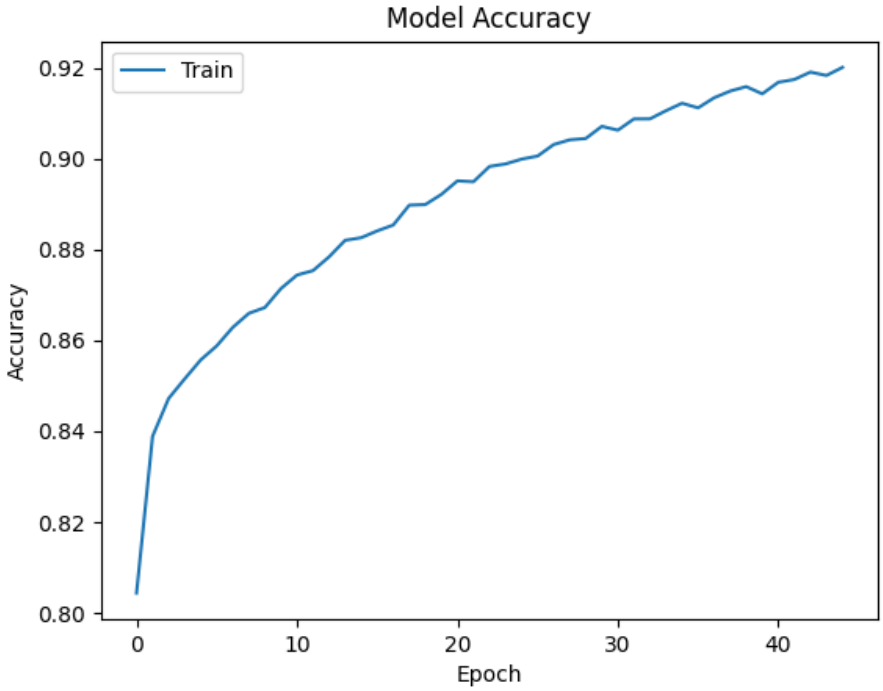


Figure 3.8: Increasing neural network's accuracy over each training epoch.

collection passes through an imitation framework, whose purpose is to offer a graphical view to the human and register its manually inserted actions. Each action is paired with the correspondent UAV's state and form the basis for the creation of the dataset. A fundamental property of the framework is its inductive approach and simplicity. Although the final simulation will involve a fleet of UAVs, experiences are collected for the single-UAV case. In this way, the human can guide effectively the simulation while providing a real-time critic to its strategy. Since the human can judge the quality of the exploration, the self-critic mechanism that he establishes during the experience collection is exactly what the neural network aims to imitate. At each time of this offline simulation, the human can observe the UAV's state and select manually what action fits best in that situation; the framework stores the state-action paired information and waits for the next action. A fundamental feature of the framework is that the human expert must select the best action to take with only the information provided by the state of the UAV, hence with a limited view on the map and a point of view like interface. This approach allows the human expert to have further feedback on the neural network design and should be considered as a best practice for the design of state: in this way it is possible to design a feature vector that includes the exact amount of

information needed. By using Human Priors dataset, the learning process reached a 92.01% of accuracy over a database with 54131 labeled entries. After establishing the quality of the training process, a key element of the UAVs' performance relies upon the strategy used by the human expert to create the database, hence to train the network. Three guidelines drive the strategy:

- Minimize the expected discounted return.
- Reduce the number of turning maneuvers.
- Avoid going twice through the same cell.

The expected discounted return is defined as:

$$G_t = \sum_{t=0}^{\infty} \gamma^{t-1} R_t \tag{3.2}$$

, where $\gamma \in [0, 1)$ is the discount factor, t is the discrete time-step, R_t is the reward at time t, equal to the number of explored cells at time-step t. Since $\gamma < 1$, future rewards decay in time, and strategy is encouraged. If γ was equal to 1, the UAVs would be free to explore the map without the time constraint, as long as they carry out a full exploration of all unexplored cells. By discounting future rewards, it is possible to select strategies able to cover the whole target area with the minimum number of timesteps, hence distinct good strategies from bad ones. The number of turning maneuvers is another relevant evaluation metric since the UAV's energetic payload is limited. Finally, not using the same cell twice should reduce the number of redundant movements. Taking into account all these elements while manually driving a multi-agent system in a complex environment is unsustainable for a human. For this reason, the experiences are collected in a single-agent and single sub-area simulation. Noticed that a human expert can't follow strictly the three guidelines, but the final strategy will result in a mix of all of them. The final result of the training is a network able to choose the best action to take at each time step and for each UAV, imitating the strategy of the expert system stored in the database. With an accuracy of over 92%, the final result is highly reliable and, if the network would decide to take actions that deviate from the standard strategy, the outcome would be aligned with the learned strategy in any case, thanks to the generalized learning capabilities. The Imitation Learning technique overcomes the problem of pre-existing large labeled databases, while encourages its creation, as long as there is a capable expert system able to guide the solution. When it comes to Supervised Learning, the strategy of the expert system can be imitated at best, without any variation or improvement. Conversely, Reinforcement Learning is able to use the guided solution as a basis of further learning steps, and explore new solutions, at the cost, however, of intensive use of computational resources.

3.4 Data Pipeline

Before using the labeled dataset, each entry of the database is elaborated through two main processes: augmentation and cleaning. The term pipeline indicates serial data processing, where the output of one elaboration is the input of the next one: the dataset is firstly augmented and then cleared.

Data augmentation allows the creation of new data starting from existing. Since the number of collected experiences of the dataset is limited compared to the huge set of possible states, the neural network learns on a small subset called training dataset and then predicts never-seen data by using generalized learning. The resulting

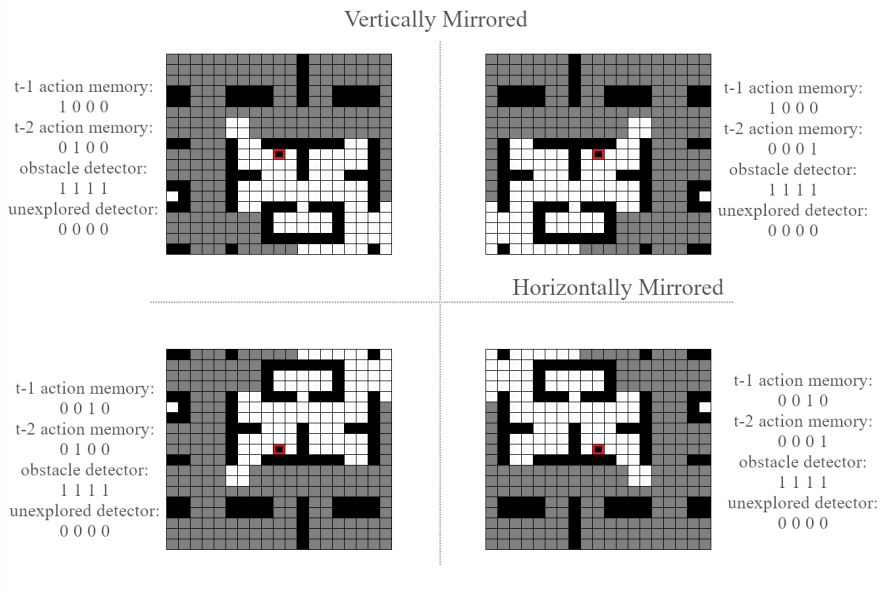


Figure 3.9: First four times data augmentation through mirroring process - not only map information are mirrored, but also action memory and cell detectors.

solution is approximated since, for the majority of the use cases, the neural network will predict solutions over never-seen data. The data augmentation allows better training by exploring a broader segment of states. This process multiplies the number of collected experiences by eight-time. Starting from a dataset of 6906 entries, the augmentation led to an increase of entries up to 55248. Consider a single entry of the labeled dataset. The entry is composed of two paired information: the state, or feature vector, and the corresponding action, or label. Since the network will learn to predict that label when fed with that state, there is no guarantee that the network is able to generalize to a mirrored situation. By rotating and flipping the state, it is possible to create new data from the existing one. Usually, those kinds of operations are associated with image elaboration. Actually, the

first 360 elements of the state, Subsection 3.2.1, can be treated as an image since they contain relevant information about the nearby local view. Importantly, when flipping or mirroring a state, the whole state must be accounted for, as well as the corresponding action. An example of data augmentation is shown in Figure 3.9. The state is mirrored three times: vertically, horizontally, and diagonally. This leads to a fourfold increase in experiences. By rotating the original state by $\pi/2$, it is possible to mirror again on the vertical, horizontal, and diagonal. The whole process creates seven other experiences starting from the original.

While data augmentation leads to a drastic increase in the number of labeled experiences, data cleaning defines the criteria by which some of the augmented entries are deleted. Data augmentation often leads to an incongruous situation: if several entries with different labels correspond to the same state, learning is undermined. Database cleaning removes both redundancies and incongruous states, resulting in increased consistency and training accuracy. Data cleaning deleted 1117 entries from the dataset, leading to a final augmented and cleaned dataset with 54131 collected experiences, as shown in Figure 3.10

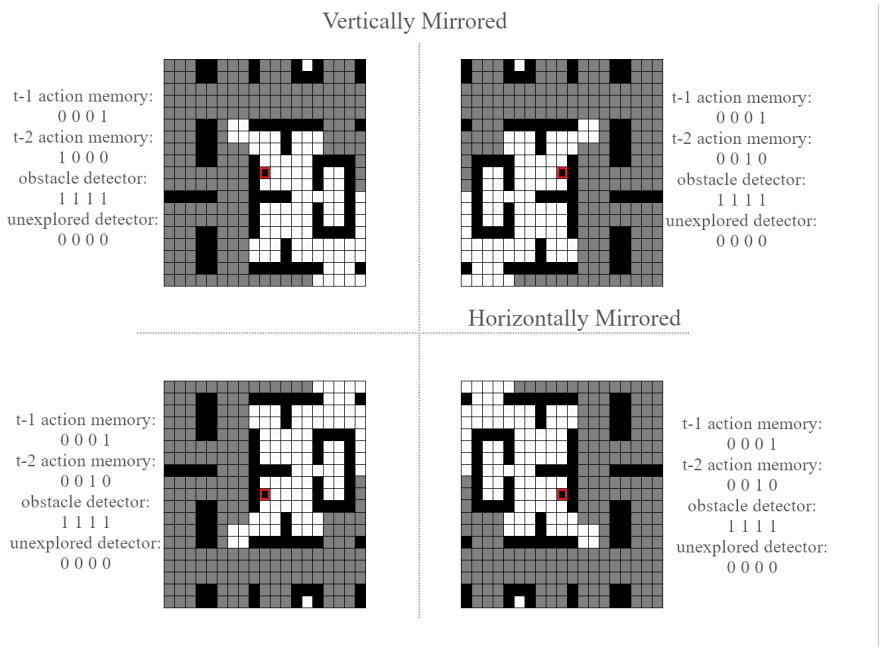


Figure 3.10: Second four times data augmentation through mirroring process - the state of Figure 3.9 is rotated by $\pi/2$ and mirrored again.

Chapter 4

Algorithm

The algorithm relies on a mixed-use of decentralized Artificial Neural Networks which confers elementary cognitive skills to each UAV, and a modified version of the famous A* pathfinder. This chapter introduces the Explorative A* pathfinder and how its combined use with neural networks can approach the Coverage Path Planning problem. The whole logical flowchart is presented, highlighting both the potential and the criticalities of the algorithm, presenting an innovative approach that aims not only at the cooperation of UAVs but also at their collaboration. Moreover, it is presented the algorithm designed to divide the target explorable area into sub-zones of exploration, and the use of the Lloyd Algorithm and the Voronoi Diagrams involved in this process.

4.1 Assumptions

In the proposed work, the simulation is discrete in time and space. The space discretization uses an occupancy grid over a matrix. Each cell of the matrix can be in only one of the following states: unexplored, explored, obstacle. Since maps are inspired by real scenarios, real obstacles are approximated due to the nature of the matrix map: if even part of the obstacle lies inside a cell, then the whole cell is considered as an obstacle cell. The obstacle cells' distribution and shape are both known a priori. The path is planned offline as obstacles are fixed. The fleet consists of a variable number of agents with the same technical capabilities - each agent occupies always one and only one cell on the map, and the correspondent cell is an obstacle cell. Consider one single UAV, equipped with a camera pointing downwards to capture the scene underneath. The camera has a square footprint, as shown in Figure 4.1. If H [m] is the height from the ground and FOV is the camera Field Of View [deg], it's possible to obtain the Side S [m] of the camera

footprint, by calculating:

$$S = 2H \cdot tan\left(\frac{FOV}{2}\right) \tag{4.1}$$

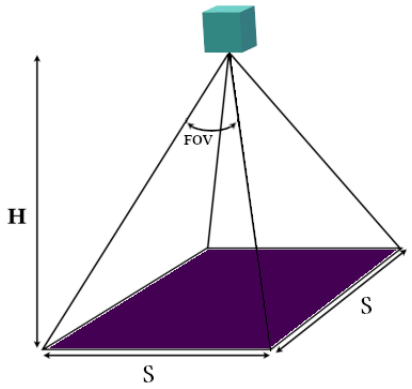


Figure 4.1: UAV's camera cone of visibility: in purple, according to the Field Of View, the camera ground footprint; in teal, a single UAV.

