UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

TRABAJO FIN DE MÁSTER

Máster Universitario en Teoría de la Señal y Comunicaciones

Design and Implementation of a Self-Navigation System using Deep Reinforcement Learning

Daniel Tapia Martínez

2019

TRABAJO FIN DE MÁSTER

TÍTULO (Español): Diseño e Implementación de un Sistema de Navegación Autónoma usando Aprendizaje por Refuerzo Profundo

TITLE (English): Design and Implementation of a Self-Navigation System using Deep Reinforcement Learning

AUTOR: D. Daniel Tapia Martínez

TUTOR: D. Juan Parras Moral

DEPARTAMENTO: Signals, Systems and Radiocommunications (SSR) UPM

TRIBUNAL:

Presidente:

Vocal:

Secretario:

Suplente:

FECHA DE LECTURA:

CALIFICACIÓN:

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

TRABAJO FIN DE MÁSTER

MÁSTER UNIVERSITARIO EN TEORÍA DE LA SEÑAL Y COMUNICACIONES

Design and Implementation of a Self-Navigation System using Deep Reinforcement Learning

Daniel Tapia Martínez

2019

Summary

Navigation is central to many research areas as well as many industrial interests. As a research topic, it attracts the attention of different fields like Artificial Intelligence, Robotics or Sequential Decision Making. And it is key to many potentially groundbreaking applications in Autonomous Driving or Domestic Robots.

In this work, we approach the problem of partially observable navigation with a reactive system trained by model-free Reinforcement Learning. The advantages of this learned approach include reducing the engineering effort at the cost of more computing power during training. We designed and built from scratch an agent and an environment focusing on being able to navigate independently of the map. We used advanced Reinforcement Learning algorithms to learn to navigate and achieved promising results. We performed other experiments to test the generality and transfer-ability to other tasks.

Our results show that several algorithms can reach the target in our navigation setup more than 85% of the episodes. Hence, these algorithms may provide a significant step forward towards autonomous navigation systems. We make use of techniques like Curriculum Learning or Transfer Learning to speed up and improve training. Finally, we achieved our original goal of making a Neural Network control a drone flying in a simulator without colliding with obstacles.

Keywords

Reinforcement Learning, Navigation, Partial Observability, UAV, Curriculum Learning, Transfer Learning, Control Theory

Resumen

La navegación es fundamental para muchas áreas de investigación, así como para muchos intereses industriales. Como tema de investigación, atrae la atención de diferentes campos como la Inteligencia Artificial, la Robótica o la Toma de Decisiones Secuencial. Y es clave para muchas aplicaciones potencialmente revolucionarias como la conducción autónoma o los robots domésticos.

En este trabajo, abordamos el problema de la navegación en un entorno parcialmente observable con un sistema reactivo entrenado por el Aprendizaje por Refuerzo sin modelo. Las ventajas de este enfoque aprendido incluyen la reducción del esfuerzo de ingeniería a cambio de una mayor carga computacional durante el entrenamiento. Diseñamos y construimos desde cero un agente y un entorno enfocados a poder navegar independientemente del mapa. Utilizamos algoritmos avanzados de Aprendizaje por Refuerzo para aprender a navegar y logramos prometedores resultados. Realizamos otros experimentos para probar la generalidad y la capacidad de transferencia a otras tareas.

Nuestros resultados muestran que varios algoritmos pueden alcanzar el objetivo en nuestra configuración de navegación en más del 85% de los episodios. Por lo tanto, estos algoritmos pueden suponer un importante paso adelante hacia los sistemas de navegación autónomos. Utilizamos técnicas como el Aprendizaje Curricular o la Transferencia de Aprendizaje para acelerar y mejorar el entrenamiento. Finalmente, logramos nuestro objetivo original de hacer que una Red Neuronal controle un dron volando en un simulador sin chocar con obstáculos.

Palabras clave

Aprendizaje por Refuerzo, Navegación, Observabilidad Parcial, UAV, Aprendizaje por Curriculum, Transferencia de Conocimiento, Teoría de Control

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Navigation is central to many research areas as well as many industrial interests. As a research topic, it lays in the intersection between different fields like Artificial Intelligence, Robotics or Sequential Decision Making. And it has many potentially groundbreaking applications in Autonomous Driving, Domestic Robots or the many applications that will be unlocked with autonomous aerial vehicles.

It has been a challenge for many years to create systems that can traverse an environment without having previously explored it as humans do. Some well-known names, like Claude Shannon, gave their shot at it with "Theseus" a magnetic mouse controlled by relay circuits that navigated a maze [Shannon, 1950]. Classical Navigation approaches developed algorithms to traverse a known environment, this field of study is called Planning Algorithms [LaValle, 2006]. Later other approaches to navigating unknown environments relayed on creating an internal representation of the environment and then use planning algorithms to find the best route. This is the example of Simultaneous Localization And Mapping (SLAM) approaches [Cadena et al., 2016]. However, these methods face the additional problem of accurately estimating the representation of the environment apart from navigating in it. This puts them in a disadvantage with the computation cost and memory tax of these systems, which makes them harder to fit in small and low-powered embedded systems such as the processor unit in a small Unmanned Aerial Vehicle.

There is another approach called reactive system, that draws inspiration from animals, where the agent navigating the environment does not have an internal representation but acts accordingly to the observations. This is the approach we are interested in.

To do so, we will be using a series of learning algorithms that lets us tackle complex problems without too much domain knowledge, and let the machine figure out the best solution. More specifically we will be using Reinforcement Learning (RL). RL distinguishes from other branches of machine learning because it learns by trial and error. Meanwhile, other types of algorithms learn by trying to adjust to already labelled data or trying to find hidden structures in unlabeled data, but ours will get the data from experience. By training in a simulator, our agent has almost unlimited experience at its disposal to learn. In this work, we dive deep into the theory and practice of using RL methods, and we try to solve the complex task of Navigation with the intent of creating a system that can make an Unmanned Aerial Vehicle fly inside a simulator avoiding obstacles.

During the development of this thesis, we go from the basic intent to a solid theoretical framework that proves that it is possible to do it. Then we implement several systems we will use to teach the agent. For this, we had to research the latest related work and find what were the best tools to make everything work. Some of the skills involved and improved over the course of the thesis include algebra, statistics, programming or system administration.

Finally, the experiments carried out aim to fulfill the original intent of flying a UAV inside a simulator without colliding with obstacles, doing it without explicitly programming it how to do so but using learning algorithms. To complete this goal we had to combat the limitations of the learning methods the simulator, and other problems that shaped the course of the thesis. The experiments answer questions like "which algorithm is better for the task?", "is it possible to learn navigation?" or "how can we make everything faster so that it is feasible to use RL?". The results are quite positive although not ready for industry-standards, but they prove that RL may be a valuable research direction in the path for better navigation algorithms.

A video of the final result can be found in Fig .1.1 or in the url: `https://youtu.be/CfLkdpxGZgA`.

1.1 Problem Description

Now to specify more in depth what are we exactly solving we are going to define the problem.

Navigation is the field that studies the skills and techniques needed by a mobile system to traverse an environment. The task of going from one point to another without colliding with the environment is called planning [LaValle, 2006]. This involves creating paths at different ranges of distances. To start

Fig. 1.1: Video of the Transferred policy. To watch it scan the QR code or go to this url: `https://youtu.be/CfLkdpxGZgA`.

simple, we are going to focus on a 2D map, short distances and few obstacles. This is usually referred to as local planning. The reason to have relatively small distances and few obstacles is that the agents do not need too many extra skills like memory to reach the target. Opposed to this we would have mazes which require not only the motor part but also identifying where you are inside the maze. But for now, we are more interested in the tasks of obstacle avoidance, which consist of moving without collisions, and PointGoal [Anderson et al., 2018], which consists on getting to the goal when the agent knows where is it but not necessarily the environment.

We are going to formalize the definition of PointGoal as a task in where the agent will have to move in the environment to reach the goal point $\|\vec{d}\| < \varepsilon$ with a controlled speed $\|\vec{v}\| < \epsilon$, with $\vec{d}$ being the distance to the goal vector, $\vec{v}$ being the velocity vector, ε being a distance threshold and ϵ a speed threshold. Sometimes we will only require the agent to reach the target and not care about the speed. This task is trivial if there are no obstacles in the environment but [LaValle, 2006] mentions that navigating in a cluttered environment is an NP-hard problem as shown in Fig. 1.2.

Obstacle Avoidance is achieved when there are no collisions in the trajectory. We will measure it in the training by the number of episodes that end in a collision, the episodes that end in the target, the episodes that end with the maximum time allowed T_{max} or the episodes that end with the agent out of bounds.

1.2 Applications and Impacts

Technology is shaping the social, economic and other landscapes that affect us. Technologies like machine learning can provide a huge increase in efficiency in different processes, bring great impact to numerous industries and change the world as we know it today. The field of Reinforcement Learning offers great potential in taking part in all those changes. This family of methods provides generality and the ability to leverage compute power to reduce engineering efforts in solving control problems. And with Moore's law this advantage takes special importance [Dean, 2017].

The good part about using Reinforcement Learning is that in exchange for computing power we can make more robust and fine-tuned systems, automatically and without expert knowledge. But we also have to take into account some difficulties that, if they are not taken care of, can make our system work in unintended and erratic ways.

As for today in 2019, there has not been a match from the industry side with the use of RL in academia. We have seen some industrial applications, but not to the extent of supervised learning methods. Some examples include reducing the cooling bill of data-centers [DeepMind, 2016] or improving the notification sending system in Facebook [Gauci et al., 2018]. Other achievements have been in the field of games or video-games, which pose interesting and challenging control problems from a research perspective [Silver et al., 2016], [OpenAI, 2018], [Vinyals et al., 2019].

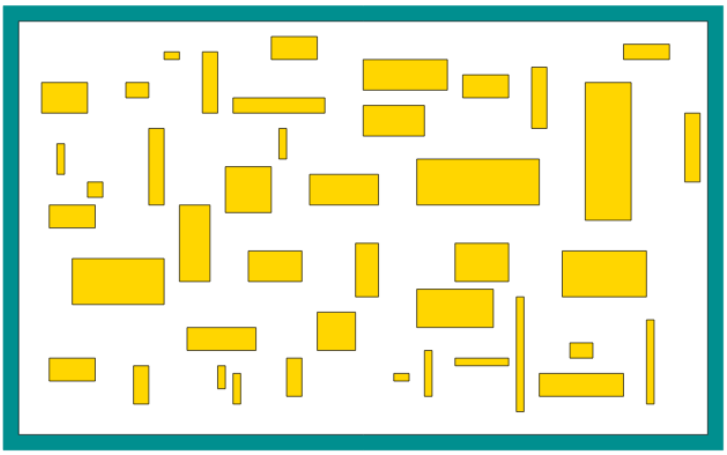

Figure 6.41: Even motion planning for a bunch of translating rectangles inside of a rectangular box in $\mathbb{R}^2$ is PSPACE-hard (and hence, NP-hard).

Fig. 1.2: Figure 6.41 of the book [LaValle, 2006] where the computational complexity of 2D Navigation is mentioned in the chapter of Complexity of Motion Planning.

Regarding the Unmanned Aerial Vehicles, we are talking about a technology that has the advantage of a small, mobile and long range tools that have little action-ability but can be used for getting visual information. So the applications where this would excel could be related to surveillance, maintenance or exploration. But some other UAV's may have different characteristics like a larger autonomy and more carrying power, so they could be used for delivery or even human transport.

(a) Manteinance *(b)* Blood Delivery *(c)* Military

Fig. 1.3: Concrete applications of drones are a reality.

Again, we are talking about a technology that could disrupt and improve many sectors like Agriculture, Emergency Response, Infrastructure Maintenance or Delivery. But when used wrong or with bad intent can cause harm and problems for society. As with most of the things in life, the famous quote from Uncle Ben in Spider-man applies: "with great power, comes great responsibility" and we have the responsibility to make this world a better place for all of us.

Some more concrete examples of these applications shown in Fig 1.3 are quickly inspecting the electric grid and windmills with drones [1], delivering blood transfers by drone [2] or a more negative one with drone terrorist attacks[3].

The structure of the work goes as follows: in Chapter 1 we have introduced and defined the problem, in Chapter 2 we discuss the related work. In Chapter 3 we set the theoretical background we are following. Chapter 4 describes our system and the design decisions involved. In Chapter 5 we analyze and comment on the results and finally in Chapter 6 we draw conclusions and point future lines.

