



Politecnico di Torino

I FACOLTÀ DI INGEGNERIA
Corso di Laurea in Ingegneria Aerospaziale

TESI DI LAUREA

Neural networks for modelling and guidance of UAVs in urban environments

Candidato:
Francesco Piccoli

Docenti:
**Alessandro Rizzo
Carlos N. Perez M.
Elisa Capello**

Contents

1	Ringraziamenti	3
2	Abstract	4
3	Introduction	5
3.1	State of the art	6
4	Chosen model and architecture	7
4.1	Initial idea	7
4.2	Final implementation	8
4.3	Environment description	8
4.4	UAV model	9
4.5	Deep reinforcement learning and neural network model	10
5	Code and user interface	14
5.1	Code description	14
5.1.1	Training	14
5.1.2	Partial training	15
5.1.3	Main	15
5.2	User interface	16
6	Results	17
7	Difficulties	21
8	Software versions	22
9	Conclusions	23

1 Ringraziamenti

Un giorno scrissi che la vita universitaria era come un viaggio in mare. Ci sono periodi di vento sfavorevole in cui sfrecci sull'acqua, momenti di bonaccia dove non ti muovi di un metro e momenti di vento contrario, in cui ti sembra di perdere tutti i progressi fatti. È proprio allora che devi affidarti alla tua barca e al suo equipaggio, ai tuoi compagni di viaggio che stanno lottando insieme a te per portarti alla fine del percorso. E quando alla fine arrivi a destinazione, Itaca ti avrà dato un bel viaggio e la consapevolezza di essere approdato nel posto giusto.

Oggi sono arrivato a Itaca, e come tradizione vuole, sono pronto a rimettermi in viaggio verso nuove esperienze e conoscenze. Un parte dell'equipaggio mi seguirà, e quel viaggio lo continuerà a fare al mio fianco, nel mio cuore e nei miei pensieri. Di certo non sarei qui oggi senza le persone che ho incontrato durante la traversata, e che insieme a me hanno deciso di sfidare Poseidone ed Eolo.

In primis i miei genitori, Andrea e Serenella, che mi hanno incoraggiato a e accompagnato fin dall'inizio, donandomi la barca con cui salpare, credendo che ce l'avrei fatta. E quando la barca cominciava a imbarcare acqua sono sempre stati in grado di ripararla e farla sfrecciare più veloce di prima. Insieme a loro i nonni, che nonostante sia ormai il 2019 inoltrato, si sono sempre preoccupati per la mia sazietà, ricordandomi delle mie radici e dei sapori di casa. Subito dopo vengono loro, i miei compagni di viaggio. In primis Ilaria, la più intima, forse quella che mi ha cambiato più di tutti, quella con cui ho condiviso soddisfazioni, pianti, gioie e sforzi da ormai un anno e mezzo. Poi i miei compagni di Camplus, che sono stati la mia quotidianità e la mia motivazione negli ultimi tre anni, seduti al mio fianco sulle sedie dell'aula studio. Last but not least, i miei amici ferraresi, con cui basta una frase dal nulla o un'immagine per scatenare una risata e che nonostante le centinaia di chilometri di distanza sono sempre stati un supporto vicino.

Ora inizia un'avventura nuova, e dentro di me sento che con questo equipaggio sono pronto per spingermi oltre le mie colonne d'Ercole alla ricerca di nuova virtù et conoscenza.

2 Abstract

Il lavoro di un ingegnere è orientato alla semplificazione delle nostre vite. Le macchine sono nate per aiutarci nei lavori più duri e usuranti e, negli anni, le abbiamo introdotte in molti aspetti della nostra quotidianità, cercando di affidare loro un livello di indipendenza sempre maggiore. Uno dei più importanti obiettivi degli ultimi decenni è stato, e tuttora è, il raggiungimento dell'autonomia nei movimenti di un veicolo, sia esso da trasporto persone o merci, stabilmente a terra o volante. Perché l'obiettivo si possa definire raggiunto, è necessario ottenere soluzioni affidabili ed efficienti per una serie di step intermedi. Uno di questi step è rappresentato dalla determinazione della traiettoria del veicolo tra due punti predefiniti, un ambito di ricerca noto come problema di path planning. Se questo veicolo si muove volando nello spazio tridimensionale senza equipaggio, prende il nome di Unmanned Aerial Vehicle (UAV). Vari metodi sono stati impiegati e nuove soluzioni proposte, ma il problema di path planning tuttora rimane un ambito di studio aperto.

Questo lavoro si pone l'obiettivo di analizzare la fattibilità della determinazione della traiettoria di un UAV attraverso l'utilizzo di reti neurali. In particolare, simulando un ambiente urbano con presenza di ostacoli fissi, si è utilizzato un metodo noto come deep reinforcement learning per ottenere una traiettoria che ottimizzasse il tempo impiegato per coprire la distanza tra due punti della mappa, visualizzando il risultato in una interfaccia grafica sviluppata in linguaggio QML. La parte di calcolo è invece scritta in Python, con l'ausilio della libreria TensorFlow di Google.

3 Introduction

The utilization of robots that can displace autonomously in a given environment have continuously increased in the past years. Potential applications are several and range from delivery vehicles, to robots assistants for elderly people, autonomous cars, surveillance drones, robots removing bombs in military areas, and planet exploration robots.

Privates as Tesla, Uber, Google, or Amazon are investing millions of dollars trying to develop safe, reliable and effective technologies which can help them offer better services at lower prices. One of the main complications and actual goals is represented by cities, where fixed and moving obstacles are abundant and risk of crashing against one of them is high, and would lead to legal problems.

Both for university research and industries, one of the most interesting type of autonomous robot is represented by the Unmanned Aerial Vehicles (UAVs). UAVs are reusable aircrafts designed to operate without an onboard pilot [1]. They do not carry passengers and can be either remotely piloted or preprogrammed to fly autonomously. Autonomous flight is a major target goal for all technologists. The ability to take-off, execute a given mission with high maneuverability and return to its base without human intervention, promises to enhance UAV deployment in many application domains.

In order to accomplish these tasks, the UAV must have perception of the surrounding environment, know with precision its position, have a path to follow from a start to an end point, and a control system. Each of these blocks is a vast research field, so we decided to concentrate this work on path planning.

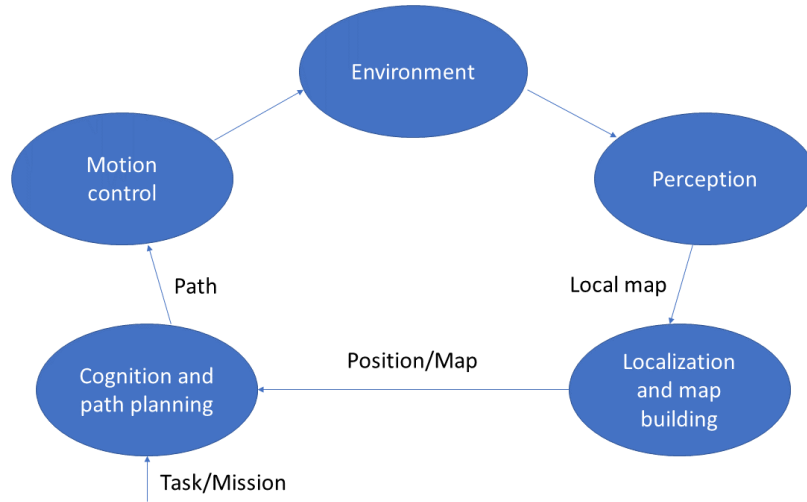


Figure 1: Building blocks of mobile robots navigation

A more detailed taxonomy of unmanned aerial system is represented by the Aerostack architecture, consisting of a layered structure, where every layer correspond to a level of abstraction. According to this classification, path planning is located in the Deliberative layer and is part of the planning system, which is responsible of generating global solutions to complex tasks by means of planning.