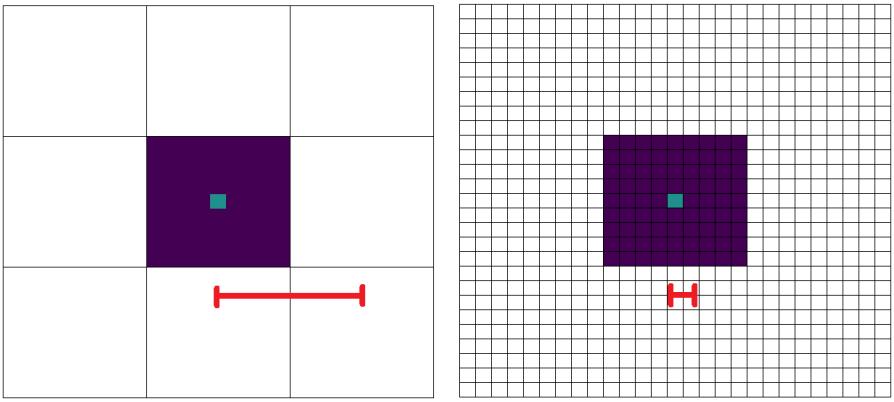


Figure 4.2: Top View: (a) Traditional approaches' space discretization (b) Space discretization in this thesis work.

At each time-step, each agent explores all nearby cells in the line of sight with its camera FOV: the camera, always perpendicular to the ground, has a squared footprint with side $S \in \mathbb{R}$. Furthermore, without loss of generality, it is assumed that the movement of the agent is defined with four directions, specifically $\{Up, Right, Down, Left\}$ in the adjacent cells. Since no rotation action is allowed, the UAVs are always heading Up. The FOV takes 9×9 cells and is always centered in the UAV. Compared to the traditional approach, where each action would lead

to a displacement equal to S, in this work the displacement at each time-step is equal to $S/9$, equivalent to 1 *cell*, as shown in Fig 4.2. The height of flight from terrain is fixed so that the problem can be treated in two dimensions. Finally, it is assumed that all unexplored cells can be reached from at least one agent.

4.2 Algorithm Overview

A trained Artificial Neural Network is the core of decision-making. It acts as the artificial brain of the UAV and decides what action to take. The input of the ANN is the locally sensed state of a UAV. The sensed state is then forward propagated through all the hidden layers until it reaches the output layer, where one of the four classes of action is selected. Despite the number of agents A being variable, the same network is used for every UAV. This means that all agents share the same weight set hence the same way of making decisions: at each time step, the network performs several forwarding propagations, up to a maximum of one per agent. Since each agent has a limited view of nearby cells, the ANN is not always used, as described below. The discriminating factor of this choice is the state. The state is a feature vector described in the Subsection 3.2.1. The first 360 entries contain information about the local view of the UAV. This local view is a 19x19 matrix, reshaped into a column vector. Since the local view is always centered on the UAV, the central cell does not contain relevant information. By counting the number of unexplored cells in the local view, the agent decides whether to use ANN or the A*/Explorative A*. If the number of unexplored cells inside the local view is zero, the agent does not have enough information on unexplored areas, and any action taken will not lead to a strategy: therefore Explorative A* is chosen. As described in the following Section 4.4, Explorative A* is used to aid the UAV in finding and reaching an unexplored area when no unexplored cells lie inside the local view. The unexplored areas of the map are not explored chaotically, but the target space is divided into several sub-areas equal to the number of drones, and each UAV must explore its assigned area since can to sense/view only its unexplored cells. Once a UAV completely explores its assigned area, its sight capabilities increase as it becomes able to sense the overall status of the map, hence exploring also other sub-areas. Increased sight capabilities should not be confused with a wider local sight, rather than when any unexplored cell of any sub-zone lies inside its local view, the UAV can actually sense its unexplored cell. This mechanism enhances collaboration and synergy between parts, in addition to cooperation. UAVs still can't communicate with each other, but several controls are implemented to increase trajectories' efficiency.

4.3 Zone creation

The target explorable space is decomposed into sub-regions of exploration. The idea is to assign each UAV to a certain sub-region of exploration: this allows achieving a better UAVs spatial distribution while enhancing coverage performance. The decomposition is guided by Lloyd's algorithm, an unsupervised algorithm also known as K-means. The K-means clustering algorithm attempts to split a given anonymous data set into a fixed number K of clusters or subsets. In this work, the anonymous data is the set of unexplored cells in the map that should be grouped in a number of subsets equal to the UAVs' number. Let $X = x_1, x_2, ..., x_U$ be the set of unexplored cells, each one defined by its coordinates in term of row and column of the matrix map, with $U \in N$ the number of the target cells to be explored. Let K be the number of clusters in which the original set will be partitioned, such that $1 < K < U$. "A partition of a set is a grouping of its elements into non-empty subsets, in such a way that every element is included in exactly one subset." Let $P = p_1, p_2, ..., p_K$ be the set of partitions such that:

- $\bigcup_{i=1}^{K} p_i = X$
- $p_i \cap p_j = \emptyset, \qquad$ with $i \neq j$
- $\emptyset \subset p_i \subset X \qquad$ with $i = 1, ..., K$

The number of clusters K is equal to the UAVs' number. Partitions can be represented by a matrix $V \in \mathbb{Z}^{U \times K}$, where a generic element $v_{i,j}$ is set to 1 if the *j-th* unexplored cell belongs to the *i-th* cluster, otherwise is set to 0. An unexplored cells belongs to one and only one cluster per iteration. Each cluster has an associated centroid, an object that represents the average properties of a cluster in terms of coordinates. Let $O = o_1, o_2, ..., o_K$ be the set of centroids. Each centroid moves according to the virtual center of gravity of the correspondent cluster. The purpose of the algorithm is to move iteratively the centroids' position according to the value function until convergence is achieved. The value function is described as:

$$F = \sum_{i}^{U} \sum_{j}^{K} \sqrt{x_i^2 - o_j^2} \cdot v_{i,j} = \sum_{i}^{U} \sum_{j}^{K} D(x_i, o_j) \cdot v_{i,j} \tag{4.2}$$

The value function computes the Euclidean distance $D(x_i, o_J)$ in the 2D space between the unexplored cell x_i and its assigned centroid o_j. Each iteration aims to decrease the value of the value function: when the value function comes into a minimum value, then convergence is achieved and centroids' position is reached an equilibrium point. After each iteration, the partition matrix is updated and each

unexplored cell is assigned to the closest centroid, therefore:

$$v_{i,j} = \begin{cases} 1, & \text{if } D(o_i, x_j) = \arg\min D(o_z, x_j), \\ & \quad \text{with } z = 1, ..., K,\ z \neq i \\ 0, & \text{otherwise} \end{cases} \tag{4.3}$$

At the end of each iteration, the *i-th* row of the matrix V contains as many 1 as there are unexplored cells associated with the centroid o_i: let the this number be Q_i, to indicate the quantity of assigned cells to the centroid o_i. Each centroid moves toward the centre of its assigned cells by means of the average position averaged over the number of cells:

$$o_j = \frac{1}{Q} \cdot \sum_{i=1}^{U} x_i \cdot v_{i,j}, \quad \text{with } j = 1, ..., K \tag{4.4}$$

A best practice is not to use the unexplored cells as the target space to split, since U, the number of elements in X, can be very large and the computational time may be unnecessarily high. Instead, it is possible to approximate the space of the unexplored cell by randomly spawning a number of $N_p << U$ points over the target space, and then use this set of points as a representative set of unexplored cells. The procedure can be summarized in the following pseudo-code:

Algorithm 1 K-Means clustering algorithm pseudo-code.

```
Place points x_1, ..., x_P over unexplored cells, randomly
Place centroids o_1, ..., o_K in each UAV's starting position
while not convergence do
    for each point x_i, i = 1, ..., P do
        find nearest centroid o_j                    ▷ arg min D(x_i, o_j)
        assign the point x_i to cluster o_j          ▷ Equation 4.3
    end for
    for each centroid o_j, j = 1, ..., K do
        count the number Q_j of its assigned points x_assign.,
        compute mean position of all the assigned point x_a,   ▷ Equation 4.4
        move c_j to that new mean position:
    end for
    Stop when none of the cluster assignment change
end while
from o_1, ..., o_K, calculate Voronoi Diagram and divide the target area.
```

Once the position of all centroids does not change from one iteration to another or is contained within a threshold value, convergence is reached. Starting from the

centroids' final position, the Voronoi Diagram is used to achieve the partition of the unexplored cells and create the sub-maps. Consider as an example the case study shown in Figures from 4.3. The K-means algorithm and some of its iteration are highlighted from Figure 4.4 to Figure 4.8.

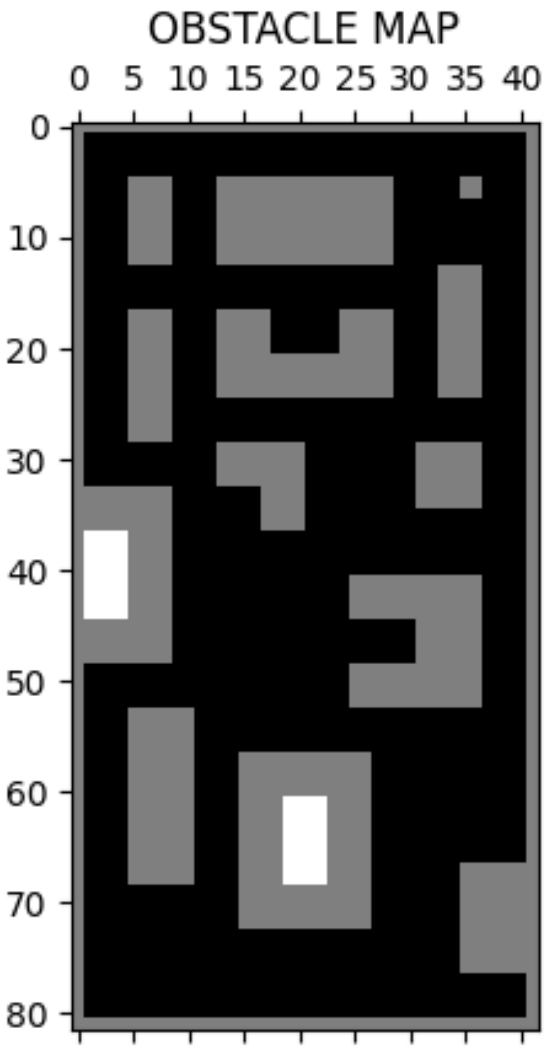


Figure 4.3: Example case study map - In black, the target area composed by "unexplored" cells. In gray, "obstacle" cells. In white, unreachable cells, set as "explored" according with the assumptions described in Section 4.1. The algorithm will attempt to create K partition of the set of unexplored cells.

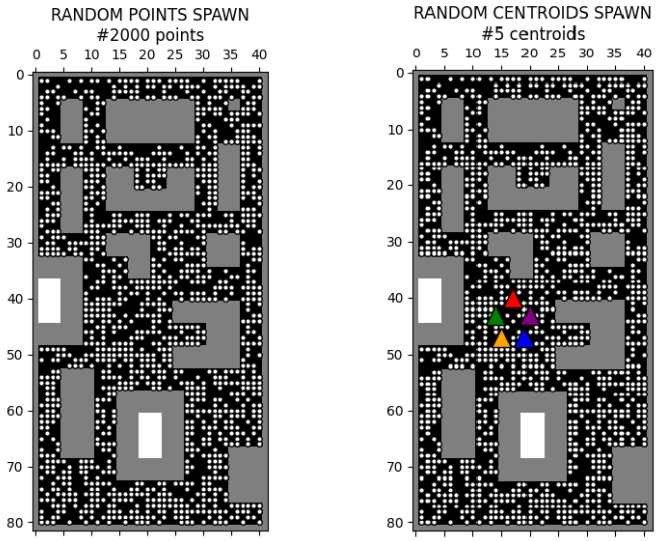


Figure 4.4: In the left figure, 2000 white points are spread over the unexplored target area. In the right figure, five centroids, represented as triangles, are positioned in each UAV's starting position.

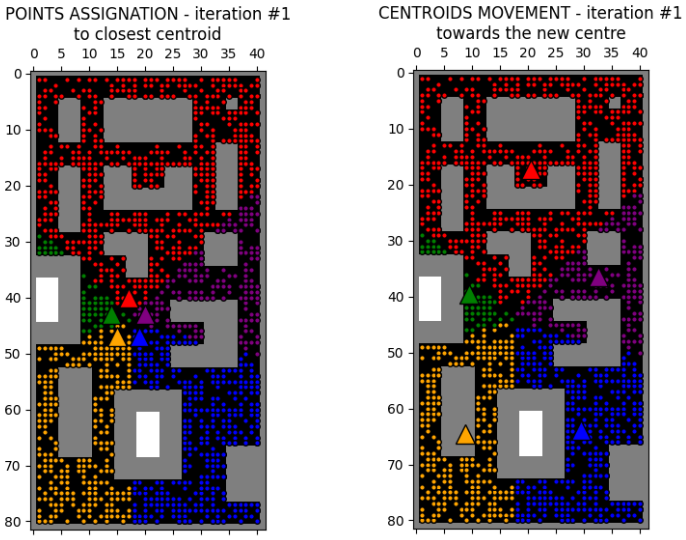


Figure 4.5: In the left figure, each point is assigned to the closest centroid. In the right figure, each centroid is moved toward the averaged points' position. This process represents one complete iteration of the K-means algorithm.

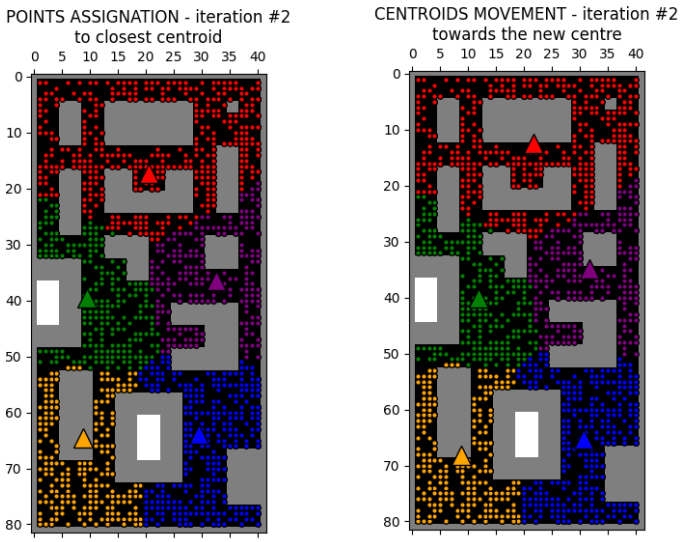


Figure 4.6: Second iteration of the K-means. In the left figure, each point is re-assigned to the closest centroid. In the right figure, each centroid is moved toward the averaged points' position

.

