POLITECNICO DI TORINO

Master's Degree in Mechatronic's Engineering

Master's Degree Thesis

Reinforcement Learning Application on an Autonomous Quadrotor's Object Seeking Trajectory Planning

Supervisors

Prof. Giorgio GUGLIERI

Prof. Larissa DRIEMEIER

PhD Francesco MARINO

Candidate

Eduardo EIRAS DE CARVALHO

April 2023

Summary

Deep Reinforcement Learning has been successfully applied to several computer games, but it is still rarely used in real-world applications. One of the main reasons is the lack of generalization since it is still challenging to produce robust agents generally capable of RL. This is an even greater issue considering UAVs, since a full assessment of the UAV capabilities is complex. With the current advancements applying reinforcement learning (RL) in robotics and promising methods for producing more robust agents such as curricula and unsupervised environment design, this work proposes to harness the work of designing several environment configurations to train agents with greater capability of generalization. Our approach seeks to automatically generate useful training levels, proposing a new Unsupervised Environment Design (UED) method. This thesis introduces a configurable 2D python quadrotor simulation to ease the training with RL, reaches a marginal step forward in the research of unsupervised environment design, and successfully trains an agent capable of seeking an object while avoiding obstacles in our 2D simulation.

Acknowledgements

Com o fim deste trabalho, se encerra uma fase muito importante da minha vida, de muito aprendizado, conquistas e de muitas relações. Gostaria de aproveitar este momento para agradecer algumas das pessoas que fizeram parte desta jornada e que de alguma forma contribuíram com seu andamento.

Grazie al Prof. Guglieri, al PIC4SeR e al DRAFT per l'opportunità di svolgere questa tesi, che è stata un grande momento di crescita sia professionale sia personale.

Grazie a Francesco, per i tanti consigli, la disponibilità, le tante conversazioni e la motivazione per portarmi a finire questo lavoro.

Thanks to Matteo and everyone at DRAFT for all the time and knowledge that we shared.

Obrigado aos meus pais e meu irmão, que sempre me apoiaram e fizeram de tudo para que eu pudesse seguir meus sonhos.

Ai miei amici di Torino, Alessandra, Alessandro, Alice, Caterina, Martin e Rita, che mi hanno regalato un anno che non dimenticherò mai.

Aos meus amigos Gustavo e Michele, com quem dividi tantas aventuras nesse intercâmbio, e à Giovanna, que sempre me mostra um lado diferente da vida.

Aos meus amigos da mecatrônica Fabiano, João, Luana, Luan, Luciano, Rafael e Vitor. Do time de tênis Fabrizzio, Gabriel, Helena, Koji, Paulo, Rafael, Sophia e Sthefani. E do Turing, sem os quais a faculdade não teria sido a mesma.

Table of Contents

List of Tables

List of Figures

Acronyms

AI
artificial intelligence

A2C
advantage actor critic

DQN
deep q-learning network

LIDAR
light detection and ranging

MDP
markov decision proccess

ML
machine learning

NN
neural network

ReLU
rectified linear unit

RL
reinforcement learning

SF
success factor

SGD

stochastic gradient descent

SLAM

simultaneous localization and mapping

TP

trajectory planning

UAV

unmanned aerial vehicles

UED

unsupervised environment design

Chapter 1

Introduction

The topic of Unmanned Air Vehicles (UAVs) poses a significant research challenge, demanding from different areas such as mechanics, electronics, geolocalization, control and planning. The latter usually requires extensive work in studying and designing the application dynamics for developing and testing the planning solution.

With the advancement of computational power, reinforcement learning has emerged as a leading technique in designing solutions of planning and control in the robotics field [1] [2], thanks to its ease in gathering data from a simulation for training. Indeed, UAVs have gained widespread use across several industries, such as surveillance, aerial photography, agriculture, inspection, construction, among others. But it is usually common that mobile robots must deal with several environment configurations and tasks, which makes the designing of a training environment complex, since once the data is gathered and the agent is trained, it is difficult and expensive to retrain and get a second solution if the task changes.

It is also difficult to model the real world, it has several possible scenarios and edge cases and numerating, studying, and designing all of them is difficult and time consuming. On the other hand the distribution of some parameters in an environment can severely affect the result of an agent trained on them. We also do not want to spend time training an agent on tasks that are not useful or that it already knows how to solve on ease. To avoid wasting time and computational resources on irrelevant tasks or on data that the agent already knows how to solve, it is crucial to develop solutions that handle the generalization of problems efficiently and focus on what is really relevant.

In order to train robots that can handle different environment configurations, without having to study and design every possible configuration, it is necessary to automate the generation of different environments, focusing only on the useful

configuration parameters. The goal is to teach a quadrotor that automatically reaches a target position avoiding obstacles in several environment configurations without specifying them all explicitly.

To this end, this Master Thesis aims to explore the use of Reinforcement Learning in solving the trajectory planning and obstacles avoidance problems for UAVs. The focus will be on developing a system that can handle different environment configurations and tasks without having to design and study each scenario explicitly. The objective is to train the UAV to automatically reach a target position while avoiding obstacles in diverse environment configurations, using only the useful configuration parameters, generated by a Unsupervised Environment Design algorithm. The results of this work will provide insights forthe feasibility of using Reinforcement Learning for UAV trajectory planning and demonstrate its potential for real-world applications.

1.1 Problem Statement

Autonomous drones are an interesting research topic for mobile robotics, control, perception, and path planning since they must be energy efficient, light and small. To be able to fly, they cannot be heavy, but due to their limited computational capabilities, which relies on onboard computers, they must be computationally fast, which consumes relevant power, besides the energy of motors and controllers. Therefore, they also need sufficient energy on the batteries, which can increase its weight. It is easy to notice that designing drones is a hard task, since ground robots usually have less restrictions on weight, can have faster computers and larger batteries. Therefore, an algorithm for an autonomous quadrotor is usually also a good solution for ground robots for example, due to the lack of the above mentioned restrictions.

Regarding navigation algorithms, we can generally say that they are composed by building blocks of localization, control and path planning [3]. Each block is linked to the other to have a fully autonomous navigation system. The planning block needs a working control block to follow the desired coordinates, and the control block needs a state estimation for localization to check if the actions are being followed, additionally, the planning also needs the state to compute next actions. Designing efficient trajectory planning algorithms usually demands a long study on the dynamics of the specific problem and the implementation of complex control laws. The problem with this approach is the lack of generalization in the solution. While performing trajectory planning, an autonomous drone has to plan several movements between rotating, adjusting altitude, accelerating, avoiding crashing

with an obstacle, between others. If you train a drone to inspect agricultural areas in a field without elevation changes, the same algorithm will probably have problems in terrain with altitude changes. Similarly, a drone trained to avoid electricity pylons, which probably would require only rotations, will not be able to surpass an obstacle like a building if necessary.

It is challenging to develop general solutions, but recent work shows an increasing effectiveness in the use of machine learning based algorithms. The role that ML presents in the field of robotics, specially reinforcement learning, is that it gives to the robot the ability to learn, adapt and reproduce behaviors for challenging tasks based on autonomous exploration and learning. Considering industrial robotics, it is way more usual to find solutions based on programming instead of learning, since there is not so much necessity for constant and autonomous adaptation for different situations and constraints. ML gives most robots this ability, the ability to learn rather than just reproduce the programmed actions, and in an even easier way, since it is autonomous, the adaptation is more straightforward than other methods that would probably require the adjustment of already learned parameters.

In the field of autonomous aerial vehicles, the use of machine learning techniques is of much use. UAVs have plenty of constraints that are difficult to model and test, and the challenges faced by them usually require great ability of adaptation and generalization. For this reason, using ML to model and solve UAV tasks has been an increasing topic of research, even since, relatively speaking, artificial intelligence the way it is today is still a new topic with plenty of room for research and improvement. Hence, with the good results shown by recent work, and the known challenge that it is to build robust algorithms for autonomous vehicles, this work aims to discuss and propose a machine learning solution for an important task for autonomous aerial vehicles.

Specifically, the main purpose of this work is to teach a quadrotor to reach a target location avoiding obstacles in a simulation using reinforcement learning with a training algorithm that automatically provides useful environment configurations for convergence. This work does not focus on hardware implementation, energy and computational efficiency, but only on the designing of the trajectory planning problem using Reinforcement Learning methods and the most recent methodology for generalization.

This work proposes to solve the stated problem by creating a computer simulation of a quadrotor, designing reinforcement learning algorithms to train an agent in the simulation and developing a training algorithm that automatically generates different relevant environment configurations to the agent's learning, exploiting the

concept of unsupervised environment design, which will be furtherly discussed in another section.

Chapter 2

Reinforcement Learning Obstacle Avoidance Applications

This chapter aims to introduce the problem of trajectory planning for mobile robots, with emphasis on aerial robots, but also to give a brief introduction to reinforcement learning. This chapter then focuses on the state of the art of reinforcement learning for trajectory planning and obstacle avoidance and then introduces a training approach called unsupervised environment design.

2.1 Introduction to trajectory planning for mobile robots

Mobile robots can be said to be any device capable of locomotion, they can move using wheels, wings, rotors and others. The usage of mobile robots is really extensive, from transport of payloads and organization in distribution centers, cleaning and disinfection in houses or hospitals, fruit and vegetable picking in farms, military usage, to autonomous delivery systems. This section is referenced by [3],[4] and [5].

Flying robots or unmanned aerial vehicles (UAV) are increasingly becoming more common and acting in different sectors, not only in the industry and robotics research, but also for photography and personal use. The main types of aerial robots are fixed wing UAVs, similar to an aircraft, with wings and a propeller; and rotorcraft UAVs, in several configurations, being one of the most popular in the research field, the *quadrotor*. One of the reasons for its popularity is the ability to take-off and land vertically and that it is possible to fly it indoors.

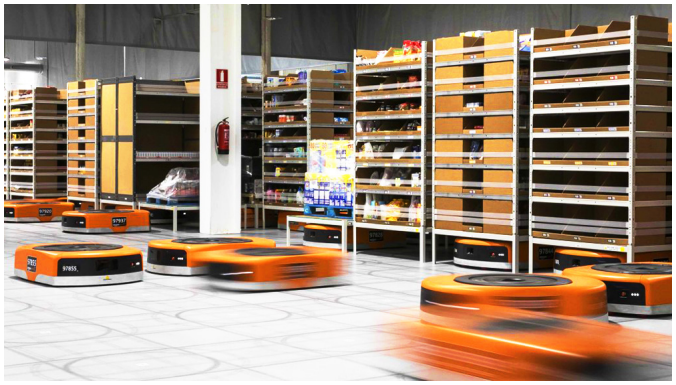


Figure 2.1: Mobile Robot at an Amazon fulfillment center [6]

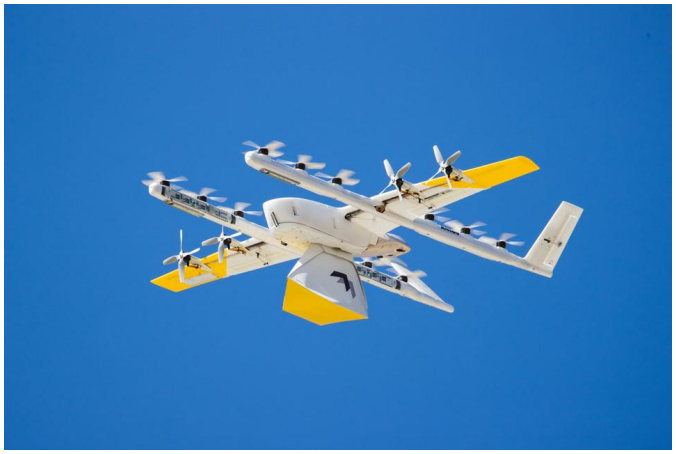


Figure 2.2: Delivery UAV from Alphabet's Wing company [7]

Quadrotors have six degrees of freedom, being three transitional movements, upward and downward, forward and backward, and right and left; and three rotational movements, roll, pitch and yaw.

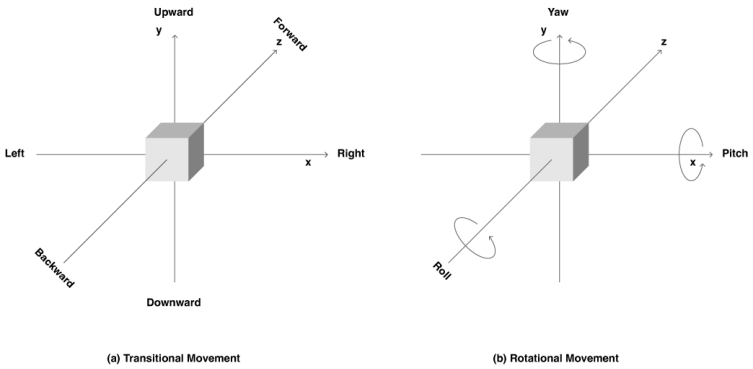


Figure 2.3: Degrees of freedom of an UAV

For manipulating those configurations, flying robots are actuated by forces, in the case of quadrotors, actuated by four accelerations of the rotors. This means it is necessary to use a dynamic model of the flying robot for its control.

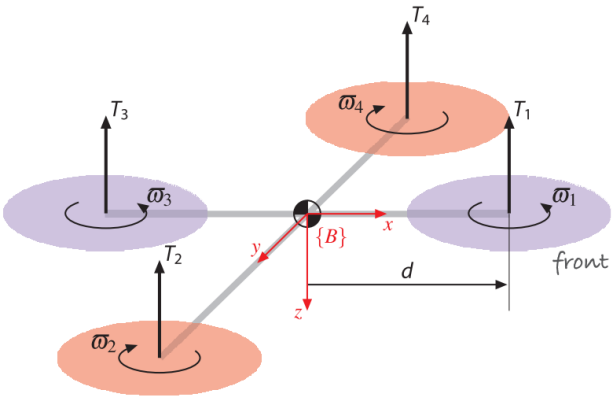


Figure 2.4: Dynamic modelling of a Quadrotor [5]

The process of planning and moving the robot along a path, safely between two points, that is, without colliding, is called navigation. Navigation in robotics, differently to most human approaches, does not necessarily include or are map dependants. This kind of navigation, without a map, is referred to as *reactive navigation*. Some examples include a line-follower robot, vacuum cleaning, or a racing drone that uses gates to localize itself. However, a map is not necessary to successfully complete several tasks, since it is not the only available way for a robot to sense and perceive the environment. In the study of navigation, the tasks are usually divided in three subgroups: *localization*, *motion control* and *path planning* .

2.1.1 Localization

Localization is the process of estimating the location of the robot in the world, or more specifically, in the task environment. The task of localization can vary in difficulty depending on the problem and also in the available information. A robot in the outdoors with a GPS available probably does not face major difficulties to estimate its location. However, besides the inaccuracy of the GPS, in indoor tasks, problems where a map is not available or situations where there are few sensors are more likely to depend on advanced techniques to estimate its location. Another aspect of localization is that it is often required not to know the global position of the robot, but the relative position of some object, or target. Those tasks also require the robot to understand the data from its sensors to exploit relevant information, a process known as *perception*.

One way of estimating a mobile robot position is through the integration of the robot's velocity, if it is known, over the sampling interval Δt, which consists in the technique of *odometry*. For a differential robot for example, it is possible to update its location through integration by knowing the traveled distances for each wheel, which can be measured by an encoder.

A more advanced way of estimating the robot's state is by using a *Kalman filter*, which combines the information from the measuring devices, the knowledge of the system and measurement errors and uncertainty in the system modeling. At each step the system gathers information through its sensors, predicts the state with the dynamical modeling and then combines it by filtering the information. The result is a more robust estimation of the robot's state, but requires modeling the dynamics of the problem and estimating the system's uncertainty and measurement errors.

Though maps are not always available, it is possible to localize itself by constructing a map while facing the task, that is, to autonomously build a map. A popular technique of location while also building the environment map is called *Simultaneous Localization and Mapping* (SLAM). Starting from an arbitrary point, the robot physically explores the environment and estimates its location while also gathering information from its sensors to build a map, and localize itself relative to this map.

2.1.2 Path and Trajectory Planning

Path planning is the task of finding an optimal path from the initial position until some desired goal, or target, position. Trajectory planning has the same idea of path planning but with the added time constraint, that is, instead of only geometrical goals there is also a restriction in time. An ideal path or trajectory

planning must be good enough to deal with adversities and uncertainties in the way. Therefore, given the robot's perception of the world, it is also important that it is able to sense and change the trajectory to avoid obstacles in the way to prevent any impact that can prevent the robot to reach the goal or damage itself. The previous definition is called *obstacle avoidance*. Hence, in order to successfully fulfill the task of reaching a target position, it is not only necessary to plan a trajectory or path, but also to gather information during it, perceive the environment and react to any eventualities, since there can always be imperfections in the modeling of the problem, noise in the sensors and even due the fact that the world is dynamic, not static.

One way of solving the path planning problem is through *graph search*, which consists of constructing a graph configuration of the environment and then finding the optimal path between the initial and final points. Though there are several algorithms to solve this task, only the A* (pronounced "a star") algorithm is going to be shown in this work. A* combines ideas of the *Dijkstra's Algorithm* by looking for vertices that are closer to the starting point, and ideas of using heuristics like in *Greedy Best-First-Search*, by looking for vertices that are close to the goal. That is, A* uses the actual distance from the starting point combined with the estimated distance to the goal. Below follows the A^* algorithm, based on [8].