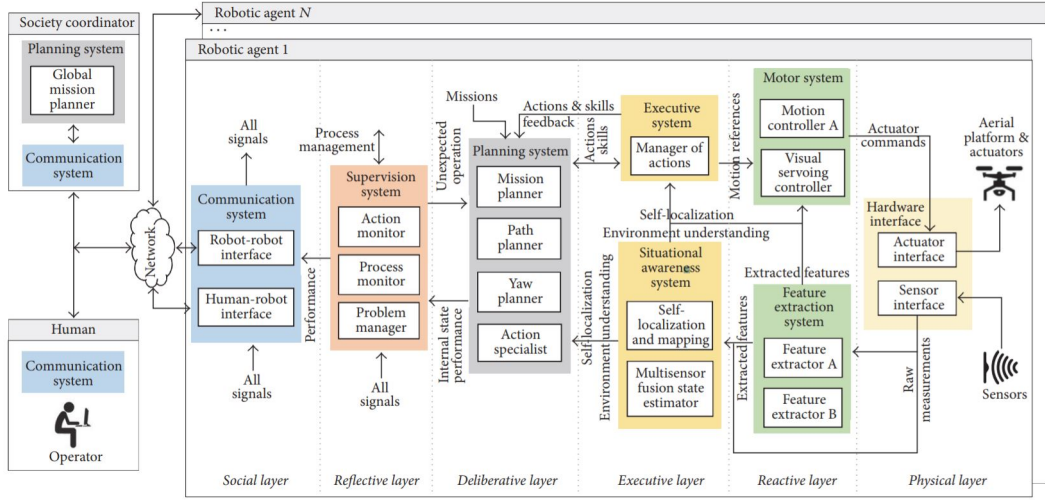


Figure 2: Aerostack architecture [2]

When defining a mission, path planning is the crucial element of the entire system. More precisely, path planning involves the determination of a collision-free global route between a start point A to an end point B, optimizing a performance parameter, such as distance, time, or power. Depending on the kind of information possessed, we can have on-line and off-line path planning. Off-line path planning requires complete information about static and dynamic objects in the studied environment: this is known as *global path planning*. When we do not possess complete information about the surrounding environment, so we do not know the position or velocity of obstacles, the vehicle gets data through sensors and cameras. This approach is called on-line path planning, or *local path planning*. Usually, robots start their path off-line and switch to on-line when a change in scenario is detected.

3.1 State of the art

From the optimization point of view, finding the 3D complete path of a robot is a Non-deterministic Polynomial time hard (NP-hard) problem, so a common solution does not exist. During the years, as reported in [3], two categories of off-line and on-line path planning algorithm have been developed to solve the problem. The first category is known as *classic approach*, while the second one is referred as *evolutionary approach*. While the latter has proven to obtain better results in shorter computational time, the two approaches are often used in combination in order to overcome the NP-hard complexity.

Classic approaches, such as *configuration space (C-space)*, *roadmap (Visibility graph and Voronoi diagram)*, and *cell decomposition* were found to be effective in the determination of collision-free paths, even if they take more time with respect to the evolutionary methods, such as *genetic algorithm*, *particle swarm optimization*, *neural networks*, *ant colony optimization*, or *simulated annealing*, where their common denominator is finding an optimal result starting from zero and evolving over time. The main complication of the classic approach algorithms is that they become incompetent when the studied environment is complex and contain dynamic obstacles. For this main reason, evolutionary approach algorithms are increasingly being used.

Another algorithms classification is the one made [4]. In that paper they classify UAV 3D path planning algorithms into five categories, each of them containing various similar methods:

1. **Sampling based algorithms:** they necessitate environment's information and usually sample it as a set of nodes, or other forms, in order to map it and find an optimal

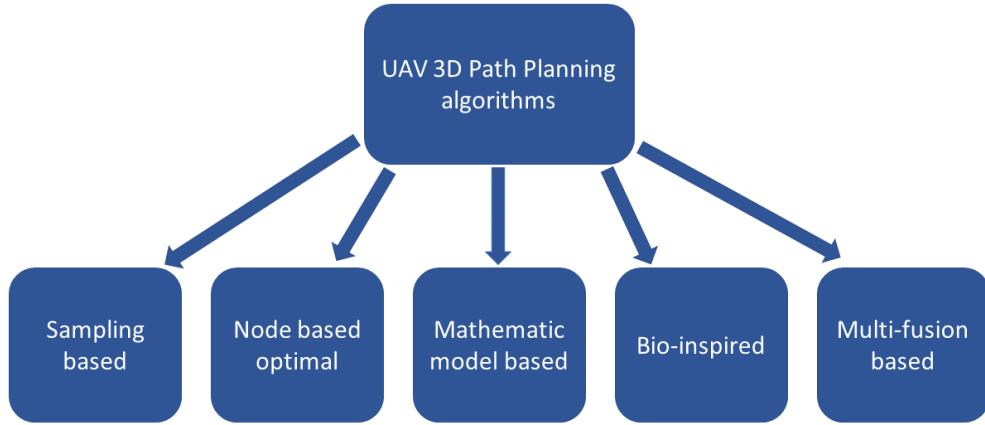


Figure 3: UAV 3D path planning algorithms

path. Among them are *3D Voronoi*, *visibility graphs*, and *artificial potential field*.

2. **Node based optimal algorithms:** they generate paths based on a set of nodes. Among them is the well-known Dijkstra’s algorithm, capable of obtaining the shortest route on a given graph.
3. **Mathematic model based algorithms:** they model the environment, take into consideration kinematic and dynamic constraints and bound a cost function to maximize or minimize in order to obtain an optimal solution. Among them are *linear programming* and *optimal control*.
4. **Bio-inspired algorithms:** they mimic biological behavior to deal with complex problems. They do not build complex environments models, concentrating on finding an effective searching method. They are divided into *evolutionary algorithms* (EA) and *neural networks* (NN). EA imitates natural selection, starting from a random generation, adding mutations and crossovers between the best individuals of each generation, in order to eventually obtain an optimal solution. NN are a set of nodes simulating human neurons. Neurons are divided in layers [5] and each layer is connected with the following and previous ones by connections that remind synapses. Thanks to a training process, connections’ parameters are modified in order to obtain a final result minimizing or maximizing a cost function.
5. **Multi-fusion based algorithms:** they combine existing algorithms, that alone could not achieve an optimal result, to obtain a global optimal path.

4 Chosen model and architecture

For this work, an off-line bio-inspired path planning approach has been used. In particular, a cell decomposition approach is used together with a machine learning technique called deep reinforcement learning.

4.1 Initial idea

The initial idea was to obtain a 3D map of Madrid’s city center and train a drone to go from any point A to any point B of the map. Work was divided into intermediate steps.

First step was starting with a 2D map and a drone’s model only capable of 90 degrees movements. As a starting point an open source Python code [6] of a 2D environment has been used. In this code a rat learned how to move from a fixed point A to a fixed point B (where a piece of cheese was located) of the environment using just 4 actions (North, South, East, West).

Second step was adding 45 degrees movements. Total actions had to augment to 8.

Final step was switching to a 3D environment and add tridimensional actions. Total actions had to augment to 26.

4.2 Final implementation

Neural network’s training has been done using the TensorFlow library, with the objective of minimizing the number of actions and time needed in the path from A to B.

Firstly, a 4x4 matrix simulating a bidimensional urban environment has been created. Using the open source code we obtained the drone arriving to the final cell. At that point, we started complicating the map, increasing dimensions till a 10x10 matrix and having the drone get the target cell.

Next step was adding other bidimensional actions to the initial four. So, we implemented the four missing diagonal movements. The integration was not easy, but it was essential for the following steps, since it helped us learn which parameters and functions of the code needed to be changed in order to add any type of action.

At this stage, we decided to make the transition to a 3D environment. This resulted to be the most difficult step to be implemented, as the initial code and the results’ visualization had been constructed for a bidimensional environment. In order to let the drone move in three dimensions we needed to add other actions and all the limitation to prevent the drone from going out of the map, crashing against an obstacle or make a movement opposite to the latest one. We ended up with 26 possible actions, namely a 3x3x3 cube without the drone’s occupied position. Once obtained a successful transition to three dimensions with a 4x4x4 cube, we started once again complicating the geometry, terminating with a 8x8x8 maze. We also tried a 10x10x10 geometry, but due to the high computational cost it would have taken too long to process the latest step: the possibility to change the final point.