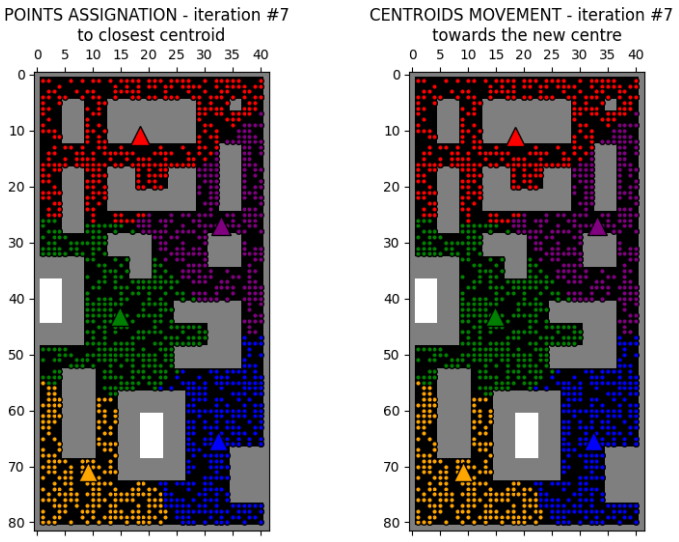


Figure 4.7: Seventh and last iteration of the K-means. In the left figure, each point is assigned to the closest centroid. In the right figure, each centroid is moved toward the averaged points' position.

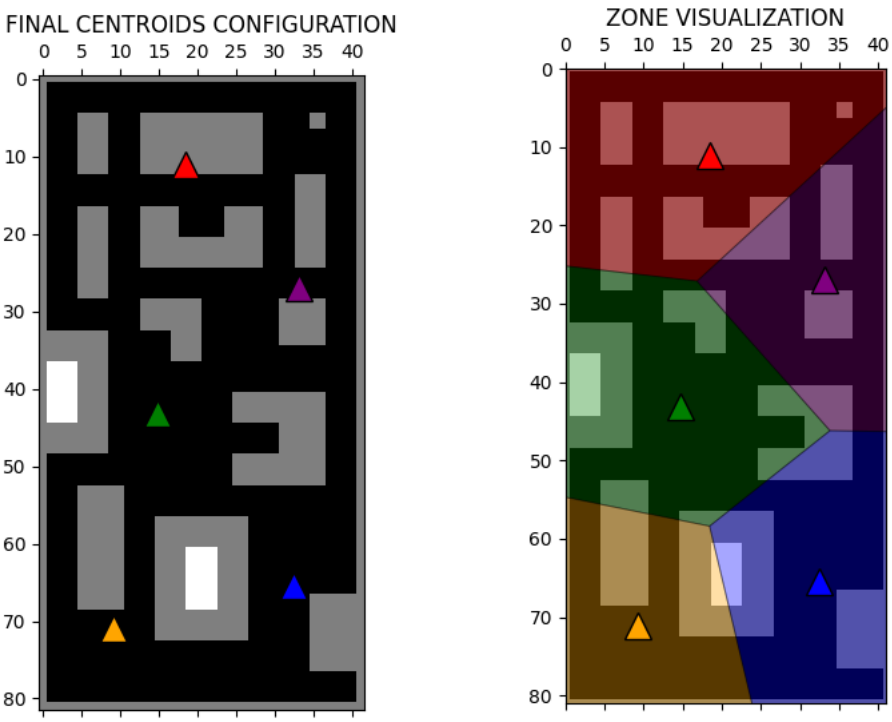


Figure 4.8: In the left figure, the centroids' final position cleared from the points. In the right figure, the correspondent Voronoi tessellation.

The final centroid position is obtained in seven iterations of the K-means algorithm, as shown in Figure 4.7. Using the final position of K centroids as seeds, a Voronoi tessellation splits the map into K zones as shown in Figure **??**. Thanks to this algorithm it is possible to create a partition of a complex distributed target set. It is important to underline that the partition is not unique, nor is the equilibrium configuration of the centroids, but this depends on the initial position of the centroids. Instead of placing them randomly on the map, it was considered more useful to initialize them on the starting cell of the corresponding UAV. In Figure 4.11 it is shown the final partition, according to the matrix nature of the map. In the example, each of the five UAVs will be assigned to one unique zone, and its goal will be to fully explore it. Note that the algorithm is set as "unexplored" only the target area of the correspondent map. Although the non-target area is set as "explored", it is a fictitious operation, since the exploring UAV must give priority to its partitioned zone. Explorative A*, Section 4.4 and the Flowchart, 4.5, will describe in detail how collaboration is enhanced.

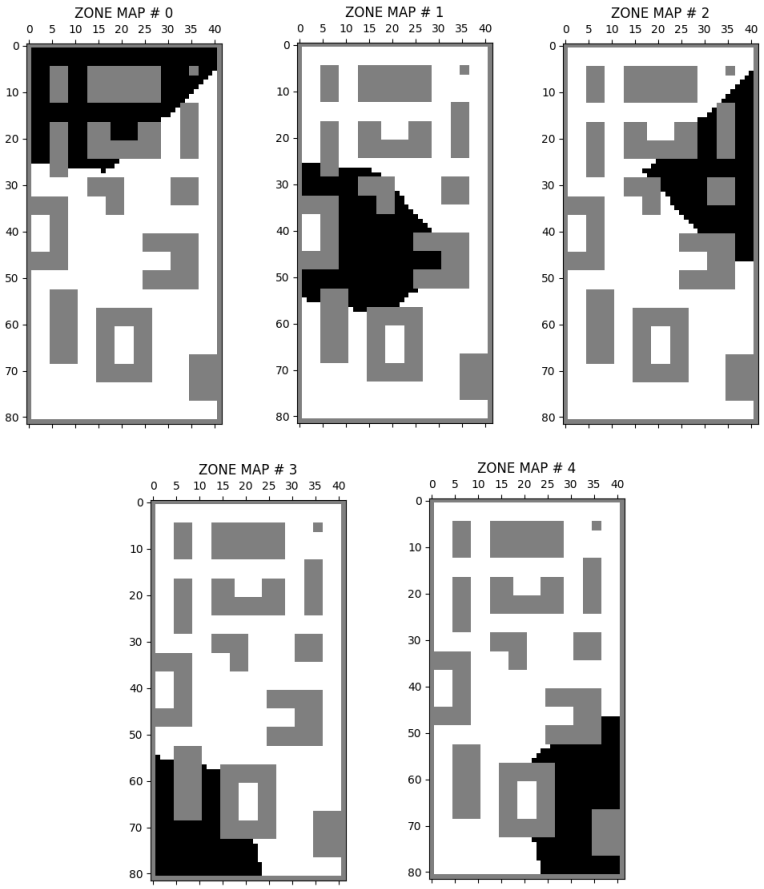


Figure 4.9: Zone visualization - every single zone is highlighted, showing in black the correspondent target area, and in white the non-target cells. In this example, five UAVs will explore the map, each one assigned to a particular zone.

4.4 Path-finding algorithm

The local limited sight of each UAV on the map often leads to situations where no unexplored point lies inside their local view and for which the neural, in which case and for which the use of neural network approach is not useful. This issue occurs especially towards the end of the exploration when few cells are missing. In these cases, an explicit path-finder is used to aid the UAV to find the nearest interesting cell and reach it planning an optimal path. More specifically, the two path-finding algorithms are Explorative A* and, in case of non-convergence, traditional A*. The algorithm is articulated in three main steps:

Step 1: Goal cell – enlarge the local view in order to detect the nearest unexplored point and set it as the goal.

Step 2: Explorative A* – check if Explorative A* convergences from the UAV position toward the goal.

Step 3: A* – if Explorative A* did not converge, then try to use the classic version of A*.

Step 4: Convergence – If neither traditional A* does not converge, then all steps are repeated from *Step* 1

[17]The A* algorithm, pronounced A star, is one of the most established and efficient search algorithm used in the path planning field, which uses a heuristic function to guide its search. The high speed of convergence is guaranteed by construction, as it implements a "best solutions-first" policy. This property suits perfectly the urban scenario, where the high interconnection of streets and paths plays a fundamental role in the speed of convergence. Since A* relies on graphs, each cell of the matrix map is considered as a node, and the equation that governs the choice of the path is structured as follows:

$$f(n) = g(n) + h(n) \tag{4.5}$$

, where n is the next node, while $g(n)$ is the cost of the path from the start node n_0 to the generic node$n$ in terms of distance, describes as:

$$g(n)|_{n_0} = (n.x - n_0.x) + (n.y - n_0.y) \tag{4.6}$$

The function $h(n)$ is a heuristic function that estimates the cost of the cheapest path from n to the goal. The algorithm explores the map by seeking a path that minimize the overall overall travel cost. For matrix maps, as in our case, it is possible to select a consistent heuristic by means of a taxicab distance between a generic cell n and the goal cell n_g:

$$h(n)|_{n_g} = (n.x - n_g.x) + (n.y - n_g.y) \tag{4.7}$$

Compared to the classic A*, where the aim is to find the path that minimizes the number of movements, the Explorative A* has been developed ad-hoc for in this work of the thesis. Its purpose is to search for the shortest path that reaches the goal while exploring the unexplored area with continuity. A visual comparison between A* and Explorative A* is shown in Figure 4.11. The resulting path is a path that passes only through unexplored cells. Since this is not always possible, when Explorative A* does not converge, the traditional A* is used. Although dynamic versions of A* exist, capable to re-plan a route when they encounter

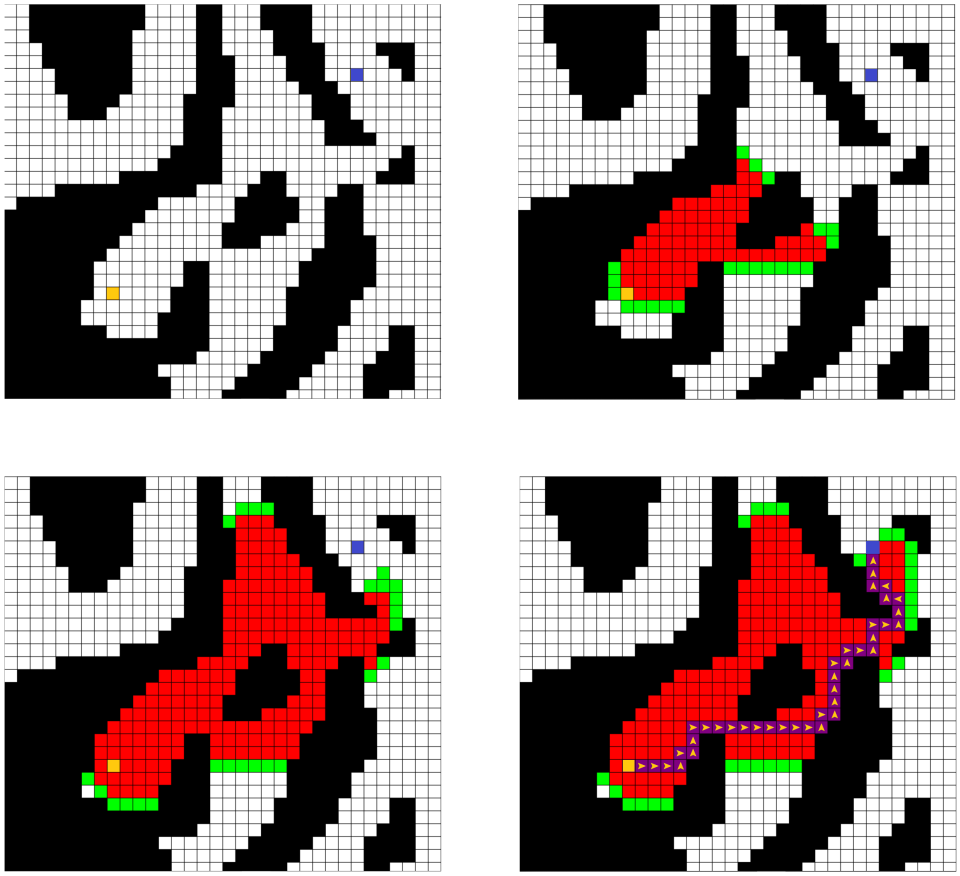


Figure 4.10: Traditional A* finds the path - the start cell is represented in orange; the goal cell is represented in blue; red cells are the nodes visited by the algorithm; green cells are the cells that the algorithm can expand; the final purple cells represent the minimum and optimal path to reach the goal from the start.

an unexpected object, this work uses the static version. If a UAV encounters an obstacle while following a planned path, that obstacle can only be another agent: in those cases, the UAV hovers for a time-step and waits for the other agent to move. Several measures are coded to improve efficiency and avoid congestion issues. For instance, if an agent is following its A*/Explorative A* planned path towards the goal, and another agent explores its goal cell, then a new path is planned toward a new goal cell; if a UAV encounters unexplored cells during the planned path such that at least one unexplored cell lies inside its local FOV, then A*/Explorative

A* is deactivated and Neural Network decision-making activates. A more precise overview of the algorithm is shown in the following Section 4.5.