[1] "https://cleantechnica.com/2018/09/06/drones-ai-used-to-quickly-inspect-wind-turbines/"
[2] "https://www.youtube.com/watch?v=bnoUBfLxZz0"
[3] "https://www.forbes.com/sites/zakdoffman/2018/12/27/forget-gatwick-why-the-deadliest-terrorist-threat-from-drones-is-no
"

2. RELATED WORK AND STATE OF THE ART

2.1 Related Work

In this section we are going to review the previous work done on the fields of Navigation, Deep Reinforcement Learning applied to Navigation and a few more specific topics like Curriculum Learning or Transfer Learning.

2.1.1 Classic Navigation

Navigation is an old field with extensive research literature. There have been many approaches throughout history. The two most used ones in robotics are planning and mapping and sometimes the integration of these two [Tordesillas et al., 2018].

Planning means creating a path to follow when the environment is already known. The basics are well covered in sources like [LaValle, 2006]. More advanced methods include creating feasible smooth trajectories by solving a minimization problem [Van Nieuwstadt and Murray, 1998] and [Mellinger and Kumar, 2011] or searching over the state-space of solutions [Liu et al., 2018] or [Liu et al., 2017].

These approaches assume the global map is known, others called *Mapping* use a global planner or an exploration based planner that gives the next way-point to the local planner. These approaches include memoryless methods that only act on instantaneous sensing data [Dey et al., 2016], like in our approach, but then using planning methods to create trajectories. Similar approaches called *reactive* use a memoryless representation of the environment and choose its trajectory from closed form solutions [Lopez and How, 2017]. This last type of methods is the ones we are trying to learn with Reinforcement Learning.

It is also important to name the SLAM methods [Cadena et al., 2016] mentioned in the introduction, that consists on the construction of a model of the environment and the estimation of the state of the agent moving in it. Which are a key part in navigation systems to apply planning methods in previously unknown environments.

2.1.2 RL impact applications

In Section 1.2 we briefly talked about some of the impact and applications of Reinforcement Learning. We mentioned the academic impact RL is also being consequent to the growth of Machine Learning research being done in the last few years [MIcrosoft,]. Although for RL there has not been an equal growth in industrial or commercial applications we can mention again the work from Deepmind with games like go or chess [Silver et al., 2016], videogames like Starcraft [Vinyals et al., 2019] or reducing the cooling bill of datacenters [DeepMind, 2016]. Work from OpenAI with videogames like Dota [OpenAI, 2018] or robotics [Andrychowicz et al., 2018]. Or work from Facebook improving production systems like the notifications in a social platform [Gauci et al., 2018]. And another very interesting trend is RL systems to find the best combination of parameters or pieces to perform a task, similar to black-box-optimization. And it is being used in the search for better Neural Net architectures [Zoph and Le, 2016] for example.

2.1.3 Learning Navigation

In this work we use Reinforcement Learning to learn a navigation system like the ones on 2.1.1 without having to program them, there is similar work on the research community. The survey of [Anderson et al., 2018] was very helpful to understand the problem and find new approaches.

There has been recent interesting work on trying to learn to navigate with different representations of the map, we analyze some.

[Oh et al., 2016] poses simple tasks of navigating a grid-like maze in a Minecraft(videogame) environment and solves it with different DRL architectures that use external memory.

In [Parisotto and Salakhutdinov, 2017], they navigate a grid-maze using an architecture that uses writable memory along with a neural network and compares it to standard DRL methods. [Gupta et al., 2017a] and [Gupta et al., 2017b] purpose some kind of "learned-SLAM" where the feature synthesis and path planner are learned and used for goal-driven navigation.

Other approaches have been learning to navigate with implicit representations using general Reinforcement Learning (RL) algorithms. For example:

We have [Jaderberg et al., 2016] and [Mirowski et al., 2016] where an RL agent improves his training time and performance by learning with auxiliary tasks. In [Zhu et al., 2017] apart from using Actor-Critic methods to learn target driven navigation, they can transfer the learned policy with some fine tuning as we do in this work.

[Sadeghi and Levine, 2016] is interesting to us because they train a Collision-free indoor flight model in real 3D CAD environments from images to actions, they also transfer to a real setting by using domain randomization. Our work also transfers domains and shares some points with this work.

[Karkus et al., 2018] brings together the ideas of RL and motion planning by integrating a differentiable motion planning algorithm in the policy that can be trained end-to-end.

There was a very interesting approach to navigation by trying to predict future frames in [Dosovitskiy and Koltun, 2016] but it was using supervised methods, their performance was good in previously unseen DOOM(videogame) levels in a competition.

Although some of these works see the problem of navigation from a more high-level perspective, they provide interesting points and references to understand this work.

2.1.4 Curriculum Learning

We talk more in-depth about Curriculum Learning in section 3.3.3, but it is the nature-inspired idea that learning progressively harder tasks from a simple start makes learning faster and better. We make use of this technique and measure its effects on the learning. It was originally introduced in 2009 by [Bengio et al., 2009] and there are a couple of works whose ideas inspire some of the techniques we used.

One is [Jiang et al., 2015] where they unify Curriculum Learning with a similar idea called self-paced-learning which takes feedback from the learning agent when changing the environment. This paper inspired the way we designed and implemented the curriculum learning part of UAVenv, the environment we created.

The other one is [Molchanov et al., 2018] where they use their curriculum-learning implementation for maze navigation in a grid environment, which sparked a couple of ideas on how could we implement the curriculum in our environment.

2.1.5 Transfer Learning

We also talk more in-depth about transfer learning in section 3.3.4, but it is the idea of transferring knowledge from one task to another. In our case, we use it to learn to navigate in the environment we created and then transfer that knowledge to the realistic simulator. The idea not only applies to Reinforcement Learning but more types of learning use it too [Pan and Yang, 2009].

One good source of information on how can we address this concept when using RL is [Taylor and Stone, 2009] which mentions many ways it can be performed. Transfer Learning is also parallel to another concept from robotics called Domain Adaptation, we also explain it in 3.3.4, but it comes from the need in robotics of deploying controllers that come from simulation to the real robot.

A couple of works we discussed previously make use of this technique zhu2017target, [Sadeghi and Levine, 2016] but others also bring good ideas.

[Müller et al., 2018] uses transfer learning for autonomous driving tasks by training in a simulator and transferring the learned policy to a 1/5 sized robotic truck.

It has also been used in UAV's by [Daftry et al., 2016] where they train by Imitation Learning (supervised learning over expert trajectory data) a policy and then deploy it on a small UAV with some adaptation techniques based on regularization with good results.

3. THEORETICAL BACKGROUND

As we discussed previously there have been different approaches to Navigation. The most used ones related to Motion Planning rely on having full information about the map and making a plan for it. But we want to navigate without knowing the map previously, so we will make use of different models and tools to arrive at a solution.

Reaching the objective depends on all the previous actions taken, thus making Navigation a sequential decision problem where the sensors provide the input signal and after we process them, our system outputs an action. We will use a mathematical model that suits this problem called Partially Observable Markov Decision Processes (POMDP). But first, we have to explain the underlying model from where the POMDP derive some of their convenient properties.

By using these models (our Markovian assumption) we will be able to use tools like Reinforcement Learning with some theoretical proofs (for example that it converges to optimal policies in the limit) for certain algorithms. It is also enclosing the intuition that the information received each time is enough to make the correct decision. In practice, there is research work where RL algorithms are run and evaluated in settings where the Markov property does not hold (or it is unclear if it holds).

3.1 Mathematical Models

3.1.1 Markov Decision Processes (MDP)

Markov Decision Processes [Sutton and Barto, 2011] formalize the problem of sequential decision making, where taking actions influences beyond immediate rewards. It also affects subsequent situations, states and future rewards. MDPs are the basis for the model we are interested in solving, the partially observable version of MDP's.

An MDP is a model of an agent interacting synchronously with an environment. Each time-step t, the agent takes as input the state of the world and generates output actions, which affect the state of the world. We can see a diagram of this process in Fig. 3.1.The state s contains all the information of the environment, thus making it fully-observable. In the MDP framework, it is assumed that, although there may be a great deal of uncertainty about the effects of an agent action, there is never any uncertainty about the agent current state. It has complete and perfect perceptual abilities. A Markov decision process are defined [Sutton and Barto, 2011] as a tuple $\langle \mathcal{S}, \mathcal{A}, T, R \rangle$ where:

- $\mathcal{S}$ is the set of states of the world. In our case, it is infinite because the space is continuous.
- $\mathcal{A}$ is a finite set of actions.
- $T\colon \mathcal{S} \times \mathcal{A} \to \Pi(\mathcal{S})$ is the state-transition function, giving for each state and action, a probability distribution over arriving at the next states. We write $T(s, a, s')$ for the probability of ending in state s' , given that the agent starts in state s and takes action a.
- $R\colon \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function, that gives the expected immediate reward given to the agent for taking each action in each state. We write $r(s, a)$ for the instantaneous reward for taking action a in state s.

As defined, the next state and the expected reward depend only on the current state and the previous action taken. Even if it was defined to condition on additional previous states, the transition probabilities and the expected rewards would remain the same. This is known as the **Markov property**. The state and reward at time $t + 1$ are dependent only on the state at time t and the action at time t.

The actions can be low-level controls, such as the forces applied by the rotors of a drone, or high-level decisions, such as moving to one location or another.

The states can also take a wide variety of forms. They can be completely determined by low-level inputs, like direct sensor readings, or they can be more high-level and abstract, such as symbolic

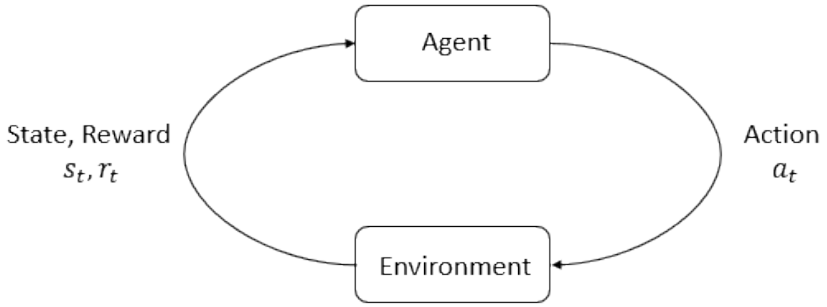

Fig. 3.1: Markov Decision Process and RL Loop. Image from [Achiam, 2018]

descriptions of buildings in front of the agent. Some of what makes up a state could be based on past sensations or even be entirely mental or subjective.

3.1.2 Partially Observable MDP (POMDP)

We can formulate the problem as a POMDP, which is an extension of Markov Decision Processes where the agent is unable to observe the current state. This means that for each time step t the agent only has an observation o which may not contain all the information of the state s.

A partially observable Markov decision process is formulated [Kaelbling et al., 1998] as a tuple $\langle \mathcal{S}, \mathcal{A}, T, R, \Omega, O \rangle$, where:

- $\mathcal{S}$ is the set of states of the world. In our case, it is infinite because the space is continuous.
- $\mathcal{A}$ is a finite set of actions.
- $T\colon \mathcal{S} \times \mathcal{A} \to \Pi(\mathcal{S})$ is the state-transition function, assigning each state and action, a probability distribution over arriving at the next states. We write $T(s, a, s')$ for the probability of ending in state s' , given that the agent starts in state s and takes action a.
- $R\colon \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function, that gives the expected immediate reward given to the agent for taking each action in each state. We write $r(s, a)$ for the instantaneous reward for taking action a in state s.
- Ω is an infinite set of observations the agent can experience of its world.
- $O\colon \mathcal{S} \times \mathcal{A} \to \Pi(\Omega)$ is the observation function, which assigns, for each action and resulting state, a probability distribution over possible observations. We write $O(s', a, o)$ for the probability of obtaining observation o given that the agent performed action a and arrived at state s'.

In the POMDP we don't have all the information about the state in the observation, and taking actions based only on the current observation at time t can lead to bad results. One example can be navigating through a maze, where two places may look the same but if you are not able to correctly identify your position(estimate state s) you may end up lost. One way to solve this is to estimate the state by a belief state b.

The agent makes observations, generates actions and keeps an internal belief state, b, that summarizes its previous experience. Every time step the belief state would be updated based on the last action, the current observation, and the previous belief state. This model has markovian properties over the space of beliefs instead of the space of states, but a good state-estimation is required to solve the problem [Kaelbling et al., 1998].

The POMDP framework is general enough to model diversity of real-world sequential decision processes where planning under uncertainty is needed. In order to behave effectively in a partially observable environment, it is necessary to use memory of previous actions and observations for the disambiguation of the states of the world.

There are two main ways of dealing with partial observability [Hausknecht and Stone, 2015] and getting to solve a POMDP like an MDP:

1. One way is to introduce past information in o_t, like past frames, to better estimate the state and reduce the uncertainty of the state. Making the POMDP behave like an MDP. This is the case for DQN [Mnih et al., 2015] in their famous Nature paper where they include 4 frames in each observation to solve the game of Atari. It is proven to be a simple and effective solution.