Algorithm 1 Implementation of A* algorithm

```
1: procedure A*(Queue)
2:     ▷ Queue contains the root
3:     ▷ Initialization
4:     path dictionary ← {}                                   ▷ Empty
5:     cost_current ← 0                                       ▷ Execution
6:     while Queue is not empty do
7:         Get current from Queue
8:         ▷ Gets the path from the priority queue
9:         if current is goal then
10:            Break
11:        end if
12:        for All neighbors do
13:            cost_new ← cost(current) + cost(current, neighbor)
14:            Update cost_current
15:            Update Queue on heurist
16:            Update path dictionary
17:        end for
18:    end while
19: end procedure
```

Therefore, A* works by searching the graphs by a priority queue based on heuristics of the estimated distance to the goal and the traveled path from the start.

Another approach for path planning is *potential field planning*, which creates a gradient across the environment's map that directs the robot though the target position. The robot is seen as a point in the influence of a potential field and follows the direction of the field just until a point of minimum, which is the goal position. The obstacles are designed as peaks in the field, repelling the robot away from it.

Both graph search and potential field planning solve both the path planning and the obstacle avoidance problem, the first can find optimal solution and the later can also be used to derive a control law for the robot, since the potential field can be understood as forces actuating in the robot. However, both solutions need prior information of the environment, such as the map, which in many cases is not possible to have. As mentioned above, there are techniques that can construct a map during the locomotion, but this requires an expensive amount of time and that the robot explores the entire environment. In the recent development of robots, several approaches use a combination of perception with machine learning techniques to solve the problem of online trajectory planning.

2.1.3 Motion Control

After the robot is able to understand its relative position and where it should go, it is necessary to assure that the given inputs to the motors or rotors are actually leading the robot to the desired location. In order to guarantee that the followed trajectory is in fact the desired one, a control strategy is necessary.

One strategy to make the robot go from its initial location to a final desired location is to divide the computed path into several segments and apply those inputs. If the actual robot trajectory is not being measured, or at least estimated, this is called an *open-loop control*. The main disadvantage for this technique is that it is not possible to automatically adapt the trajectory if there is any error or changes in the environment. A more robust approach is to use a feedback control, a so-called *closed-loop control*, in which the robot's state is fed back to the computation of inputs for the motors. In that way it is possible to track trajectory errors and automatically readapt the inputs to correct this behavior.

A common and basic feedback control strategy, known as *Proportional, Integrative, Derivative* (PID) control, consists of minimizing the robot's position (could also be another target variable for a different problem) error, instead of just the

position. The error e will be computed as the desired position p_d minus the actual measure, or estimated, position p:

$$e = p_d - p$$

The PID controller will consist of three parameters K_P, K_I ,and K_D, one for each kind of error, the proportional error, as shown above, the integrative error and the derivative error, which is, in order, the integral of e and the time derivative of e. The control law is:

$$PID(e) = K_P \cdot e + K_I \int e dt + K_D \frac{de}{dt}$$

There are several other feedback control laws, each one with their advantages, but the main reason that PID is so common is that it is easy to implement and does not require a dynamic model of the system.

Without motion control it is not possible in the real world to follow a trajectory planning strategy since the environment is often subject to variations, the sensor readings are noisy and there are probably several simplifications when designing the dynamical model of the system.

2.2 Introduction to Reinforcement Learning

The idea of having machines that automatically solves complex problems for us is not new, in fact, humans have been since the beginning trying to build more and more complex tools for its use. Computers were not different, created to aid us and nowadays most jobs would not even exist without them, but since they were conceived, people wondered if one day computers would be as intelligent as us. In the primitive work of artificial intelligence, the initially solved problems were tasks that could be written as a series of logical commands, usually problems that are more difficult for humans than machines, since they were described by mathematical rules. The difficult problems arise when trying to solve tasks that are not a logical description, but rather something more intuitive for humans. One example is that it is easier to show a computer how to solve mathematical equations than teaching it the difference between an image of a dog and a cat, something that a toddler can do. The first can be described by a list of commands, while the later is something more intuitive, there is no easy mathematical rule to describe what a dog and cat are.

Better solutions for those more intuitive problems came with machine learning, where the computer learns how to solve problems by experience or existing data, which avoid the need of a human to formally describe all the knowledge needed by

the computer to solve it. There are mainly three types of learning in the machine learning field. If the computer learns to map input examples from labeled data into the target variable, that is, the answer, then it is said to be a *Supervised Learning* problem. For example, a set of pictures of dogs and cats, and each one containing its label, if it is a dog or cat picture, and the goal is that, given a picture, whether it is a dog or cat picture.

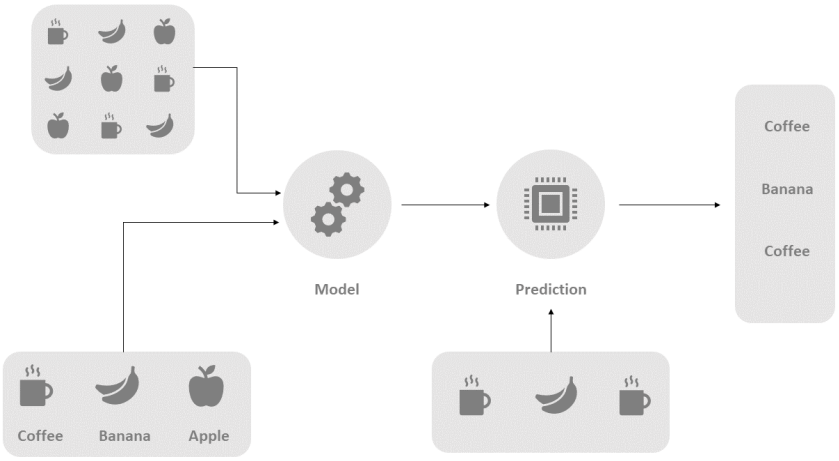


Figure 2.5: Scheme of Supervised Learning, representing a classification problem

When a model is taught to describe or extract relationships in the data, without any labels given, then this is a *Unsupervised Learning* problem, the algorithm learns by itself to make sense of the given data. Given a table of different customers from a shop and their characteristics, an unsupervised learning algorithm can try to identify the different existing types of customers of that shop, without any answer given in the data.

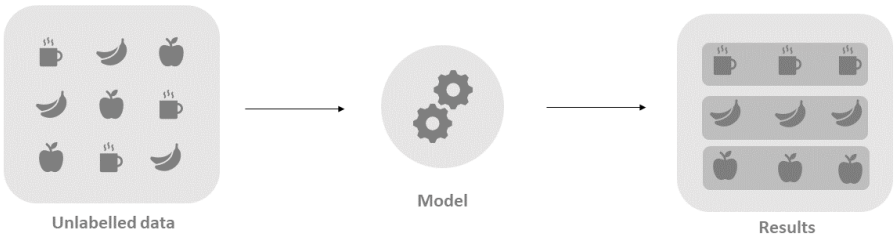


Figure 2.6: Scheme of a Unsupervised Learning problem

Now, when an algorithm is used to interact with an environment by taking actions and receiving numerical rewards for each action taken in each environment state, and tries to learn the best actions to take in each situation, then this is a reinforcement learning problem. *Reinforcement Learning* includes trial-and-error search in an unknown environment and the problem of trading off instant and future rewards, without being taught which actions to take, but only whether these were good or bad actions based on the given reward.

The goal of this chapter is to give an introduction of the reinforcement learning framework, its basics concepts and some of its methods.

2.2.1 Brief story of Reinforcement Learning

The first studies in Reinforcement Learning can be traced back to the 1980s, when psychology researchers began exploring the concept of learning through trial and error. Another thread was the study of optimal control using value function and dynamic programming, although it did not involve learning. The RL known as this day, which focuses on teaching an agent to make decisions in an environment by maximizing a reward signal through trial and error, came with the advancement of those studies.

Over the years, RL has experienced significant growth and development, becoming

a well-established field with numerous real-world applications. Today, RL is the topic of research in fields such as robotics, gaming, and recommendation systems, among others. The advancements in machine learning algorithms and techniques have led to significant improvements in RL, allowing it to tackle more complex problems. The increasing use of big data in various industries, such as healthcare, finance, and retail, has also driven the development of RL, as it allows for the analysis and prediction of large amounts of data. The continued advancement of RL is expected to lead to further breakthroughs in fields of autonomous systems.

2.2.2 Elements of Reinforcement Learning

The basic elements of reinforcement learning are the agent, environment, policy, reward signal and a value function. The given definitions here are based on [9]. The *agent* is the learner and decision maker of the RL framework. Everything the agent interacts with, that is, everything outside the agent, is called the *environment*. In a computer chess game, the agent is the machine taking actions, the environment comprises the board, the adversary and the pieces, thus everything the agent can interact with.

The *policy* is the agent's rule for acting, it maps what it can sense of the environment state into the action to be taken in that particular scenario. The policy describes how the agent should behave. In the reinforcement learning framework, usually the policy is what we are interested in discovering, because it is what describes how to solve a given problem.

The *reward signal* is a way of describing the goal of the problem, it is what at each time step will give a numerical value, the reward, to describe how close or far from the goal a given action is. The agent's objective is to maximize all collected rewards, just as a student wants to maximize all its grades along its semester.

As the *reward* describes how good an action is at each time step, the value function describes the goal in the long term, that is, after many time steps. Each state will have its value, that describes how good it is to be in that state. The value function can act as a prediction of the future rewards, given that the agent chooses the best actions. Therefore, estimating value functions for state is a way of finding how to maximize its reward in the long-run.

As an example of the game chess, by moving the pawn to get the opponent's pawn can give a good reward, but getting the queen will yield an even better reward. Nevertheless, if by getting the queen we end up being in a state where the opponent can give us a checkmate, then the reward was good but the value of that

state is bad.

On the other hand, we can sacrifice a piece, getting a bad reward, but giving a check to the opponent, ending up in a good state. The policy in that example is the set of rules that the agent should follow based on each state.

2.2.3 Markov Decision Process (MDP)

In the reinforcement learning framework, an agent interacts with an environment, taking actions and receiving rewards for them, analyzing those values and doing everything once again. The formalization of this sequential decision making process is a Markov Decision Process, where an action does not only influences the immediate rewards, but also the future returns and future states.

At each time step t=0,1,2,3... the agent has some representation of the environment's state at that time step, called St $\in S$, and takes an action based on that state, $A_t \in A(s)$, receiving a numerical reward $R_{t+1} \in R \subset$, then changing to a new state $S_{t+1} \in S$, repeating this process.

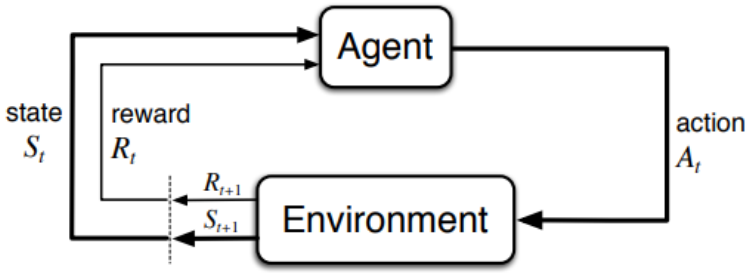


Figure 2.7: Agent-environment interaction in a finite MDP [9]

In most cases we deal with finite MDPs, which means that the sets of states, actions and rewards (S, A, R) all have a finite number of elements. The reason for that is that the real world is too complex to be solvable in an easy way for most problems, it is easier to deal with a model of our problem, and thus it is useful to model it as a finite problem.

For some problems, given a state s and an action a, it is not clear which is the next state s' the agent will fall into and the reward r it will receive, we can define the probability of that happens:

$$p(s', r|s, a) = Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}$$

$$p(S, R, S, A) \rightarrow [0,1]$$

That function p, describing the probability of an agent in the state s taking an action a, receiving a reward r and ending up in the state s' is called the *dynamics* of the MDP. In an MDP, it is said that it has the *Markov Property* if the probability of each possible value for S_t and R_t depend only on the present values of state and action S_{t-1} and A_{t-1}, and not on the past values as well. That is, the next state depends only on the present state and action, not on earlier values. The Markov Property is assumed to solve RL problems.

The goal of the agent in reinforcement learning is the maximization of the expected value of the sum of rewards throughout the long run. The reward signal is the only thing describing the agent's purpose, and the only way of meeting that goal is to maximize the sum of those values, which means the agent is doing a good job. This is called the reward hypothesis. In supervised learning tasks we want to get the right value of the target given our input, the equivalent of that in RL is that given our state we want to maximize the cumulative sum of rewards.

To formalize that idea, we define the *expected return*, which is the sum of rewards for each state the agent has been in, from t to the final step T, which we call episode:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T$$

, this sum starts on $t + 1$ since there is no reward for the initial state.

Each episode consists of the run that the agent makes through the environment, from the initial timestep until the last state, called *terminal state*, in which the episode ends. Every time an episode ends, a new episode begins, not depending at all on the last episode. Tasks that can be easily formulated with a very specific ending, such as chess where the end is a win, loss or draw situation, are called *episodic tasks*. On the other hand, tasks without a defined end, such as the control of a robot's motor, are called *continuous tasks*.

Since an episode can have an undefined and unpredictable number of steps, and that a continuous task may have an infinite number of steps, it is difficult to try to maximize an infinite sum. Therefore, it makes sense for the agent to try to maximize the discounted return, defined as:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Where $0 \leq \gamma \leq 1$ is called the *discount rate*. The discount rate determines the relevance we give for future rewards, that is, how much we prioritize the long term sum of rewards in respect to the instantaneous reward. Higher values of γ give higher importance for future rewards, while if γ equals 0, then the agent is only trying to maximize the current reward.

The above function can also be computed recursively:

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \\ G_t &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + ...) \\ G_t &= R_{t+1} + \gamma G_{t+1} \end{aligned}$$

2.2.4 Policy and Value function

Value functions tell how good it is for the agent to be in that state, most RL algorithms involve estimating value functions, but the concept of good is only based on the future rewards, the expected return. This expected return will depend on the actions chosen by the agent and, of course, it wants to choose the actions that give the best expected return. The way the agent chooses those actions is defined by the policy. Hence, the expected return is dependable on the policy that the agent is following.

The policy is a function that will map the current state into the probability of selecting an action at that time step. Policies will be denoted by π. So $\pi(a|s)$ is the probability that the agent chooses the action given that the current state is s.

The *value function* of a state s under a policy π is the expected return if an agent starts at the state s and follows the policy π.

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$$

If the agent follows a policy π indefinitely, saving each reward if received, computing an average of the returns it found from each state, then this average will convert to the actual value function. Methods that use this procedure to estimate $v_\pi(s)$ are called *Monte Carlo Methods*.

It is also possible to define the expected value of taking an action a in the state s under the policy π, called *action-value function*, defined as:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$$

The relation between the two above functions is given by just multiplying $q_\pi(s, a)$ by the probability of taking action a at state s given by the policy, for all actions:

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s, a)$$

Another useful way of computing $v_\pi(s)$ is relating it with the value of the next state:

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\
v_\pi(s) &= \mathbb{E}_\pi[R_{t+1} + G_{t+1}|S_t = s] \\
v_\pi(s) &= \mathbb{E}_\pi[R_{t+1}|S_t = s] + \mathbb{E}_\pi[G_{t+1}|S_t = s] \\
v_\pi(s) &= \sum_r \sum_{s'} \sum_a r \cdot \pi(a|s) \cdot p(s', r|a, s) + \\
&\qquad \gamma \cdot \sum_r \sum_{s'} \sum_a \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s'] \cdot p(s', r|a, s) \cdot \pi(a|s) \\
v_\pi(s) &= \sum_a \pi(a|s) \sum_{s', r} p(s', r|a, s)[r + \gamma v_\pi(s')
\end{aligned}$$

This resulting relation is called *Bellman equation* for $v\pi$, and express the relation between the value of a state and the values of its successor states. This equation acts like an average of all possibilities of rewards and next states the agent could fall into based on the environment model, and all possible actions it could take at the present state given its policy. If the agent is on state s and takes action a, there is a probability $p(s', r|a, s)$ that the next state is s' yielding a reward of value r. We sum for all possible values of s' and r given that probability, then sum for all possible actions in respect to the probability $\pi(a|s)$ that the agent takes this action. For all those probabilities we compute the next reward and the next state's value.

2.2.5 Optimal Policies

When solving a reinforcement learning problem, the goal is to find a policy that yields the maximum of rewards over all timesteps, so basically finding the best existing policy, the one that achieves the higher amount of reward. The policy that is better than or equal to all other policies is defined as *optimal policy*, and there may be more than one, all of them are denoted equally by $\pi*$ A policy π is said to be better than or equal to another policy π' if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in S$.

Therefore, the value of the the optimal policy is:

$$v_*(s) = max_\pi v_\pi(s)$$

For the optimal policy, the expected return is the sum of all rewards achieved from the current state by following an optimal policy, hence, it is equal to the value of the next state. Thus, applying the Bellman equation for the optimal policy we get:

$$
\begin{aligned}
v_*(s) &= max_a q_{\pi_a}(s,a) \\
v_*(s) &= max_a \mathbb{E}_{\pi_a}[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\
v_*(s) &= max_a \mathbb{E}_{\pi_a}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \\
v_*(s) &= max_a \sum_{s',\gamma} p(s', r|a, s)[r + \gamma v_*(s')]
\end{aligned}
$$

This equation is called Bellman optimality equation for v_*. This relation is actually a system of equations, since there is one Bellman optimality equation for each state of the system, therefore, for finite MDPs there is a unique solution, which is possible to solve if the dynamics $p(s', r|a, s)$ of the environment is known.

Usually this dynamics is not known and the environment has a large state space, making it hard to find the optimal policy. Reinforcement Learning methods try to solve this problem in different ways to find the optimal policy, which usually rarely happens, or a policy that is good enough even if the environment dynamics is unknown and the state space is large or even infinite.