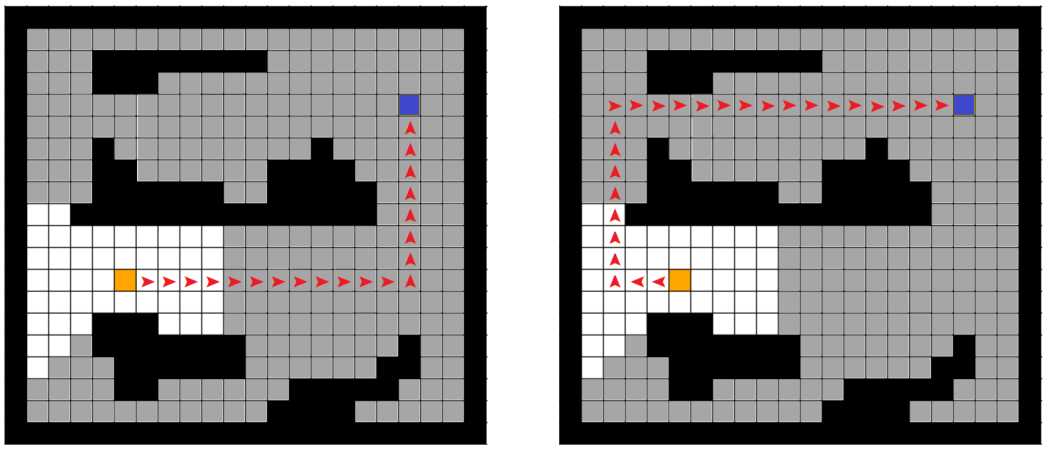


Figure 4.11: Traditional A* and Explorative A* in comparison, same situation. The UAV is located in the orange square; the goal cell to reach is represented in blue. The left figure shows the path with Traditional A*, which is the shortest path; the right figure shows the path with Explorative A*, which accounts for both the length of the path and the exploration of new cells (in grey).

4.5 Flowchart

The complete flowchart is shown in Figures 4.12 and 4.13: it describes how the coverage path planning is carried out, highlighting the dual use of neural networks and path-finders. Before the exploration starts, the target explorable map is split as described in Section 4.3. Once obtained N different zones, each of the N agents are assigned to one of those, prioritizing the exploration in the correspondent area. Each agent has a limited view on the map, and it represents a consistent part of the input state fed into the network, as explained in Subsection 3.2.1. The UAVs can sense as "unexplored" only the cells that belong to their assigned zone. Thanks to this artifice of sight, each agent will be focused on the exploration of its correspondent zone. The iteration continues until the map is fully explored. At each time-step, each agent is asked to take one of the four actions $\{Up, Right, Down, Left\}$ over the 2D map. In the initial time-step, it is assumed that each agent does not have any planned path, so A* flag is set to False. If the local view of the $i\text{-}th$ UAV contains unexplored and visible cells, then the decision-making is entrusted to the

neural network; conversely, the path-finding algorithms aid the UAV to search the closest unexplored cell. When it uses the neural network approach, the UAV senses the information needed for the input state, and then a forward propagation predicts the probability distribution over the four actions. The agent is encouraged to take the action with the highest value, checking in advance if that action would lead to a crash. If the action is admissible, then it is selected, otherwise, the next action with the highest probability is considered and analyzed. If the local view does not contain unexplored cells, the UAV is guided toward an unexplored cell: Explorative A* and traditional A* aid the UAV in this task. First of all, since there are not any target cells in view, the algorithm searches for a potential goal cell that is unexplored, enlarging progressively the view of the UAV. When the closest potential goal cell is founded, then the algorithm checks the convergence with the path-finding algorithms. It is sufficient that at least one of the two converges to be able to set the flag A_j* to "true" since the path toward the goal cell is planned. This is the process described in Figure 4.12. On the other hand, if the A_j* flag is true at the start of the iteration, then a path has been already planned. The first conditional block checks if the UAV has encountered unexplored cells during the planned path. If so, then the neural network approach is followed since the unexplored cell is in the local view, and the selection of action is the one described below. Conversely, if the path is planned toward a goal cell and no unexplored cells are in view, then the algorithm checks if the goal cells have been already explored by another agent. In this case, the path is re-planned toward a new goal cell. Once a UAV explores entirely its assigned zone, then its sight capability increase as it becomes able to explore each unexplored cell of each zone. In this way, UAVs can help each other as they finish their exploration. This cycle iterate until the map is fully explored. Further improvements are not showed in the flowchart include an anti-congestion function, activated during the research of a goal cell that avoid two UAVs to plan a path toward the same cell or nearby. The combination of those elements allows the coverage path planning of a fleet of UAVs, the results of which are presented in the next chapter.

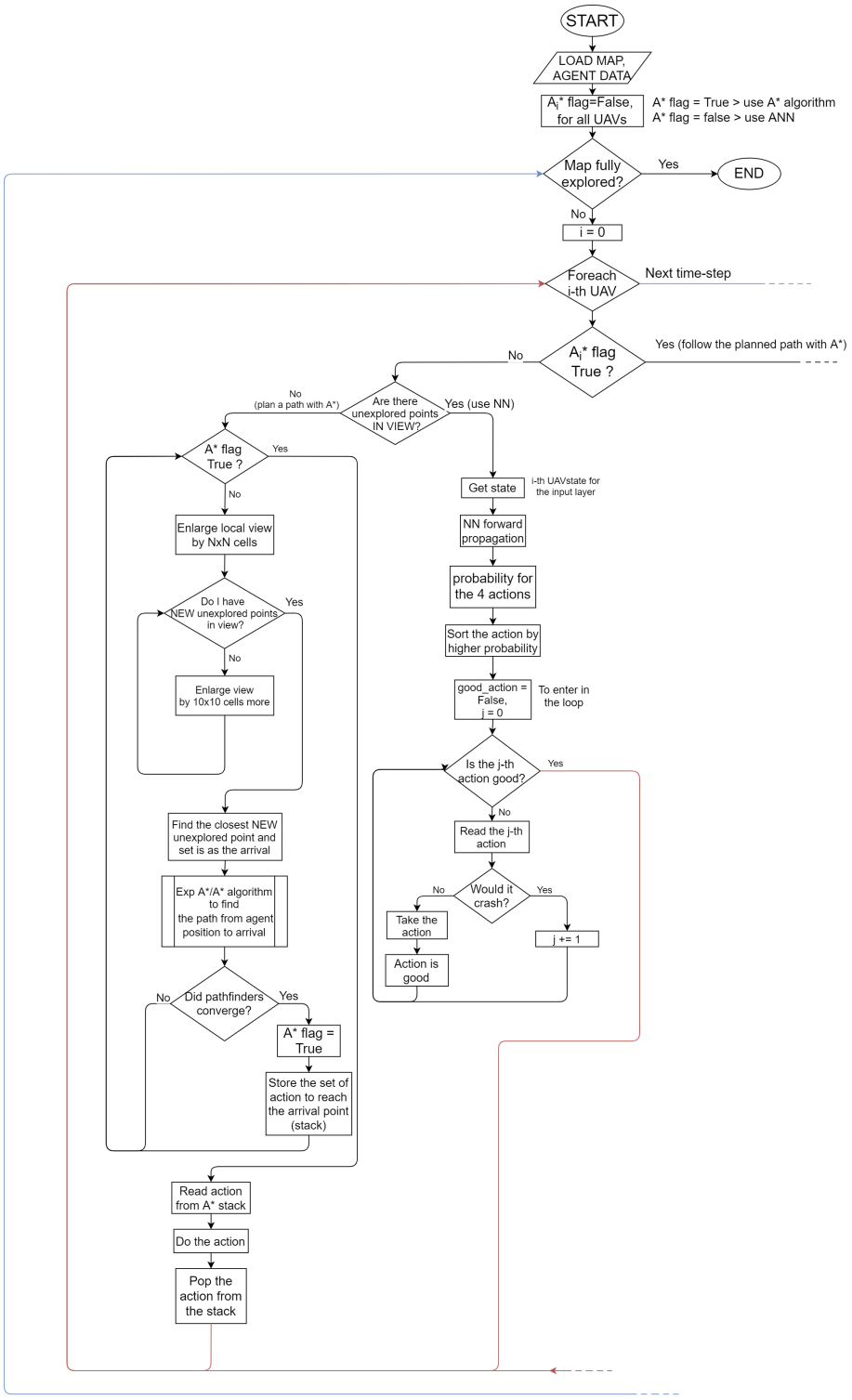


Figure 4.12: Flowchart - Part 1

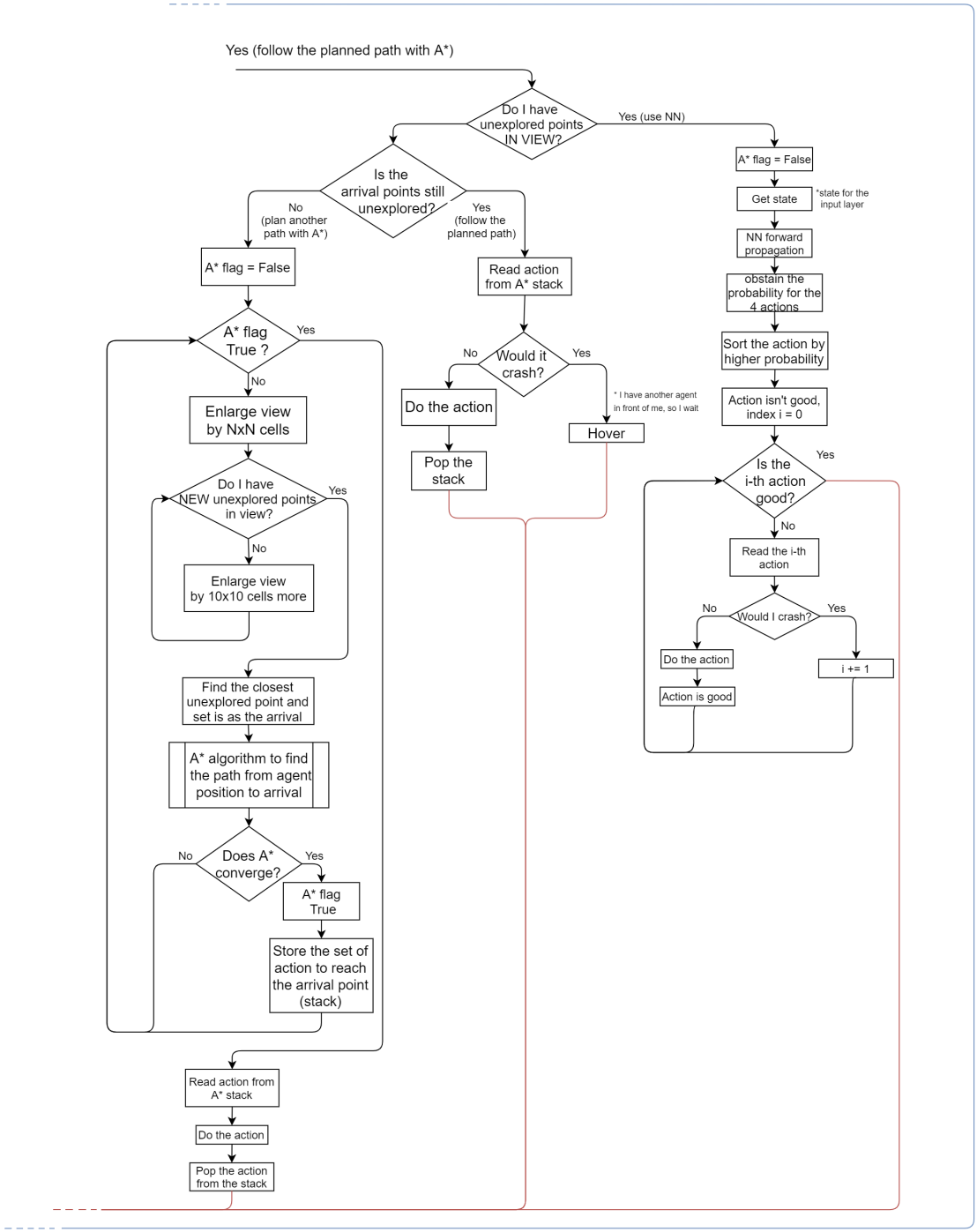


Figure 4.13: Flowchart - Part 2

Chapter 5

Results & Simulation

The performance evaluation of a coverage path planning algorithm is not trivial at all, especially when it comes to multi-agent systems where a consistent number of good solutions can coexist. Furthermore, the determination of a good trajectory is linked to the type of final application. For UAV applications, a good trajectory manages to minimize turnovers, as well as being efficient and minimizing flight time. This chapter presents the simulation results, following the basis of evaluation metrics. The algorithm is tested over real urban case studies, with a particular focus on Leonardo's indoor case study. For this latter map, it is presented also a trajectory corrector whose purpose is to smooth the simulated trajectory. Moreover, UAVs are simulated in a 3D environment run with ROS - Robotic Operating System.

5.1 Evaluation Metrics

The evaluation metrics offer a clear and objective basis to judge the quality of the results. Six parameters are considered. These parameters can be grouped into two main categories, as a function of the time interval considered. Some parameters are averaged over the whole simulation regime, hence at each time-step; other ones, indicated with an asterisk*, account for only a part of the time-steps. For the latter category, it is considered only the steady-state exploration, empirically defined as the set of time-steps in which the exploration percentage is contained between 40% and 70% of the complete exploration: in this way metrics are not influenced by the boundary starting conditions of the simulation, e.g. UAVs' starting position, thus isolating the performance characteristics of the algorithm and resulting in a more clear and accurate performance evaluation. The following table introduces the evaluation basis:

$Mean\ Moves\ Number.\ (MMN)$	$\sum_i M_i \frac{1}{A}$
$Trajectory\ Efficiency\ (TE)$	$\sum_{i,t} \frac{E_{i,t}}{9} \cdot \frac{1}{A \cdot MMN}\ \%$
$Mean\ Number\ of\ Turnovers (MNT)$	$\sum_{i,t} T_{i,t} \cdot k_\theta \frac{1}{A \cdot MMN} \%$
$Mean\ UAVs\ Distance^*\ (MUD)$	$\sum_{i,j,t} D(a_{i,t}, a_{j,t}) \frac{2}{(A^2 - A) \cdot T_r}$
$Mean\ Distance\ from\ Unexplored\ cells^* (MDU)$	$\sum_{i,j,t} min(D(a_{i,t}, u_{j,t})) \frac{1}{U_t \cdot T_r}$

• Mean Moves Number (MMN): the smaller MMN, the more successful the overall strategy is: the parameters is calculated dividing the sum over the number of total moves M_i of the $i-th$ UAV by the UAV number A. Reducing the flight time is mandatory due to energetic constraints in UAV application.