The possibility of changing the initial point was not hard at all. The machine learning technology we used is named deep reinforcement learning, with which the drone learn the optimal action using a statistical process called Markov decision process. In fact, at the end of the training, it is like if every cell of the map contained an arrow indicating to the drone which is the action to take in that cell in order to maximize the reward at the end of its path. Rewards are given to the drone after every action. For the time to be minimum, we decided to give a negative reward after each movement, and this reward is proportional to the required time of that specific action. For example, the drone receives a higher penalty if the action is diagonal 3D than if it was diagonal 2D or lateral 2D, because the covered space, and corresponding time, are higher. When the drone gets to the final point, it receives a positive reward.

So, as every cell contains an ideal arrow representing the action maximizing the final total reward for a chosen final point, moving the initial point is an easy achievement since the drone will already know which direction it needs to take from the new cell. The real difficulty comes when trying to move the final point, as it means breaking the arrow-made maze and make a new one, namely train again the neural network and save its coefficients in another file, which will then be read by the Python code.

4.3 Environment description

Our environment is a three-dimension 8x8x8 array simulating the map of a city and composed of two types of cells:

- Free cells: they are identified with a 1
- Obstacles: they are identified with a 0 and their altitude ranges from 1 to 8

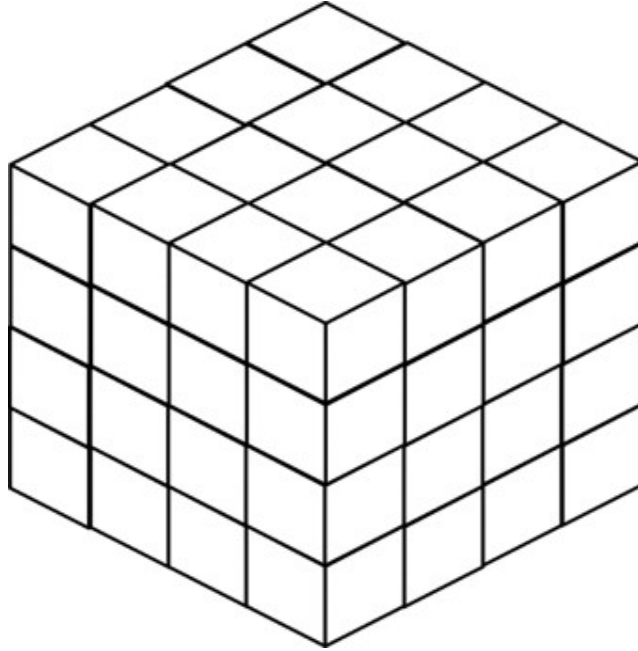


Figure 4: A 4x4x4 map

Selected start and end points must be among the free cells, otherwise an error message is displayed. Moreover, the end point must always be set at ground level, simulating a docking station. Every square cell has been considered to have a 5 meters size.

4.4 UAV model



Figure 5: Un unmanned air vehicle (UAV)

Our agent is a UAV capable of moving along the three dimensions in the created environment. The possible actions are 26 (numbers 9 and 19 are missing for symmetry):

- | | | |
|-----------------|---------------------|---------------------|
| • UP = 1 | • ZP_UP = 11 | • ZM_UP = 21 |
| • UPRIGHT = 2 | • ZP_UPRIGHT = 12 | • ZM_UPRIGHT = 22 |
| • RIGHT = 3 | • ZP_RIGHT = 13 | • ZM_RIGHT = 23 |
| • DOWNRIGHT = 4 | • ZP_DOWNRIGHT = 14 | • ZM_DOWNRIGHT = 24 |
| • DOWN = 5 | • ZP_DOWN = 15 | • ZM_DOWN = 25 |
| • DOWNLEFT = 6 | • ZP_DOWNLEFT = 16 | • ZM_DOWNLEFT = 26 |
| • LEFT = 7 | • ZP_LEFT = 17 | • ZM_LEFT = 27 |
| • UPLEFT = 8 | • ZP_UPLEFT = 18 | • ZM_UPLEFT = 28 |
| • ZP = 10 | • ZM = 20 | |

ZP means going up along the z dimension, while ZM means going down along the same dimension.

The agent has a maximum horizontal velocity of $14\frac{m}{s}$ and a maximum vertical velocity of $5\frac{m}{s}$. Drone's maximum total velocity is equal to its maximum horizontal. Every map's square cell has $5m$ sides. Battery consumption is proportional to time required for the action. We adopted the following rewarding scheme for our problem.

1. Rewards are ranging from -0.9 to 1.0.
2. Every drone's move will be rewarded with a negative amount (penalty) proportional to the needed time for that action. This would prevent the drone to move around and push it to the target cell in the shortest time possible.
3. North, South, West and East 2D actions cost the drone -0.118 points.
4. Diagonal 2D actions cost -0.125.
5. North, South, West and East 3D actions cost -0.160 points
6. Diagonal 3D actions cost -0.164.
7. When the drone gets to the end point it receives 1.0 point.
8. Attempts to move outside the map's boundaries or against an obstacle are invalid actions and produce a severe -0.9 penalty, so that the drone learns not to execute them.
9. Returning to an already visited cell is a counter productive move and cost the drone a -0.7 penalty.
10. In order to avoid infinite loops, if the drone's punctuation is below a negative threshold (proportional to map's size) the game ends. The policy is taken assuming that under this threshold the drone has already "lost its way" and made so many errors from which it learned enough, and can proceed to a new game.

4.5 Deep reinforcement learning and neural network model

Deep reinforcement learning is a machine learning technique whose mathematical apparatus was developed by DeepMind, an AI startup acquired by Google in 2015 for 500M\$. It consists of an agent which interacts with an environment, trying to find the best action to take in every possible state, before reaching a final pre-defined state.

The agent, in our case a UAV, must balance *exploration* and *exploiting* past experience, called *episodes* in our code, in order to reach its goal. It may fail several times in achieving the objective, but thanks to a correct choice of rewards and penalties it would eventually reach the solution of the problem. This solution is an optimal sequence of states (also known as *optimal policy*) maximizing the accumulated reward. This process is called **Q-learning**. Instead, **deep learning** is a machine learning method based on artificial neural networks (ANN) and it uses layers of nodes in order to extract high-level features from raw data. When Q-learning is used in tandem with deep learning we obtain the method patented by DeepMind as **deep reinforcement learning**.

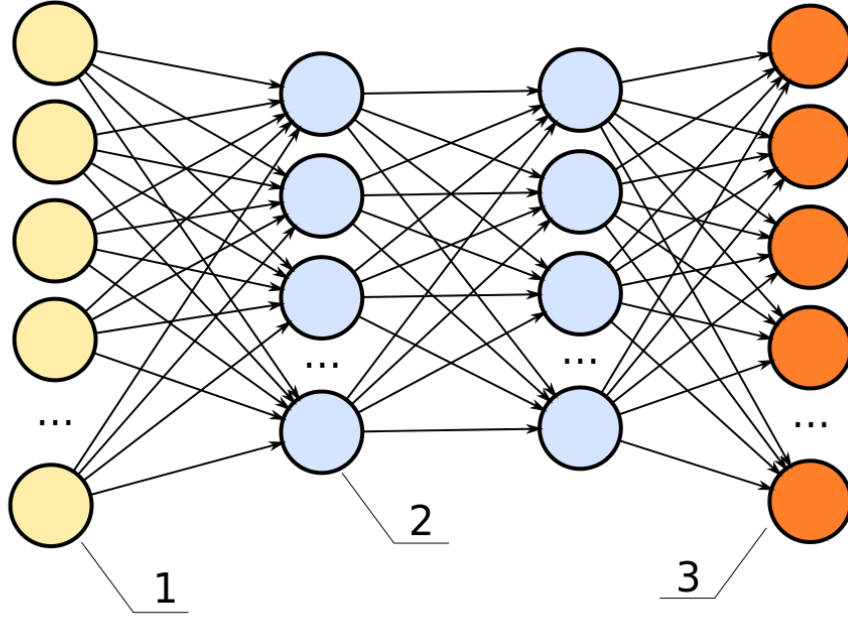


Figure 6: An artificial neural network (ANN) made of an input layer, two hidden layers and an output one [7]

Regarding Q-learning, as previously said, its objective is developing a policy π that permits the agent to successfully move in the map [6]. This policy π is simply a function, obtained after thousands of games, which takes as input the actual agent-environment state and returns the best action to take in order to maximize a given parameter (time in our case):

$$\text{next action} = \pi(\text{state})$$

Starting with a random initial policy, we use it for playing thousands of games from which we hope to learn how to improve π . At the early stages, the policy will for sure contain a lot of errors and make the UAV loose many games. The rewarding policy is responsible for modifying and perfecting the function through a feedback system.