2. The other way is to introduce some type of memory in the models, like using a Recurrent architecture for the model. This is the case for [Hausknecht and Stone, 2015], where they introduce a recurrent neural network in the DQN architecture and solve it taking only the current frame. This solution involves dealing with implementation problems for the recurrent architectures.

3.2 Reinforcement Learning

In the big picture, RL is the study of agents and how they learn by trial and error. Solving the Reinforcement learning problem is learning what to do and how to map states to actions in order to maximize a numerical reward signal. It formalizes the idea that rewarding or punishing an agent for its behaviour makes it more likely to repeat or forego that behaviour in the future.

The agent is not programmed or told which actions to take but he has to discover which actions produce the largest reward by trying them. In some of the most challenging cases, actions may affect not only the immediate reward but also future situations and subsequently future rewards. These characteristics of *trial-and-error* search and *delayed reward* are two of the most important distinguishing features of reinforcement learning.

It proposes that any problem of learning goal-directed behaviour can be explained with three signals communicating an agent and its environment.It follows the same model we introduced in 3.1.1 as we can see in the Fig 3.1.

- a signal to represent the actions made by the agent (the actions),
- one signal to represent the information on which the decisions are made (the states),
- and one signal to define the agent goal (the rewards).

This framework encompasses the details of the sensory, memory, and control apparatus, and whatever objective one to achieve. It may not be sufficient to express all decision-learning problems usefully, but it has proved to be widely useful and applicable.

MDPs are a mathematically idealized form of the reinforcement learning problem for which precise theoretical statements can be made. As in all of Artificial Intelligence, there is a tradeoff between breadth of applicability and mathematical tractability. The MDP framework is a problem of goal-directed learning from interaction. But we can reformulate the MDP into a Reinforcement Learning [1] framework we will use to solve our problem.

To understand RL better, we need to introduce additional terminology to later understand how is it done in this work. The theory and notation here follow [Achiam, 2018] conventions:

3.2.1 States and Observations

The definition of state and observation are similar than the $\mathcal{S}$ and O: for MDP's and POMDP's.

- A state s is a complete description of the state of the world.[2]
- An observation o is a partial description of a state, which may omit information.

When the agent is able to observe the complete state of the environment, we say that the environment is fully observed and we are dealing with an MDP. When the agent can only see a partial observation, we say that the environment is partially observed and we are dealing with a POMDP.

However, we have to take into account the size of the space of states and observations when working with it. For classical methods and small and countable state spaces, we can just enumerate them. But for big state spaces like a continuous plane, we represent the space by a real-valued vector or higher order tensor. Images can be represented as the RGB matrix of its pixel values.

The dimensionality of the state and observation space completely determines the tools available and the complexity of the problem.

[1] RL is simultaneously a problem, a class of solution methods that work well on the problem, and the field that studies these problems and its solution methods.

[2] Reinforcement learning notation sometimes puts the symbol for state, s, in places where it would be technically more appropriate to write the symbol for observation, o.

3.2.2 Action Spaces

The definition of action space is the same as the one for MDP's $\mathcal{A}$. It is the set of all valid actions in a given environment.

Different environments allow different kinds of actions. Some environments (Chess or Tic-Tac-Toe for example), have discrete action spaces, where only a finite number of moves are available to the agent. In discrete action spaces, actions are a choice of the possible actions represented by a number. Other environments, like where the agent controls a robot in a physical world, have continuous action spaces. In continuous action spaces, actions are real-valued vectors.

This distinction has some quite-profound consequences for methods in RL, limiting the families of algorithms available and the complexity of the problem.

3.2.3 Policies

A policy is a rule used by an agent to decide what actions to take [3]. It can be deterministic or it may be stochastic.

We define the policy π as the function that maps from states to actions.

If the observation space is finite we could use a look-up table for the policy. But when we have a continuous observation space and we do not approximate by discretizing it, we have to use a more powerful function approximator. In our case, we use a simple Neural Network, commonly called Multi-Layer Perceptron [Goodfellow et al., 2016], mapping $a_t = f(o_t; \theta)$, where θ are the Multi-Layer Perceptron parameters. When using neural networks for RL people often refer to it as Deep Reinforcement Learning (DRL).

In DRL, we deal with parameterized policies: policies whose outputs are computable functions that depend on a set of parameters (eg the weights and biases of a neural network) which we can train to change the behaviour via some optimization algorithm.

We often denote the parameters of such a policy by θ, and then write this as a subscript on the policy symbol to refer to a parametric policy:

$$a_t \sim \pi_\theta(\cdot|s_t). \tag{3.1}$$

3.2.4 Trajectories

A trajectory τ, also frequently called episodes or rollouts, is a sequence of states and actions in the world,

$$\tau = (s_0, a_0, s_1, a_1, ...). \tag{3.2}$$

where Actions come from an agent according to its policy and, state transitions (also defined T for MDP's as what happens to the world between the state at time t, s_t, and the state at $t+1$,s_{t+1}), are governed by the natural laws of the environment, and depend on only the most recent action, .

3.2.5 Reward and Return

The reward function R is critically important in reinforcement learning. It is defined the same way as in MDP's:

$$r_t = R(s_t, a_t) \tag{3.3}$$

In our case, it is represented by a scalar given to the agent each time it takes an action. The goal of the agent is to maximize some notion of cumulative reward over a trajectory. We will notate it with $R(\tau)$.

There are two types of returns, finite-horizon undiscounted and infinite-horizon discounted. They differ by the number of elements added, as one is finite and the other is infinite and by a discount factor $\gamma \in (0, 1)$:

- finite-horizon undiscounted return:

$$G(\tau) = \sum_{t=0}^{T} r_t. \tag{3.4}$$

[3] Because the policy is essentially the agent's brain, it's not uncommon to substitute the word policy for the word agent, eg saying The policy is trying to maximize reward

- infinite-horizon discounted return:

$$G(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t. \tag{3.5}$$

3.2.6 The RL Problem

Given the return measure, and whatever the choice of policy, the goal in RL is to select a policy which maximizes expected return when the agent acts according to it.

To talk about expected return, we first have to talk about probability distributions over trajectories. Let us suppose that both the environment transitions and the policy are stochastic. In this case, the probability of a T-step trajectory is, with ρ_0 being the starting state distribution:

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t). \tag{3.6}$$

The expected return denoted by $J(\pi)$, is then:

$$J(\pi) = \int_{\tau} P(\tau|\pi)R(\tau) = \mathop{\mathbb{E}}_{\tau\sim\pi}[G(\tau)]. \tag{3.7}$$

The optimization problem in RL is expressed by

$$\pi^* = \arg\max_{\pi} J(\pi), \tag{3.8}$$

with π^* being the optimal policy.

3.2.7 Value Functions

It is often useful to know the value of a state or state-action pair. By value, we mean the expected return if you start in that state or state-action pair, and then act according to a particular policy forever after. Value functions are used in most RL algorithms.

There are two main functions of note here.

1. The Value Function,$V^{\pi}(s)$, which gives the expected return if you start in state s and always act according to policy π:

$$V^{\pi}(s) = \mathop{\mathbb{E}}_{\tau\sim\pi}\left[G(\tau)|s_0 = s\right]. \tag{3.9}$$

2. The Action-Value Function,$Q^{\pi}(s, a)$, which gives the expected return if you start in state s, take an arbitrary action a (which may not have come from the policy), and then forever after act according to policy π :

$$Q^{\pi}(s, a) = \mathop{\mathbb{E}}_{\tau\sim\pi}\left[G(\tau).|s_0 = s, a_0 = a\right]. \tag{3.10}$$

If the policy of the value functions is the optimal policy π^*, they will be called the optimal value function V^* or optimal action-value function Q^*.

3.2.8 Bellman Equations

The Bellman Equations are equations that work with the value functions and that are the basis for many RL algorithms. Its main idea behind is:

> The value of your starting point is the reward you expect to get from being there, plus the value of wherever you land next. [Achiam, 2018]

The Bellman equations for the value functions are

$$V^{\pi}(s) = \mathop{\mathbb{E}}_{a\sim\pi\ s'\sim P}[r(s, a) + \gamma V^{\pi}(s')],$$
$$Q^{\pi}(s, a) = \mathop{\mathbb{E}}_{s'\sim P}\Big[r(s, a) + \gamma \mathop{\mathbb{E}}_{a'\sim\pi}[Q^{\pi}(s', a')]\Big],$$

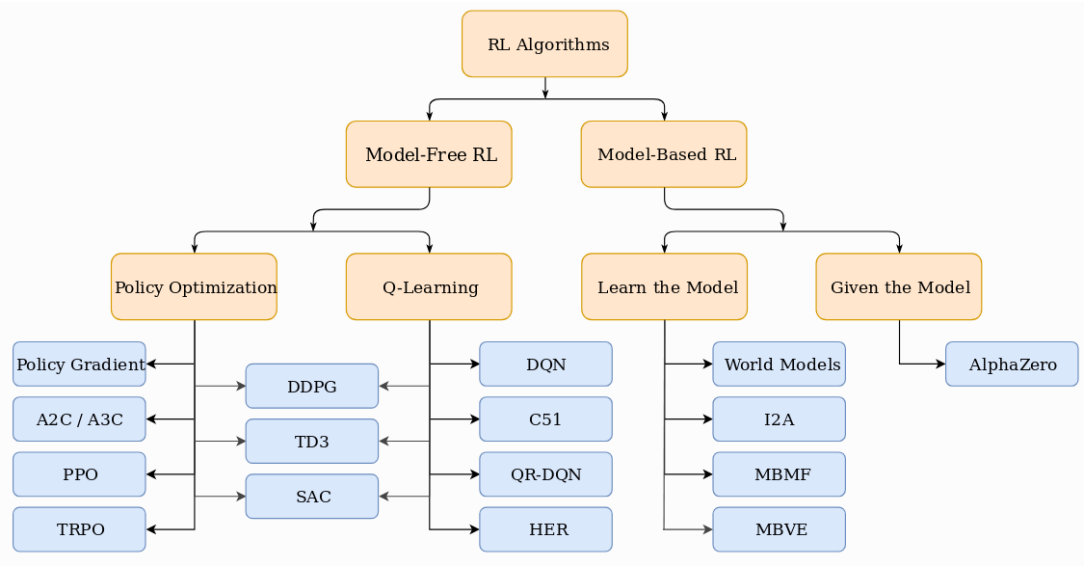

Fig. 3.2: Tree of RL Algorithm Families. Image from [Achiam, 2018]

(3.11)

where $s' \sim P$ is shorthand for $s' \sim P(\cdot|s,a)$, indicating that the next state s' is sampled from the environment's transition rules; $a \sim \pi$ is shorthand for $a \sim \pi(\cdot|s)$; and $a' \sim \pi$ is shorthand for $a' \sim \pi(\cdot|s')$.

The Bellman equations for the optimal value functions are

$$V^*(s) = \max_a \mathop{\mathbb{E}}_{s' \sim P}[r(s,a) + \gamma V^*(s')],$$
$$Q^*(s,a) = \mathop{\mathbb{E}}_{s' \sim P}\left[r(s,a) + \gamma \max_{a'} Q^*(s',a')\right].$$

(3.12)

The difference between the Bellman equations for the value functions and the optimal value functions is the absence or presence of the max operator over actions. Its inclusion reduces the choice of actions, in order to act optimally, to the action that leads to the highest value.

3.2.9 Kinds of Learning Algorithms

One of the most important points of the algorithms we are using is that it doesn't have a model of the environment. It is learning completely from scratch, it doesn't care if it is learning to navigate a drone, managing the electric consumption of a data centre or improving notification in a social network.

The algorithms we use are called **model-free**: they do not have a model of the environment transitions and are very general purpose. By a model of the environment, we mean a function which predicts the transition function T defined for MDPs. Knowing a model of the world doesn't mean the RL problem is solved just as we may know exactly how a puzzle like Rubik's cube works, but still be unable to solve it.

This is the first division in RL algorithms, whether they use or learn a model of the environment (model based) or not (model-free). The main differences right now (during the writing of this thesis) are that even tough model-based are much more sample efficient, model-free methods tend to be easier to implement and tune, are more popular and have been more extensively developed and tested than model-based methods.

Another dividing point in RL algorithms is the question of what to learn. Learning implies approximating one function with an increase in performance respect to some metric. The list of elements that are usually learned includes policies, either stochastic or deterministic, action-value functions (Q-functions), value functions and/or environment models.

In Fig 3.2 we can see a taxonomy of some of the algorithms used in Reinforcement Learning.

There are two main approaches to representing and training agents with model-free RL:

Policy Optimization

Methods in this family represent a policy explicitly as $\pi_\theta(a|s)$ with a Neural Network with parameters θ. They optimize the parameters θ directly by gradient ascent on the performance $J(\pi_\theta)$, or indirectly, by maximizing local approximations of $J(\pi_\theta)$. This optimization is almost always performed **on-policy**, which means that each update only uses data collected while acting according to the most recent version of the policy.