• Trajectory Efficiency (TE): considers the exploration quality: since each UAV can explore up to 9 cells for action, the exploration rate $E_{i,t}$ of the $i-th$ UAV at timestep t is divided by 9. The overall sum of E_i, t is then divided by both the number of agents A and the MMN. The reference trajectory is an ideal trajectory where each UAV explores 9 cells per timestep for all timesteps, TE = 100%. Since the map features make it impossible to reach this percentage, it is desirable to obtain a high TE value anyway.

• Mean Number of Turns (MNT): this metric is considered since the energetic constraints must be accounted for: turnovers are one important factor of energy dissipation. For this reason, the sum of all turnovers $T_{i,t}$ at time t by the agent $i-th$, is multiplied by a factor k_θ which can be equal to 1 if the turnover is a 180° or equal to 0.5 if the turnover is a 90°. The overall sum is divided by the number of agents A and by the Mean Moves Number. The lower MNT, the smoother trajectory is planned.

• Mean UAVs Distance (MUD^*): indicates how distributed are the UAVs at steady-state exploration. The function $D(a_i, a_j)$ is the Euclidean distance between the agent a_i and the agent a_j. The positions are given in terms of the correspondent row and column of the UAV position in the matrix map so the MUD is expressed using the distance between two adjacent cells as the unit of measurement. The sum of distances is then divided by the number of all possible combinations of the relative distances between the UAVs. For A UAVs, the number of possible pairs is $(A)(A-1)/2$. The larger the MUD, the more the drones are spaced. To get an idea of the strategic distribution that can cover all areas of the map, the MUD must be considered in conjunction with the MDU.

• Mean Distance from Unexplored cells (MDU^*): considers the overall coverage at steady-state exploration. For each unexplored cell of the map u_j, it is considered the Euclidean distance between the cell u_j and the nearest UAV a_i. The overall sum of these distances is then divided by the number of timesteps a regime T_r and the number of U_t at timestep t. The smaller the MDU, the better distributed are the UAVs since each unexplored cell can be rapidly reached and explored.

5.2 Results

The algorithm focuses on occupancy grids inspired by real urban maps. Results are presented in four case study maps. Each map is simulated four times, with an increasing number of agents from 3 to 6, and results are compared in terms of evaluation metrics. Particular attention is paid to Case Study I, whose relative grid is representative of Leonardo Drone Contest indoor scenario presented in October 2020. This map covers a $20 \times 10[m^2]$ area and has been modeled with an 84×42 cells matrix. A 3D model of this map is shown in Figure 5.1. Other maps include a more complex occupancy grid since they are inspired by real urban maps.

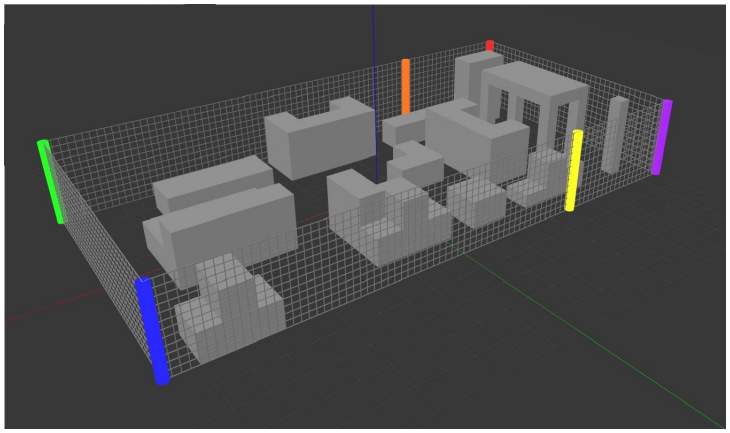


Figure 5.1: Case study I - a 3D view of the Leonardo Drone Contest urban environment

The first batch of results is presented in Leonardo's Case Study, as it presents a simple geometry to appreciate a visual representation of the trajectories. Figure 5.1 shows how the 3D model of the environment is modeled with the matrix map: the tighten grid offers a wide range of trajectory planning possibilities. Since the exploration is bi-dimensional, the grid is created for a fixed height following the assumptions introduced in 4.1; unreachable cells are considered as explored, represented in white in Figure 5.2, since there is no interest in visiting them and avoiding the neural network to predict wrongly. In the following table, evaluation

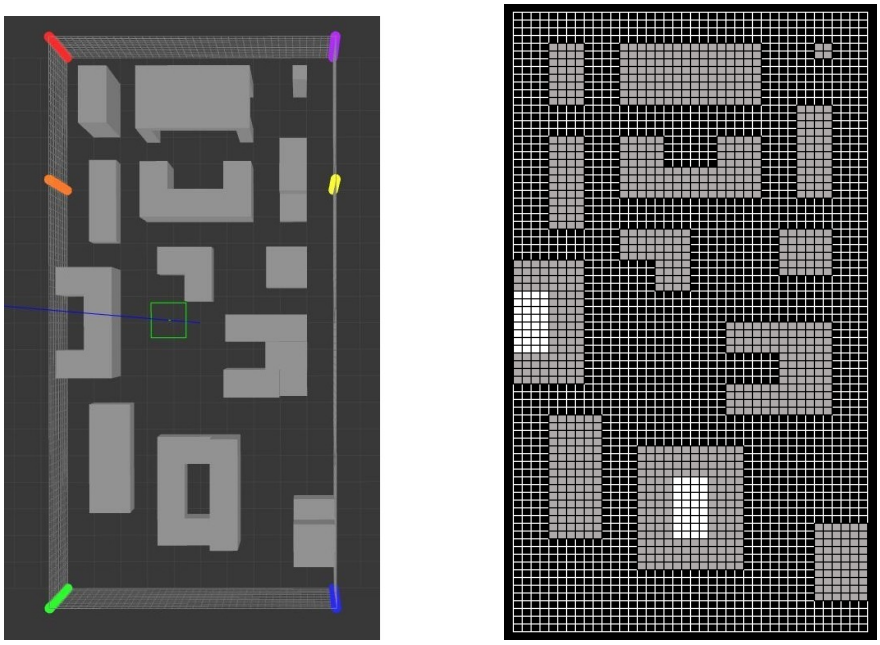


Figure 5.2: In the left figure, Case Study I - inspired by Leonardo Drone Contest. In the right figure, the matrix map used in the simulation.

metrics are implemented for the Case Study I map, with an increasing number of agents from 3 to 6:

Table 5.1: Case study 1: Leonardo's Map - Fig 5.2

$Num.\ of\ UAVs$	3	4	5	6
MMN	183	168	133	126
TE	41.1%	32.8%	32.4%	27.1%
MNT	6.73%	7.21%	6.31%	5.88%
MUD	22.25	31.48	38.54	35.31
MDU	19.55	14.04	13.39	13.61

The Figure 5.3 contains a plot of the metrics with an increasing number of agents. As shown in Table 5.1, the higher the number of UAVs that simultaneously explore the map, the lower the number of movements is required for the complete exploration, as expected. On the other hand, the increased number of agents leads to a performance reduction in terms of trajectory efficiency - UAVs tend more frequently to pass over cells that have been already explored. The weighted number of turning maneuvers appears to be independent of the UAV's number and the behaviour does not allow to go for a generalization of concepts: this is a

consequence of sharing a single neural network hence that same way of acting. For tiny maps as this Case Study 1, both Mean UAVs' Distance (MDU) and

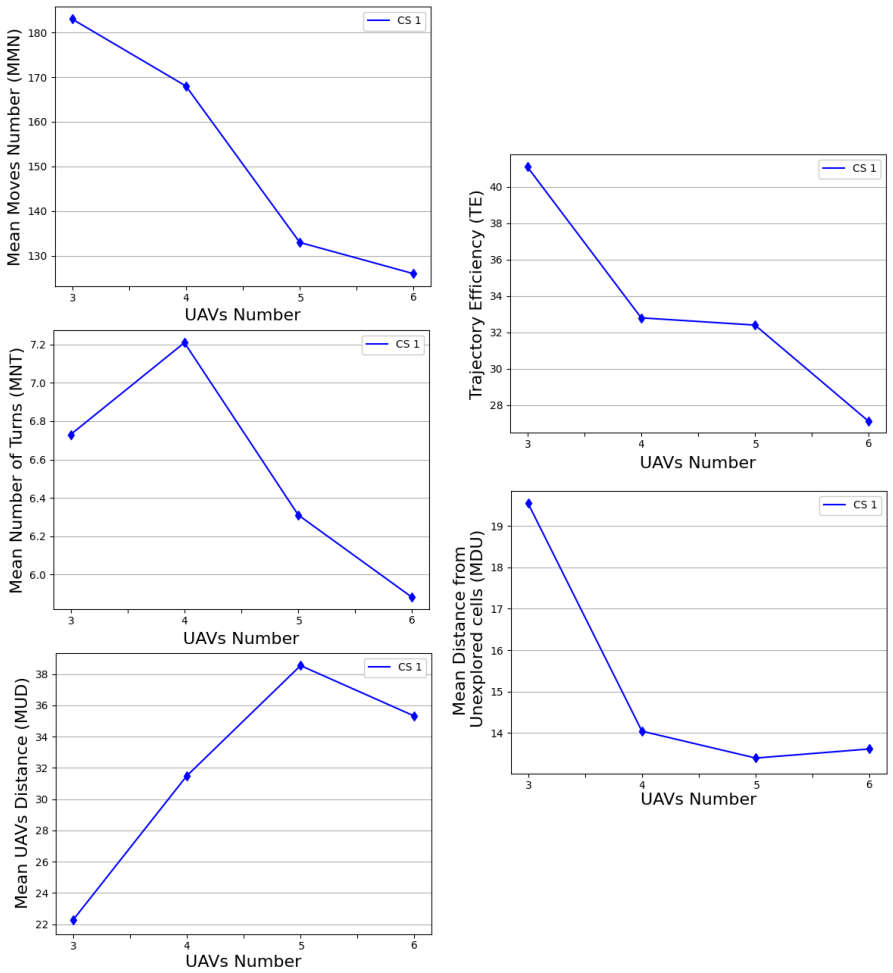


Figure 5.3: Case study I - results in terms of evaluation metrics

Mean Distance from Unexplored cells (MUD) are dependent on the UAVs starting position. However, to achieve performing explorations, MDU should be high and MUD low, simultaneously, in order to have scattered UAVs in strategic positions. In Figure 5.4 it is shown the track of each UAV's trajectory for the complete coverage path planning with a number of agents/UAVs equal to 5 over the Case Study I. For the same simulation, in Figure 5.5 it is shown the frequency by which Neural Network and A*/Explorative A* are used during the simulation: the graph shows an increased use of A*/Explorative A* towards the end, as expected, since the number of unexplored is reduced and the UAVs need the aid from an explicit path-finding algorithm.

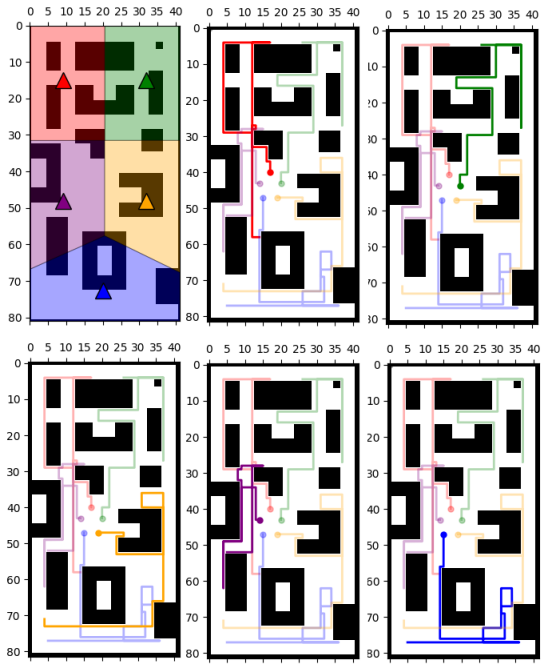


Figure 5.4: Case study I - trajectories in the simulation with 5 agents. The upper left image shows the area division, according to the K-means algorithm . In the other images, the overall UAVs' track, where each singular UAV trajectory is highlighted.

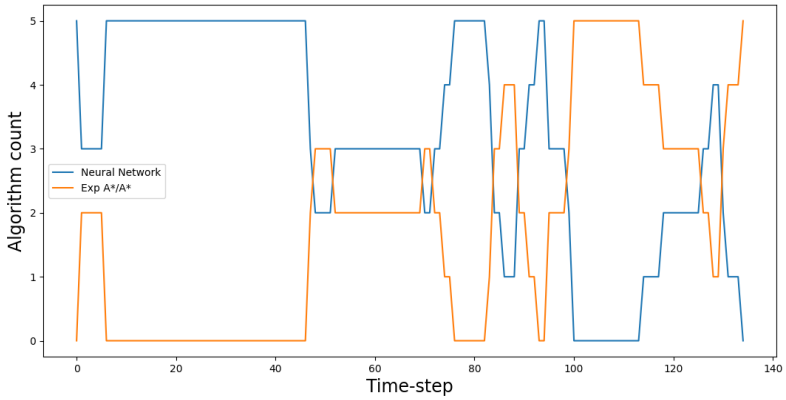


Figure 5.5: Case study I, 5 agents - On the x axis, the time-steps. For each time-step, the graphic shows the number of agents that use the neural network approach, in orange, and the explicit path-finders, in blue.