The type of action chosen in the Q-learning process depends on two factors:

- **Exploitation:** actions are chosen based on the previous experience the agent has accumulated. They are dictated by the policy π .
- **Exploration:** actions are taken on a completely random base. It is helpful to experience new path and possibly meet bigger rewards that otherwise we would have not obtained.

The balance between these two factors is represented by the exploration factor epsilon, which is set to the value 0.1 in our code. Its name is due to the fact that it represents the frequency of the exploration actions to take. In other words, every 10 actions the agent will pick a random action from the possible ones.

In 90% of the cases, our UAV will make its choice in an experienced manner, thanks to the policy function π . In general, the agent will move through the map in a finite discrete list of time steps. At every time step t , the agent is in a state s and needs to choose between one of the possible actions a_i (in our case 26). This decision should be completely independent from the previous states and should only depend on the current state s . In literature, this operation is referred to as **Markov Decision Process (MDP)**, a discrete time stochastic process used for modeling decision making.

When we take action a in a state s at time t , we obtain a new state s^* at time $t+1$ and a reward r . Both are clearly functions of the previous state and the action taken:

$$s^* = T(s, a)$$

$$r = R(s, a)$$

where T is called **transition function** and R is the **reward function**.

The UAV's objective is to obtain the maximum total reward at the end of the path. In order to accomplish it, the total sum A of the rewards r_j has to be maximum:

$$A = R(s_1, a_1) + R(s_2, a_2) + \dots + R(s_n, a_n)$$

This is exactly what the policy function π does. When we apply it to a current s we obtain action a to take in that state to maximize the function A :

$$a = \pi(s)$$

It is like if the policy function assigned to every cell of the map an arrow indicating the agent the action that needs to be taken. This is well illustrated by a policy diagram: Once obtained π , we just apply it to every state until we get to the final point. Unfortunately, obtaining it could be a very complex process, especially for large environments.

DeepMind uses a particular method to find π . It starts by obtaining a **best utility function $Q(s,a)$** , or **best quality function**, from which the name Q-learning comes from. $Q(s,a)$ is defined as the maximum total reward we can get by choosing action a in state s . It can be mathematically proved that such a function exists. The policy function is then easily determined:

$$\pi = \arg \max_{i=0,1,\dots,n-1} Q(s, a_i)$$

Practically, we need to calculate $Q(s, a_i)$ for all the 26 possible actions and then pick the action a_i that produces the highest value of Q . This way we turned the problem of finding π to determining the function Q . This new function has an interesting recursive property that helps us approximate it. It is known as **Bellman's Equation**:

$$Q(s, a) = R(s, a) + \max_{i=0,1,\dots,n-1} Q(s^*, a_i)$$

where R is the previously defined reward function for action a in state s , and s^* is the subsequent corresponding state. It can be proved that if a function satisfies Bellman's Equation, then it has to be the best quality function. To obtain it we build a neural network N with the Google TensorFlow library. We give as an input the agent's state s and obtain as output a vector q containing an approximation of the value $Q(s, a_i)$ for all the possible actions a_i . Therefore, vector q is of the form $q = (q_0, q_1, \dots, q_{n-1})$ and, once the network has been accurately trained, it provides us an approximation of the policy function π with the following three steps:

$$q = N[s]$$

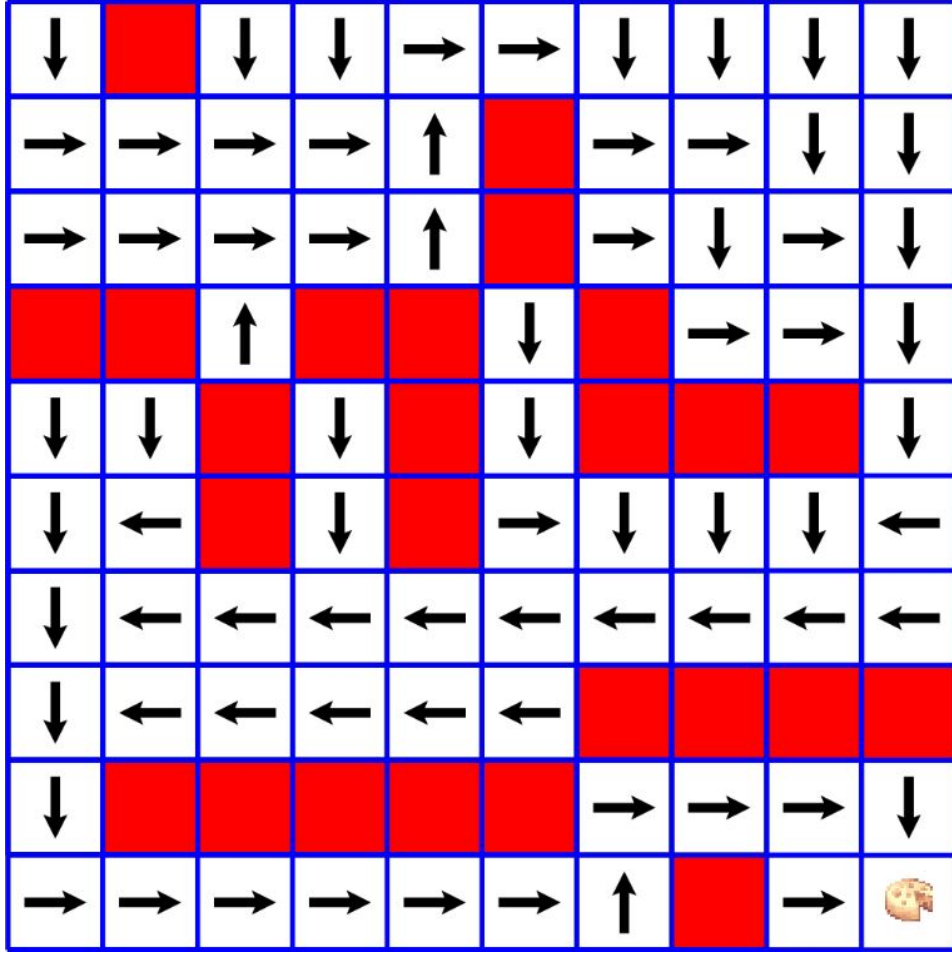


Figure 7: Policy diagram [6]

$$j = \arg \max_{i=0,1,\dots,n-1} (q_0, q_1, \dots, q_{n-1})$$

$$\pi(s) = a_j$$

Assuming that our neural network converges (demonstrations can be found in literature) [6], it will define a function Q which satisfies Bellman's equation, therefore it should be the function we are looking for.