Policy optimization also usually involves learning an approximator $V_\phi(s)$ for the on-policy value function $V^\pi(s)$, which is used in figuring out how to update the policy. This is central to actor-critic methods. Sometimes this approximator shares some parameters with the policy.

Here we explain the calculations used for gradient ascent on the policy. The agent in our problem acts following a stochastic, parameterized policy, π_θ.

We aim to maximize the expected return $J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[G(\tau)]$

We optimize the policy by gradient ascent, e.g.

$$\theta_{k+1} = \theta_k + \alpha \, \nabla_\theta J(\pi_\theta)|_{\theta_k} . \tag{3.13}$$

The gradient of policy performance, $\nabla_\theta J(\pi_\theta)$, is called the policy gradient,

Deriving the expression of the Basic Policy Gradient we get to:

$$\begin{aligned} \nabla_\theta J(\pi_\theta) &= \nabla_\theta \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}[G(\tau)] \\ &= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) G(\tau)\right] \qquad \text{Expression for the gradient} \end{aligned} \tag{3.14}$$

The derived expression is an expectation, which means that we can estimate it with a sample mean.

If we collect a set of trajectories $\mathcal{D} = \{\tau_i\}_{i=1,\dots,N}$ where each trajectory is obtained by letting the agent act in the environment using the policy π_θ, the policy gradient can be estimated with

$$\hat{g} = \frac{1}{|\mathcal{D}|} \sum_{\tau\in\mathcal{D}} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) G(\tau), \tag{3.15}$$

where $|\mathcal{D}|$ is the number of trajectories in $\mathcal{D}$.

This last expression in (3.15) is the simplest version of the computable expression we aimed for. For more details of those derivations, the full process can be found in [Achiam, 2018] or [Weng, 2018].

Assuming that we have represented our policy in a way which allows us to calculate $\nabla_\theta \log \pi_\theta(a|s)$, and if we are able to run the policy in the environment to collect enough trajectories, we can compute the policy gradient and take an update step in the optimization.

The algorithms that optimize the policy this way are called policy gradient algorithms. The policy optimization algorithms that we use in this work are presented. Each uses slightly different methods :

- **TRPO** [Schulman et al., 2015] updates policies by following the largest step possible to improve performance while satisfying a special constraint on the KL-Divergence over the new and old policies are allowed to be. KL-Divergence is a measure of (similar, but not exactly) distance between probability distributions.
- **PPO** [Schulman et al., 2017], whose updates indirectly maximize performance, by instead maximizing a surrogate objective function which gives a conservative estimate for how much will change as a result of the update. The advantage over TRPO is that PPO is less computationally expensive.
- **A2C / A3C** [Mnih et al., 2016], which performs gradient ascent to directly maximize performance, it uses an updating scheme that operates on fixed-length segments of experience and then uses these segments to compute estimators of the returns and advantage function. Advantage function defined as $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$. It introduces architectures that share layers between the policy and value function, making the first layers work as a feature extractor.
- **ACER** [Wang et al., 2016] is an off-policy actor-critic model with experience replay, greatly increasing the sample efficiency and decreasing the data correlation. It is the Off-Policy version of A3C.

- **ACKTR** [Wu et al., 2017] combines three different techniques: actor-critic methods, trust region optimization for steady improvement, and distributed Kronecker factorization to improve sample efficiency and scalability.

Q-Learning

Methods in this family learn an approximator $Q_\phi(s,a)$ for the optimal action-value function, $Q^*(s,a)$. In which the Q function is represented by a function approximator (a neural network) with parameters ϕ . Typically they use an objective function based on the Bellman equation 3.11.

This optimization is almost always performed **off-policy**, which means that each update can use experience collected at any moment during training, regardless of how the agent was exploring the environment when the data was obtained.

The corresponding policy is obtained via the connection between Q^* and π^*: the actions taken by the Q-learning agent are given by

$$\pi(s) = \arg\max_{a'} Q_\phi(s,a'). \tag{3.16}$$

The development of Q-learning [Watkins and Dayan, 1992] is a big success in the early stages of Reinforcement Learning.

1. At time step t, we start from state S_t and pick action according to Q values, $a_t = \arg\max_{a\in\mathcal{A}} Q(s_t,a)$.
2. With action a_t, we observe reward and get into the next state .
3. Update the action-value function: $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha(r_{t+1} + \gamma \max_{a\in\mathcal{A}} Q(s_{t+1},a) - Q(s_t,a_t))$.
4. $t = t+1$ and repeat back to step 1.

Unfortunately, Q-learning may be inestable and diverge when combined with a nonlinear Q-value function approximation and bootstrapping(The Deadly Triad chapter in [Sutton and Barto, 2011]).

Examples of Q-learning methods include Deep Q-Network ("DQN"; [Mnih et al., 2015]), a classic algorithm that launched the field of deep RL. DQN derives from original Q-Learning, however, the Q function is represented by a function approximator (a neural network) with parameters ϕ .

DQN aims to greatly improve and stabilize the training procedure of Q-learning by two innovative mechanisms:

- Experience Replay: All the episode steps $e_t = (s_t, a_t, r_t, s_{t+1})$ are saved in one replay memory $D_t = \{e_1, \ldots, e_t\}$. D_t has experience tuples over many episodes. During Q-learning updates, samples are picked from a uniform distribution from the replay memory and thus one sample could be used multiple times. Experience replay helps with data efficiency, reduces correlations in the observation sequences, and smooths over changes in the data distribution.
- Periodically Updated Target: Q optimizes towards target values that are only periodically updated. The Q network is copied and kept unmodified as the target for optimization every C steps (with C being an hyperparameter). This modification makes the training more stable as it overcomes the short-term oscillations.

The loss function looks like this:

$$\mathcal{L}(\phi) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\Big[\big(r + \gamma \max_{a'} Q_{\phi^-}(s',a') - Q_\phi(s,a)\big)^2\Big] \tag{3.17}$$

where $U(D)$ is a uniform distribution over the replay memory D; θ^- is the parameters of the frozen target Q-network.

There are many modifications to DQN to improve the original design, like DQN with duelling architecture ([Wang et al., 2016]) which estimates value function $V(s)$ and advantage function $A(s,a)$ with shared network parameters. Or [Hausknecht and Stone, 2015] which adds a recurrent model to avoid having to include several frames in the state.

3.3 Limitations of Learning Algorithms

Deep Reinforcement Learning can be horribly **Sample Inefficient**. Atari games are used as a baseline to compare RL algorithms. In the DQN papers, for example, the results they claim are after 200M timesteps, considering the game runs at 60 frames per second, it would take approximately 38 days to learn compared the minutes a human needs to learn to play those games. The Sample inefficiency problem affected us in the sense that the last experiments took 2 weeks to complete.

DRL is exciting because of its generality, it is amazing that it can learn to solve that wide variety of problems. However, there is almost always a tradeoff between **generality versus performance**. For purely getting good performance, DRL methods are not that great, because it consistently gets beaten by other methods. Generality comes with the price that it is hard to exploit any problem-specific information that could help with learning, decreasing the sample efficiency. One example is Boston Dynamics[4] Robots, they have impressive results, but they stated that they didn't use learning in their robots[5].

RL needs a reward function, and it must capture exactly the problem you are trying to solve. This is important, because the type of reward function may affect sample efficiency. Having a dense reward makes the experiment converge faster but may not capture the problem completely, or having a sparse reward makes the efficiency decrease but captures better the problem. Reward function design is difficult, the design decisions must contemplate all the ways in which the agent may exploit it without fulfilling the goals. This problem is called **Reward Hacking**. One example was in the first iteration of our problem, the policy learned to be suicidal because the negative reward was plentiful, the positive reward was too hard to achieve, and a quick death ending without too much time penalty was preferable to a long life that risked negative reward.

The ever-present **Exploration-Exploitation Dilemma**, sometimes the agent gets stuck in local optima and fails to get to better solutions because they are difficult to explore. Your data comes from your current policy. If your policy explores too much you do not get meaningful data and learn nothing. Exploit too much and you settle in behaviours that are not optimal.

It may **over-fit** to the environment. In Navigation, we do not have this problem specifically, because we can generate random start points and locations and add enough randomization. But for other cases, the agent is just memorizing the environment. And when taken to an unseen environment it may perform poorly.

Reliability and Reproducibility are an issue too. As some parts of the algorithm are random, and the nature of the environments can be stochastic too, reproducing the same result of an experiment is complicated if you do not aim explicitly to it when designing everything. Reliability is an Issue, because some algorithms just fail to converge sometimes. For one of the algorithms when we ran 10 seeds 6 of them got a score similar than the other algorithms that didn't converge, the other 4 converged properly. So from that point on, we started to run several seeds for each algorithm.

To read more about the limitations of RL I recommend the Alex Irpan blog post: "DRL doesn't work yet" [Irpan, 2018] which was an eye-opener on what failures to expect from RL. Next, we are going to review some tools and techniques that let us deal with these limitations.

3.3.1 Function Approximation with Neural Networks

One way to fight the curse of dimensionality without simplifying too much the problem is to use more powerful tools. However, they usually come at the cost of something like the need for more compute power for example. Neural Networks [Goodfellow et al., 2016] are a powerful function approximator that lets us deal with large Action and State Spaces, even continuous ones.

Firstly, they take an input vector. Then they perform matrix multiplications with the matrices of parameters of each layer. Then, they apply non-linear functions to the output of those layers. And finally, they output the number or vectors we are trying to approximate as the output of one of those layers. The parameters are adjusted by a method called Backpropagation [Goodfellow et al., 2016] which is optimizing the parameters iteratively by gradient descent with respect to a loss function. These methods scale good with great amounts of data (Fig. 3.3), and in our case, we have them because we take the data from a simulator.

[4] `https://www.youtube.com/watch?v=rVlhMGQgDkY`

[5] Marc Raibert, CEO Boston Dynamics answering a question around 48 minute mark `https://www.youtube.com/watch?v=LiNSPRKHyvo`.

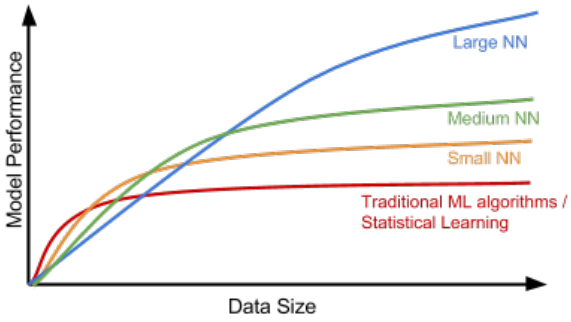

Fig. 3.3: Scaling of Model Size vs Data Size. Image from [Weng, 2017]

The combination of Modern Neural Networks and classic Reinforcement Learning techniques gave birth to the field that is called Deep Reinforcement Learning. We discuss the use of different algorithms in the results section.

3.3.2 Reward Shaping

As we mentioned in the Limitations of RL, Sample Inefficiency is one of the main challenges we are facing. Making us spend long times training even when implementing and testing the system. One way to combat this is to make a synthetic reward function that captures the objective we are trying to achieve but gives reward more often. This process is called Reward Shaping, and it g has been discussed since 1999 in [Ng et al., 1999]. This is a delicate task and is very easy to get it wrong. We discuss the use of different reward functions in the results section.

3.3.3 Curriculum Learning

Another tool to combat Sample Inefficiency and the Exploration vs Exploitation Dilemma is the hypothesis of Curriculum Learning. It is a concept discussed since 2009 in [Bengio et al., 2009]. It involves a process for which the task learned starts simple and becomes increasingly difficult during training. This process helps the exploration of more successful policies since the beginning. And this all translates into a speedup in training. However, this process requires the design of the curriculum and it may bias the agent towards policies that are not optimal. We analyze the effect of curriculum learning in training in the results section.

3.3.4 Transfer Learning, Domain Adaptation and Domain Randomization

One of the key points in the development of this work was to find that the realistic environment we were attempting to train on, could only run in real-time. That meant it could only work at 5 timesteps per second , and that it would take around 55 hours to complete 1M timesteps (which depending on the task, it is not too much training). Then speed became a big concern along with sample efficiency. In order to overcome this problem, we created an environment that replicated the dynamics of the realistic environment but could run $\times 100$ times faster.

However, there is always a tradeoff. And if we wanted to go back at some point to the realistic environment, we would have to transfer the knowledge from one environment to the other. This is close to the ideas of Transfer learning, and it is called Domain Adaptation.