Taking again chess as an example, some estimates say that there are 10^{46} [10] legally possible chess configurations, being impossible with current computers to compute the optimal policy for chess. It is then necessary to compute approximate estimates of value-function to find a good policy for chess, which is usually done with RL achieving great results, even because there are usually states, or chess board configurations, with improbable chance of happening, and there is no great gain in computing the value for those states, there is a higher gain in having good estimates for state that happen more often.

2.2.6 Model free learning

In reinforcement learning, a model-free learning algorithm is an algorithm that does not use the the transition probability distribution $p(s', r|s, a)$, which, together with the reward function of the MDP, can be said to be the model of the problem. Thus, model free algorithms are useful because they do not require a model of the environment to act, being more straightforward to be used, since the process of obtaining all the formulas that describe the problem can be challenging, specially depending on the size of the problem. On the other hand, if it is possible to acquire

the model of the problem, it can be easier to find an optimal solution, since there is more information to solve the problem.

One problem of model-free learning is that it is totally dependent on the data used for training, or on the simulation that the learning algorithm runs, since the model of the problem is not available the algorithm will converge on the distribution presented by the data, and if any dynamics of the real problem is missing in the simulation or the available data, then the policy will not englobe those features.

2.2.7 Tabular and approximate methods

When the state and action spaces are small enough such that the value function can be represented as arrays or tables, it is possible to use *tabular methods* to find the MDP's value function and estimate the optimal policy. Tabular methods can often find optimal solutions, that is, the optimal policy, but only in the stated case where the problem is small enough. When this is not an option, it is needed to use *approximate methods*, which does not necessarily find the optimal policy, only approximate solutions, but can be applied to a higher range of problems.

In the tabular methods, we store each possible state, or state-action pair and explore the environment, saving the received rewards for better estimation of $V(s)$ or $Q(s, a)$. The three main classes of tabular methods are Dynamic Programming, Monte Carlo and Temporal Difference.

Dynamic Programming are model-based methods in which the transition probability distribution is used to compute each value with the values from surrounding states and the Bellman Equation. The process is repeated for every state until it converges.

Monte Carlo are model-free methods in which at each episode we explore through the environment storing state, action, reward and next state at each time step of the episode. At the end of each episode we average the saved values to estimate $V(s)$.

Temporal Difference are also model-free methods that compute the value of the current state based on the estimates of other states at each time step exploring the environment. Both at Monte Carlo and Temporal Difference the exploration is usually done epsilon-greedly.

Approximate methods are useful when the problem presents a large state or action space, since by using tabular methods it would be needed a large amount of memory to store all state-values but also an enormous time to explore several times every possible state. Approximate methods try to estimate those values even if

it has never seen a state in the training process. Those methods benefit from the previous study done in other machine learning fields, function approximation. A common family of methods for approximate solutions in reinforcement learning are *Policy Gradient* (PG) methods. Those methods do not take actions consulting action-values, but rather try to learn the policy directly based on the gradient of some performance measure.

2.2.8 Exploration and Exploitation

One problem that arises in the reinforcement learning framework, not encountered in other types of learning, is the *exploration vs exploitation* trade off, being present in every RL problem. The goal of the agent is to find a policy that maximizes the discounted expected return and, in order to get the maximum possible value for that, it must take actions that reflect what it already knows from previous exploration, that is, the agent exploits to maximize its return in the long run. In other words, the agent chooses the actions that it thinks are the best.

On the other hand, there is a chance that those actions are not the best ones, are just a local maximum and not the global, that maybe there are better actions to be taken. Thanks to that, the agent also needs to explore, take different actions that lead to unknown states or that improves the accuracy of action-values for the policy, and those actions may lead to even better states than the previously learnt policy.

With only exploration or exploitation, the agent will certainly fail. It must alter between those two to find good policies. To solve that, some RL methods use ϵ-greedy action selection. A parameter $0<\epsilon<1$ is defined such that, at learning time, the best-known action is selected with probability 1-ϵ, and a random action is selected with probability ϵ. Selecting an initial value of ϵ close to 1 will lead the agent to initially explore more and learn more accurate action-values and then, decreasing through the learning process will lead the agent to exploit more in the end, following the best-known actions.

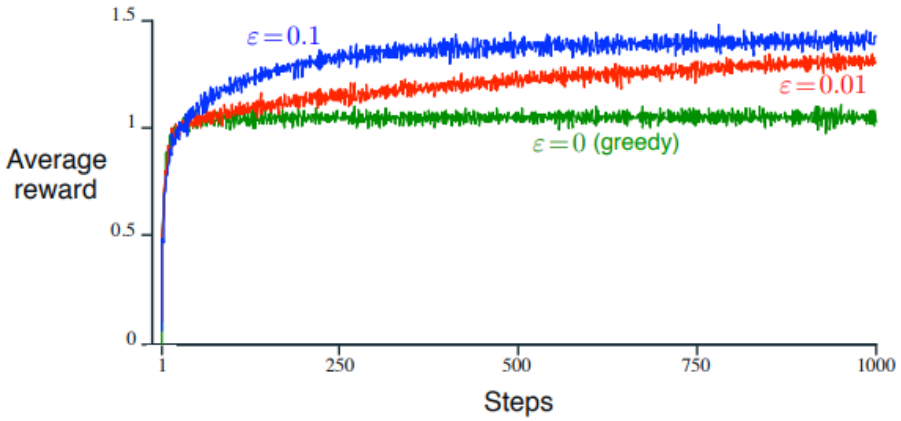


Figure 2.8: Example of the evolution in return for different values of ε in the multi-armed bandit problem [9]

2.2.9 On-policy and Off-policy algorithms

On-policy methods evaluate, or try to improve, the policy that is used to make decisions, that is, the policy used to choose actions based on the current state. Those methods estimate the policy being followed.

Off-policy methods find the optimal policy based on actions chosen by a different policy. They follow actions taken by a second policy to estimate action-values. For example, if a learning approach uses -greedy action selection, than the policy it follows is different from the policy it is trying to estimate, since we are not choosing the best action from this policy, but actually randomly alternating between this policy and a random action, which is by itself a different policy.

2.3 Neural networks and function approximation

As discussed previously, some reinforcement learning tasks have a large state space, where it is not possible to find the optimal policy through tabular methods, and therefore we need to estimate the optimal policy with different methods. One way of approximating nonlinear functions, that is very studied in supervised learning, are neural networks (NN).

The goal of a neural network is to approximate some function $y = f^*(x)$, that is, a map from an input variable x to an output y. The neural network will then learn

some parameter θ to best approximate this function as $f(x, \theta)$.

Neural networks, also referred as feedforward neural networks, have this name because feedforward refers to a series of computations that flows from the input, to some intermediate computations and then the output; the term network because it is a composition of several functions; and neural since the motivation was to somehow represent a simplified working of the brain, though its actual purpose is not to replicate the brain functioning, but rather to learn and generalize functions.

2.3.1 Brief History of Neural Networks and Deep Learning

Neural Networks first appeared as an attempt to mathematically model the principle of brain cells, neurons [11]. The McCulloch-Pitts neuron (1943) was an early linear model of the brain function, but the weights of the linear model had to be set manually by an human operator. Later, the Rosenblatt perceptron arrived in the late 1950s, and was the first model that could learn the weights with the usage of input examples. In the 1960s appeared the first version of the known algorithm called stochastic gradient descent, used to learn weights. Machine learning did not gain much attention at that time since the known algorithms could not learn non-linear functions, such as the exclusive OR (XOR). A wave of advancements in neural networks and machine learning came with the popularization of the back-propagation algorithm and the introduction to nonlinear units in the feedforward NN, and then being actually able to learn more complex functions and train larger models.

Recent history has benefited from the advancements of the study of large artificial neural networks, well known as deep learning, mainly after the pioneer use of deep Convolutional Neural Networks [12] that won the ILSVRC-2012, a visual recognition challenge. One of the reasons why deep learning and neural networks have become more useful lately is due to the increasing amount of available useful training data with the digitalization of society. The world has been storing all kinds of data in the last decades, and now researchers have found a tool that can understand and use all this huge amount of knowledge. Deep learning has been shown to be a very useful method for several applications, but it is highly dependable on large amounts of data. Another explanation of the improvement of neural networks was the increase in its size. The possibility to build larger models with a higher number of layers was only possible with the advancement in hardware and software, mainly on GPU's, that made possible the training of deep learning in large amounts of data in a feasible time.

2.3.2 Neural Networks Architecture

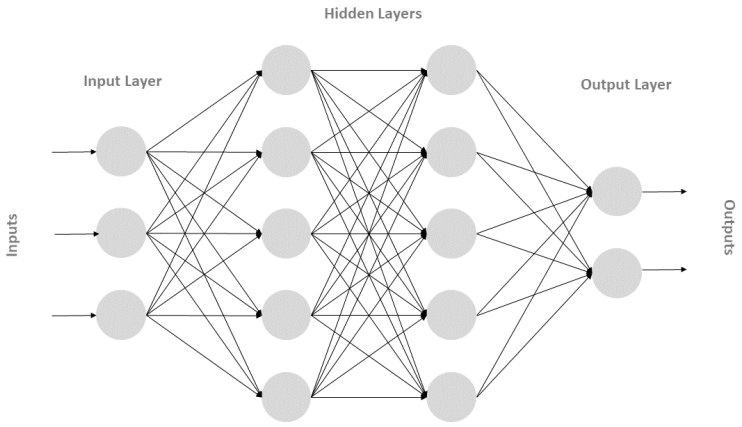


Figure 2.9: Simplified scheme of the Neural Network architecture

The general architecture of a neural network is composed of layers, each layer represents a function that will be performed on the output of the previous layer. The first layer is the input layer, and the final layer is the output layer. All intermediate layers are called hidden layers, and the number of layers in a NN defines the depth of the model, which is why the study of "deep learning" is the study of neural networks with more than one hidden layer. The name hidden comes from the fact that it is not known or specified what the model should learn to do in the intermediate layers. The first layer deals with the input and the last layer must take the output from the previous layer and map it into the target variable. However, the working of the hidden layers is not known, in the sense that the learning algorithm learns by itself what to do in the intermediate layers.

At each layer, a matrix multiplication will be performed. Suppose the first layer parameters are composed by a weight w_1 and a bias b_1, and that the second layer parameters are w_2 and b_1. The functions of each layer are:

$$f^{(1)}(x, w_1, b_1) = x^T w_1 + b_1, f^{(2)}(x, w_2, b_2) = x^T w_2 + b_2$$

This is actually the same as linear regression in each layer. Combining the two functions to compute the output we would actually have:

$$y = f^{(2)}(f^{(1)}(x, w_1, b_1), w_2, b_2)$$

This will end up being also a linear regression, with a new weight $w_1 \cdot w_2$. The gain of NNs compared to other models is that, by adding a nonlinear function at the end of each layer it is possible to learn complex nonlinear functions. These functions are called activation functions.

2.3.3 Activation Functions

Activation functions are nonlinear functions placed at the end of each layer in a neural network so it is able to learn to approximate nonlinear functions. Being $g()$ the activation function, the output of a layer is:

$$h = g(W^T x + b)$$

The main activation function used in hidden layers nowadays is the rectified linear unit, ReLU (Rectified Linear Units Improve Restricted Boltzmann Machines, Vinod Nair 2009). ReLU is defined by $g(z) = max(0, z)$.

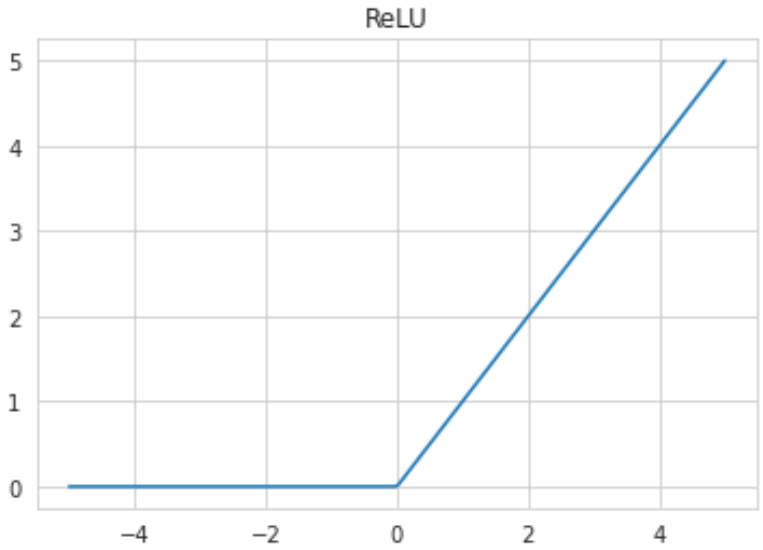


Figure 2.10: Output of the ReLU function

Other important to mention activation functions, but that are not so used in modern deep learning are:

Sigmoid (or logistic) function:

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

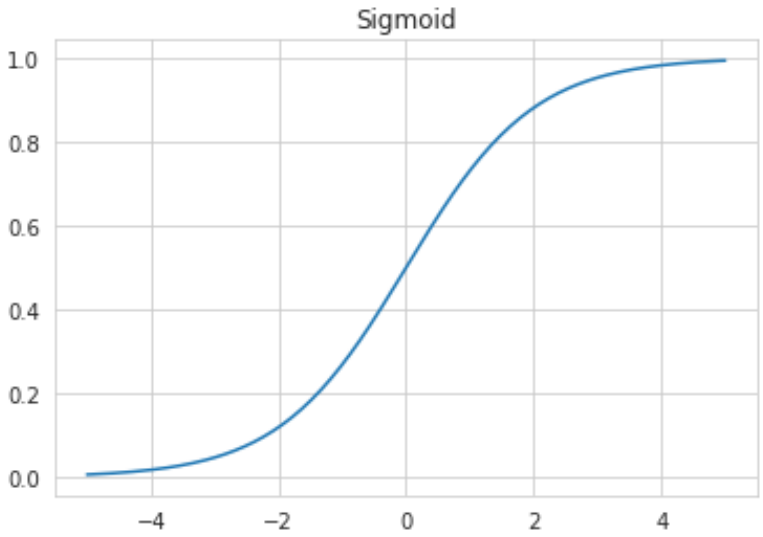


Figure 2.11: Output of the Sigmoid function

Hyperbolic tangent

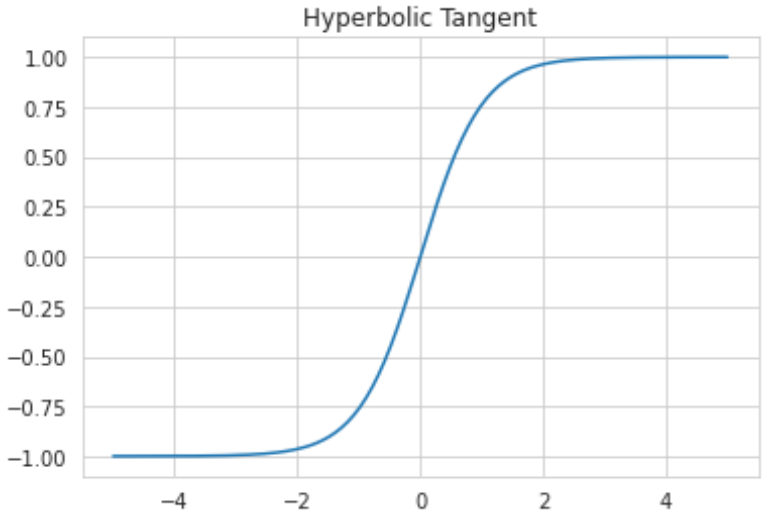


Figure 2.12: Output of the Hyperbolic Tangent function

The two functions above were mainly used before the introduction of ReLU, and the reason they were replaced is that they both saturate in the extremities, which can make gradient-based learning very difficult.

Now considering the output layer the most common function for classification

functions is the softmax function. For each output category j, or neuron, it is computed as:

$$softmax(z_j) = \frac{e^{z_j}}{\sum_j e^{z_j}}$$

This function can be understood as the probability function for each class of the output, that is, the probability that each category corresponds to the target.

2.3.4 Gradient Descent Algorithm

Machine learning problems usually require finding the maximum or minimum of some function, for example minimizing the error function to increase accuracy, this task is called optimization. A very popular optimization algorithm is the *Gradient Descent*. For neural networks, we often deal with the minimization of a function $f(x)$, which can also be called by cost function. It is known from calculus that usually the minimization or maximization of some function requires finding the value that its derivative equals to zero, since the derivative gives the slope of the function at some point, if the slope is zero this point will be a local maxima or minima.

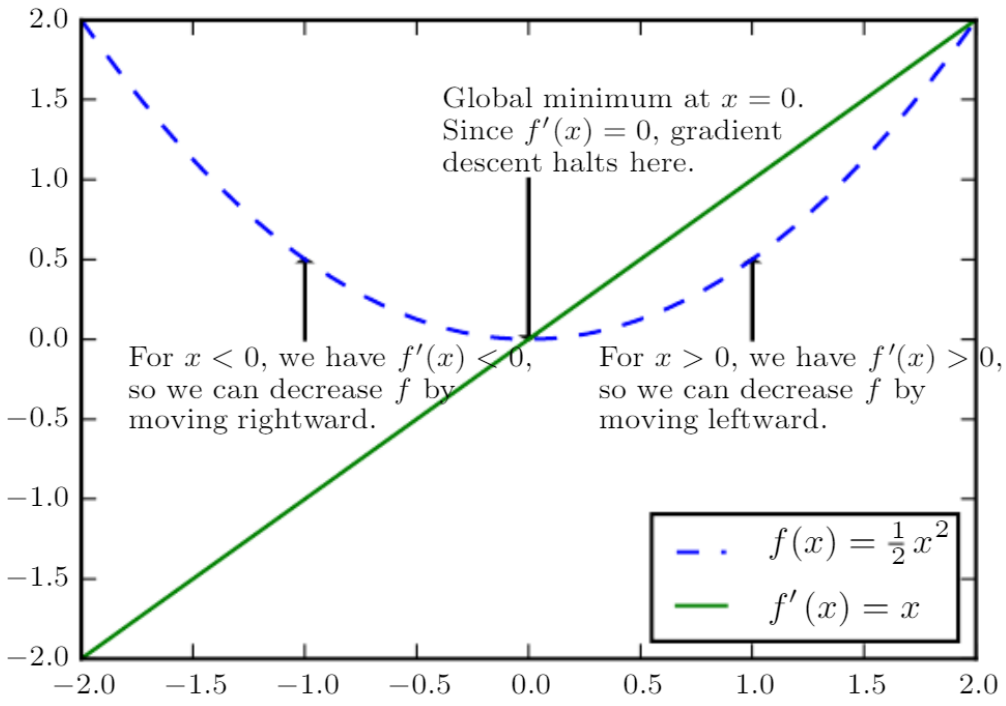


Figure 2.13: Illustration of the use of the derivative function to follow the function to a minimum [11]

The derivative is also useful to understand what a small variation of x will correspond to the function:

$$f(x + \epsilon) \approx f(x) + \epsilon f'(x)$$

It is then possible to minimize this function by giving small increments with the opposite sign of the derivative. This technique is called gradient descent.