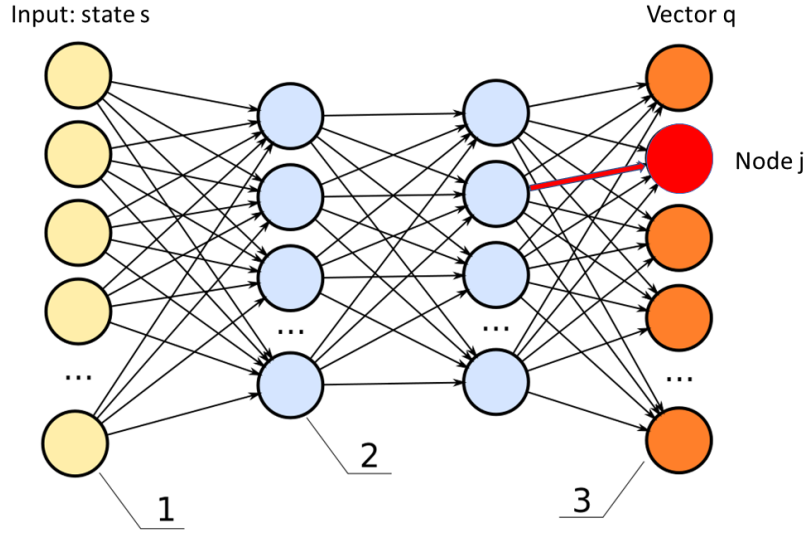


Figure 8: Process for obtaining policy π ; The orange nodes of the output layer represents the possible actions: the chosen action will be the one with the highest connection coefficient.

5 Code and user interface

5.1 Code description

Our Python code is divided into three main parts:

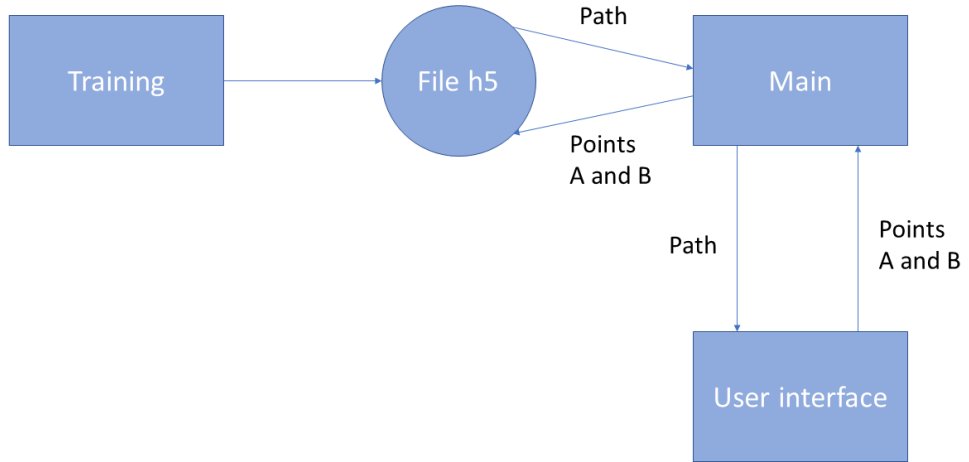


Figure 9: Code scheme

5.1.1 Training

There is a code uniquely used for the neural network's training. It contains the 3D map, which in our case is an 8x8x8 array of zeros (obstacles) and ones (free spaces), and some

drone’s characteristics, such as speed (horizontal and vertical) and unitary power consumption. Moreover, it also contains all the 26 possible actions and the class *Qmaze*. *Qmaze* receives the map (*maze*) as a parameter and has some specific functions, such as *get_reward*, *valid_actions*, *update_state*, which are used in the training process.

This is the central part of the code and it is represented by the function *qtrain*. It consists of a while loop which updates the coefficients of the neural network at every iteration; these coefficients will be saved in a h5 file at the end of the training. Each iteration is called “Epoch”, and each epoch is made of “Episodes”, which are five-element tuples describing the state of the system drone-map after each action.

During each episode this is what happens:

- Out of all actions, the impossible for the actual drone’s position are eliminated (function *valid_actions*), namely the ones that make the drone go out of the map or crash against an obstacle.
- One action is selected among the possible ones in a randomly or experienced manner, depending on the parameter *epsilon*, named exploration factor.
- Drone’s position and system’s state are updated by the function *update_state*.
- Required action’s time is calculated (function *calcolo_tempo*) together with its relative reward, proportional to the required time and always negative, unless the vehicle arrived to the final point.

If the drone arrived to the final point (state *win*), or if its total reward is less than a predetermined value which depends on the maze’s size (state *lose*), the epoch ends and the drone starts a new epoch from a random cell. There is a variable counting the number of times the drone wins or o loses, and training ends when the drone wins for a certain number of times in a row, depending on the map’s size.

Training is made for every final point and the ANN’s coefficients are saved in a h5 file, which will be read by the code named *Principal*. There is an h5 file for every final point, and the code *Principal* reads the h5 file corresponding to the final point chosen on the user interface.

5.1.2 Partial training

The *Partial Training* code is very similar to the previous one, except for the fact that it does not start training from zero, but it starts training the network from coefficients generated from an anterior training, and saved in a h5 file. Code is almost the same, but a lot more shorter is computational time, because the drone already knows how to get to the final point.

5.1.3 Main

In the file named *Principal*, map is also defined and is the same one previously defined in the *Training* file; first of all there is the *Graphics* class, which contains the functions concerning the visualization, in other words the functions which open the User Interface and modify it.

In the User Interface, it is possible to select the points A and B on the map; subsequently, those points are passed to the function *play_game* when we push the button *Calcola!*. This function takes as an input the initial UAV cell, a map of the class *Qmaze*, and a trained neural network, whose coefficients are saved in an h5 file. It reads the h5 file corresponding to the final point B selected and computes the theoretical least-time path between A and B, adding step by step to a variable the vectors corresponding to drone’s position, and calculating the required time, the space travelled and the battery consumed.

The *play_game* function ends when the drone arrives to the final point or when the drone got a total reward less than a pre-established value, in which case the drone did not get to

the final point for any reasons: in this situation an error message is displayed.

```

1 def play_game(drone_cell, arrivo):
2     #build the neural network
3     model = build_model(maze)
4     #get the NN coefficients for the chosen ending point "arrivo"
5     qtrain(arrivo, model, epochs=1000, max_memory=8 * maze.size, data_size=32)
6     qmaze = Qmaze(maze, drone_cell, arrivo)
7     if not drone_cell in qmaze.free_cells:
8         raise Exception("Invalid Rat Location: must sit on a free cell")
9     global spazio
10    spazio = 0
11    global batteria
12    batteria = 100
13    global tempo_totale
14    tempo_totale = 0
15    global elenco_posizioni
16    elenco_posizioni = [drone_cell]
17    qmaze.reset(drone_cell)
18    envstate = qmaze.observe()
19    while True:
20        prev_envstate = envstate
21        # get next action and calculate time and space traveled
22        q = model.predict(prev_envstate)
23        action = np.argmax(q[0])
24        tempo = calcolo_tempo(action)
25        tempo_totale += tempo
26        calcolo_spazio(action)
27
28        # apply action, get rewards and new state
29        envstate, reward, game_status = qmaze.act(action, tempo)
30        h, r, c, s = qmaze.state
31        elenco_posizioni.append((h, r, c))
32        if game_status == 'win':
33            return True
34        elif game_status == 'lose':
35            return False

```

Listing 1: Function play_game

Once drone's path has been obtained, it is displayed on map, together with an altitude graph, which improves visualization, and the space travelled and time required. In the case the user selected other two points A and B and push another time the *Calcola!* button, map updates and shows the new path.

5.2 User interface

In the realization of the user interface our main objectives was making an interface that was the more user friendly as possible, in order to facilitate its usage.

The screen is divided into two areas: on the left there is the map, on the right the commands and indicators (figure 10). Adopted technology is Qt, a specific program for creating dynamic interfaces. In particular, the interface is a *.ui* file. The main advantages are the easy graphic development and the optimal communication between Qt and Python. In the *mainwindow.ui* file all the objects and fundamental properties are declared. With QtDesigner it is also possible to define the relationship between some objects of the screen: for example, if a slider moves, its correspondent numerical value needs to change. The *Principal* Python file contains the screen's commands in the class *Graphics*.