The second case study is inspired by a view of Porta Nuova, Turin, as shown in Figure 5.6, modeled with a matrix of 195×247 cells. This case study results more complex compared to the first one. but the urban structure of Turin preserves sharp curves as Case Study I. Note that the density of obstacles and buildings in the real map is drastically higher compared with the Case Study I. Results are shown in Table 5.2 and plotted in Figure 5.7. In this case study, the high density

Table 5.2: Case study 2- Results - Turin, Porta Nuova - Fig 5.6

$Num.\ of\ UAVs$	3	4	5	6
MMN	1019	854	827	591
TE	37.5%	33.5%	27.5%	32.1%
MNT	4.01%	4.51%	5.27%	4.01%
MUD	153.10	118.96	132.97	120.33
MDU	72.61	54.18	50.37	48.76

of obstacle and the structure of the explorable space is very rigid. Since streets are very long, the path must be planned carefully, since a single wrong turning maneuver could lead to several time-steps without exploration. The Mean Moves Number parameter decreases as expected, showing a drastic reduction when the number of UAV is equal to 6, from 827 to 591 moves per agent. The trajectory efficiency has been strongly influenced by the MMN, showing that six agents performed better than five agents. As anticipated, this could be attributed to a combination of some sub-optimal planned maneuvers and the nature of the map which does not allow mistakes. The MNT is in the order of 4% due to the multitude of straight paths, less if compared to Case Study I, in which it reached the 7.2%. Note that the original scale of the map has a strong influence on the resolution of the grid. In each map, the step of movement has an amplitude determined by the height from which the UAVs fly; a possible solution to avoid those long straight corridors could be a change in grid resolution hence higher UAVs, to achieve a wider step. The MUD and MDU must be analyzed in conjunction with the dimensions of the map. Both MDU and MUD decrease drastically when the number of agents passes from 3 to 4, while for a higher number of UAVs the coverage appears to be uniform and well distributed. The track of the UAVs is not showed for real urban maps since it would be difficult to understand the path from a figure. Comparing each metric, it seems that for a number of agents equal to 5 the exploration was not completed efficiently since the correspondent point shows the worst performances and it can be considered an outlier. Particularly, the number of turning maneuvers is particularly high, leading to less trajectory efficiency.

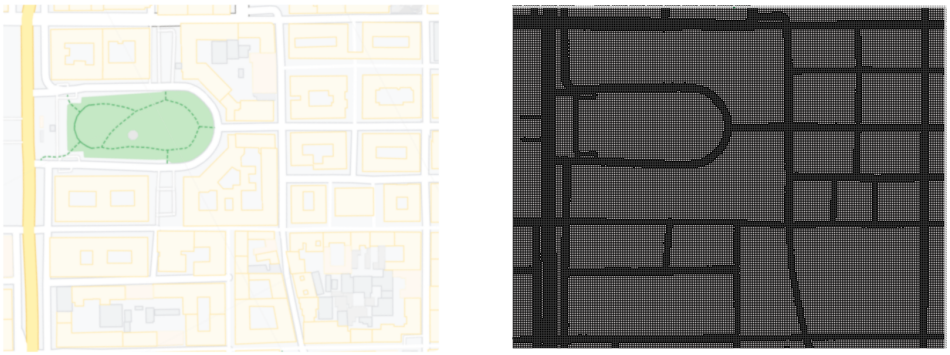


Figure 5.6: In the left figure, a satellite view of Porta Nuova, Turin. In the right figure, the Case Study II map used for the simulation.

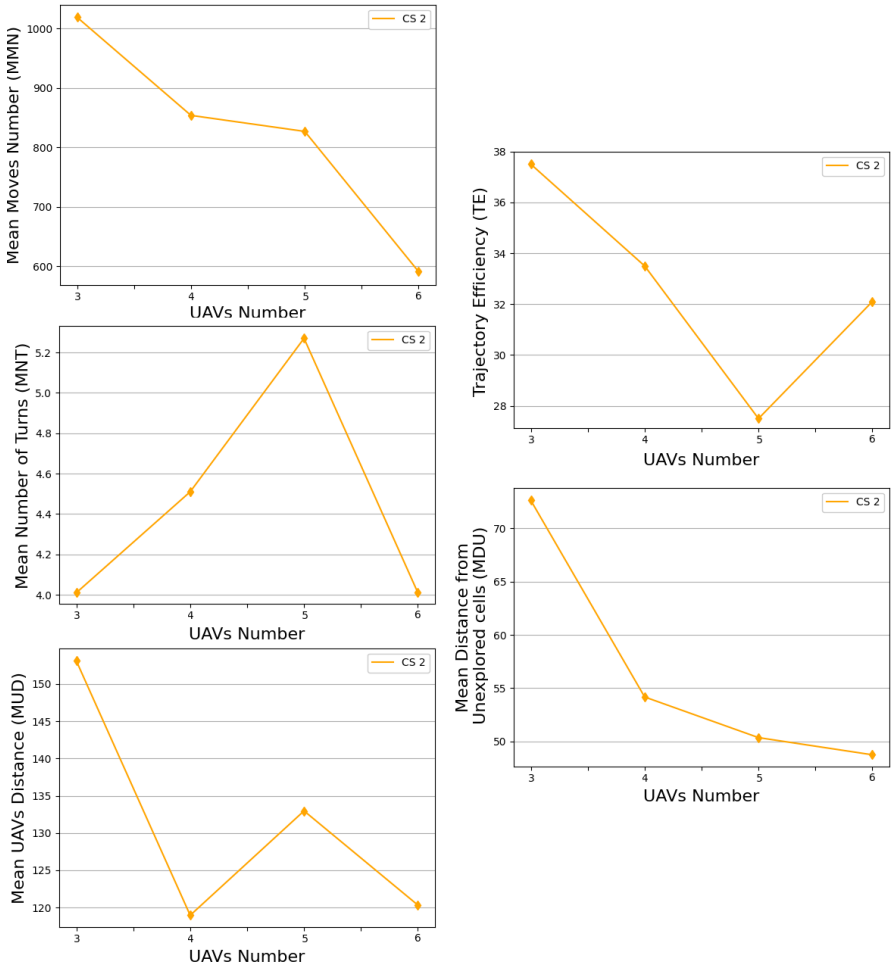


Figure 5.7: Case study II - results in terms of evaluation metrics

The third case study is inspired by Porto Antico, Genoa, as shown in Figure 5.8. Compared to Case Study II, this presents a strong interconnection between the various areas although the dimension of the map is 104×161 cells. Results are presented in Table 5.3 and plotted in Figure 5.9. The number moves per agent decreases as the number of agents increases, with a drastic reduction when the number of agents passes from 3 to 4. On the other hand, the simulation with 3 agents presents a very low trajectory efficiency, which raises for the simulation with 4 and 5 agents and then collapses when the number of agents becomes 6. The 5 agents case is particularly interesting since both trajectory efficiency and the number of turns are low, while also preserving excellent coverage on the map. The

Table 5.3: Case study 3- Results - Genova, Porto Vecchio Fig 5.8

$Num.\ of\ UAVs$	3	4	5	6
MMN	1012	674	553	482
TE	23.6%	26.3%	26.4%	24.23%
MNT	21.2%	20.8%	19.39%	20.40%
MUD	53.76	69.48	62.99	65.73
MDU	41.91	25.33	26.96	22.09

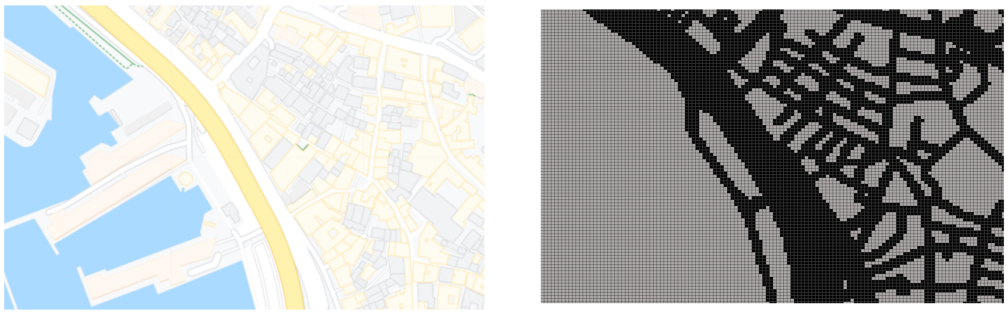


Figure 5.8: In the left figure, a satellite view of Porto Antico, Genoa. In the right figure, the Case Study III map used for the simulation.

fourth case study is inspired by Porto, Spain, as shown in Figure 5.4. Results are presented in Table 5.4 and plotted in Figure 5.11. This map presents a new critical issue since the Douro rivers that cross the entire map splits the explorable surface in two. These two areas are connected only by a bridge so the crossing is forced. It should also be noted that drones are not bound to fly over only the walking areas but could use virtual corridors to fly across the river and access the other side more easily. Although the grid is not the thickest, with only 133×199 cells, the covered areas are the larger of all the four case studies. For this reason, UAVs have an easier possibility of planning the route between urban blocks, although the movement step

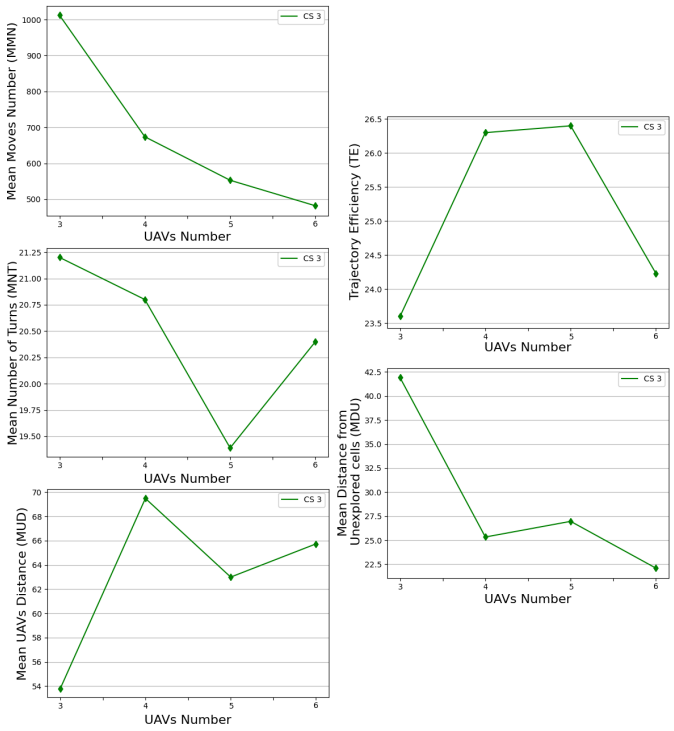


Figure 5.9: Case study III - results in terms of evaluation metrics

is much larger than in previous cases. This property flows into a more homogeneous distribution, showing the actual improvements in exploration performances brought about by the addition of every single UAV. The MMN decreases in a quasi-linear way, while the other parameters highlight an increased performance trend as the number of agents increases, with a particular efficient trajectory planned with 5 UAVs. The exploration has given excellent results despite the constraint of the river, whose merit is also attributable to the division algorithm by areas, which has managed to divide the areas to avoid that the same area does not correspond to unexplored points straddling the river.

Table 5.4: Case study 4- Results - Porto, Douro River - Fig 5.10

$Num.\ of\ UAVs$	3	4	5	6
MMN	1537	1157	911	791
TE	21.7%	21.6 %	21.9%	20.8%
MNT	19.43%	18.56%	17.56%	18.05%
MUD	131.11	98.27	95.34	90.13
MDU	46.01	38.01	38.45	32.71

Figure 5.10: In the left figure, a satellite view of Porto, Portugal. In the right figure, the Case Study IV map used for the simulation.

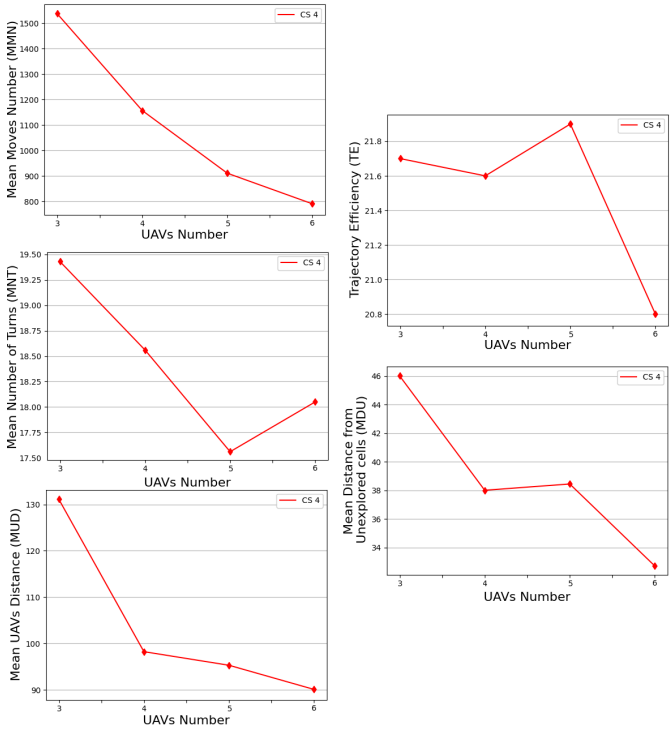


Figure 5.11: Case study IV - results in terms of evaluation metrics

Finally, the overall results are showed in Figure 5.12, where each of the previous tables from Table 5.1 to 5.4 are plotted together. The results are not normalized to the size of the map, as the density and distribution of obstacles are much more influential on the performance results. The number of UAVs influences positively the Mean Moves Number, MMN, decreasing the time required to achieve the complete exploration. For larger areas, this performance characteristic is inhibited by a decrease in the efficiency of the trajectory which makes the UAVs tend to

explore areas already explored several times. On the other side, the energetic constraint must be accounted for, and the mean number of turnovers seems to be independent of the number of UAVs. A trade-off analysis also considers the contribution to a homogeneous distribution of UAVs in space, thus showing how the algorithm is effectively able to plan a trajectory capable of intercepting the criticalities of the maps presented and adapting the path even to unconventional obstacles.