In neural networks however, this optimization happens with multiple inputs, being necessary to compute how the function changes with respect to several variables. The gradient is the function responsible for computing the derivative of a function in the case where the input is a vector, that is, more than one input, it will show the direction of increase of the function for each variable. The gradient of a function in respect to the variables x is $\nabla_x f(x)$.

Therefore, the update of the gradient descent to find the variables x that minimizes $f(x)$ is:

$$x' = x - \epsilon \nabla_x f(x)$$

Where ϵ is a small scalar that represents the size of the update step, called learning rate.

The cost function used in machine learning often will be a sum over all training data,of size m, of some loss function $L(x, y, \theta)$:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} L(x^{(i)}, y^{(i)}, \theta)$$

And the gradient becomes:

$$\nabla J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta L(x^{(i)}, y^{(i)}, \theta)$$

One problem of the gradient descent though is that it becomes very slow for large training datasets, since at each update step we compute the cost for all data points. A variation of gradient descent, called *Stochastic Gradient Descent (SGD)*, is usually used. It consists on, instead of computing the actual gradient of the cost function, it computes an approximate by computing the gradient of a small set of the data, called minibatch. Now, even if the dataset increases the SGD remains relatively small.

The approximation for the gradient, g, in SGD is:

$$g = \frac{1}{m'} \sum_{i=1}^{m'} \nabla_\theta L(x(i), y(i),)$$

Where m' is the size of the minibatch.

2.3.5 Backpropagation

To compute the output of a feedforward neural network it is necessary to compute a series of matrix multiplication, each one followed by an activation function. This process is called *forward propagation*. To do so, however, it is necessary to learn the matrices weights and biases values. During training, forward propagation produces a cost function, in order to update the value of the parameters to minimize the cost function, that is, to have the output of the network closer to the target, it is necessary to know how the change of those parameters affect the output, using the partial derivatives of the cost function relative to the parameters.

Since we know the function for each layer, it is possible to manually compute the partial derivative of the cost function in relation with the weights and biases and perform stochastic gradient descent. The numerical evaluation of those operations, however, is too expensive to be performed computationally. What is actually done for computing the gradient in the training of neural networks is to use the algorithm of *backpropagation* which, as the name suggests, is the operation of, once computed the cost, compute the gradient values from the output to the input layer.

The difference for the backpropagation algorithm is that it employs computational graphs and the chain rule of calculus to evaluate the partial derivatives during training, and a recursive method flows each partial derivative back through the network by each layer, until the gradient is computed for the input layer.

2.4 Overview of the Reinforcement Learning methods

There are several reinforcement learning methods between model-free and model-based, on-policy and off-policy, tabular and approximate. In this work two algorithms will be presented, Q-Learning, due to its simplicity and efficiency, being one of the most used for simple tasks, and the Actor-Critic algorithm, due to its increasing usage in several fields and ability to solve complex tasks, also the algorithm used in this work, an evolution of Actor-Critic.

2.4.1 Q-Learning algorithm

Q-learning [13] is an off-policy, model-free, and temporal difference reinforcement learning algorithm created by Watkins in 1989. In its paper, it is described as "a simple way for agents to learn how to act optimally in controlled Markovian domains."

Q-Learning works by interacting with the environment, tries an action at a particular state and updates its estimate of q-value (action-state value) based on the received reward and its previous estimate of the q-value. Being a tabular method, by trying all states and actions repeatedly, this method eventually converges to the optimal policy.

Algorithm 2 Q-learning algorithm

procedure Q-LEARNING(ϵ)
▷ small $\epsilon > 0$ is for the ϵ-greedy policy
▷ Initialization
$Q(s, a)$ inits arbitrarily ▷ $Q(s, a)$ stores all state-action values
$\alpha \in (0,1]$ ▷ Is the step size
$\gamma \in [0,1]$ ▷ Is the discount factor
▷ Execution
for Each episode **do**
Initialize state S
for Each step **do**
Choose action A with probability ϵ
Otherwise choose A from $max_a Q(S, a)$
Take action A and observe R, S' ▷ Reward and next State
$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma max_a Q(S', a) - Q(S, A)]$
$S \leftarrow S'$
end for
end for
repeat until S is terminal
end procedure

As seen from the above algorithm, Q-Learning is a temporal difference method, since it updates its policy during the episode, and not only when it ends, and it is also off-policy, since the policy used to generate the agent's behavior, in this case a ϵ-greedy policy derived from the actual policy, is different from the policy that is being learnt. This method is also model-free, since it is non dependable of the environment's model.

The update rule for the Q-Learning is a derivation of the Bellman Optimality Equation for action-value function but not taking the model of the environment into account:

$$q_*(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma max_{a'} q_*(s',a')]$$
$$q^{update}(s,a) = [1-\alpha] \cdot q^{old}(s,a) + \alpha \cdot [r + \gamma max_{a'} q^{old}(s',a')]$$

2.4.2 Deep Q-Learning algorithm

As discussed previously, tabular methods are not feasible when the environment's state space is very large, being necessary to use approximate solutions. In 2013 DeepMind presented a variant of Q-Learning, calling it Deep Q-Networks (DQN) [14], using deep learning to successfully teach an agent to play Atari. In this modification, the state space of the problem are the pixels from the game's image and the neural network is used to map the input state into an approximation of the action-state value.

In its releasing paper, the approach is used with something called experience replay buffer, which is basically a storage of all played transitions of state, action, reward and next state and then, instead of training the policy sequentially, it randomly selects a transition from this buffer.

Algorithm 3 Deep Q-learning with Experience Replay

procedure DQN
Initialize replay memory $\mathbb{D}$ to capacity $\mathbb{N}$
Initialize action-value function $\mathbb{Q}$ with random weights
for episode=1,M **do**
Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\sigma_1 = \sigma(s_1)$
for $t = 1, T$ **do**
With probability ϵ select a random action a_t
otherwise select $a_t = max_a Q^*(\sigma(s_t), a_t; \theta)$
Execute action a_t in emulator and observe reward r_t and image x_{t+1}
Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\sigma_{t+1} = \sigma($s_{t+1}
Store transition $(\sigma_t, a_t, r_t, \sigma_{t+1})$ in $\mathbb{D}$
Sample random minibatch of transitions $(\sigma_j, a_j, r_j, \sigma_{j+1})$ from $\mathbb{D}$
Set

$$y_j = \begin{cases} r_j & \text{for terminal } \sigma_{j+1} \\ r_j + \gamma max_{a'} Q(\sigma_{j+1}, a'; \theta) & \text{for non terminal } \sigma_{j+1} \end{cases} \tag{2.1}$$

Perform a gradient descent step on $(y_j - Q(\sigma_j, a_j; \theta))^2$
end for
end for
end procedure

The benefit of this approach in respect to tabular Q-Learning is that the learnt policy can generalize for unseen states in the training, with the use of the neural network, since it will try to predict the q-value for any given input.

2.4.3 Advantage Actor-Critic (A2C) algorithm

Actor-Critic is a family of on-policy, model-free, policy gradient reinforcement learning algorithms. Different from Q-Learning, which takes actions consulting action-values, Actor-Critic tries to learn the policy directly.

Policy gradient methods learns a parameterized policy, which is dependable on a set of parameters denoted θ, thus the policy is written as $\pi(a, s, \theta)$, that is, the actual policy is being learnt as an estimate of the policy with some function represented by the parameters θ. The parameters are then updated with the gradient of some metric function called $J(\theta)$, the performance measure. Since the objective is to maximize performance, the updates are done with gradient ascent in respect to $J(\theta)$:

$$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t)$$

Policy gradient methods are any method that follow this general idea of learning a parameterized policy. A subfamily of PG are Actor-Critic methods, which learn an approximation for both policy and value-functions. The name actor-critic comes from the fact that there are two learned functions, the actor, which is the approximate of the policy, since it tells the agent how to "act", and the critic, the value-function, which tells how good a state is. The actor-critic variation shown here will be the Advantage Actor-Critic (A2C), which is a variation of the Asynchronous Advantage Actor-Critic algorithm [15] shown by DeepMind, but which they later stated that the asynchronous part brings no great advancement.

The main difference between Policy Gradient methods is how the performance measure is estimated. This function, $J(\theta)$, is defined as being the state-value of the start state of the episode, for episodic cases but the idea hold also for continuous tasks, and is estimated with the policy gradient theorem, which states the performance measure is proportional to the distribution of states under the policy θ, the action-state value function and the gradient of the policy:

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\theta)$$

For the actor-critic, this relation is replaced by an expectation of gradient of the policy and a comparison between the return and a baseline. For the advantage actor-critic algorithm, the comparison will be the advantage function:

$$A(s) = G_t - v(S_t, w),$$

where w is the parameter for the value function.

To estimate the advantage function, it can be computed as:

$$A(s) = R_{t+1} + \gamma v(S_{t+1}, w) - v(S_t, w)$$

The update for the A2C algorithm ends up being:

$$\theta_{t+1} = \theta_t + \alpha[R_{t+1} + \gamma v(S_{t+1}, w) - v(S_t, w)] \cdot \frac{\nabla \pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)}$$
$$\theta_{t+1} = \theta_t + \alpha[R_{t+1} + \gamma v(S_{t+1}, w) - v(S_t, w)] \cdot \nabla \pi(A_t|S_t, \theta_t)$$

The pseudocode for the A2C is as follows:

Algorithm 4 Advantage Actor Critic algorithm

procedure A2C(θ,θ_v)
▷ θ and θ_v are parameter vectors
$t \leftarrow 1$ ▷ Initialize step counter
$E \leftarrow 1$ ▷ Initialize episode counter
for each Episode **do**
▷ Reset gradients
$d\theta \leftarrow 0$
$d\theta_v \leftarrow 0$
$t_{start} = t$
Get stat s_t
for each step until terminal **do**
Perform a_t according to policy $\pi(a_t|s_t;\theta$
Receive reward r_t and new state s_{t+1}
$t \leftarrow t + 1$
end for

$$R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta_v) & \text{for non terminal } s_t \end{cases} \tag{2.2}$$

for $i \in \{t-1, ..., t_{start}\}$ **do**
$R \leftarrow r_i + \gamma R$
Accumulate gradients
$d\theta \leftarrow d\theta + \nabla_{theta} log\pi(a_i|s_i;\theta)(R - V(s_i;\theta_v)) + \beta_c \partial H(\pi(a_i|s_i;\theta))/\partial\theta$
$d\theta_v \leftarrow d\theta_v + \beta_v(R - V(s_i;\theta_v))(\partial V(s_i;\theta_v)/\partial\theta_v$
end for
Perform update of θ using $d\theta$ and of θ_v using $d\theta_v$
$E \leftarrow E + 1$
end for
end procedure

For the parameterized policy and value-functions it is common to use neural networks, which are usually defined with the same architecture. In the pseudocode above, the NN can be represented as the vector of parameters parameters , which would be the weights and biases of the neural network.

2.5 State of the art of RL Trajectory Planning Applications

Reinforcement learning has been gaining notorious importance for control problems, mainly after DeepMind taught an agent to play several Atari games with its new

algorithm Deep Q-learning [14]. Since then, RL has shown to have great success in games or game-like problems, with the benefit of learning policies that directly map state into action, without any prior knowledge of the environment.

With that progress in learning how to play video-games, and in some cases even achieving superman-like performance, like the AlphaGo, a computer program by DeepMind that was the first time a computer has defeated a human professional player in the game Go [16], combined with the progress in GPUs, which allowed faster onboard inference for neural networks, reinforcement learning also gained attention in the robotics field, and has been replacing classic control strategies in some tasks.

Considering the many existing challenges on solving trajectory planning for mobile robots, such as finding the solution of the robot dynamics, finding all the constraints, designing a robust control strategy and solving the optimization problem for the planning, many current works are focusing on training reinforcement learning algorithms on computer simulations, simulating real world sensors and focusing on the specific desired task.

Deep RL has been applied for autonomous navigation tasks, trained in a small set of simulated environments and tested in real life afterwards and have shown good performance with only the simulated training [17], or performed a small second training in the real world [18], but with no great advancements. Some also perform end-to-end trajectory planning and obstacle avoidance in quadrotors [19] training in a realistic simulation changing the agent's and obstacles positions for better generalization, showing better results than other finite state machine methods.

Another activity of high interest in the field of trajectory planning is autonomous racing, which also involves, besides the trajectory constraints, some time constraints, usually solved with rewards functions that penalizes the time spent on track [20] or rewarding the course progress [21]. In this first work, the generalization is accounted for by generating random tracks with increasing difficulty, based on the crash ratio the curvature between two gates would increase. In the later, they deploy the fastest agent trained on one track layout to train in different settings, such as changing the car model, the track, adding uniform noise and delaying the agent's inference, but the agent however was not able to extrapolate its behavior to some of the unseen tracks.

In most of the above works the algorithm used was a policy gradient method, like actor-critic variations, deep deterministic policy gradient and proximal policy optimization. At [19], DQN was used for a smaller memory utilization, since the

algorithm was deployed onboard in a micro drone.

2.6 Problem of the Unsupervised Environment Design

Using reinforcement learning to learn path planning with obstacle avoidance presents several challenges, especially with deep learning methods that are very data dependent. Specifically in the case of using RL for aerial vehicles, it is difficult to train policies online, that is, during the real world flight, since the battery is limited and probably each battery charge would not be enough to gain considerable knowledge, being required several flights and battery charges. Secondly, the initial policy does not yet know how to avoid obstacles, being dangerous to train during real flight since the UAV will most likely crash and break. Those two reasons by themselves are sufficient to recommend that the reinforcement learning training for aerial vehicles in obstacle avoidance tasks must be done in simulated environments. A new problem, however, arises when considering training in simulation. Since the goal of learning this policy is probably to fly a UAV in the real world, and not only in the simulation, the training must guarantee that the efficiency of the learnt algorithm is also good enough in the real world, which is a problem since the training only consists of simulated images.

Yet some approaches trained only in simulation work in the real world, it is not guaranteed. In the computer vision field it is often common to train a neural network in one dataset configuration that is easier to gather a large sample, and then perform transfer learning in another target configuration which is more difficult to find good data. In the robotics field this could also be done by training an algorithm in simulated data and then training in a small sample of real world data, but gathering good data for aerial vehicles is difficult and time consuming, usually requiring extra sensors like motion capture systems.

Some works try to address this issue without the need of capturing real world data, but instead exploring the advantages of simulated environments. One technique is called Domain Randomization, which consists of randomly changing simulation parameters so that the policy is less likely to overfit to the simulated environment, but rather be able to generalize and transfer knowledge to another environment, in most cases the real world. At [22] the authors randomize dimensions, color and lightning conditions, and dynamic configurations to train a robot hand to manipulate a colored cube so that it is able to transfer to the real world. And in [23] the authors randomize the visual properties of the environment, like background, illumination

and shapes to teach a drone to fly through gates in a simulation and then apply it in the real world, since the learnt policy is very robust to environment changes.

To properly work in the real world, the training must provide the ability of great generalization to the policy, that it is able to find solutions for not seeing before environments or configurations. A trained policy robust to environment changes will most likely be able to pass the task in the real world. For that to happen, it is necessary to have a useful distribution of several environment configurations, which is hard to achieve, since the developer needs considerable effort and time to design valid and useful distributions of environments. To address this issue, a different approach was proposed at [24] with the name of Unsupervised Environment Design (UED) to create environment configurations, from unknown parameters, that can adapt the current knowledge of the agent and be able to generalize to a large set of environments, since they claim that methods like Domain Randomization may create levels that are too difficult for the current knowledge of the agent. They define UED as "the problem of using an underspecified environment to produce a distribution over fully specified environments, which supports the continued learning of a particular policy".

Their approach consists of having a set of two agents, one denoted protagonist, the one we are interested in training, and another called antagonist, which is a pre-trained agent responsible for adapting the environment's difficulty, and a third policy, called adversary policy, responsible for generating environment parameters θ for new configurations. The third policy tries to maximize regret, defined as the difference between the rewards obtained for a set of decisions and the rewards obtained for a different set of decisions. In this case, the regret is computed between the both agents, and maximizing the regret between the antagonist and the protagonist, that is:

$$REGRET^{\overrightarrow{\theta}}(\pi^P, \pi^A) = U^{\overrightarrow{\theta}}(\pi^A) - U^{\overrightarrow{\theta}}(\pi^P)$$

Means that the adversarial policy is trying to output parameters that contribute to the antagonist and disturb the protagonist, such that the new environment is a useful difficult environment for the protagonist to learn.