To understand the logic flux of the program, we analyze what happens when the user click on the button *Calcola!*:

- Python execute the *calculate* function
- At the beginning of the function *calculate* there is an *if* which activates the function *allcorrect*

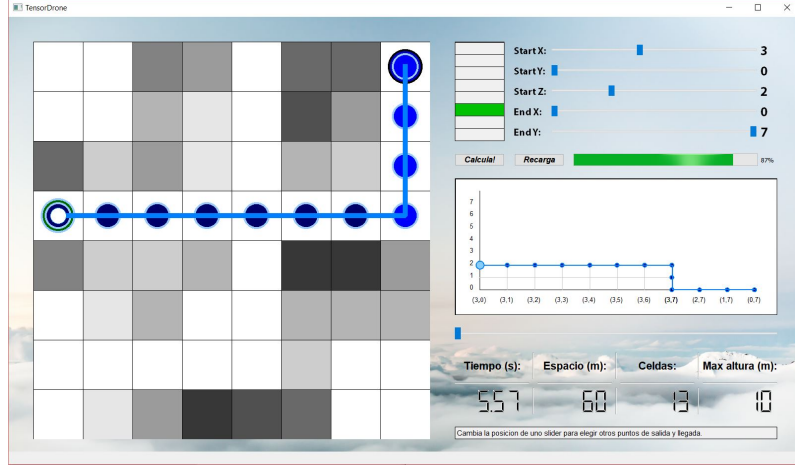


Figure 10: User interface; on the left the map, on the right, from top to bottom, an indicator with drone's initial altitude and underneath obstacle, the sliders to choose start and end points, buttons to calculate path and recharge battery, battery indicator, altitudes graph, time and space indicators and info window

- *allcorrect* receives the chosen starting and ending points and verifies that they are correct (for example, starting and ending points coincide or are inside an obstacle); if the user make a correct initial choice, the function returns a *True*
- *calculate* saves drone's positions in two variables and executes function *play_game*
- *calculate* updates parameters, map, altitude graph, etc. with their functions.

6 Results

In this section we will proceed visualizing some of the obtained results. Selecting as a starting point (3,0,2) and as a final point (0,7,0) (figure 11), the altitude indicator visualizes the starting point's altitude, and also shows the altitude of the obstacle in that cell (figure 12). Let's click on the button *Calcula!*.

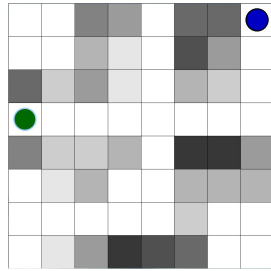


Figure 11: Map with selected points

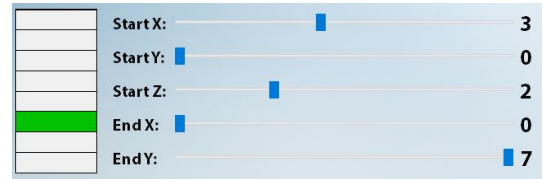


Figure 12: Altitude indicator and position slider

Figure 13 shows the computed path and figure 14 displays the altitudes graph, together with the battery percentage. The selected point is (3,6), where the robot flies at an altitude of 2.

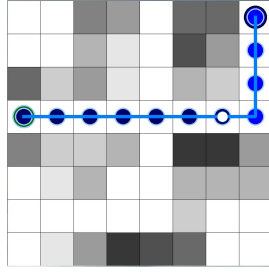


Figure 13: Computed path and selected point (3,6)

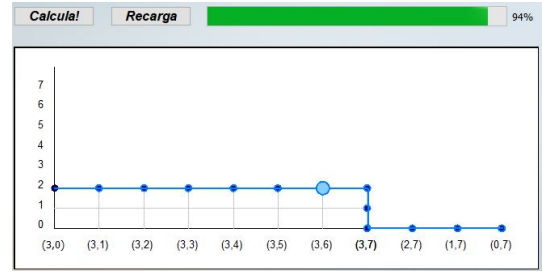


Figure 14: Altitude and battery graphs

Finally, below are shown some path's features: time, total space (every cell correspond to 5 meters), number of cells and maximum altitude. The text below alerts the user with some important messages, for example, when the chosen points are not valid or a power recharge is necessary.

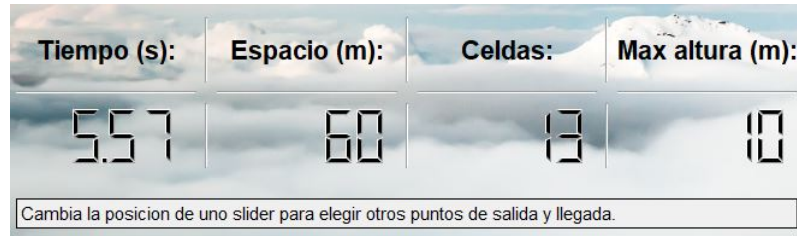


Figure 15: Features and info

Let's analyze some other examples:

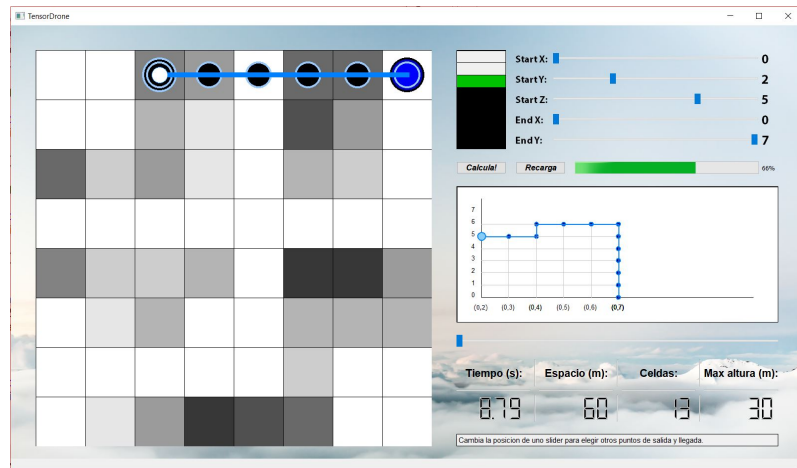


Figure 16: Start (0,2,5), end (0,7,0); below starting point there is an obstacle with height 4

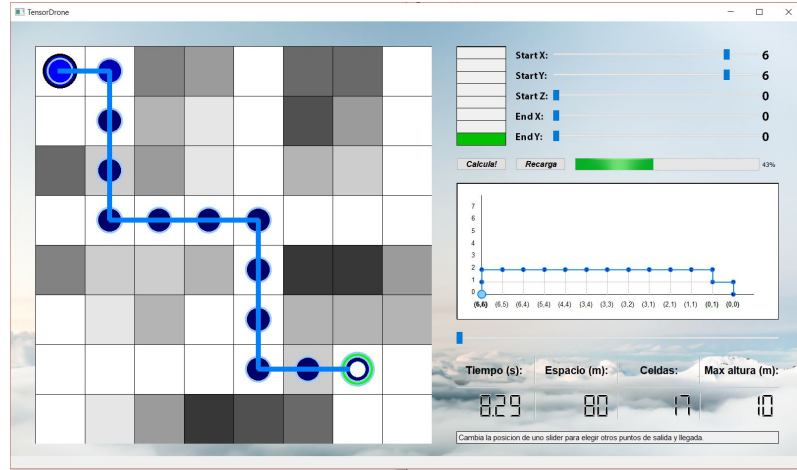


Figure 17: Start (6,6,0), end (0,0,0).