Transfer learning [Pan and Yang, 2009] is a set of techniques that aim to be able to improve the performance in one task after learning from a different one. Domain Adaptation [Weng, 2019] refers to a set of transfer learning techniques developed to update the data distribution in simulation to match the real one through a mapping or regularization enforced by the task model.

One of those strategies is Domain Randomization [Andrychowicz et al., 2018] which involves randomizing some properties of the simulated environment so that the real one is expected to be inside the distribution of training environments. And when the transfer happens the agent still performs well in the real environment. We evaluate the transferred policies in the results section. And discuss the domain randomization ideas behind the design decisions in the description of the system section.

4. DESCRIPTION OF THE SYSTEM

In this chapter, we are going to go from the theoretic models we talked about in Chapter 3 to the details of the system and the implementation. This involves applying the theory to our idea of trying to make a UAV learn to navigate. We will discuss the design decisions and tradeoffs involved in the creation of the system as well as some implementation details which were fundamental to the development of it.

In order to implement the system, we have to work with a programming language like Python, which provides ease of learning and using, a big amount of libraries available (including the simulator and the Machine Learning Frameworks) and an enormous community .

The model studied in the experiments consists of an agent, which represents the mobile that navigates, and an environment, which represents the world in which the agent is located. The interactions between agent and environment happen at discrete time steps. For each time step t, the agent receives an observation o_t and a reward r_t, and interacts with the environment by taking an action a_t, which may affect the state s_{t+1} of the environment and the future observations. This feedback loop allows us to define episodes of a maximum length T_{max} where the RL algorithm will try to maximize the performance score $R(\tau_{\pi_\theta})$ enough to solve the task of reaching the goal point without collisions.

4.1 Agents

First of all, we have to make a distinction in the way these UAV's can fly. These ways have implications in how we approach the problem. A diagram is found in Fig 4.1

- One way a drone can fly is with respect to an $x - y - z$ axis without facing the direction it is moving. This implies that the perception systems have to be omnidirectional. This approach is simpler because it only requires moving in the right direction, whether the other requires facing the right direction and then moving.
- The other way it can move is facing the direction it moves. This is more realistic in the context of a drone navigating with the input from a camera. But it is also more complex to solve because it implies both looking and moving.

There is, of course, the problem of the z axis. As we wanted to reduce the complexity of the problem especially in the early stages, we stated that the problems we would be solving were going to be 2D navigation. So in the simulations, the agent will move at a constant altitude. And the two flight modes we talked about become what we refer to as Cartesian Dynamics and Polar Dynamics. We will talk

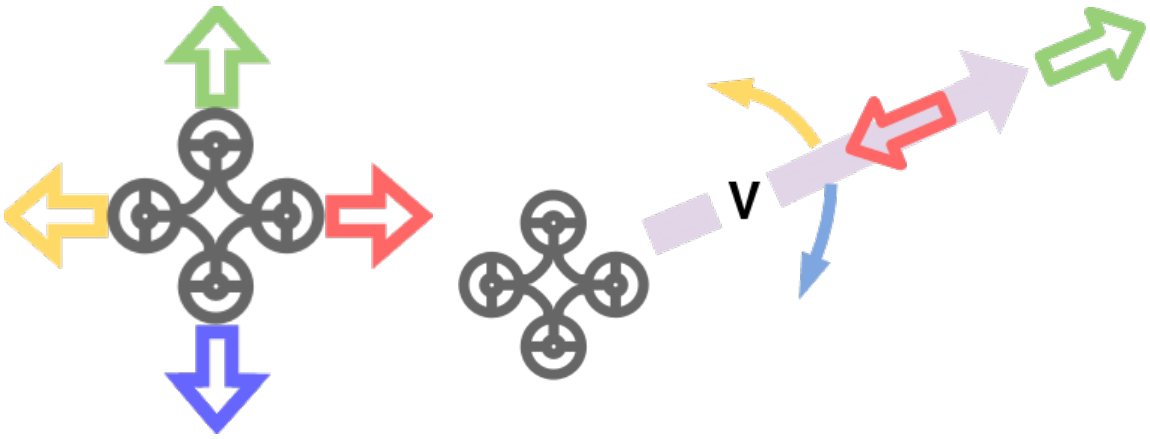

Fig. 4.1: Types of flight

about them when in the Action and Observation spaces. But we will mainly use Cartesian actions for the experiments because the mapping from observations to actions is more straightforward and the observability is richer.

Tools: Baselines

When faced with the implementation of the agent, it had to have the policy and the RL algorithm for the learning. As we saw in the Theory Section, these learning algorithms are not simple, and implementing them from scratch would have introduced another possible failure dimension in our task. So we decided to look for a consistent implementation that was well tested, performed well, was well documented and was easy to use.

With all these requirements in mind we first looked for the most famous open collection of these algorithms: Baselines from OpenAI [Dhariwal et al., 2017]. But we found that even being the first to do an open collection like this and performing well, they were not properly documented, they didn't share a common interface and definitely, they were not easy to use. It often happened that there was an error, and it was hard to see if it was on our part of the code or in theirs.

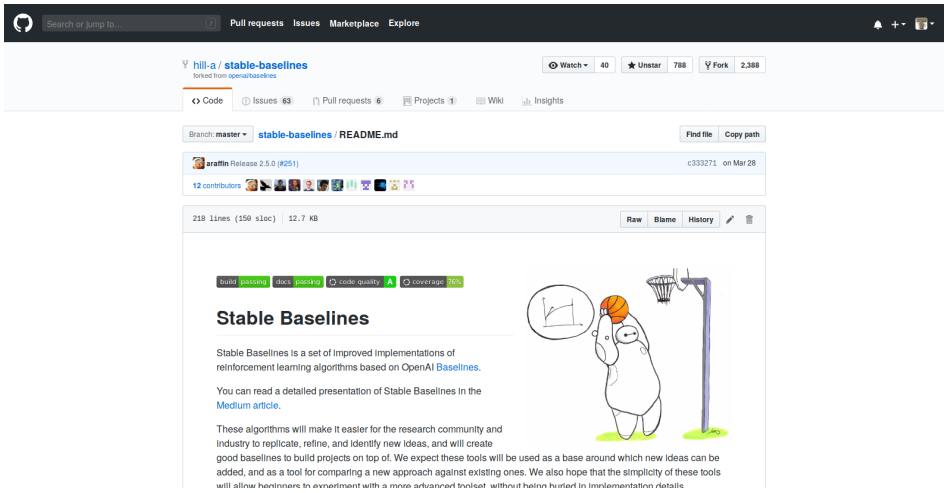

Fig. 4.2: Stable Baselines github.

So when we found that a group of researchers from INRIA Paris were solving these problems with a well-documented implementation of this collection [Hill et al., 2018], we started using it. This sped up development time significantly. In Fig 4.2 we can see the Github page of the project and in Fig 4.3 we can see the list of algorithms implemented and the features they have available.

All the algorithms follow a similar interface. Following the Object-Oriented paradigm of programming, we programmed from scratch the Agent object, which is one of the implementations of an algorithm that inherits from BaseRLModel. This agent object is then assigned, among other objects and parameters, a Policy object and an Env. object. In Fig 4.4 we can see the structure of the code and how each component interacts with each other. It is important to mention that we also had to develop and implement all the visualization tools and metrics because they were not properly developed in the tools we were using.

Policies

The policy is the function that maps from observations to actions. In our case, it will be an object that contains a Neural-Network. The NN is defined using the library TensorFlow [Abadi et al., 2016] (very popular for working with Neural Network and differentiable graphs).

The NN we use for most of the experiments consists of 2 fully connected layers of 64 neurons each. We tried different sizes, from 16 to 256 and there was not a difference in performance, so we kept the 64 size parameter. We can see a diagram of the architecture in Fig 4.5, with only 16 neurons per layer (for representation purposes).

The architecture of the policies for some algorithms (Actor-Critic) includes another output for estimating the value of the state. These architectures share the first layers for the Value and the Policy so we could interpret them as feature extractors. The value function apart from being used for the learning

Implemented Algorithms

Name	Refactored(1)	Recurrent	`Box`	`Discrete`	`MultiDiscrete`	`MultiBinary`	Multi Processing
A2C	✔	✔	✔	✔	✔	✔	✔
ACER	✔	✔	✖ (5)	✔	✖	✖	✔
ACKTR	✔	✔	✖ (5)	✔	✖	✖	✔
DDPG	✔	✖	✔	✖	✖	✖	✖
DQN	✔	✖	✖	✔	✖	✖	✖
GAIL (2)	✔	✖	✔	✔	✖	✖	✔ (4)
HER (3)	✖ (5)	✖	✔	✖	✖	✖	✖
PPO1	✔	✖	✔	✔	✔	✔	✔ (4)
PPO2	✔	✔	✔	✔	✔	✔	✔
SAC	✔	✖	✔	✖	✖	✖	✖
TRPO	✔	✖	✔	✔	✔	✔	✔ (4)

Fig. 4.3: Table of features for the algorithms from stable-baselines.

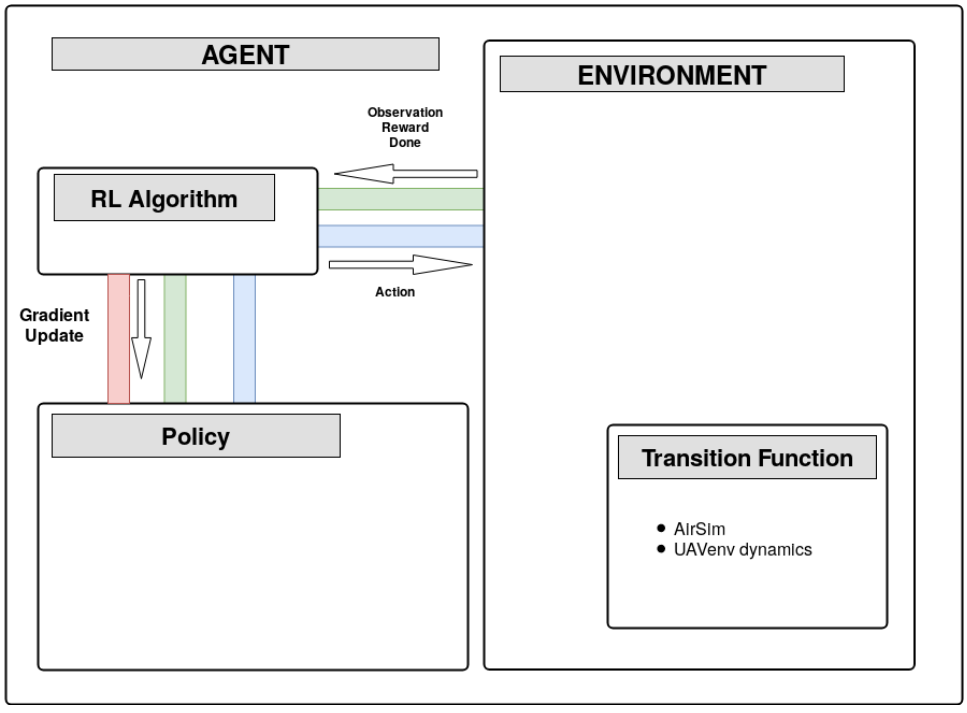

Fig. 4.4: Code Structure

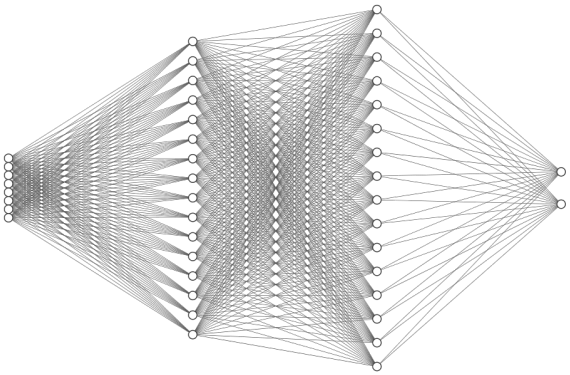

Fig. 4.5: Policy Architecture.

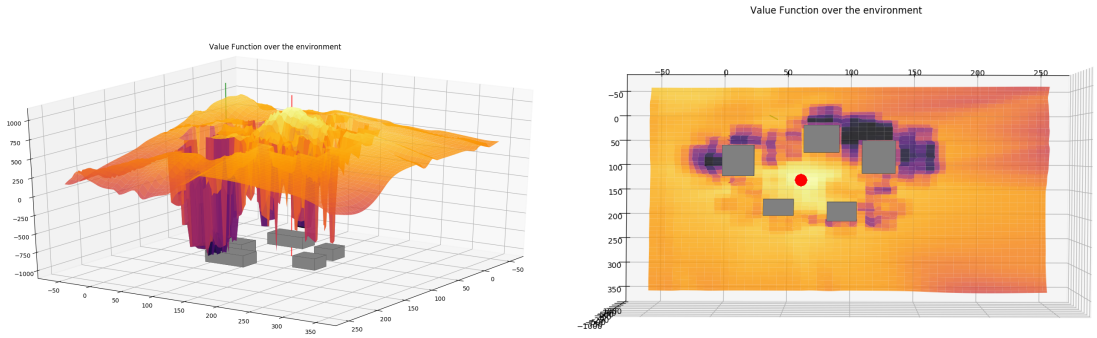

Fig. 4.6: 3D representation of the value function of starting in every position of the space. Grey boxes are obstacles.

can be used to check if the agent is learning something, as we can see in the Fig 4.6. In these maps we have a z and colour value for the Value of starting in each position of the map.