Algorithm 5 PAIRED

Randomly initialize Protagonist π^P, Antagonist π^P, and Adversary $\tilde{\Lambda}$
while not converged **do**
    Use adversary to generate environment parameters: $\theta \sim \tilde{\Lambda}$. Use to create POMDP $\mathbb{M}_\theta$
    Collect Protagonist trajectory τ^P in $\mathbb{M}_\theta$. Compute: $U^\theta(\pi^P) = \sum_{i=0}^{T} r_t \gamma^t$
    Collect Antagonist trajectory τ^A in $\mathbb{M}_\theta$. Compute: $U^\theta(\pi^A) = \sum_{i=0}^{T} r_t \gamma^t$
    Compute: $REGRET^\theta(\pi^P, \pi^A) = U^\theta(\pi^A) - U^\theta(\pi^P)$
    Train Protagonist Policy π^P with RL update and reward $R(\tau^P) = -REGRET$
    Train Antagonist Policy π^A with RL update and reward $R(\tau^A) = REGRET$
    Train Adversary Policy $\tilde{\Lambda}$ with RL update and reward $R(\tau^{\tilde{\Lambda}}) = REGRET$
end while

In their research, they show that their method provides better results than other tested approaches to surpass never seeing before environment configurations, meaning that the learnt policy has the ability to generalize for different environments.

Another work on UED by [25] proposes a regret based adversarial training, that is, a student and a teacher agent, to make small edits in previous high-regret levels to produce increasingly complex configurations that are in the frontier of the student's capability. Their work consists of randomly sampling levels from a level generator, running both agents to compute the level's regret value, then a curator selects levels to be replayed. The student only trains on replay levels, and after training, the replayed levels are edited and evaluated again to compute regret.

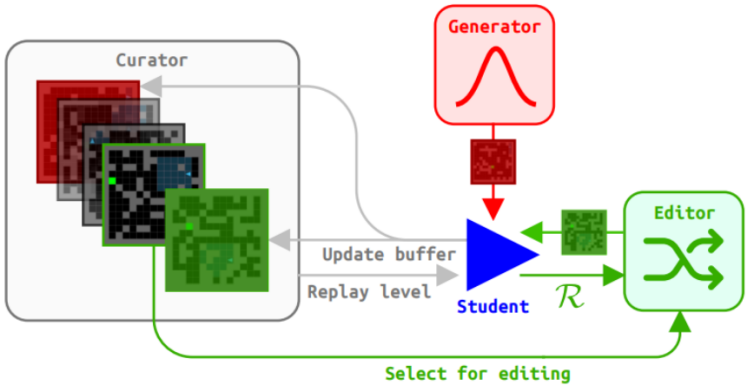


Figure 2.14: Diagram of their proposed algorithm, ACCEL [25]

Algorithm 6 ACCEL

Input: Buffer size K,initial fill ratio ρ, level generator
Initialize: Initialize policy $\pi(\phi)$,level buffer Λ
▷ Initial Data Collection
Sample $K \cdot \rho$ initial levels
▷ Main Training Loop
while not converged **do**
  Sample replay decision $d \sim \mathbb{P}_D(d)$
  if $d = 0$ **then**
    ▷ Evaluate DR levels but do not update
    Sample level θ from level generator
    Collect π's trajectory τ on θ,with a stop gradient $\phi_\perp$
    Compute approximate regret S
    Add θ to Λ if score S meets threshold
  else
    ▷ Train student agent on curated high regret levels
    Sample a replay level $\theta \sim \Lambda$
    Collect policy trajectory τ on θ
    Update π with rewards $\mathbb{R}(\tau)$
    ▷ Edit previously high regret levels and evaluate
    Edit θ to produce θ'
    Collect π's trajectory τ on θ',with a stop gradient $\phi_\perp$
    Compute approximate regret S
    Add θ' to Λ if score S meets threshold
    (Optionally) Update Editor using score S
  end if
end while

They show that their method is able to produce complex levels that facility the zero-shot transfer to human-designed environments in their tested tasks.

2.7 Proposal for new UED method

In this work, it is proposed a new and simpler method for Unsupervised Environment Design, with the goal of generating new and solvable methods with increasing complexity to achieve policies with high capability of generalization for zero-shot transfer in new levels.

Other methods rely on training two adversary agents, one to actually learn the desired policy and another to serve as a comparison for the level difficulty, and also

a third policy to learn how to design new levels, or to produce a level generator. The present work uses the same approach on adversary agents, one pre-trained antagonist and one protagonist, but tries a different method on generating new levels that does not depend on a third policy. This work also has the goal of trying to understand what is the best way to pre-train the antagonist agent.

For this present approach, it is necessary to have an unstructured environment with some free environment parameters. The antagonist agent is trained in a set of levels with the same parameters, in our case of obstacle avoidance for an UAV, the parameters of the environment are the level's size and the density of obstacles in the level. After the antagonist is trained in a set of levels with fixed size and number of obstacles, the protagonist is initiated, without training, and in a random environment with "easy" parameters, in this case the level begins small in size and with only one obstacle. To test the feasibility of this generated environment, the A* algorithm is applied to check if there is a path between the initial point and the target without collisions. If no free path is found in three trials changing the location of the obstacles, then the algorithm reduces the difficulty of the level by reducing the obstacle density. If a path is found, both the antagonist and the protagonist are evaluated in this level, without training, to compute the regret between the two agents. The agents are then both trained in this set of environments.

At the end of this step of evaluating and training, if the regret computed in the evaluation is higher than a threshold regret, then the levels are reduced in difficulty, otherwise the level is increased in difficulty, by increasing the obstacle density until we reach a limit, in this case the level's size is increased. This iteration runs for some episodes, if the algorithm is not able to increase in difficulty for the limit of episodes then the environment changes in configuration.

Algorithm 7 Proposed UED method

procedure UED($SF, E_{tot}, E_{train}, E_{val}, REGRET, tries$)
▷ SF is the desired Success Factor to change difficulty
▷ E_{tot} is the total trained episodes
▷ E_{train} is the total trained episodes at each level
▷ E_{val} is the total evaluating episodes at each level
▷ $REGRET$ is the threshold between antagonist and protagonist
▷ $tries$ is the maximum tries generating a new level configuration
▷ Initialization
Initialize pre-trained antagonist, protagonist
$episodes \leftarrow 0$
current $SF \leftarrow 0$
while E_{tot} not reached **do**
while environment not feasible **do**
Initialize environment
Evaluate environment with A^*
end while
while current $SF < SF$ and $episodes < E_{tot}$ **do**
Evaluate agents for E_{val}
Compute Success Factor and $regret$
Train agents for E_{train}
if $median(regret) \geq REGRET$ **then**
Decrease environment difficulty
end if
if maximum tries in environment is reached **then**
Change environment configuration and maintain difficulty
else
Increase environment difficulty
end if
end while
end while
end procedure

The idea of this algorithm is to maintain the assumption that if the protagonist's regret, that is, the difference in discounted return of the antagonist and the protagonist, is too high then the environment is too difficult yet for the protagonist, and it is not useful for its learning. However, this work tries to maintain that idea without the need of training a third policy, but by only checking the feasibility of the generated environment with known methods, such as the A* algorithm, and checking by using a flag to increase or decrease the level's difficulty, based on the

regret evaluation and the number of times the protagonist tried to surpass this level with the requested success factor. In our case, the success factor is the amount of times the agent succeeds in reaching the target without collisions divided by the number of trials.

The law used to increase and decrease the difficulty of the environment may depend on the application, since it may make sense that some parameters possess limits or constraints. In this work's case, the update law is simple and just increases the number of obstacles until some defined limit is reached, and then randomly increases the size of the environment. This law is further explained in the training section.

2.8 Requirements and Constraints to Implement an RL application

Developing UAVs with classic control techniques usually requires a dynamic formulation of the problem, which can be complex but usually brings greater reliability when testing, since the user knows the inner workings of its application. Reinforcement Learning applications however may not require dynamic formulation, with the advancement of open source programs, it is easier to find off-the-shelf simulations that may be harder to check if its inner working is precisely the same of your end application, but may be good enough with some adaptations to train a policy. This makes the train easier but does not guarantee that the test in the real world will be fail proof, even having the dynamic formulation it is not possible to guarantee that errors will not happen.

The above problem is a recurrent one in the robotics field, but requires special attention when employing RL techniques in simulated environments to be transferred to real applications. The search for a robust policy that succeeds in different environments is essential but demands several extra steps in the training. It is possible to try to solve this problem by re-training the policy with data from the real world or to manually add other commands to adapt to the real world. The concern of transferring a policy from simulation to the real world is a complex problem by itself and will not be covered in this work.

Chapter 3

Creation of the Simulation

Drones must be low in weight in order to be less energy consuming and fly, thus they are often made with fragile components and are not resistant to falls and collisions. Therefore, it is not a good idea to use the real-life drone to train a reinforcement learning policy, specially in an obstacle avoidance task, obviously it is preferable to crash models than crashing real life robots. Moreover, training a model in a simulation can avoid the tedious process of collecting good quality data and can be changed and modeled with more facility to fulfill the desired requirements of the application. Often a realistic computer simulation is designed to train a policy and then deployed in the real hardware. Probably a realistic scenario with more detailed physics can help the agent to achieve better results in the real world. Sometimes it is also useful that real-world data is collected to perform transfer learning and fine tune the model trained in simulation. However, using realistic simulations and real world data is not sufficient for success if the model is trained in only a few situations or if the simulation and real data cannot exploit the generality needed to solve our problem. It is also necessary that we train policies considering a high range of situations. In this work, we are proposing to develop an algorithm able to generate useful environment configurations to train a trajectory planning solution for a quadrotor. Since the training of machine learning models is already time consuming, we decided to develop and test our solution in a 2D environment because it is already sufficient to validate our results and less computationally consuming than a realist 3D environment.

Therefore, we saw the need of designing our own simulation to be able to generate different environment configurations and test the trajectory planning algorithm with more ease.

3.1 Assumptions of the developed simulation

To test our solution, our simulation had to be simple and fast, since our focus is to generate different environment configurations with ease, so it does not contemplate the real dynamics of a quadrotor in real-life, such as:

- Friction
- Inertia
- Gravity
- Battery life of drone
- Delay of computing actions
- Sensors precision
- Non-linearity
- Natural environmental conditions

Certainly simulating a quadrotor including its dynamics and non-linearity, and also including friction factors, imperfections of sensors and natural conditions such as wind, contribute when deploying the agent into the real world because it is more similar to reality. However, those factors have a huge impact on the simulation complexity, harming the computational speed and delaying the model's training.

3.2 Development of the 2D simulation

For our work, it was important that our simulation generates different environment configurations, simulates necessary sensors and has an easy interaction when training machine learning algorithms, since our main purpose is to provide useful and feasible configurations for training. We decided that two important environment parameters would be the size and shape of our environment and the number of obstacles, in that way our agent would face different situations in several difficulty scales. It was also important that we could simulate different perception possibilities, sensor configurations, and drone movements, to find the best parameters for our problem.

3.2.1 Simulation Objects

Inside the simulation, the drone, the target and the obstacles are objects, making it easy to store attributes, functions and checking its interactions.

Each object stores its size, position, sprite image (necessary for rendering the simulation), and an init function.

Drone

The drone class also stores the necessary functions for performing its actions, such as rotating, accelerating and moving. This class is the first object to start in the simulation, and can be initialized at a random spot or at the upper left corner.

This class holds the actions for moving the drone's state, such as forward motion, lateral motion, rotation and acceleration. To discretize the actions, they are divided into 12 possible actions, being combinations of those 4 main actions. It is worth mentioning that each one can be positive or negative, that is, positive forward motion and the negative being backward motion. The resulting action is then inserted in the simulation as an array. The division into lateral and forward motions are to account for the drone's perspective when rotated.

The forward update is:

$$x \leftarrow x + \cos(angle) \cdot velocity$$
$$y \leftarrow y - \sin(angle) \cdot velocity$$

And the lateral motion:

$$x \leftarrow x - \sin(angle) \cdot velocity$$
$$y \leftarrow y - \cos(angle) \cdot velocity$$

Obstacles

Obstacles are initialized after the drone and they are placed at a random position, checking if there is superposition with the drone and, if so, a new object is created at a different spot.

Target

A target is created checking if there is no superposition with the drone nor with the obstacles.

3.2.2 Target position sensor

The information regarding the target's position is not given by a specific sensor, but it can be understood as information given by a camera or GPS. This sensor gives the true direction of the target relative to the drone's forward-looking side and

can be configured to provide only the direction or also the distance in each direction.

Another possible application of the "target sensor" is of a light seeking robot, such as in search and rescue tasks in closed environments, so it is possible to provide the direction and intensity of the light source.

In our script it works by getting the target position and drone's position, calculating its relative position and then converting to the drone's own coordinate system, since for the agent's point of view it is easier to learn thinking on its own coordinate, rather than a "general" coordinate system. In our study we then normalized the relative distance of the target.

The only parameter configuration for this sensor is its range, that is, the maximum distance the sensor can provide an accurate distance from the quadrotor and, above it, it only returns the direction of the target.

$$
\begin{aligned}
x_{dist} &= x_{target} - x_{drone} \\
y_{dist} &= y_{target} - y_{drone} \\
forward_{dist} &= \frac{x_{dist} \cdot \cos(drone_{ang}) - y_{dist} \cdot \sin(drone_{ang})}{target_{range}} \\
lateral_{dist} &= \frac{-x_{dist} \cdot \sin(drone_{ang}) - y_{dist} \cdot \cos(drone_{ang})}{target_{range}}
\end{aligned}
$$

To be able to avoid obstacles and walls, a robot must have a reliable sensor to perceive its surroundings and update at each step how close it is to other objects. This task can be done usually with cameras or distance sensors, such as ultrasonic sensors, laser sensors and LIDARs. In our simulation we developed our distance perception inspired by laser sensors. Some possible configurations are:

- How many sensors surround the quadrotor
- Range of distance
- If the laser surrounds the entire robot or just its front

Thinking in search and rescue tasks or racing tasks where time is crucial, the agent barely has time to go backwards, since it can optimize its time going only forwards. Sometimes it is also crucial that the agent only perform forward moves because of some sensor limitation, such as a single camera at the front, with that configuration it is dangerous to perform several backward moves.

With that reasoning, we used 9 sensors taking place at 180^{o} degrees around the quadrotor's front to perform this work analysis.

The laser rays of the sensor are evenly divided around the drone and each one provides the distance between the drone and the closed object found, including the simulation walls. This sensor is also designed with a limited range, defined in the simulation.

To compute the distance, each ray is treated as a line, and each object has its surrounding lines. At each timestep, for each laser ray, it is computed the point of intersection for each object line in the simulation, then it is stored the closest point from the drone.

Algorithm 8 Computation of distance and point of intersection

```
▷ Initialization
distance ← ∞
▷ distance stores the minimum distance of the laser and the obstacles
Obstacle list obstacles
for each obj in obstacles do
    for each line in lines of obj do
        if line intersects with laser_lines then
            x_d, y_d, dist = point of intersection of line and laser_lines
            if dist < distance then
                distance ← dist
                store points of the closest intersection
            end if
        end if
    end for
end for
```

Then, each laser ray is drawn with its starting point around the drone until the intersection point, that can be an obstacle, the simulation walls, or, if there is no intersection, the laser's end according to its range. Since the target is not a physical object, but more a desired location, the laser sensor does not detect the target. To compute the point of intersection between lines, we define:

$$A = F_y - I_y$$
$$B = I_x - F_x$$
$$C = A \cdot I_x + B \cdot I_y$$

Being Ix, Iy the x,y coordinates of the init point and Fx, Fy the x,y coordinates of the end point.

If $line_1.A \cdot line_2.B - line_2.A \cdot line_1.B = 0$ then there is no intersection. Otherwise,

$$x = \frac{line_2.B \cdot line_1.C - line_1.B \cdot line_2.C}{det}$$
$$y = \frac{line_1.A \cdot line_2.C - line_2.A \cdot line_1.C}{det}$$

3.2.3 Simulation Parameters

- Number of obstacles

  The number of obstacles in the simulation is necessary for starting, every obstacle has a random position and it is not possible to have zero obstacles.

- Size of environment

  There are three possible predefined sizes for the environment

- Size of objects

  The drone, the obstacles and the target's size are defined by a single variable that can be changed. In our work we used the size of 32 pixels

- Number of possible actions for the quadrotor

  The quadrotor can move up, right, left, down, rotate clockwise and counter-clockwise, speed up and speed down. It is possible to choose between combinations of those movement to discretize the action space of the agent

- Easy/hard

  If the simulation is at easy mode the target will always spawns closer to the drone

- Change obstacles positions

  It at the end of each episode the objects change position or not

- Quadrotor spawn position

  It is Possible to choose between random spawn, a predefined position or, as default, the top right corner

- Agent state type

  If the state to train the agent is the laser sensor measures or the general position of the drone in the environment

- Number of laser rays for distance sensor

  The number of rays for distance sensors around the drone. If an even number it will go around 360º the drone, and if it is an odd number it will go around 180º in the front of the drone.

- Reward function

  There are several defined reward functions that we used to tune our agent, but it is possible to write other reward functions and choose the wanted one

- Mode of usage (training, running, evaluation, keyboard)

  To be more easily developed and tested, the simulation has to option to be playable by the keyboard, this is defined by the mode of usage, also possible to choose between the training of an agent, running the trained agent in some environment, evaluating the agent in several environment configurations and saving the results, and also other training configurations

- Rendering ON or OFF

  For faster training it is useful to not render the simulation, and therefore not image is produced, but for checking results or debugging it is necessary to see the simulation

- Randomness of simulation

  Since the objects are randomly placed, it is possible to define the seed for the random generator and make the randomness controlled, useful to keep track of results.