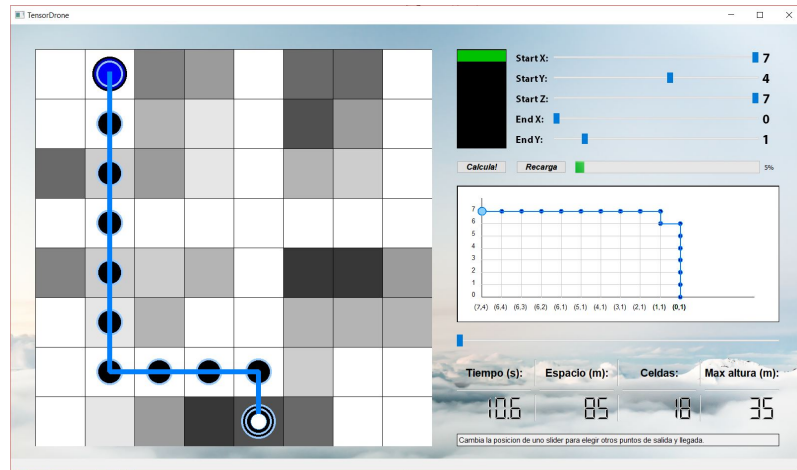


Figure 18: Start (7,4,7), end (0,1,0); below starting point there is an obstacle with height 6

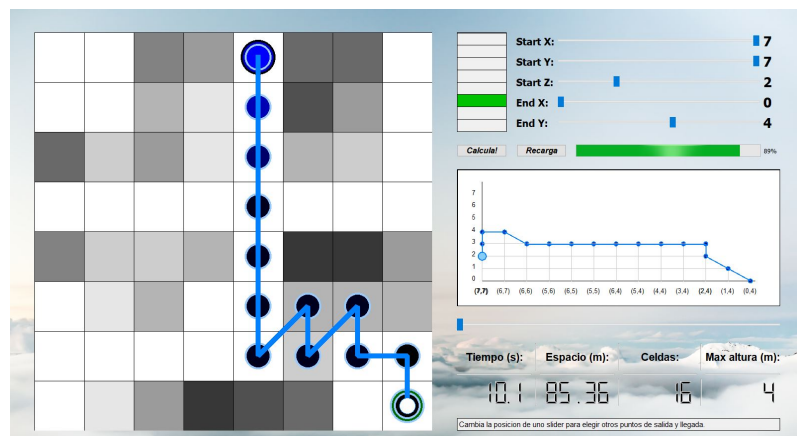


Figure 19: Start (7,7,2), end (0,4,0).

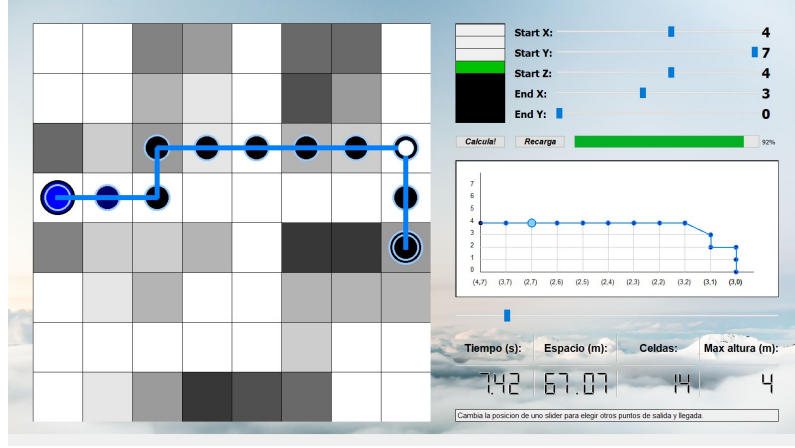


Figure 20: Start $(4,7,4)$, end $(3,0,0)$.

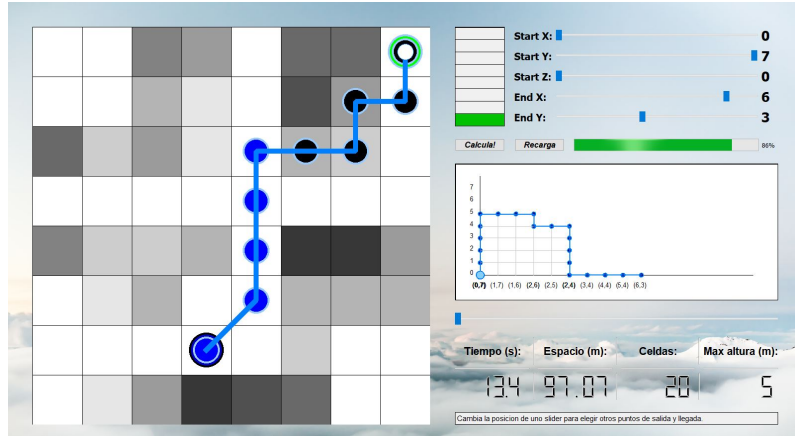


Figure 21: Start $(0,7,0)$, end $(6,3,0)$.

It is immediate to note that some paths seem to be good, while others are clearly not the optimal ones, but appear to be suboptimal. We tried to figure out what the problem was exactly throughout the whole project but we did not find a complete answer. A more detailed comment can be found in the conclusions section.

7 Difficulties

While writing the code the encountered complications were several:

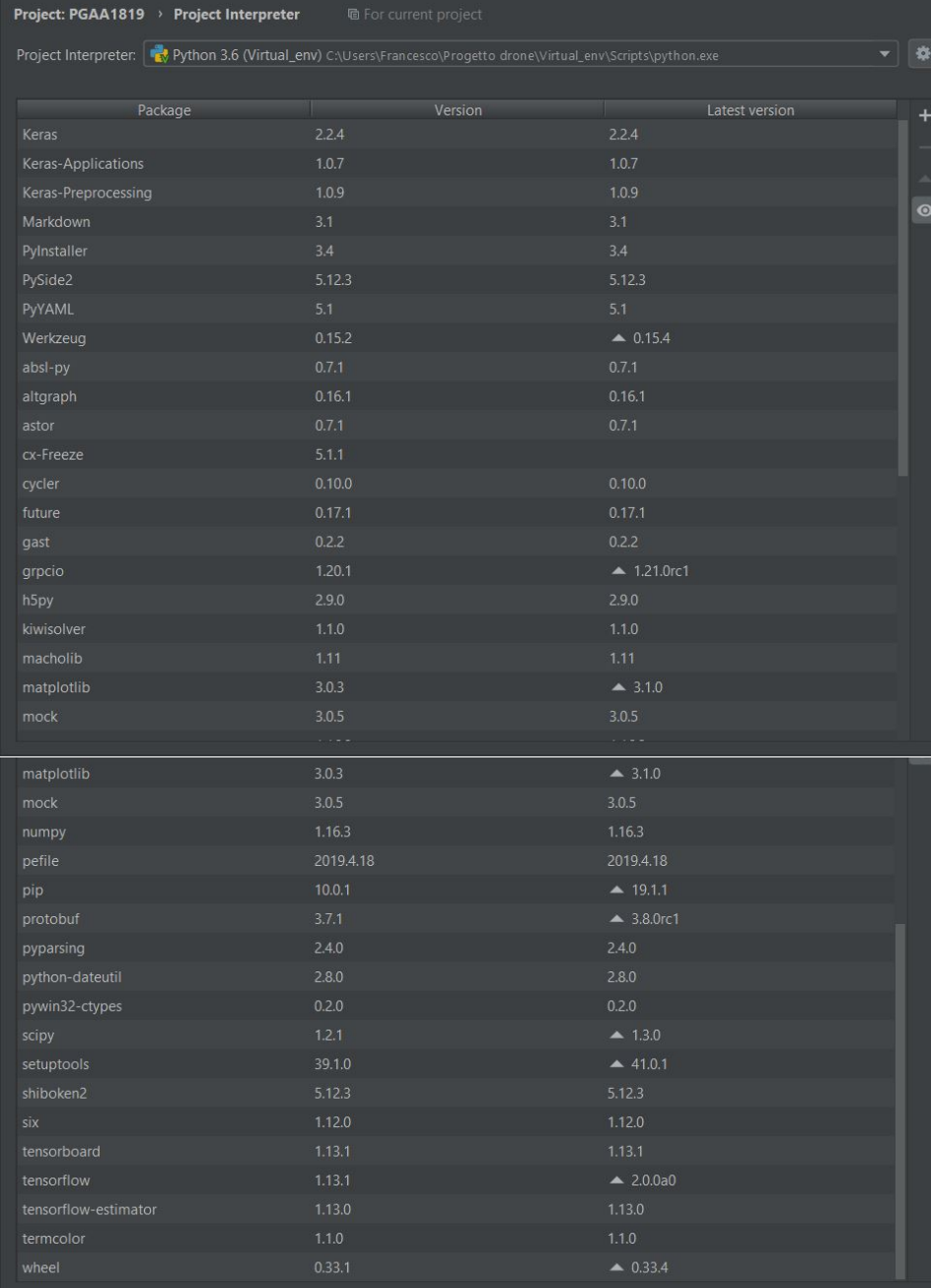
- The first one was installing all the required libraries: we had a hard time trying to install PySide2 and TensorFlow from different platform, and we eventually accomplished it by installing them directly from the Integrated Development Environment (IDE) PyCharm.
- With respect to the code itself, we spent a lot of time trying to understand how the initial code worked, in order to be able to correctly modify it.
- First of all, we had to add a great number of actions and the main problem was modifying the part of the code which controlled that the drone did not crash against an obstacle or go out of map.
- Subsequently we had to add a third dimension (altitude), modify the map and a great part of the code, which had been thought to run in two dimensions. This step was accomplished studying each function and adding the required modifications.
- Once the transition to 3D had been made, another complication was the visualization, which is not as immediate as in two dimensions, making hard to understand if the code is working correctly.
- Moreover, computational complexity in three dimensions is higher than in two, so we were just able to try the code for very small maps before launching the final training for an 8x8x8 maze.
- Training time was huge and we had to leave it training for several days, different times.
- We thought that using TensorFlow GPU instead of TensorFlow CPU would have required less time, nevertheless, as the neural network is read as a vector, doing a lot of small parallel calculations (as the GPU does) it is not necessary. In fact, computational time was much higher using GPU instead of CPU.
- Choosing the program for the user interface was complex. The requisites were a good integration with Python and the possibility to draw and move objects. We had three possibilities: QML, PyQt5, PySide2.
 - QML permits to create an intelligent interface, but its syntax is hard and the integration with Python complex.
 - PyQt5 is a fully developed technology, with several online tutorials. Nevertheless is not very used, as PySide2 is more efficient.
 - PySide2 permits a complete communication between Python and Qt. From Python it is possible to control objects on the screen and from the screen it is possible to activate Python functions. Almost all the logic is developed in Python and Qt just needs to activate Python's functions.
- Selected PySide2 for the user interface, as it is a relatively new technology, it is hard to find online tutorials and examples. Moreover it is not possible to use PyQt5 and PySide2 together, because several libraries have the same names and there is a conflict between them. So, our unique source of information was the official documentation, even if sometimes it was not very complete.
- The last complication was generating the executable file, since the default Python version used by Windows was not well-matched with the downloaded libraries. Additionally, there was a PySide2 wrapper impossible to find by Windows. We eventually accomplished to generate it with PyInstaller.