In Section 3.1.2 we talked about the ways to solve a POMDP and we opted for including temporal information in the state by giving it the $\vec{v}$ similar to what [Mnih et al., 2015] does because it is simpler and gives better results. This lets us use a MLP without memory and get good results approximating the POMDP.

4.2 Environments

So far we have talked about how to solve the problem with the agent, but the definition of the problem is encapsulated in the definition of the environment. If we want to solve a specific problem we have to define a specific environment. But first of all, we have to specify the way it is going to work by defining an interface.

Tools: OpenAI Gym

Gym [Brockman et al., 2016] is an open source interface to reinforcement learning tasks. It is the most commonly used interface for RL experiments. It also provides a collection of environments, but we created our own using their interface.

The main ways an agent interacts with the environment are with a step method which takes an action as a parameter and returns an observation, reward and a flag for when the episode is over. This step method represents a whole cycle to the RL Loop. Then there is the reset method for when the episode is over and the agent has to start again.

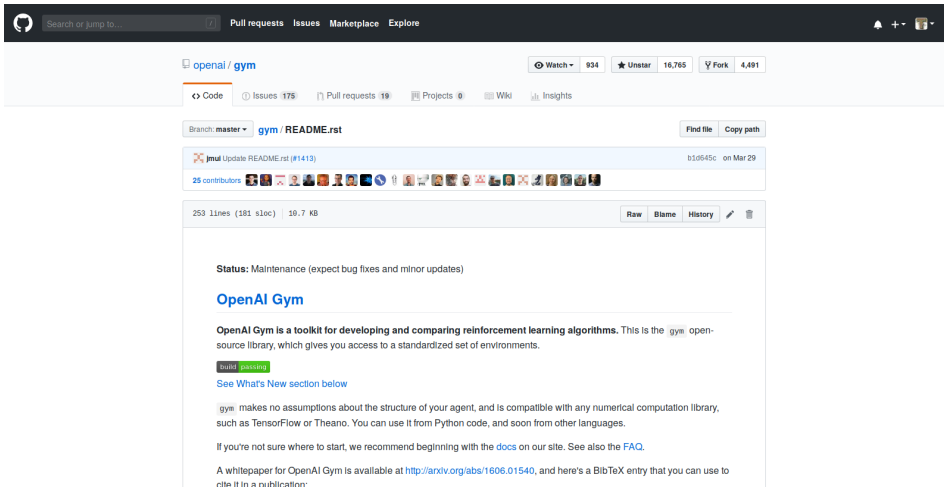

Fig. 4.7: OpenAI:Gym github.

4.2.1 AirSim

We searched for an open source realistic UAV simulator that had a Python interface and found AirSim [Shah et al., 2017]. It is developed by Microsoft and is built on top of Unreal Engine. Their goal is to be a research platform for AI, so their API permits to retrieve data and control vehicles in a platform independent way.

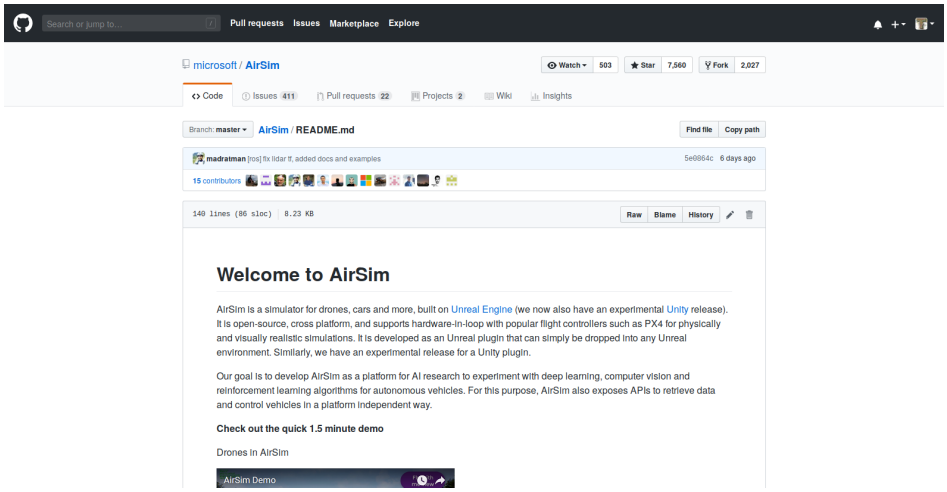

Fig. 4.8: Microsoft AirSim Github.

We can see some images of how the simulator looks like in Fig 4.9.

And some of the API's it exposes are to check the GPS, the position, speed, and accelerations of the UAV, the orientation and if it has collided with an object. There is also a vision API that lets you get the images of the frontal camera with: natural camera, object segmentation camera and depth camera (an example is shown in Fig 4.10).

There is also a control API which lets you control the drone by different levels of actions.

- It lets you take off, land, and go automatically to the place where you took off.
- It lets you move by angles. Controlling pitch, roll and yaw.
- It lets you move by velocity, specifying a velocity in x, y, z direction.
- It lets you predefine a path to follow.
- It lets you select a position to move and go in a straight line.

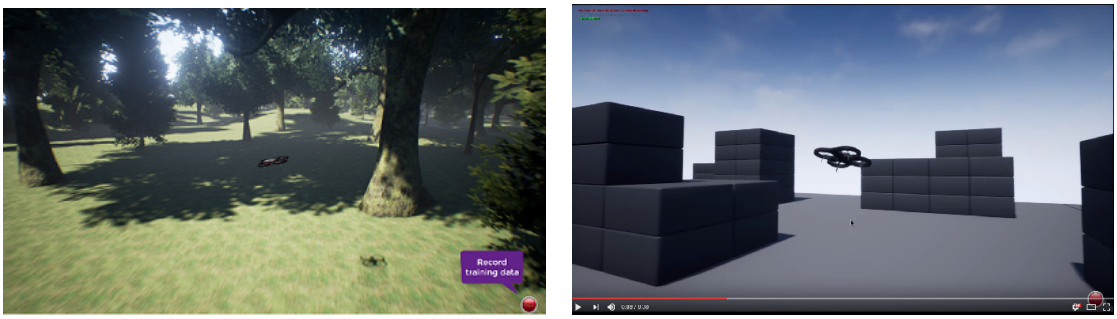

Fig. 4.9: Some examples of how the simulator looks like.

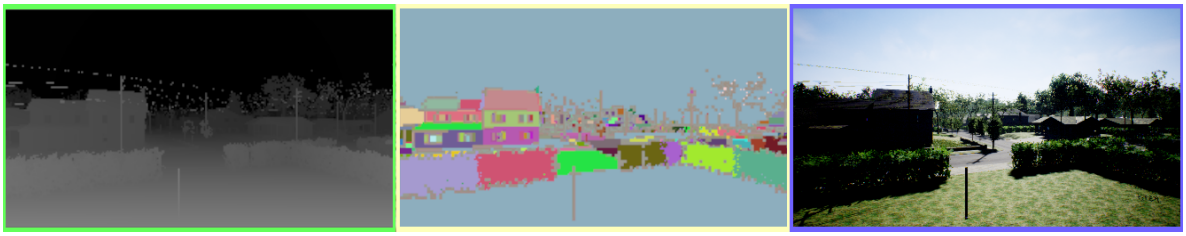

Fig. 4.10: Image modes for airsim. From Airsim Github [Shah et al., 2017]

With these tools, we arranged them and created an environment that captures our problem and follows the Gym Interface. In this environment, there are 11 Block obstacles that we want to avoid, and there are 6 colour balls that are going to represent the start and end of our trajectories. In each episode 2 balls will be selected at random and the UAV will have to go from one to another without colliding with the obstacles. We can see the map and an image of the simulator in Fig 4.11.

4.2.2 UAVenv

As we mentioned before, when running the experiments in AirSim environment we experienced the need for more speed in development. The way to achieve this was to create from zero another environment without graphical interface but with similar dynamics. This way we programmed and created what we call **UAVenv**. A simple and fast environment for Navigation and obstacle avoidance tasks. It was created entirely in Python using numeric libraries like Numpy and geometrical ones like Shapely, and following the interface of OpenAI Gym. This resulted in a performance increase of $\times 100$, going from 5fps to 500fps in training.

The environment consists of a 2D rectangular map with rectangular obstacles placed randomly inside, the starting point and the goal location. To mitigate the problem of sample efficiency we did the

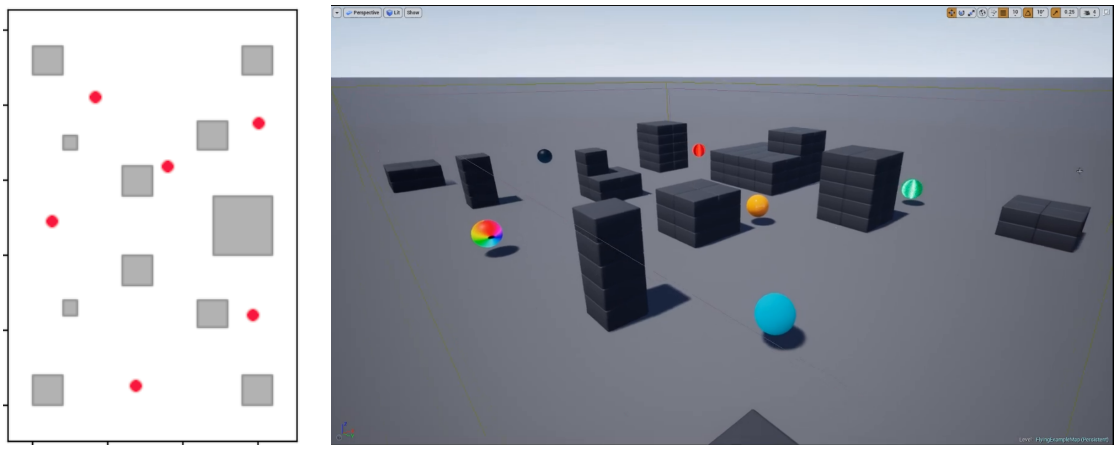

Fig. 4.11: The map and how it looks like inside the simulator. Red dots represent the starting and origin points, the grey squares are the obstacles..

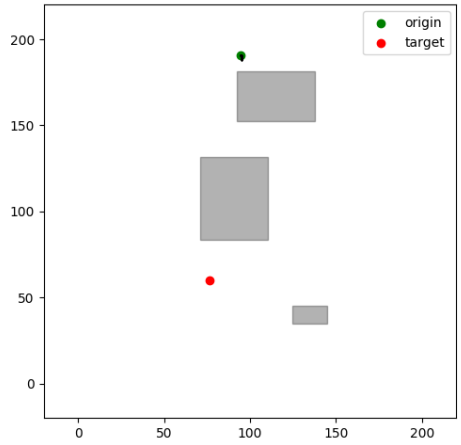

Fig. 4.12: Example of a rectangular map with 3 obstacles.

environment very simple so that it could run fast and the agents could be trained in a couple of hours. The environment also supports a wide variety of modes created to accelerate the research iteration process. Some of the features implemented are:

- discrete and continuous actions,
- cartesian and polar actions,
- dense and sparse reward functions,
- domain randomization on the goal placement (allowing you to not reset the objective till the episode is completed),
- images as observations (an example in Fig. 4.13) ,
- custom sized maps with variable number of obstacles and different levels of difficulty,
- curriculum learning mode and
- an interface to work with AirSim dynamics as the transition function.

We can see an example of a generated environment in Fig. 4.12 where the grey rectangles are the obstacles, the green dot is the starting point and the red dot is the goal.

We found that other environments [Savva et al., 2017], [Kolve et al., 2017], [Chevalier-Boisvert et al., 2018], [Chang et al., 2017] used to benchmark this type of task were either discrete grids, where the obstacle avoidance problem is trivialized or lacked the diversity in environments making the agent memorize the map and not being able to generalize across different scenarios.

Every episode the environment will change the location of the obstacles, starting point and goal location making the agent unable to memorize the environment and forcing it to develop a strategy that enables it to reach the objective independently of the disposition. The idea of randomizing the environment in training was a key point in [Henderson et al., 2018] and proved to be an effective yet simple solution to over-fitting the environment.