3.2.4 Simulation Working

The simulation initially initiates all objects in the according environment size, making sure no superposition occurs. Then, at every agent action a step is performed, which applies the given action, updates all the objects, and then checks if the target is reached or if any collision happened. If so, the episode is terminated.

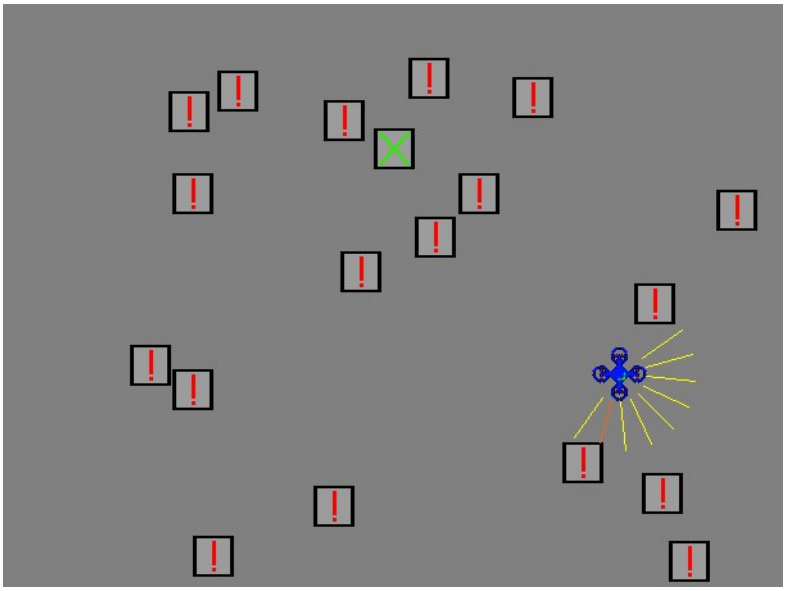


Figure 3.1: Example of an environment of the simulation

To generate objects, making sure there are no superpositions between drone, obstacles and target, to reach the desired simulation design, we initialize the drone randomly and add it to a list of existing objects. While we do not have the desired number of obstacles, we add random objects and check with the collision algorithm if superposition occurs, and then add the new object into the list. First all obstacles are created and lastly the target position.

Algorithm 9 Generating objects

```
    Input: Number of obstacles N                                  ▷ Initialization
    Initialize objects ← [...]
    Superposition ← True
    ▷ Flag for superposition
    Init drone position randomly
    while length of objects < N do
        while Superposition is True do
            Initialize new object randomly
            Check collision between new object and existing objects
            Update superpostion flag
        end while
        Insert new object in objects
    end while
```

To check for collisions, and therefore also superpositions, we input the center

coordinates of the objects and their size, which is always the same, and then check if $x_1 + \frac{size}{2} > x_2 - \frac{size}{2}$ and $x_1 - \frac{size}{2} < x_2 + \frac{size}{2}$, which returns the below algorithm.

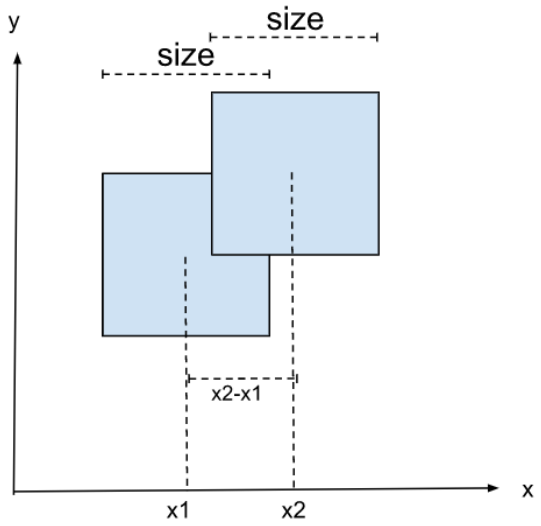


Figure 3.2: Scheme of block superposition

Algorithm 10 Collision check

Input: Object1 and Object2 (x, y) coordinates and size
if $(size > x_2 - x_1)$ and $(x_1 - x_2 < size)$ **then**
    if $(size > y_2 - y_1)$ and $(y_1 - y_2 < size)$ **then**
        Collision is **True**
    else
        Collision is **False**
    end if
else
    Collision is **False**
end if

The function responsible for the real working of the simulation, updating all states given the input actions is the step function. The step function gets the current state and the agent's action and computes the next state, reward and if the episode is done. This function is used to run the simulation, train and test the agent.

To initiate the simulation it is necessary to input the environment size, number of obstacles, the state parameters, the reward function and specify if the objects

Algorithm 11 Functioning of the Simulation

```
for each step do
    Render environment
    Get input action
    Update all objects with action
    Check for collisions
    Check if simulation is finished
    Compute reward
    Return state, reward and done
end for
```

should be placed randomly or not. If it is decided to not initiate the objects randomly, their positions must be stated.

Chapter 4

Training of the Agent

Differently from supervised and unsupervised machine learning training, the training procedure for reinforcement learning usually is not done by collecting labeling data and then applying the training algorithm. The training procedure involves letting the agent interact with the environment, take actions, and learn from the received rewards and next states. That means that when evaluating the training results, usually the training curve, the states that the agent would pass through and the outcomes of those interactions are not known beforehand. One way of analyzing the training curve in reinforcement learning is by plotting the evolution of the discounted return and not on rewards or the normal return. This is because in episodic tasks, as in this work, it is harder to deduce if the algorithm is converging. Still, when looking directly at the discounted return, it is easier to analyze if the training succeeded.

In this work, different algorithms were trained using different levels with different difficulties and different difficulty increases for different numbers of episodes. One challenge of analyzing those results is that when the difficulty of the environment changes, the environment size changes, meaning the agent will spend more steps in a single episode so that episode is finished. This changes the return's value, which is why the graphics in this section will present the discounted return.

4.1 Reward Function

One of the main challenges of developing a reinforcement learning application is the design of the reward function. The reward function is the only aspect of the application that will tell your learning algorithm what the problem needs to be solved because it is what the algorithm will use to differentiate good and bad actions, as in the supervised learning applications that the label will show the

algorithm if it is right or wrong. Hence, this means that a reward function that is not precisely designed will show the learning algorithm how to solve a different problem differently from the one desired. Besides that, currently, there are no guidelines on designing a reward function rather than intuition.

To aid the analysis of the model's evolution, the success factor of the agent was computed at each training loop, that is, the percentage of episodes in which the agent reached the target position with no collisions.

4.1.1 Design of the Reward Function

The final reward function chosen to train the agent, after several different functions, rewards the agent when it reaches the target location with a high value, penalizes with the opposite value when it collides, gives small rewards when the agent gets closer to the target and slightly greater penalties when it moves further away. The above reward tells the agent that the target location is always the best and moves towards obstacles are always the worst. It also tells the agent that moves that get closer to the target are better, but moves further are not so much worse, or on the contrary, the agent might never find a turnaround to reach the target if the path is blocked.

The reward value that accounts for the end of the episode, that is, if the target is found or if a collision happened, is:

$$reward_{finish} = \begin{cases} 50 & \text{if } target \text{ is found} \\ -50 & \text{if } collision \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

Regarding the proximity to the target, the reward is:

$$reward_{closer} = \begin{cases} 0.2 \cdot (dist_{last} - dist_{current}) & \text{if } dist_{current} < dist_{last} \\ 0.8 \cdot (dist_{last} - dist_{current}) & \text{if } dist_{last} \geq dist_{current} \end{cases} \tag{4.2}$$

To account for obstacle avoidance, there is also a penalization based on the laser distance closest to an obstacle. The penalization is worse when the agent is too close to an obstacle. Still, it accounts only for the smallest laser distance value.

$$reward_{laser} = 6 \cdot (min(\text{laser measures}) - 1)$$

Where laser measure is an array containing all laser measurements normalized by the greatest possible distance measure of the sensor. Therefore, $reward_{laser}$ will

always be a value in the range $]-6,0]$.

When training the agent, it was noticed that sometimes the agent would stop to try to find a different path and would drive in circles or back and forth for several episodes. This would slow down the training while also not discovering new possibilities. To account for that problem, a penalization was introduced when the agent spent several steps without a significant change in its location, by saving the agent's position in a buffer and computing its x and y coordinates average location.

$$reward_{stuck} = \begin{cases} -10 & \text{if } mean(|x_{buffer}|) < 7 \text{ and } mean(|y_{buffer}|) < 7 \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

Then, the final reward is:

$$reward = \begin{cases} reward_{finish} & \text{if } done \\ reward_{closer} + reward_{laser} + reward_{stuck} & \text{otherwise} \end{cases} \tag{4.4}$$

This update in the function was enough to solve the issue above. The agent would quickly stop being stuck after a few episodes, understanding that repeating moves and not changing positions was bad for fulfilling the application goal.

4.2 Methodology and parameters adopted

The RL algorithm chosen for this work was the A2C algorithm, which is a very well-known and consolidated algorithm, since the initial tests showed a faster convergence in comparison with a second chosen algorithm, DQN. The function approximation used two neural networks with two layers each, implemented with the PyTorch framework, being one NN for the actor and a second for the critic. Since the actor's output must be a probability distribution for action values, a softmax activation function was placed at the network's output.

To tune the number of neurons, a few tests were made to balance a NN with small complexity to have faster training, but not so simple that it is not big enough to learn complex functions. To present the results, it was chosen to train the agent with 32 neurons, which showed an increase in the results to smaller networks but not compromising the training time.The input size is 12, since the chosen state for the agent is composed of 9 normalized laser readings, the drone's speed, and the normalized lateral and forward distances from the agent to the target. The output size is also 12, being a discretization of 12 possible actions for the drone:

- Right or left

- Forward or backward
- Accelerate or decelerate
- Rotate clockwise or counter-clockwise
- Go forward and accelerate or go forward and decelerate
- Go forward rotating clockwise or go forward rotating counter-clockwise

The normal train loop, the one without UED, is done online with the simulation, reacting to the states that succeeded after the action chosen. The neural network, however, trains in batches of 128 observations, which makes the train faster.

Algorithm 12 Training of agent

```
▷ Initialization
done ← False
E_train, γ                                  ▷ Total number of episodes to train
E ← 0                                       ▷ Number of episodes trained
SF ← 0                                      ▷ Success Factor
while E < E_train do
    RETURN ← 0
    Get state S from environment
    while notdone do
        get action A from policy π of agent
        S', reward, done ← environment step using A
        Update parameters θ of agent's Neural Network
        RETURN ← RETURN + reward · (1 − γ)
        S ← S'
    end while
    Update SF
    E ← E + 1
end while
```

While training, the agent chooses a random action, according to the probabilities of its policy, so it explores more state-action value pairs and increases its chances of finding a global maximum instead of getting stuck in a local maximum. And then, when testing the trained agent, this option is disabled, so it only chooses the best action according to its policy.

$$action = \begin{cases} \text{random action according to policy } \pi \text{ probability distribution if } training \\ \text{choose best action according to policy } \pi, \text{ otherwise} \end{cases} \tag{4.5}$$

4.3 UED Training

When training with the UED algorithm, the training algorithm is the same, with both protagonist and antagonist training with the same parameters in the same levels. The difference is that more parameters related to UED, are needed to tune to train the agent. The needed parameters are the increase and decrease in the size of the environment and the maximum obstacle density allowed per level. Every time the UED algorithm increases the difficulty of the environment, it first increases the number of obstacles until the maximum density is reached; then, it applies a random increase in the width and height of the environment so that the environment shape is always changing.

Algorithm 13 UED training

▷ Initialization
Initialize $increase_{max}, increase_{min}, density_{max}, height, width$
▷ *increase max* and *min* are the proportions to change environment's size
▷ $density_{max}$ is the maximum obstacle density
$N_{obstacles} \leftarrow 1$
while $\frac{N_{obstacles}}{width \cdot height} < density_{max}$ **do**
    $N_{obstacles} \leftarrow N_{obstacles} + 1$
end while
▷ Increase between $increase_{min}$ and $increase_{max}$
$width \leftarrow width \cdot random(increase_{min}, increase_{max})$
$height \leftarrow height \cdot random(increase_{min}, increase_{max})$
$N_{obstacles} \leftarrow 1$

Note that the test levels will always have the same proportion of width and height. Still, the levels generated by the UED can have any shape, with the idea of generating several different environments to increase robustness.

Figure 4.1: Return evolution for the normal training during 50k episodes in the environment 2 with 8 obstacles

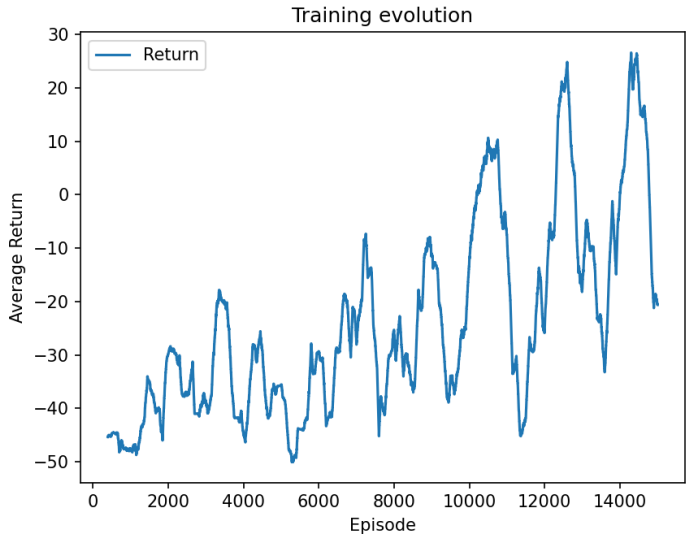


Figure 4.2: Return evolution for the UED training during 15k episodes, using as antagonist an agent trained for 50k episodes in the environment 2 with 8 obstacles

Chapter 5

Analysis of the Results

Reinforcement learning aims to find the best parameters to reach an optimal policy that maximizes the sum of discounted returns of a defined reward function. One of its main advantages is that those frameworks learn by interacting with the environment without designing and tuning complicated control algorithms or optimization functions. Of course, there is usually the need to design the dynamics of the environment and choose and tune reinforcement learning methods, but those are usually more straightforward to design.

One problem, however, in the reinforcement learning framework is to find the right parameters of your environment or at least find the parameters that yield the best results faster. Since robotics problems are usually task-based, in most cases, the parameters will be easier to choose, probably focused on one specific task. Nevertheless, several problems, even being task-based, do not rely on a single environment configuration and depend on the high ability of generalization of the robot.

In those cases, it can be difficult to find the best set of parameters that provides the best and fastest learning of the reinforcement learning agent since some parameters can just be redundant or even make an environment too difficult for the agent and therefore provides no use case for learning.

With that in mind, this chapter aims to analyze the results provided by both learning procedures, with UED and without, and check if the first approach provides an easier way to train reinforcement learning agents, not focusing on environment design by trial and error.

The goal of our reinforcement learning agent is to reach a target position avoiding obstacles in the way, and every time our agent completes its task, the episode

finishes. To test the efficiency of each trained agent, we will compute the success factor in several environment configurations of each solution, that is, the percentage of episodes that the agent reached the target without hitting any obstacle with different values for a number of obstacles and environment size.

$$SuccessFactor = \frac{SuccessfulEpisodes}{TotalEpisodes} \cdot 100[\%]$$

An episode will be successful when the agent reaches the target position without hitting an obstacle or a wall.

Moreover, suppose the drone does not reach the target in 15-time steps. In that case, the episode is finished unsuccessfully since it shows us that the drone probably does not know how to solve this level, being stuck in a succession of actions that do not lead to the episode's goal.

To compare both approaches, between environment-fix training and training with UED, the agent trained normally, without UED, will only train in one environment configuration: the same number of obstacles and the same environment size. On the contrary, the UED training will tackle a wider range of environment configurations.

Therefore, this chapter aims to check whether the agent trained with UED will succeed more efficiently in more environment configurations, showing that this technique provides more general solutions than the agent trained in a single environment.

We also want to check if the UED method converges faster to good results than the other solution.

5.1 Report of the results

In these reports, we want to compare the influence that each method of learning has on the efficiency and ability of generalization of the agent. Three environment sizes were used to test the performance of the trained agents since at every increase of size; the environment gets more difficult because at each time step, there is a chance the agent does not find the best possible move, having a greater chance of colliding. On the contrary, small environments have less space to maneuver and thus are also difficult in a different way to be passed successfully. Being necessary to test different level sizes to analyze the agent's performance and ability to pass different level configurations.

Besides the increase in size, it is also needed to increase the density factor of

obstacles in the simulation, that is, the number of obstacles for each environment size, until it reaches challenging levels for the agent since levels with more obstacles have less free moving space for the agent, and therefore have a greater chance of collision. Therefore, a test set was created to compare the agent's performance in different levels, made by three different environment sizes, each with three obstacle densities, approximately 4%, 6%, and 7.5%, that is, the ratio between the summed area of obstacles and the environment area.

The three sizes are the following:

Env Size 1	250 width, 200 height
Env Size 2	400 width, 320 height
Env Size 3	800 width, 640 height

A total of 450 evaluation levels were used, 50 for each kind of configuration, and for better visualization, the results are shown in the form of a 3x3 heatmap and a boxplot showing the distribution of the success factor of the agent in all 9 possible level configurations. It is expected that the generated heatmaps are lighter, meaning a higher success rate, on the left side, since those levels have less obstacles.

Moreover, in this analysis, we are not interested in how the agent succeeds the environment. That is, an episode is successful independently of the time it took to reach the target if it was the optimal or suboptimal path. No comments will be made on different approaches to completing an episode.