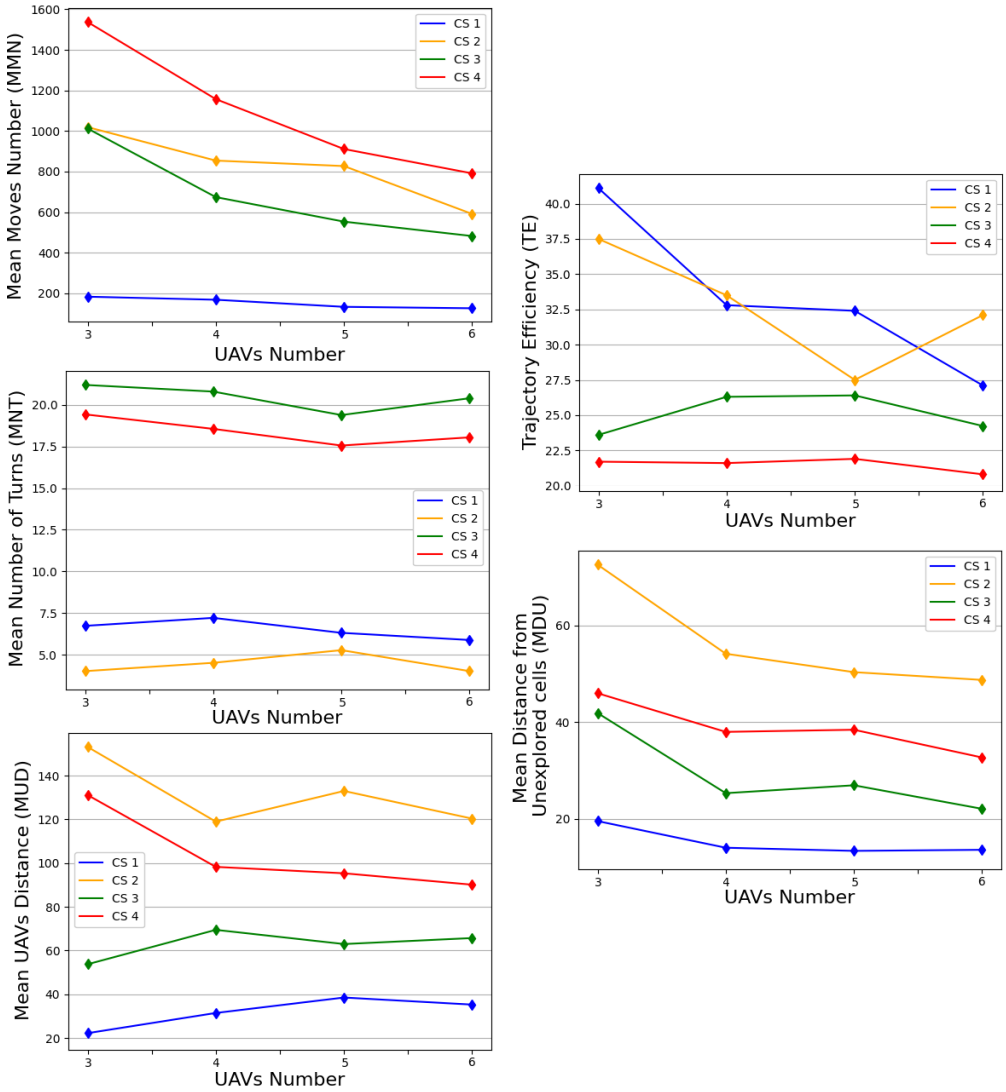


Figure 5.12: Case study from I to IV - results in terms of evaluation metrics

5.3 Simulation in ROS

The coverage path planning for cooperative UAVs has been simulated in a 3D environment using a reconstruction of Leonardo Drone Contest scenario. The path is the one shown in Figure 5.4, but in the following images, it is possible to appreciate also the dynamics of movement of each drone in accordance with the flight mechanics. The simulation has been developed using Gazebo, a 3D simulation of rigid body robots, and ROS, acronym of Robot Operating System, a framework for development and programming of any kind of robotics. ROS is an open-source meta-operating system for robotic systems[18]. Practically, ROS is a standard for robot programming and offers a general-purpose robotics library for robotic applications. Gazebo is an open-source multi-robot simulator fully compatible with ROS[19] able to simulate robots, sensors, and rigid body dynamics. In Figure 5.13 the 5 UAVs are shown in the idle state on the ground. Figure 5.14 shows the engines' start and the take-off phase, up to the initial hover state. Finally, Figures from 5.15 to 5.17 show different views angle of the exploration.

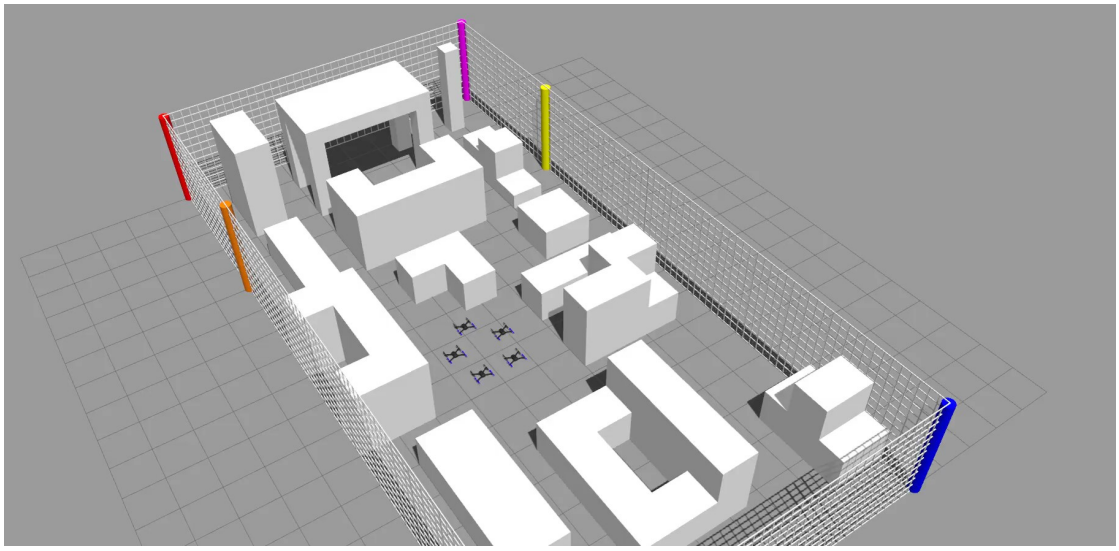


Figure 5.13: Case Study I - Leonardo Drone Contest - The UAVs are in a state of IDLE on the ground. The angled view allows a more complete perspective on the urban case, of which the occupancy grid used in the simulation allowed a planar visualization of the obstacles

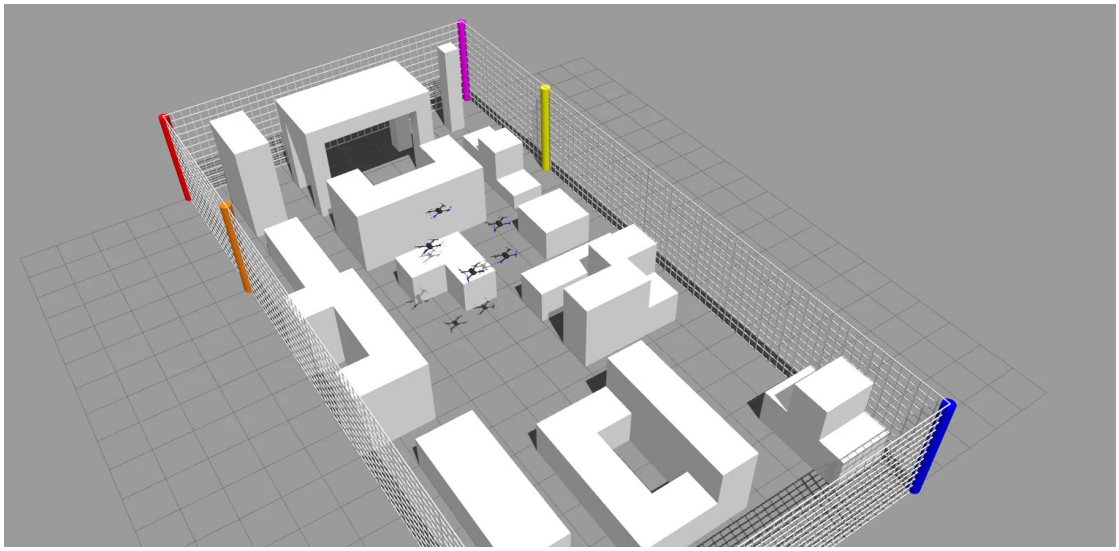


Figure 5.14: Case Study I - Leonardo Drone Contest - The UAVs in-flight formation after the take-off.

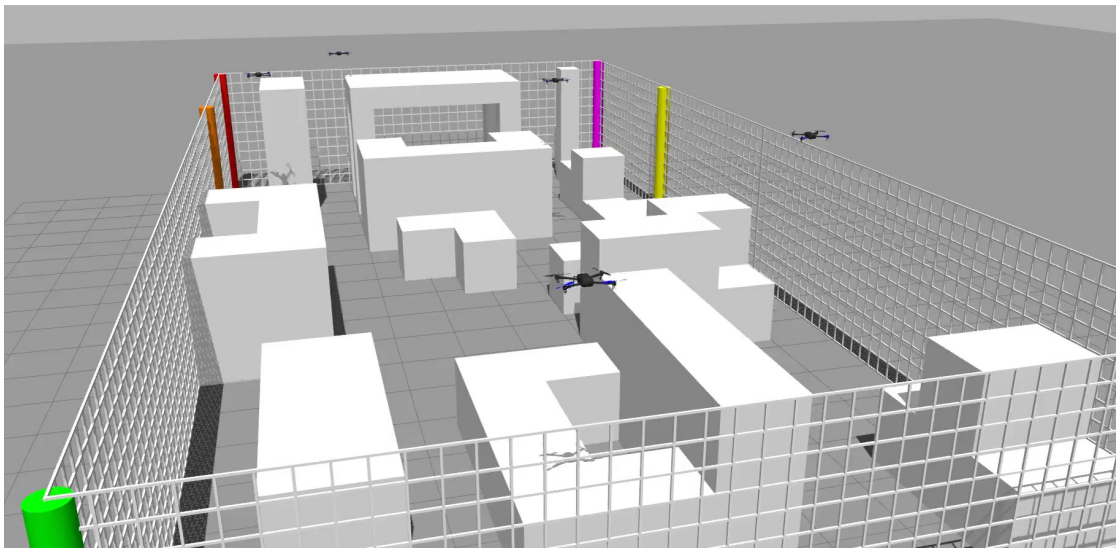


Figure 5.15: Case Study I - Leonardo Drone Contest..

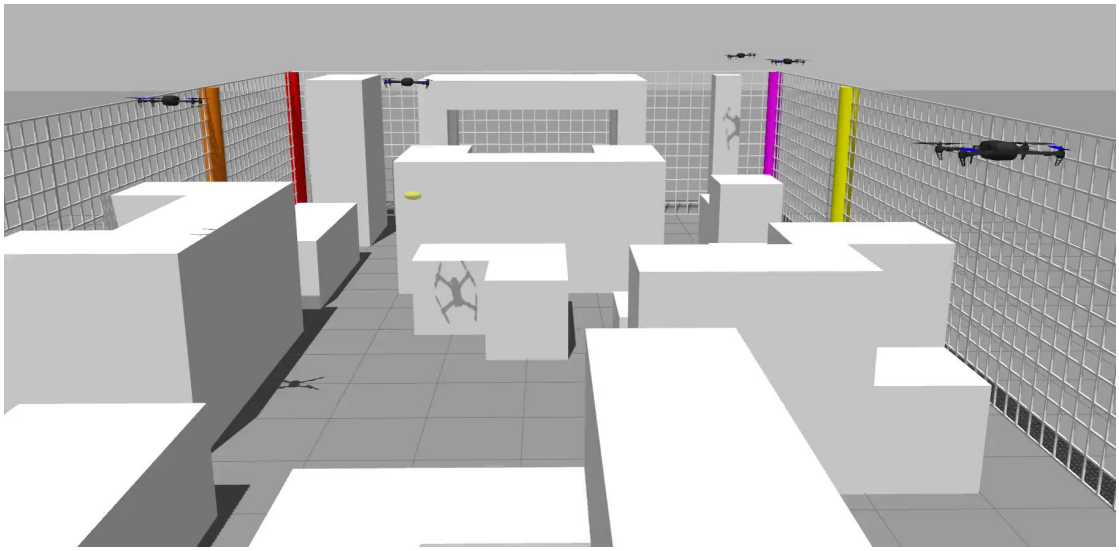


Figure 5.16: Case Study I - Leonardo Drone Contest.