Our environment follows simple yet realistic dynamic rules (Transition Function) with continuous space, which follows a double integrator equations (4.1) in which inertia is taken into account.

$$\begin{aligned} \vec{p}_{t+1} &= \vec{p}_t + \vec{v}_t \Delta t \\ \vec{v}_{t+1} &= (F\vec{a}_t - k\vec{v}_t)\Delta t + \vec{v}_t \end{aligned} \tag{4.1}$$

Where F is a constant used to regulate the force of the acceleration and $k = \frac{F}{|v_{max}|}$ is a term used to add some friction. The term $\vec{a}_t$ represents the action as a direction vector in which the force is applied. The vector $\vec{p}$ is the position vector and $\vec{v}$ is the velocity vector. The constant Δt is the time step.

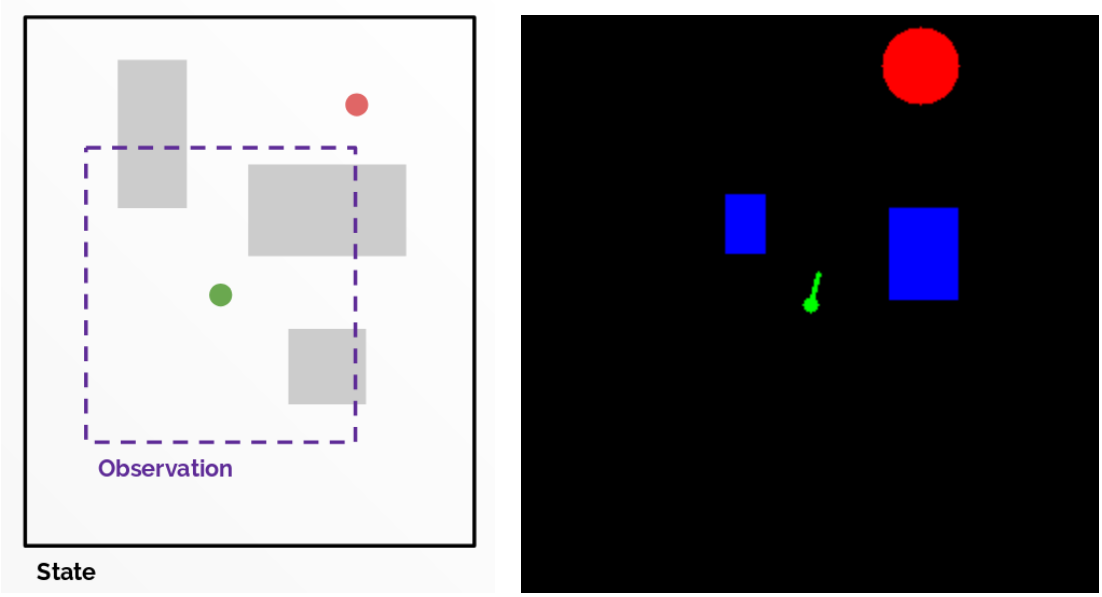

Fig. 4.13: This is what the Agent observes from the environment when set to image observations.

Finally, we have the available optionof having image as an input, generated as a render of the environment with OpenCV, where we take the 3 channels and convert them to a vector to send them to the policy. We can see a diagram of this input and an example in Fig. 4.13.

The following sections are the explanations of how those components are implemented in UAVenv.

4.3 Observations

The agent has no prior knowledge of the environment and only sees what it is received by the observations. In our system o_t are the raw sensory input and they do not contain all the information of the state of the environment. We can either have a vector or an image observation. We will mainly use vector observations because the dimensionality is much smaller thus making the problem less complex.

The input vector of both should include, some information of the state (relative position to the target, speed...), some information on the obstacle perception.

4.3.1 Cartesian

We define the observation vector as the concatenation of several elements:

- The first two elements are the Cartesian coordinates of the clipped distance vector (4.2)

$$\vec{d}_{clipped} = \frac{d_{max}(\vec{p}_{target} - \vec{p}_{agent})}{\max\left(d_{max}, \|\vec{p}_{target} - \vec{p}_{agent}\|\right)} \tag{4.2}$$

  Where $\vec{p}_{target}$ and $\vec{p}_{agent}$ are the positions of the target and the agent. Where d_{max} is a threshold distance that limits the size of the distance vector, making the agent generalize better over maps of different sizes. In practical terms d_{max} can be seen as the limited range of the sensors.

- The third and fourth elements are the components of the velocity vector $\vec{v}$.

- And the rest of the input elements are a radial collision lattice around the agent that simulates proximity sensors like a lidar device. The value of the lattice elements takes value 1 when they collide with an obstacle and 0 when they do not.

We can see a diagram of this input in Fig. 4.14.

We later improved the collision detection system to work as a collection of rays coming from the UAV that intersect with the objects around, to imitate better a lidar device. The elements of the vector take the relative length of the ray after being intersected with the obstacles. This way they range from 0 to 1. We can see a diagram in Fig. 4.14.

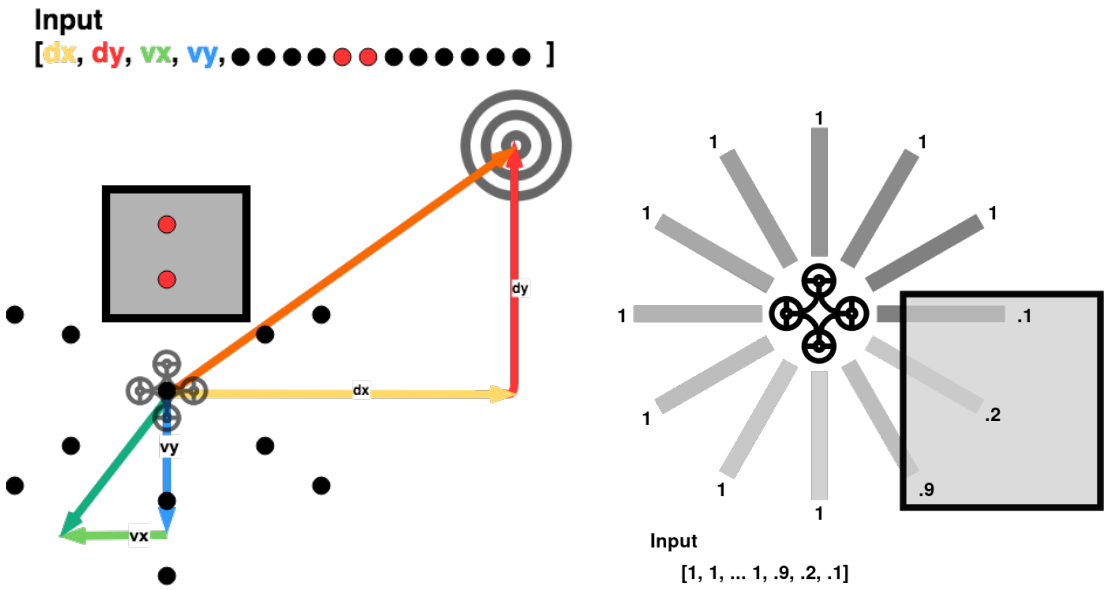

Fig. 4.14: The input diagram for cartesian dynamics. And Collision detection with rays.

4.4 Rewards

The definition of the problem in the environment is done through the reward function. This has to capture the objectives of the problem. The discipline of creating Reward Functions is called Reward Shaping. For these experiments, we used two reward functions, one Sparse and one Dense. The dense one makes the learning faster because it can help with the exploration, but it could lead to a different behaviour than the one expected. Meanwhile, the Sparse reward is very hard to make wrong, you either complete the task or do not. However, training with a Sparse reward usually adds an x10 to the training time as we will see in the results.

When defining the success case, we also define the problem. Whether we want the UAV to stop at the objective or just get close enough.

4.4.1 Sparse Reward

We want the agent to arrive at the spot and not crash, so we include positive and negative factors into the reward function with the first two cases of (4.3). For every other case, we give 0 reward.

$$r(s,a) = \begin{cases} -10 & \text{crash} \\ +10 & \|\vec{d}\| < \varepsilon \\ 0 & \text{else} \end{cases} \tag{4.3}$$

4.4.2 Dense Reward

The first two cases are the same as the Sparse reward. Then we want the reward to be dense, meaning that it takes nonzero values in places other than end-states because it speeds up training. To solve the reward sparsity problem we designed a function (4.4) for the third case of (4.5) that took into account three factors that captured the problem definition. Here we explain the motivations of this Reward Shaping.

- We want to take distance into account, the closer to the target the better. This is represented in the first element of the expression$(-\frac{d^2}{100})$.

- We want to include the speed in the reward, but only close to the target, as our goal is to reach the target with a limited speed. This is captured in the second element of the expression $(-\min((3v^2 - 80), 0)\frac{(d^2+1)}{(3d^2+1)})$ that is quadratic with the speed but only in a region close to the target.
- We also want the agent to reach the target fast so we should take time into consideration by making all the reward negative except for the successful ending. This is captured in the third element of the expression (-80) which offsets all the expression under 0.

The plot of the function of (4.4) along the distance and speed axis can be seen in Fig. 4.15 :

$$f(\vec{d}, \vec{v}) = -\frac{\|\vec{d}\|^2}{100} - \min((3\|\vec{v}\|^2 - 80), 0)\frac{(\|\vec{d}\|^2 + 1)}{(3\|\vec{d}\|^2 + 1)} - 80 \tag{4.4}$$

$$r(s, a) = \begin{cases} -10 & \text{crash} \\ +10 & \|\vec{d}\| < \varepsilon, \|\vec{v}\| < \epsilon \\ f(\vec{d}, \vec{v}) & \text{else} \end{cases} \tag{4.5}$$

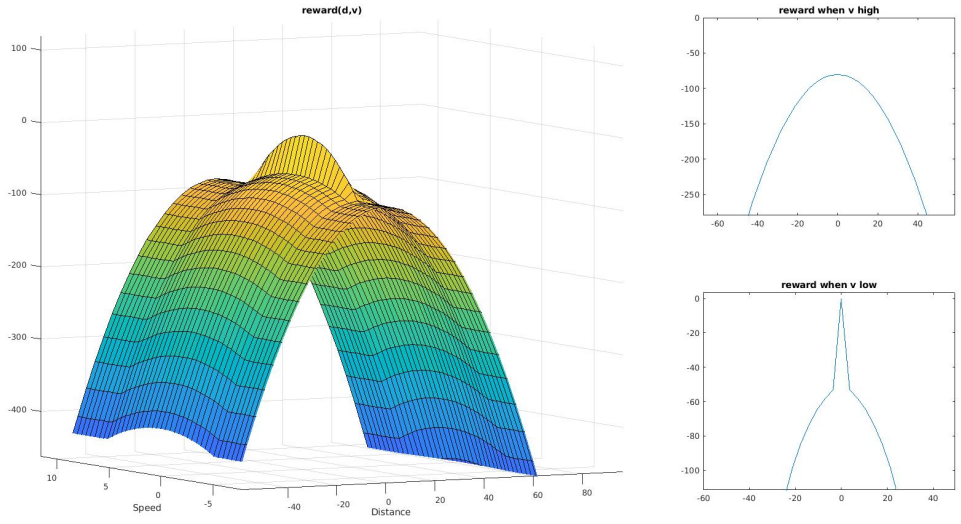

Fig. 4.15: The reward funcion along the speed and distance axis. And two sections when the speed is high and when the speed is low.

4.5 Actions

For each time step t the output of the agent after processing signal o_t is action a_t . This actions can be either discrete or continuous. For each type of dynamics, the actions will be the same in the discrete and continuous case, except that the Discrete case is the Continuous with a fixed amount for each action.

For the Cartesian case, they represent increasing or decreasing one of the components of the velocity, thus accelerating in one of four possible directions. For the Polar case, they will represent changing the orientation angle and thrusting or slowing down in the orientation direction. We mainly use the Cartesian ones in the experiments. We can see it illustrated in Fig. 4.16

4.5.1 Discrete

This action is an integer representing the choice between 4 possible actions. We chose to model the actions as a discrete choice because this conditions the algorithms available and limits the computational complexity of the problem.