5.1.1 Method

In this section, our objective is to compare the performance between models trained with and without Unsupervised Environment Design and thus compare where each trained model performs better. It is expected that if we train a model in a single environment configuration, for example, in environment size 1 with 4% obstacle density, this model will have better results at this level than bigger or more crowded levels since it is not the kind of environment where it has learned. On the other hand, it may not be obvious if an agent trained at environment size 3 with 7.5% obstacle density will have better results at env 1.1 than the agent trained only there.

Hence, the method we used to test the efficiency and generalization of our algorithm is to train six different agents, three for each algorithm (with and without UED), and for each one, we train in three environment configurations.

The chosen three levels are:

- Environment size 1 with 4% of obstacle density, denoted env 1.1
- Environment size 2 with 6% of obstacle density, denoted env 2.2
- Environment size 3 with 7.5% of obstacle density, denoted env 3.3

Hence, for each method, with and without UED, we will train agents in those three environment configurations. The difference for the UED method is that, since it does not stay at only one configuration, we will use the agent trained in that environment configuration as the antagonist agent.

Our motivation is that using those three environment configurations, we can compare how the agent generalizes in the other configurations away from that "diagonal", if we think in a 3x3 matrix, we can directly look at agent 1.1 and compare at all the levels that are higher in that matrix, such as all levels lower than agent 3.3, and in the same way agent 2.2 acts as the most central possible agent, which we can compare if is the best one for generalizing without the UED technique.

On the other hand, for the UED algorithm, we want to test which kind of agent is the best one guiding the protagonist agent, if it is an agent better or worse at generalizing, and how the environment used as training affects the agent when playing the part of antagonist.

But also considering the problem of choosing the best protagonist agent for the algorithm, we also need to check whether it is better to have a more experienced agent or one that is only slightly trained. Considering that, for each method and environment configuration, we are also training for different numbers of episodes: non-UED agents will be trained for 35k and 50k episodes. In contrast, UED agents will be trained for 15k and 35k episodes, using as antagonists the agents trained for 35k episodes.

With those tests, we want to check whether the algorithm works better with the over-trained antagonists or simpler ones. We also want to check if the UED has a faster convergence than the normal method since to train 15k episodes, we also need to train, in our case, 35k a normal agent to use as the antagonist. Therefore we need to check if this higher-cost training with UED is paid with better generalization.

5.2 Analysis of the results

Previous parts of this chapter explained the method used to compare the results obtained with two training methods in different environment configurations and

the different number of trained episodes. This part will focus on evaluating the results generated by those methods, the training with three different environment configurations, and the training using an unsupervised environment design.

For each trained agent, its performance, that is, the result of the training procedure will be analyzed with a heatmap, since it is an easier form to check the agent's generalization ability, and also in which environment configurations the agent performs better. A more generalized heat map will have colors closer to a clear green while poor performances will have a dark blue color. For each agent, a boxplot with the general performance of the agent, showing the average performance and its confidence margin is computed.

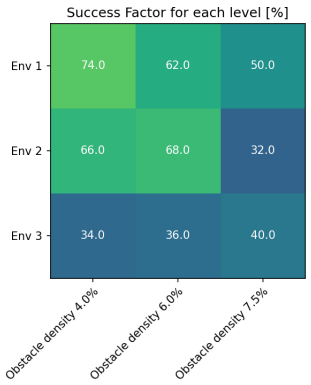
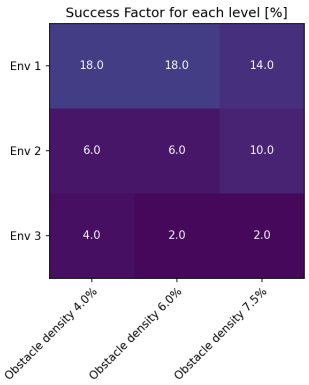
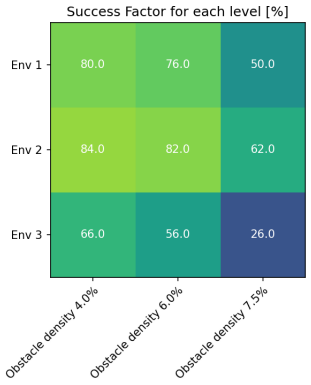
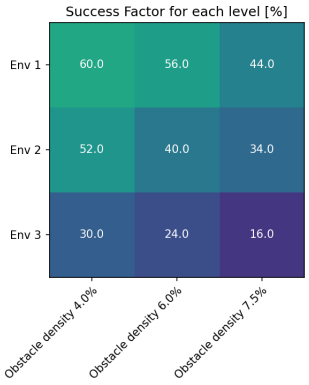
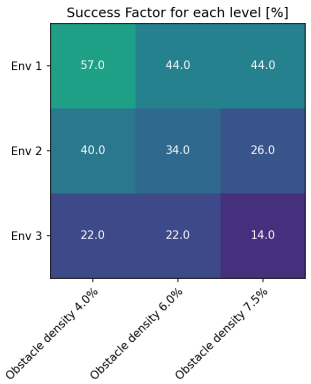
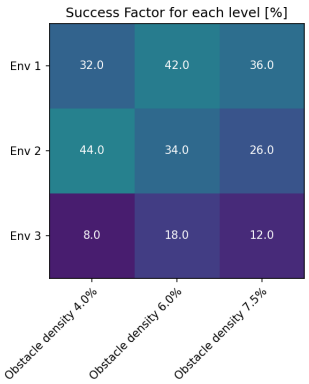
As explained in the "method" section, three environment configurations were chosen for the agent without UED, or "normal" agent, and will be denoted by env 1.1, env 2.2, and env 3.3. For each of those configurations, there are two agents, one trained for 35k episodes and another trained for 50k episodes. For the UED agent, there are three agents for each antagonist, one trained for 15k and antagonist for 35k, one trained for 15k with an antagonist for 50k, and the third one trained for 35k using an antagonist trained for 35k. Since the UED needs a previously trained agent, the idea is to compare the additional gain using the UED method after training an agent with another method. In that way, we can compare a normal agent trained for 50k with a UED agent trained for 15k with an antagonist trained for 35k. Which sums up to 50k episodes trained.

We can also compare the difference of using as antagonist a shortly trained agent for 15k with a more experienced agent trained for 50k, and then infer if this more experienced agent can guide the protagonist faster to better results.

Lastly, we check a midterm solution, using as antagonist an agent trained for 35k and then training the protagonist for 35k and comparing the gain using the UED method for more episodes, also comparing the difference with the normal agent trained for 35k.

5.2.1 Comparison in the gain of using UED

Table 5.1: Left: Normal training for 50k episodes. Right:UED Training for 15k episodes using an antagonist trained for 35k episodes

Env	Normal Training	UED Training
Env 1.1		
Env 2.2		
Env 3.3		

Comparing side by side at 5.1 the normal agents trained for 50k and the UED agents trained for 15k, using as antagonist an agent trained for 35k, it shows

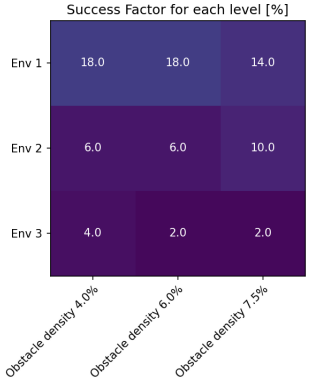
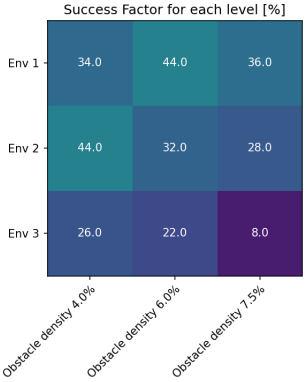
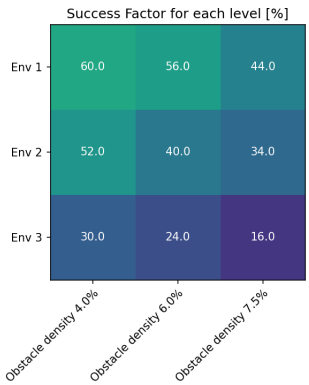
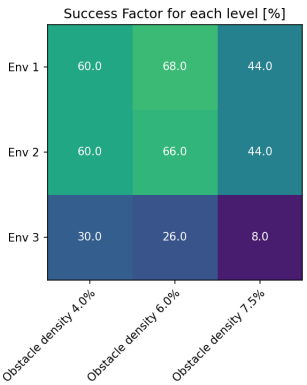
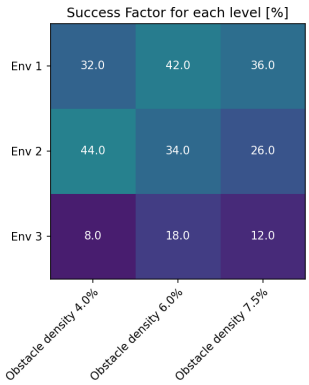
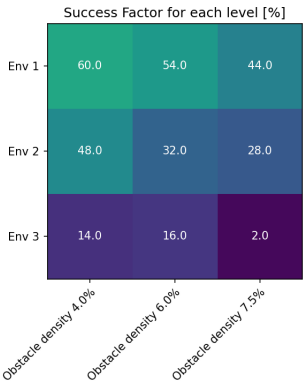
that for practically every environment configuration in the tests, the normal agent performs better than the UED agent. In fact, the normal agents trained in env 2.2 and env 1.1 have success factors surpassing 70 and 60 percent, respectively in the upper left quadrant, while in the rightmost and down most lines, the values are quite lower. One explanation is that results for environments with the highest obstacle density will naturally have lower values of success factor since they are the hardest levels. And secondly, the same explanation may apply for environment size 3. Since it is the larger one, it requires a longer path to reach the target and therefore, there is a long path where the agent can crash.

For the agents trained with env 3.3, it can be noticed that this is the configuration that has less difference between the two methods and that even the normal agent being trained only on levels with the same configuration did not have a good performance on levels of size 3.

However, despite the lower performance compared to the normal agents, the UED agents had a more consistent performance. That is, the difference in success factor between test environments is lower than the difference in the tests of the normal agents. Another point regarding the difference in performance is that 15k is probably not enough for the agent to learn a good policy, and therefore, even using an antagonist trained for 35k episodes, the summed knowledge is not sufficient to reach the same results of training an agent without UED for 50k episodes.

5.2.2 Comparison of antagonist agents trained for different number of episodes

Table 5.2: Left: UED Training for 15k episodes using an antagonist trained for 35k episodes. Right: UED Training for 15k episodes using an antagonist trained for 50k episodes

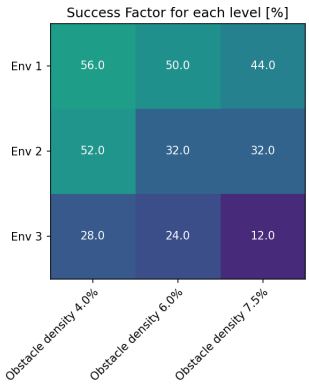
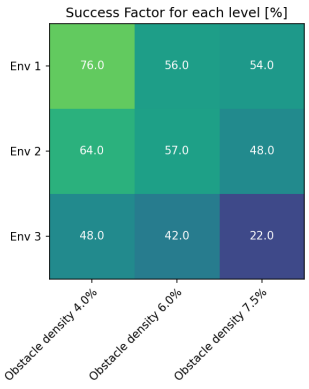
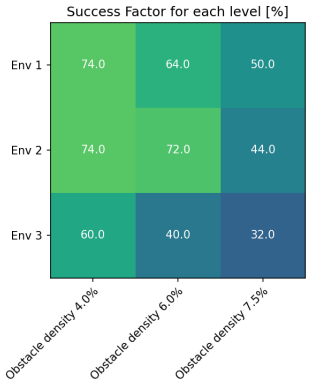
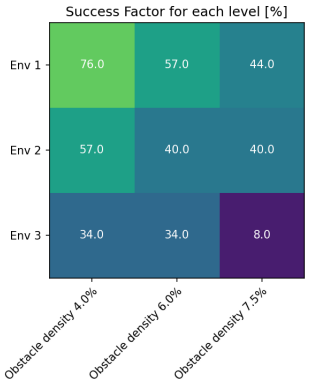
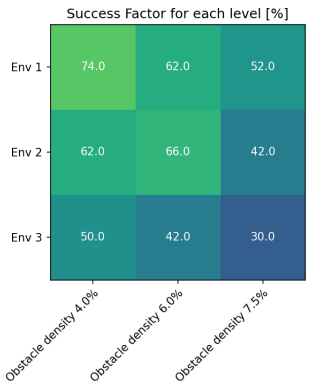
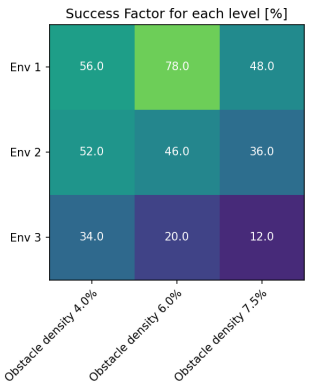
Env	35k eps antagonist	50k eps antagonist
Env 1.1		
Env 2.2		
Env 3.3		

The comparison at 5.2 between agents trained for 15k episodes with UED using antagonists trained for 35k and 50k episodes shows that for those parameters, a more experienced antagonist can lead to better success factor rates for practically every test level. The analysis, however, does not show a considerable gain in generalization since the performance in the test levels that are different from the training did not have a great gain in performance. Both approaches in every configuration showed a poor success factor in environment 3.3.

A possible explanation is that the parameters chosen for the UED training, with only 15 episodes, are not enough to reach environments with sizes close to size 3 and, therefore, may not perform well enough at this environment size.

5.2.3 Results of the midterm solution

Table 5.3: Left: Normal agent trained for 35k episodes. Right: UED Training for 35k episodes using an antagonist trained for 35k episodes

Env	Normal	UED
Env 1.1		
Env 2.2		
Env 3.3		

This analysis aims to compare agents trained for the same number of episodes, but with and without UED, and, for that, in the UED training, we use as antagonists

the same non-UED agent of the analysis.

This comparison at 5.3 shows that for the first agent, trained at env 1.1, the UED method provided considerable leverage in the overall performance. For example, every test level had better results than the normal training in environment size 2 with 6% obstacle density, going from 32% to 57% of success factor. The UED also showed a better generalization for this environment configuration since test results in the other environments different from env 1.1 performed better than the normal agent.

For the other agents of environments 2.2 and 3.3, which already had higher values of success factor to practically every test configuration, at least in comparison with every result achieved by every agent, the UED training did not achieve the same performance as the normal agent. It showed a downgrade in the generalization, with the agent not being able to pass most of the test levels at env 3.3.

It is possible that the antagonist agent was too good to help the training of the protagonist agent and that a better approach would be to use antagonists that are good in performance but not too good.

5.2.4 Difference of environment configuration for training without UED

Comparing all the trained agents without the UED algorithm, analyzing which environment configurations provided better results is useful. This analysis is not straightforward because a higher number of training episodes is usually expected to provide better results. In the tests presented in this work, some agents trained for 50k episodes had worse results than 35k episodes, which may show over-fit in training.

The agent with the highest success factor in one of the test sets was agent 2.2 trained for 50k episodes, which had 84% and 82% in environments 2.1 and 2.2, respectively. This agent also had success factors above 50% for every test set except env 3.3, which shows a considerably good ability for generalization.

Agents trained for 35k episodes at environments 2.2 and 3.3 have also shown considerably high success factors in all test sets, with values ranging from 30% to 74%, while agent 1.1 trained for 35k episodes and agent 3.3 trained for 50k episodes had lower performances.

These results show that the choice of environment can significantly affect the

agent's performance and that environment 2.2 was roughly the one that provided better results, probably for being the one in the middle of the possible parameters, since it is a mid-sized environment with a number of obstacles that is neither too high nor too low.

5.2.5 Difference in the antagonist environment configuration for training with UED

As seen in the results obtained, the choice of antagonist has shown to be an important factor in the performance of the agent trained with UED. Therefore, the environment configuration used to train the antagonist is also important.

For every training configuration, the antagonists trained in the environment 2.2 had good results in comparison with the other antagonists, and this has an effect on the protagonist as shown in 5.4, since even trained for fewer episodes, the protagonist shows better success factors for the majority of level configurations. However, it does not succeed in the hardest level. For the other protagonists, with antagonists of environment 1.1 and 3.3, both do not achieve satisfactory results when trained for only 15k episodes using antagonists trained for 35k episodes but increase the performance when using an antagonist trained for 50k, which implies that even trained for the same amount of episodes, the performance increases when led by a better antagonist.

Another aspect is that there is a clear gain when using the 35k antagonist, from training the protagonist for 15k episodes to 35k episodes in most levels but the last one, which may imply that the chosen parameters for the UED algorithm were not sufficient to generalize to the most challenging level. This may be because 35k episodes were not enough to evolve to a configuration as hard as the hardest level in the tests.

5.2.6 General results obtained for the trajectory planning task

This subsection aims to show the overall results achieved in fulfilling the trajectory avoidance task, regardless of the method used as training, but only the performances achieved in succeeding the test sets.

For every reinforcement learning problem, finding a good simulation environment, simulation parameters, and mainly finding a suitable reward function presents challenging tasks to achieve a nearly optimal policy. Although one of the goals of this work is to exploit the idea of Unsupervised Environment Design to find

robust policies, it is also worth discussing the overall performance of the trajectory planning problem.