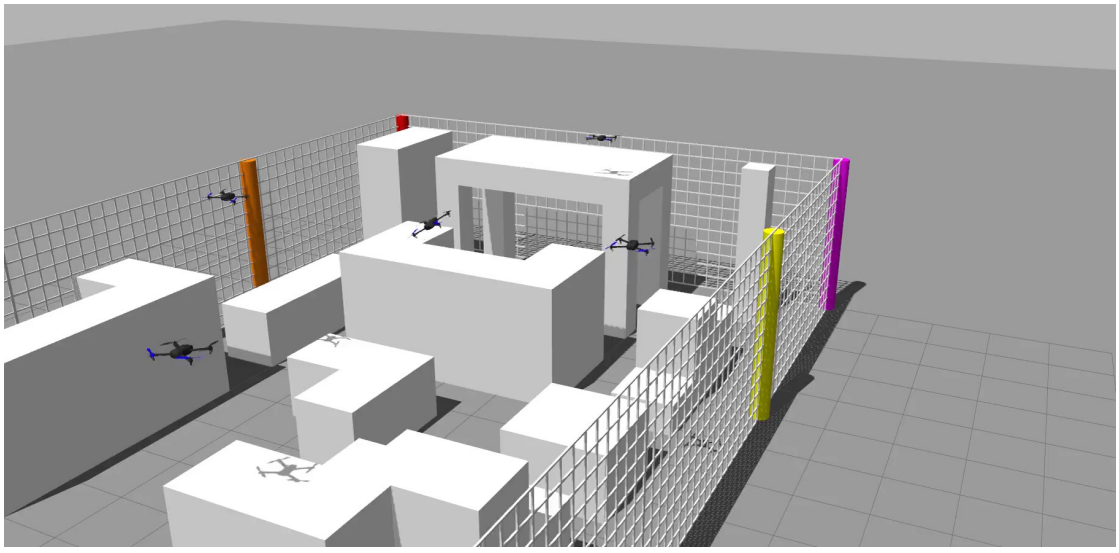


Figure 5.17: Case Study I - Leonardo Drone Contest.

Chapter 6

Conclusion

The study conducted on the planning of the path of unmanned aerial vehicles has made it possible to identify a possible area of intervention in urban scenarios. The high density of complex obstacles requires detailed planning, making the best use of spatial discretization. The simulations of this thesis project use satellite urban maps with very tight occupancy grids, in which the movement step allows precise planning even on non-convex geometries, and in which traditional approaches are unfruitful, especially in the multi-UAV case. The simulation is discrete in time and space, and each UAV is asked to select one of the four actions at each time-step, coordinating with the other agents to achieve congestion and collision-free path.

The selection of the movements renders the coverage path planning a decision problem, usually tackled with Reinforcement Learning techniques and Artificial Neural Networks, where strong decisions mathematical frameworks as the Markov Decision Process allow modeling the problem to achieve Artificial Intelligence. In this work of thesis, the problem has been reformulated, transforming the decision problem into a classification problem, exploiting the state of the art algorithm to train the neural networks with the Supervised Learning.

Potentially, the neural network model that underlies both Reinforcement Learning and Supervised Learning can be almost the same, with some changes on the output layer, while there is a substantial difference in the approximated function. The purpose of Multi-Agent Reinforcement Learning is to maximize the expected reward interacting with the environment in terms of exploration, implementing exploring paradigms as the "exploration vs exploitation", in which the network tries to predict which action might lead to the best strategy according to the designed rewarding policies. In the Supervised Learning, used in this thesis, the policy of the network does not receive any feedback from the environment, but the network is trained to imitate a pre-existing strategy, recorded from an experience basis of an expert system prior. This technique is referred to as Supervised Imitation

Learning.

Supervised learning has an excellent optimization framework and libraries, so much so that the transition from Reinforcement to Supervised Learning has made it possible to reduce the training time of the network from several tens of days on a supercomputer cluster to a couple of minutes on a domestic computer.

A strong requirement of supervised networks is the need for a large labeled database for the training session. By taking advantage of a framework designed ad-hoc for the collection of data, it is possible to record in a precise and orderly way every single action of an expert system, and thus create the dataset. This system allows an autonomous generation of data, satisfying the numbers necessary to obtain high-performance networks.

The expert system can be both a human or an expert algorithm. In this work, the experiences were collected based on human priors, in which each action was entered manually from the keyboard and coupled to the current state of the UAV. The imitation framework is used to collect the experiences from the expert system records entries for a single-agent case in simplified environments. This inductive approach shows further generalized learning capabilities of the networks, which is trained over single-agent case collected data while the final applications involve multi-agent scenarios.

The data augmentation was explored, allowing the creation of new experiences based on those already registered and gross multiplication leverage of 8 times the size of the initial database. The final database contains a total of 54131 entries, structured as pairs containing the UAVs state and the correspondent action taken by the expert system.

The network learnt the strategy of the expert system stored in the database by seeking hidden patterns and cross information among the data. The final network is capable of coupling the state of the UAV with the same action that the expert system would carry out to reproduce them when it is in the same situation and not only. The network manages to internalize the method with which the path was planned by the expert system and to propose it again in situations never seen before, thanks to the identification of recurring decision-making patterns within the database.

Thanks to an iterative try and error process, the network configuration has been designed ad-hoc to ensure excellent predictions on known data but also to preserve the generalization capabilities. The learning results are very good, reaching 92.01% percent accuracy over the 54131 entries of the database. The configuration of hyper-parameters was detailed in Chapter 3, with a total of 106684 trainable parameters. The simulations were carried out on occupancy grids inspired by satellite view on urban scenarios, some of which were virtualized

in three-dimensional environments using Gazebo and ROS. Particularly, for the Leonard Drone Contest map, the simulation has been analyzed from multiple points of view, from coverage performances to the efficiency of trajectories, from the frequency of usage of the neural network to the complete 3D simulation.

A basis of evaluation metrics has been defined to monitor both energy constraints and the distribution of agents on the map. The exploration performances show that the network, trained in the single-agent case, is also versatile in the multi-agent case, performing optimally on the satellite maps. Furthermore, the network is able to generalize planning concepts to scenarios never seen before, being able to guess the presence of other UAVs in motion and intercepting future actions in a coordinated, clean, and efficient way.

The network performs in a satisfactory manner and the output trajectories offer strategic coverage according to the limited view on the map. This feature has been tested in several scenarios with the most various obstacles, including town squares, alleys, and rivers. The simulations highlighted better performances where the step of movement was wide, hence planning with a resolution of the grid that allows the modeling of the underneath map with sufficient accuracy in which the urban blocks are not too distant and therefore the UAV is able to have a sufficiently wide local view on the set of buildings.

A drawback of having replaced Reinforcement Learning with Supervised Learning is certainly the absence of improvements compared to the strategy of the expert system. For problems such as coverage path planning, there is no single optimal solution, but the range of possibilities must be explored in accordance with the boundary conditions of the problem, for example by favoring the reduced number of turnover rather than flight time. In the case supervised by imitation, the agent does not learn to know or interact with the environment, but thanks to a training process the neural network is made capable of guaranteeing excellent predictions on both known and unseen data, preserving the capacity for generalization. Final planning then occurs with the same strategy as to how the expert system would have performed it. Moreover, an expert system is always needed to guide the solution. Conversely, in Reinforcement Learning the agent is free to explore new policies, hence improving its strategy and self-induce learning, always provided the onerous requirements of time and computational resources

It would certainly be interesting to see how imitation learning could perform in conjunction with reinforcement learning techniques, so that the agent does not have to build a strategy from scratch, but can already start from the one of the expert system, and then develop further improvements autonomously. Another point of improvement is certainly the neural network and in particular, its configuration, exploring the vast universe of neural networks and the countless

other architectures that guide modern cutting-edge robotics. Instead of using human priors, it could be possible to collect experiences from the most advanced coverage algorithms and their unique strengths together using a neural network. Moreover, also convolutional neural networks could be used in the classification operation, thus allowing to isolate the interesting features on which to base the decision-making process.

Acknowledgements

Vorrei dedicare questo spazio a chi, con dedizione e pazienza, ha contribuito alla realizzazione di questo elaborato.

Un sentito grazie al mio relatore Prof. Giorgio Guglieri, per me un mentore, per l'infinita disponibilità e tempestività dimostrate nella stesura di questo documento, oltre che per gli insegnamenti e la passione trasmessi con le sue lezioni.

Ringrazio il mio correlatore, Simone Godio, che in questi sette mesi di lavoro ha saputo guidarmi, con suggerimenti pratici e preziosi consigli, nelle ricerche e nella stesura dell'elaborato con professionalità, chiarezza e pazienza. A Simone va inoltre attribuito il merito per la simulazione 3D presentata nel Capitolo 5.

Ringrazio anche gli amici e colleghi come Michele Pio, Carlotta, Ludovico, Alice, Ginaluca che ogni giorno hanno condiviso con me gioie e sacrifici senza voltarmi mai le spalle, rendendo questo traguardo ancora più speciale.

Ringrazio i miei fedeli amici di adolescenza, Filippo e Antonello, per essermi rimasto accanto durante i miei brevi rientri a casa

Per ultima, ma non per importanza, ringrazio tutta la mia famiglia, soprattutto Nonno Giovanni che dal cielo festeggierà insieme a me questo grande traguardo.

Giovanni Sanna.

Computational resources provided by hpc@polito (http://hpc.polito.it)

Acronyms

A* A star

AI Artificial Intelligence

ANN Artificial Neural Network

CPU Central Processing Unit

CS Case Study

DLS Deep-Limited Search

DNN Deep Neural Networks

DQN Deep Q Network

GB Giga-Byte

GHz Giga-Hertz

GD Gradient Descent

GPU Graphic Processing Unit

IoT Internet of Things

MDP Markov Decision Process

MLP Multi-Layer Perceptron

MNIST Modified National Institute of Standards and Technology

MSE Mean Squared Error

MS MicroSoft

NN Neural Network

OOP Object Oriented Programming

PoC Proof of Concept

ROS Robotic Operating System

SAR Search And Rescue

SL Supervised Learning

RAM Random Access Memory

RL Reinforcement Learning

SGD Stochastic Gradient Descent

UAS Unmanned Aerial System

UAV Unmanned Aerial Vehicle

VP Vice President

List of Symbols

A	Number of exploring agents/UAVs
$\vec{a^l}$	activation vector, also knwon as input vector or feature vector or state vector, containing all the activation components a_j^l of the layer l
a_j^l	activation of the of the j-th neuron of the l layer
C	Cost function or loss function
C_0	Cost value for a particular prediction
E	number of collected experiences. Usually referred to the number of training samples plus the number of validation samples
K	number of clusters in the K-means algorithm
l	generic layer l of the the neural network
L	number of total Layers of an artificial neural network
N_{trng}	number of total training samples
N_c	number of total output classes
o_i	a generic i-th centroid
O	total centroids set
p_i	a generic i-th partition of a set
P	total partition set
U	number of unexplored/target cells of a map
w_{ij}^l	weight of the edge connecting the i-th neuron of the $l-1$ layer with the j-th neuron of the l layer
z_i^l	intermediate value of the first step of calculus of Equation 2.1. This value belongs to the i-th neuron of the $l-1$ layer

List of Tables

List of Figures

Bibliography

[1] H. Choset. *Coverage for robotics - A survey of recent results*. Springer, 2001 (cit. on p. 1).

[2] E. Galceranm M. Carreras. *A Survey on Coverage Path Planning for Robotics*. ScienceDirect, 2019 (cit. on p. 1).

[3] X. Zhou et al. *Survey on path and view planning for UAVs*. ScienceDirect, 2019 (cit. on p. 2).

[4] Lisane B. Brisolara Tauã M. Cabreira and Ferreira Paulo R. Jr. *Survey on Coverage Path Planning with Unmanned Aerial Vehicles*. ResearchGate, 2019 (cit. on p. 2).

[5] S. Godio et al. *A Bioinspired Neural Network-Based Approach for Cooperative Coverage Planning of UAVs*. information, 2021 (cit. on p. 2).

[6] Valent et al. *Near-optimal coverage trajectories for image mosaicing using a mini quad-rotor over irregular-shaped fields.* information, 2013 (cit. on p. 2).

[7] Sadat et al. *Fractal Trajectories for Online Non-Uniform Aerial Coverage*. information, 2015 (cit. on p. 2).

[8] L. Marques S. Dogru. *A*-Based Solution to the Coverage Path Planning Problem*. Springer, 2017 (cit. on p. 2).

[9] J. Valente et al. *Aerial coverage optimization in precision agriculture management: A musical harmony inspired approach*. IEEE, 2013 (cit. on p. 2).

[10] M. Kapanoglu et al. *Pattern-Based Genetic Algorithm Approach to Coverage Path Planning for Mobile Robots*. IEEE, 2020 (cit. on p. 2).

[11] et al Z. Chibin. *Complete Coverage Path Planning Based on Ant Colony Algorithm*. IEEE, 2008 (cit. on p. 2).

[12] S. Gotovac et al. *Visual-Based Person Detection for Search-and-Rescue with UAS: Humans vs. Machine Learning Algorithm*. Research Gate, 2020 (cit. on p. 7).

[13] F. Rosenblatt. *The perceptron: a probabilistic model for information storage and organization in the brain*. 1958 (cit. on pp. 8, 13).

[14] H. White K. Hornik M. Stinchcombe. *Multilayer feedforward networks are universal approximators*. 1989 (cit. on p. 15).

[15] M. Adil et al. *Effect of number of neurons and layers in an artificial neural network for generalized concrete mix design*. 2020 (cit. on p. 40).

[16] K. P. Murphy. *Machine Learning: A Probabilistic Perspective*. 2012 (cit. on p. 42).

[17] P. Hart et al. *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*. IEEE, 1968 (cit. on p. 60).

[18] M. Quigley et al. *An open-source Robot Operating System. In Proceedings of the ICRAWorkshop on Open Source Software*. 2009 (cit. on p. 78).

[19] et al N.P. Koenig. *Design and use paradigms for Gazebo, an open-source multi-robot simulator*. 2004 (cit. on p. 78).