8 Software versions

For the code implementation on Windows, we used Python 3.6.5. Particularly important are the libraries:

- TensorFlow 1.13.1
- Keras 2.2.4
- PySide2 5.12.3

All the libraries have been installed in a virtual environment created in the PyCharm IDE. A list of all the downloaded libraries of the environment is reported.



The screenshot shows the 'Project Interpreter' window in PyCharm for project 'PGAA1819'. It displays a table of installed packages, their current versions, and the latest available versions. The packages are sorted alphabetically by name.

Package	Version	Latest version
Keras	2.2.4	2.2.4
Keras-Applications	1.0.7	1.0.7
Keras-Preprocessing	1.0.9	1.0.9
Markdown	3.1	3.1
PyInstaller	3.4	3.4
PySide2	5.12.3	5.12.3
PyYAML	5.1	5.1
Werkzeug	0.15.2	▲ 0.15.4
absl-py	0.7.1	0.7.1
altgraph	0.16.1	0.16.1
astor	0.7.1	0.7.1
cx-Freeze	5.1.1	
cycler	0.10.0	0.10.0
future	0.17.1	0.17.1
gast	0.2.2	0.2.2
grpcio	1.20.1	▲ 1.21.0rc1
h5py	2.9.0	2.9.0
kiwisolver	1.1.0	1.1.0
macholib	1.11	1.11
matplotlib	3.0.3	▲ 3.1.0
mock	3.0.5	3.0.5
matplotlib	3.0.3	▲ 3.1.0
mock	3.0.5	3.0.5
numpy	1.16.3	1.16.3
pefile	2019.4.18	2019.4.18
pip	10.0.1	▲ 19.1.1
protobuf	3.7.1	▲ 3.8.0rc1
pyparsing	2.4.0	2.4.0
python-dateutil	2.8.0	2.8.0
pywin32-ctypes	0.2.0	0.2.0
scipy	1.2.1	▲ 1.3.0
setuptools	39.1.0	▲ 41.0.1
shiboken2	5.12.3	5.12.3
six	1.12.0	1.12.0
tensorboard	1.13.1	1.13.1
tensorflow	1.13.1	▲ 2.0.0a0
tensorflow-estimator	1.13.0	1.13.0
termcolor	1.1.0	1.1.0
wheel	0.33.1	▲ 0.33.4

Figure 22: Software versions

9 Conclusions

Working in team has been fundamental for the development of the final result. Work division allowed us to accomplish rapidly parallel tasks and save time. The technologies we employed are quite complex and none of us had previous experience or knowledge in the field of machine learning and path planning. The interest in TensorFlow and neural networks pushed us toward this direction.

This project is a first touch of the machine learning and neural networks worlds, with a glance to one of their applications. For these reasons, in our path planning problem geometries and drone's model are intentionally simple, even though space for future developments is left.

First of all, we need to comment the fact that the robot always successfully win the game, reaching the final chosen point. It is possible to observe that it does not always choose the optimal path, but it seems to pick sub-optimal paths. We first thought it could depend on training time, and that adding some trainings would have also increased performances. However, we noted that, after additional trainings, sometimes the time performances got better, sometimes got worse. It is still not completely clear to us if there is an optimal training time, if we need to set different rewards for actions, or if we need to pick a different neural network scheme in order to improve performances. This would be the first step of investigation in a future development.

A plus in this model of resolution of the path planning problem is surely the fact that we can add variables to the problem, making it every time more complex, without almost changing the training process. This means that, if in the future we decide to develop more our code, the inclusion of new obstacles, even dynamic, or of forbidden or preferred areas is possible without modifying too much the program's structure. Another option could be adding a second agent and study at the same time the two collision-free paths, or choose a more complex and likely drone's model, for example counting an extra time if the robot needs to change direction or stop.

A minus of this method is that computational time is high and it surely cannot be used as an on-line path planning algorithm. Training for 32 final points on a computer lab's normal computer, so with limited computational capabilities, was five days long. In order to use it as an off-line effective method, computational time could possibly be reduced changing the way the map is read. In fact, at a certain point of the code, the map is converted into a single vector and processed with the CPU. If, instead of using this method, we keep the map as a three-dimensional array, we might be able to lower the computational time by using our computer GPU and processing parallel tasks.

As a conclusion, deep reinforcement learning have proven to be able to solve a path planning problem. Nevertheless, building models that quickly converge seems to still be a tough challenge, as also underlined by our results. There still is plenty of space for improvements in this field.

References

- [1] Dalamagkidis, Konstantinos, Kimon P. Valavanis, and Les A. Piegl, "On integrating unmanned aircraft systems into the national airspace system: issues, challenges, operational restrictions, certification, and recommendations", Vol. 54. springer science & Business Media, 2011.
- [2] Carrio, Adrian, et al. "A review of deep learning methods and applications for unmanned aerial vehicles." *Journal of Sensors* 2017 (2017).
- [3] P. Raja, S. Pugazhenth, "Optimal path planning of mobile robots: A review", *International Journal of Physical Sciences* Vol. 7(9), pp. 1314-1320, 23 February, 2012.
- [4] Yang, Liang, et al. "A literature review of UAV 3D path planning." *Proceeding of the 11th World Congress on Intelligent Control and Automation*. IEEE, 2014.
- [5] M. Nørgaard, O. Ravn, N.K. Poulsen, and L.K. Hansen, "Neural Networks for Modelling and Control of Dynamic Systems", Springer, 2003.
- [6] <https://www.samyzaf.com/ML/rl/qmaze.html>
- [7] <https://wp.wvu.edu/machinelearning/2017/02/12/deep-neural-networks/>