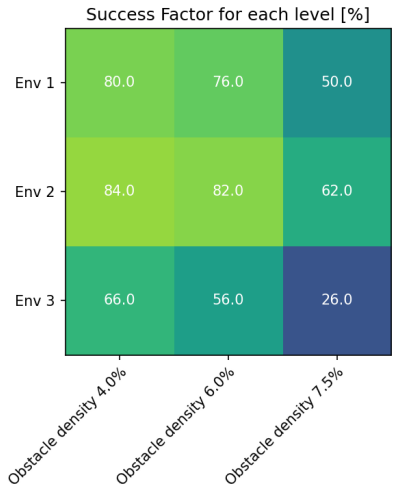


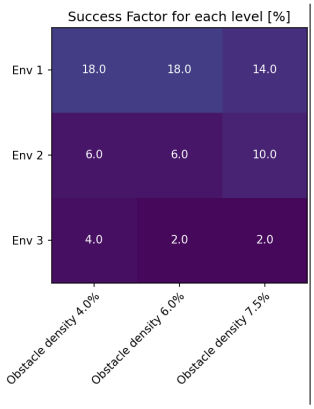
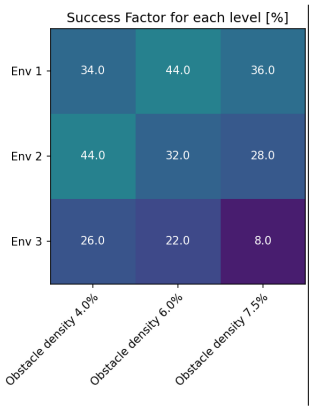
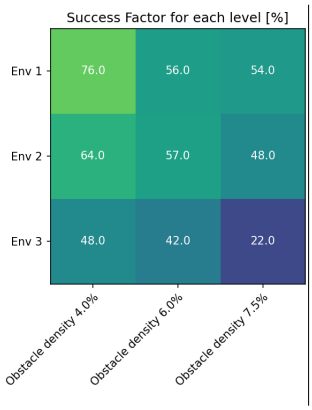
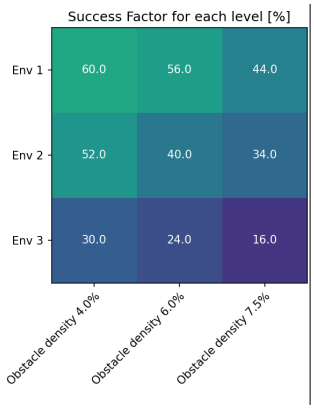
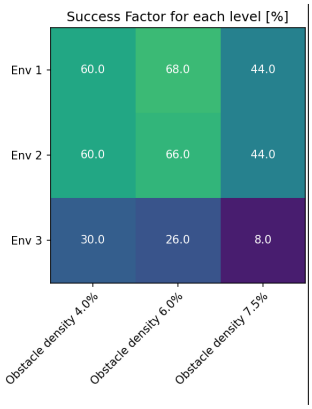
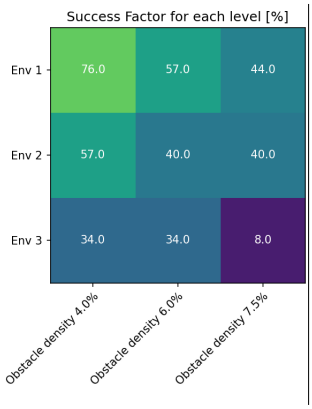
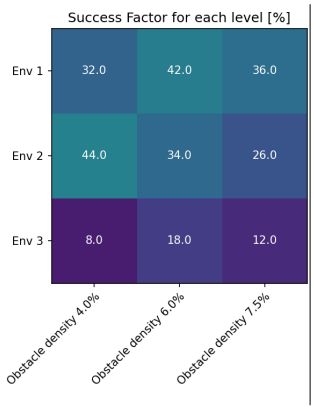
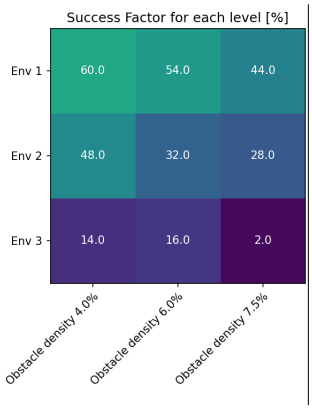
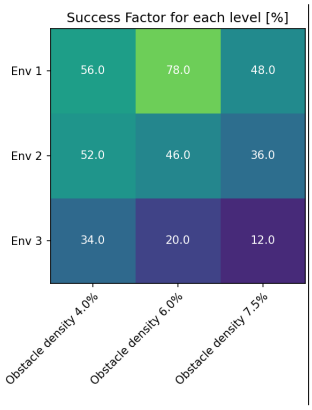
Figure 5.1: Heatmap of the agent with best overall performance: trained for 50k episodes at env 2.2

Considering only our proposed task, the trajectory planning problem, some of the agents had considerably good results at most levels. The normal agent trained for 50k episodes at env 2.2 has shown the best rates of success factor at low obstacle density levels but also great results in the hardest levels, 84% at env 2.1 and 82% at env 2.2, which are considered good rates. However, this agent was not able to generalize well enough for the hardest environment, with a SF of 26%, while the normal agent trained for 50k episodes at env 1.1 had worse results, in comparison, at easier environments but with a 40% SF at the hardest levels, which, considering the obstacle density, maybe a satisfactory result. As shown at 5.5, the overall SF, the mean of success factors, is considerably lower due to the last environment.

Episodes Trained	Env Trained	$mean(SF)$
50k	2.2	64.7%
35k	2.2	56.7%
35k	3.3	53.3%
50k	1.1	51.3%

Table 5.5: Summary of results obtained by the training without UED. Comparison of environment trained and mean of Success Factor in the test levels.

Table 5.4: Left: UED Training for 15k episodes using an antagonist trained for 35k episodes. Center: UED Training for 15k episodes using an antagonist trained for 50k episodes. Right: UED Training for 35k episodes using an antagonist trained for 35k episodes

Env	15k eps	15k eps	35k eps
Env 1.1			
Env 2.2			
Env 3.3			

Chapter 6

Conclusions and Outlook

In this section, the overall outcomes of this study regarding the intended objectives are analyzed, along with the difficulties in implementing RL solutions in real-world scenarios. Finally, future directions for this research are proposed.

The goal was to develop a 2D simulation environment to train and test reinforcement learning algorithms to solve trajectory planning for quadrotors. In addition to this, it was required to easily change environment parameters with the idea of developing more robust solutions. To promote this, an Unsupervised Environment Design framework was implemented.

6.1 Conclusions of the developed work

6.1.1 Development of a drone simulation for training and testing

The 2D python drone simulation was successfully developed. Although its development consumed a considerable amount of time, the simulation was able to simulate basic quadrotor movements, not considering the degrees of freedom related to the z-axis. It simulated a LIDAR-like sensor, which can also be seen as several distance sensors. The decision to move to a custom-designed simulation is motivated by the only purpose of testing an UED framework. Similar off-the-shelf simulations available could have been used, but none with the advantage of simulating distance sensors and also with the facility to change the environment parameters, which supports this work's initial idea to develop its simulation. Furthermore, the possibility of controlling all the simulation parameters could allow the future to add additional features, such as realistic quadrotor dynamics. The latter, more specifically, was not developed due to the increased difficulty of finding a more robust solution and

increased simulation time.

6.1.2 Solving the TP problem

Regardless of the method applied, it can be observed that the trajectory planning problem and obstacle avoidance with the aid of onboard distance sensors, considering only the 2D problem, have been solved to a certain extent, as demonstrated by the different difficulty levels available in the experiments. This is particularly noteworthy when considering the highest difficulty level, as multiple agents could attain satisfactory performances. These results demonstrate the potential of RL in solving real-world problems, such as trajectory planning and obstacle avoidance, efficiently and effectively.

6.1.3 Ability of Generalization

With respect to the general ability of generalization, it is possible to say that both UED and non-UED agents were able, at some level, to provide an increase in generalization between environments. Still, no one had the best performance at all of them; that is, the tests done in this work were not able to find an agent good enough, so it can be said that it is robust enough to all of the test levels, but only that it achieved good performance in the levels with smaller difficulty.

Therefore, the realized tests did not provide a one-shot solution since choosing and testing a few training configurations for the antagonist agent is still necessary. What this work achieved is that the results with the UED solution are more robust with respect to the initial parameters in comparison to a random trial and error training, meaning that even a not-so-good choice of parameters for the antagonist agent may still yield better results when using UED in comparison to just training the agent normally without some prior knowledge of which parameters to choose.

6.2 Constraints to bringing the agent into real environments

Simulation plays a crucial role in developing real-world applications, enabling safe testing without the risk of accidents and allowing for almost limitless experimentation. However, it is important to keep in mind that there may be limitations when transitioning from simulation to deployment, such as accurately modeling real-world dynamics such as friction and inertia. Considering all assumptions made during the development process, another aspect is that most applications have complex, multi-dimensional structures. It is important to test the application in

all possible degrees of freedom before deployment.

One reason the training of ML models is done offboard is that training is much faster on desktop computers or the cloud. Onboard computers, used for actual application deployment after training, typically have limited computational resources. Moreover, it is not guaranteed that a test done in a simulation on a computer will have the same results when deployed on the application. If an algorithm on simulation can perform two actions per second and, when deployed in an onboard computer, is only able to compute one action per second, then probably this one action will not be enough to change the robot's state to avoid a collision for example, since it would probably have been necessary to make a greater adjustment for this amount of time. Therefore it is essential to simulate and test the developed algorithms under the real conditions in which they will be deployed. Concerning neural networks, some well-known adjustments and different implementations exist when considering real-time systems that provide faster and lighter solutions. Unfortunately, these optimizations may come at the cost of reduced algorithm precision and must be carefully evaluated.

Another aspect is the presence of imperfections in real-world sensors. Sensors and cameras need calibrations to increase their accuracy and precision, being also a challenge to deal with uncertainties in data acquisition, which require special attention. Uncertainties and imperfections may not come only from sensors but also from external factors such as wind and rain; which are particularly relevant in a scenario of outdoor operating UAVs.

6.3 Next steps of this work

The next steps of this work can be divided into three main parts:

- The reward function successfully guided the agent to solve the trajectory planning problem but encountered difficulty in making complex decisions such as navigating around obstacles, making large turns, and adjusting trajectory to reach the destination. Further refinement of the reward function is necessary to address these challenging scenarios.

- The implementation of realistic dynamics and a 3D simulation is understood to be the biggest required step to deploy the decision-making agent into hardware. The ability to simulate real-world conditions is crucial for the agent to perform effectively in hardware. A 3D simulation provides a visual representation of the environment and the agent's behavior, allowing for an accurate assessment of performance and potential areas for improvement. Furthermore, implementing

realistic dynamics also requires integrating accurate sensory data, which the agent can use to make informed decisions. This helps ensure that the agent's behavior in the simulation closely reflects its behavior in the physical world, reducing the risk of unexpected results when deployed in hardware.

- Considering the development of Unsupervised Environment Design, it is recommended, to exploit the advances of robustness, to introduce more parameters to be changed in the environments. In this case, some ideas would be to make the size of obstacles changeable; introduce the possibility of making the objects move and their speeds change. Those modifications make the environment more difficult to be tuned, therefore making the UED even more parametric. It is also desirable to test the algorithm on different applications for better tuning of the parameters, such as the rate at which the environment gets more difficult.

In conclusion, this study successfully achieved its overall objective, demonstrating the potential of the proposed approach in solving complex problems that require extended environment development and design. However, it is acknowledged that there are still areas for improvement, as identified above. These areas provide valuable opportunities for further refinement and optimization, leading to even better performance in the future. In sum, this thesis provides a solid foundation for future research in the field of Unsupervised Environment Design and Trajectory Planning in the context of Reinforcement Learning and highlights the significance of ongoing advancements.

Bibliography

[1] B. Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A.Al Sallab, Senthil Yogamani, and Patrick Perez. «Deep Reinforcement Learning for Autonomous Driving: A Survey». In: *IEEE Transactions on Intelligent Transportation Systems* 23 (6 June 2022), pp. 4909–4926. ISSN: 15580016. DOI: 10.1109/TITS.2021.3054625 (cit. on p. 1).

[2] Lu Dong, Zichen He, Chunwei Song, and Changyin Sun. «A review of mobile robot motion planning methods: from classical motion planning workflows to reinforcement learning-based architectures». In: (Aug. 2021). URL: http://arxiv.org/abs/2108.13619 (cit. on p. 1).

[3] *Robotics*. Springer, 2019 (cit. on pp. 2, 5).

[4] *Introduction to Autonomous Mobile Robots*. MIT Press, 2011 (cit. on p. 5).

[5] Peter Corke. *Robotics,Vision and Control Fundamental Algorithms in MATLAB*. Springer, 2016 (cit. on pp. 5, 7).

[6] *Amazon Robotics Utiliza Amazon SageMaker e AWS Inferentia para Habilitar Inferências de ML em Escala*. 2022. URL: https://aws.amazon.com/pt/solutions/case-studies/amazon-robotics-case-study/ (cit. on p. 6).

[7] *Google Now Owns The 'Largest Residential Drone Delivery Service In The World'*. 2021. URL: https://www.forbes.com/sites/johnkoetsier/2021/08/25/google-now-owns-the-largest-residential-drone-delivery-service-in-the-world/?sh=2dac07e41197 (cit. on p. 6).

[8] Amit Patel. *Introduction to A**. May 2014. URL: https://www.redblobgames.com/pathfinding/a-star/introduction.html (cit. on p. 9).

[9] *Reinforcement learning: An introduction*. MIT press, 2018 (cit. on pp. 14, 15, 22).

[10] Shirish Chinchalkar. «An Upper Bound for the Number of Reachable Positions». In: (1996) (cit. on p. 19).

[11] *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016 (cit. on pp. 23, 27).

[12] Alberto Garcia-Garcia, Sergio Orts-Escolano, Sergiu Oprea, Victor Villena-Martinez, and Jose Garcia-Rodriguez. «A Review on Deep Learning Techniques Applied to Semantic Segmentation». In: (Apr. 2017). URL: http://arxiv.org/abs/1704.06857 (cit. on p. 23).

[13] Christopher J C H Watkins and Peter Dayan. *Q-Learning*. 1992, pp. 279–292 (cit. on p. 30).

[14] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. «Playing Atari with Deep Reinforcement Learning». In: (Dec. 2013). URL: http://arxiv.org/abs/1312.5602 (cit. on pp. 31, 35).

[15] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. «Asynchronous Methods for Deep Reinforcement Learning». In: (Feb. 2016). URL: http://arxiv.org/abs/1602.01783 (cit. on p. 33).

[16] David Silver et al. «Mastering the game of Go with deep neural networks and tree search». In: *Nature* 529 (7587 Jan. 2016), pp. 484–489. ISSN: 14764687. DOI: 10.1038/nature16961 (cit. on p. 35).

[17] Alejandro Rodriguez-Ramos, Carlos Sampedro, Hriday Bavle, Paloma de la Puente, and Pascual Campoy. «A Deep Reinforcement Learning Strategy for UAV Autonomous Landing on a Moving Platform». In: *Journal of Intelligent and Robotic Systems: Theory and Applications* 93 (1-2 Feb. 2019), pp. 351–366. ISSN: 15730409. DOI: 10.1007/s10846-018-0891-8 (cit. on p. 35).

[18] Hartmut Surmann, Christian Jestel, Robin Marchel, Franziska Musberg, Houssem Elhadj, and Mahbube Ardani. «Deep Reinforcement learning for real autonomous mobile robot navigation in indoor environments». In: (May 2020). URL: http://arxiv.org/abs/2005.13857 (cit. on p. 35).

[19] Bardienus P. Duisterhof, Srivatsan Krishnan, Jonathan J. Cruz, Colby R. Banbury, William Fu, Aleksandra Faust, Guido C. H. E. de Croon, and Vijay Janapa Reddi. «Learning to Seek: Autonomous Source Seeking with Deep Reinforcement Learning Onboard a Nano Drone Microcontroller». In: (Sept. 2019). URL: http://arxiv.org/abs/1909.11236 (cit. on p. 35).

[20] Yunlong Song, Mats Steinweg, Elia Kaufmann, and Davide Scaramuzza. «Autonomous Drone Racing with Deep Reinforcement Learning». In: (). URL: https://youtu.be/Hebpmadjqn8. (cit. on p. 35).

[21] Florian Fuchs, Yunlong Song, Elia Kaufmann, Davide Scaramuzza, and Peter Durr. «Super-Human Performance in Gran Turismo Sport Using Deep Reinforcement Learning». In: *IEEE Robotics and Automation Letters* 6 (3 July 2021), pp. 4257–4264. ISSN: 23773766. DOI: 10.1109/LRA.2021.3064284 (cit. on p. 35).

[22] Marcin Andrychowicz et al. «Learning Dexterous In-Hand Manipulation». In: (Aug. 2018). URL: http://arxiv.org/abs/1808.00177 (cit. on p. 36).

[23] Antonio Loquercio, Elia Kaufmann, Rene Ranftl, Alexey Dosovitskiy, Vladlen Koltun, and Davide Scaramuzza. «Deep Drone Racing: From Simulation to Reality with Domain Randomization». In: *IEEE Transactions on Robotics* 36 (1 Feb. 2020), pp. 1–14. ISSN: 19410468. DOI: 10.1109/TRO.2019.2942989 (cit. on p. 36).

[24] Michael Dennis, Natasha Jaques, Eugene Vinitsky, Alexandre Bayen, Stuart Russell, Andrew Critch, and Sergey Levine. «Emergent Complexity and Zero-shot Transfer via Unsupervised Environment Design». In: (Dec. 2020). URL: http://arxiv.org/abs/2012.02096 (cit. on p. 37).

[25] Jack Parker-Holder, Minqi Jiang, Michael Dennis, Mikayel Samvelyan, Jakob Foerster, Edward Grefenstette, and Tim Rocktäschel. «Evolving Curricula with Regret-Based Environment Design». In: (Mar. 2022). URL: http://arxiv.org/abs/2203.01302 (cit. on p. 38).