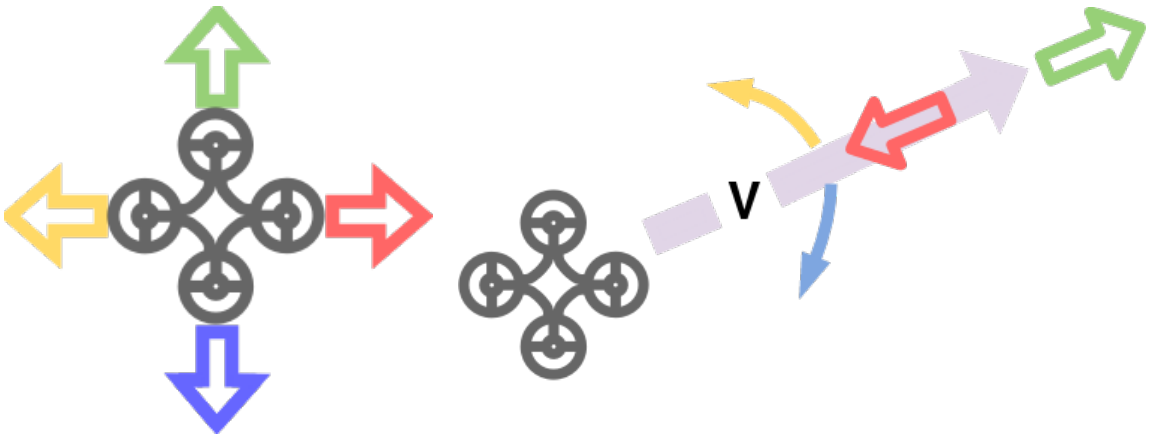

Fig. 4.16: Types of actions

5. EXPERIMENTAL SETTING AND RESULT

In this section, we are going to explain the trials performed and comment on the results and their significance. We present some major experiments already hinted throughout the work. They pose important questions that had an impact on arriving at the final goal of making a UAV to learn to navigate.

5.1 Testing Different Learning Algorithms

The first question was:

Do all the algorithms perform the same in matters of:
Achieved Score?
Speed?
Reliability?
And what is the best one for our problem?

5.1.1 Experiment Details

To test all the algorithms in terms of performance, speed and reliability we are going to set up an environment with 3 obstacles. In which they have to learn the task of getting close to the goal, with a controlled speed and not colliding in less than the T_{max} maximum time-steps. We will use the dense reward to speed up the training and be able to finish under 1M time-steps.

The training for each algorithm was run 10 times with different seeds to make sure the less reliable methods converged. At the end of 10^6 time-steps of training, the agents were evaluated for 100 episodes and its performance was measured. We evaluate the performance of the algorithms by four metrics.

- The first one, reliability refers to the percentage of the 10 seeds that did not collapse the policy to a bad local maximum, meaning the percentage of seeds in which from the 100 evaluation runs at least 10 runs reached the objective.
- The second one is time. Measuring the average time it took the algorithms to train for 1M timesteps.
- The third metric, average score refers to the mean reward in the last 100 evaluation time-steps. The evolution of this metric during the training is shown in Fig. 5.1. By design decisions of the reward function, it will be always negative except for the success-case where the agent reaches the goal, this means that the performance metric will mostly be negative. So qualitatively we can say that when the average score goes above -1.5, it means that the agent starts completing episodes. This is why we use a third metric, to clarify and explain in a more qualitative way how the algorithm is performing.
- The fourth metric is the percentage of successfully finished episodes in the 100 evaluation runs, which is much more descriptive than the average score. For the second and third metric, we chose the best performing seed of each algorithm.

5.1.2 Results

After running the different algorithms we have that the results are displayed in Table 5.1 and the training curves are shown in Fig. 5.1. Looking at them we see that the algorithms DQN and ACKTR rapidly converged to an average score close to the final performance, and steadily increased from there. The A2C algorithm is the slowest one to converge but steadily rises to higher performance than the other three. The ACER algorithm quickly converges to an average score of -3 and until later in training, it

does not leave the bad local maxima. The algorithms of PPO and TRPO didn't converge over the score of -3 in any of the training runs.

Tab. 5.1: Algorithm Performance

Alg. Name	**Performance**			
	Reliability	***Time***	***Avg.Score***	***Completed***
DQN	100%	3380s ± 12s	-1.156	85%
A2C	100%	2450s ± 6s	-0.957	88%
ACER	40%	2859s ± 36s	-0.990	87%
ACKTR	100%	2097s ± 28s	-1.190	86%
TRPO	0%	1670s ± 21s	-2.907	0%
PPO2	0%	1995s ± 11s	-3.004	0%

We could not come up with a convincing hypothesis for the bad result given by TRPO and PPO. However, other papers that deal with Navigation and RL never use them as a baseline [Mnih et al., 2016], [Mirowski et al., 2016] , [Dosovitskiy and Koltun, 2016], [Savva et al., 2017]. Even when running for longer, those algorithms could not explore out of that local maximum as seen in Fig. 5.2.

About the reliability of ACER, only four of the ten seeds managed to converge to a policy that reached the target, as seen in Fig. 5.3.. These results could be explained because ACER is trained off-policy and these methods are supposed to be more sample efficient but less stable(The Deadly Triad chapter, in [Sutton and Barto, 2011]).

In regards to the *time* category we can see that DQN is the slowest, and PPO and TRPO are the fastest computationally.

We can note that the *Completed* score of four algorithms successfully completed the task more than 85% of the times which is impressive for a general-purpose reinforcement learning algorithm that started without any previous knowledge about the task it is solving.

On more qualitative analysis, while watching the agent during training, we can point out what several stages of the training looked like. At the beginning with the randomly initialized policy, the agent moved erratically shaking or directly going outside the map. When it reaches the plateau of -3 of *Average Score*, it learned to move in opposite directions each time-step, resulting in standing still and avoiding crashes and bad outcomes. But it was also unable to reach the target. Finally, when the agent is trained we can see it avoids certain obstacles but failed to evade others when it moves fast. So we should take into account that an 85% does not represent an industry-ready navigation algorithm, but provides a strong result for model-free RL algorithms and settles a baseline to compare other methods to it.

Finally on a when deciding which algorithm are we going to choose, we find that A2C has a slightly higher score than the rest but is slow to ramp up in training. While ACKTR provides solid performance across all aspects, so for future experiments we will be using ACKTR.

5.2 Curriculum Learning

We introduced Curriculum Learning as one of the possible solutions for guiding the exploration and mitigating *Exploration vs Exploitation* and *Sample Inefficiency* problems. We are going to test the hypothesis if:

Does Curriculum Learning speed up the learning or even guides to better policies?

5.2.1 Experiment Details

For this experiment, we are going to make two identical agents learn in different environments. The task will be to reach the target without any speed restrictions. Both will be trained with a Sparse Reward function and they will be run for 60M time-steps. One of them is going to learn in an environment with 6 obstacles and the other one is going to learn in a new version that changes its difficulty whenever the agents surpass certain performance in that level. The difficulty is increased by adding obstacles and reducing the distance threshold which the UAV has to reach. The list of levels of difficulty is shown in Table 5.2.

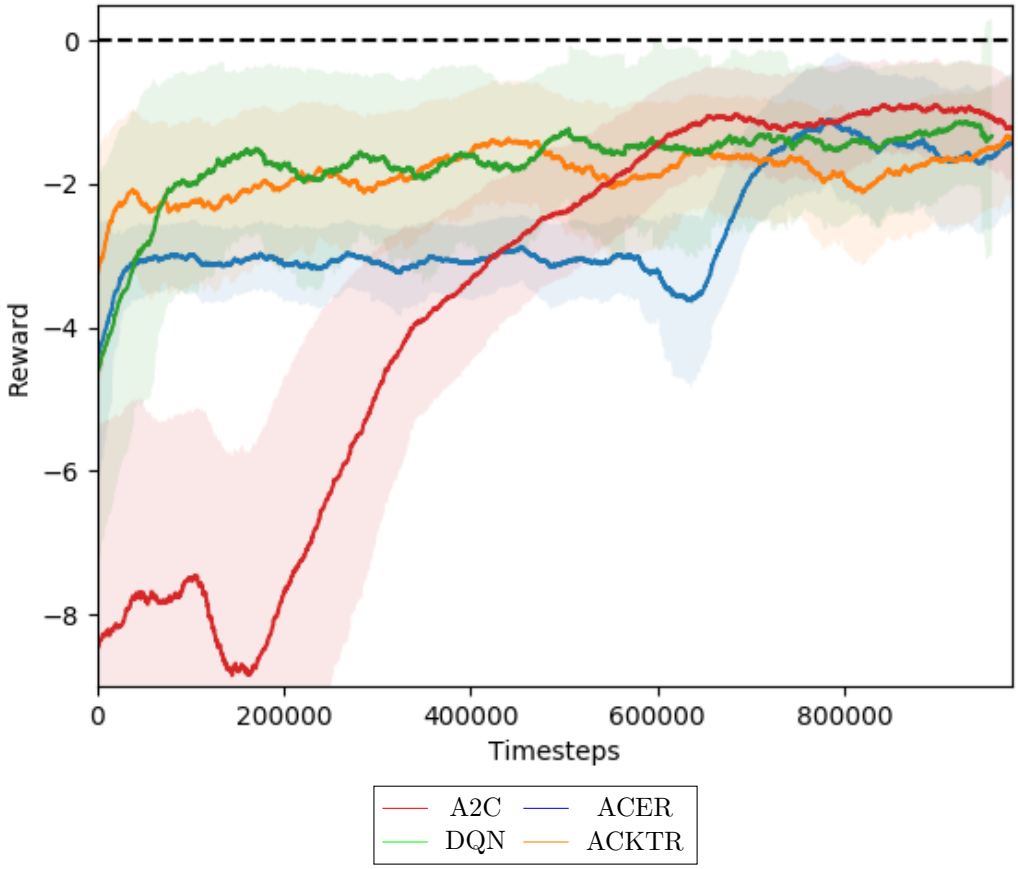

Fig. 5.1: Learning curves of the best seed for each algorithm smoothed over 400 time steps. Shadowed area represents $\pm 0.5\sigma$ over the smoothed average. PPO and TRPO not included here.

As the average score and successful episodes will be changing in the Curriculum environment with the increase of difficulty, we expect to see saw-shaped curves for the Curriculum Environment and faster improvement than the Standard one.

5.2.2 Results

We have the training curves shown in Fig. 5.4 and Fig. 5.5. The results (averaged for all the seeds) of the evaluation episodes after the training is shown in the Table 5.3. As we expected, the saw shape appears in the curriculum learning one. Both get to pretty good performances. But the Curriculum Learning approach not only ramps up faster, it also keeps a higher average result during all the training. And while this is expected during the lower levels of curriculum, once it gets to the last level it is clearly over the standard setting and it finishes with a 10 % more completed episodes than it.

We can appreciate that the policies trained with curriculum learning achieve a 10% more successful episodes than the ones trained in the standard setting. This is a more difficult thing to see in the curves in Fig. 5.4 and Fig. 5.5. However if we look closely towards the end, Curriculum Learning is on a higher performance even though the standard setting is trending towards that level of performance.

Also to test if the idea that Curriculum Learning speeds up training holds, we are going to look at the beginning of the training of the two cases in Fig. 5.6 and Fig. 5.7. In the curriculum one, we expected to have that saw-shape due to the increase of difficulty, and that level 8 is achieved and has better performance than the standard case. For the standard case, we just expected a steady increase, maybe a jump if it has learned something like avoiding easy obstacles. This supports the idea that Curriculum Learning is a useful technique when you want to improve the exploration.

Evaluating these models in a quantitative way further than the number of successful episodes is hard. Evaluating them qualitatively is a subjective task, but sometimes very useful to find what is happening

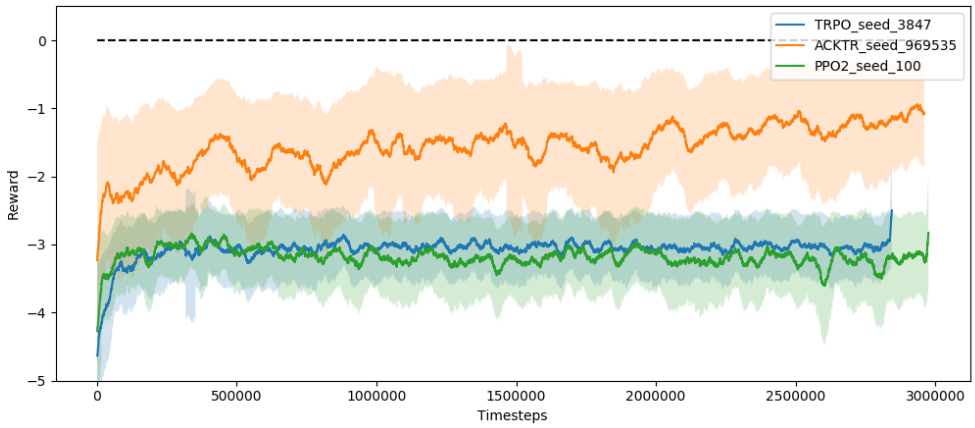

Fig. 5.2: Learning curves (smoothed over 400 timesteps) comparing ACKTR, TRPO and PPO2. Shadowed area represents $\pm 0.5\sigma$ over the smoothed average.

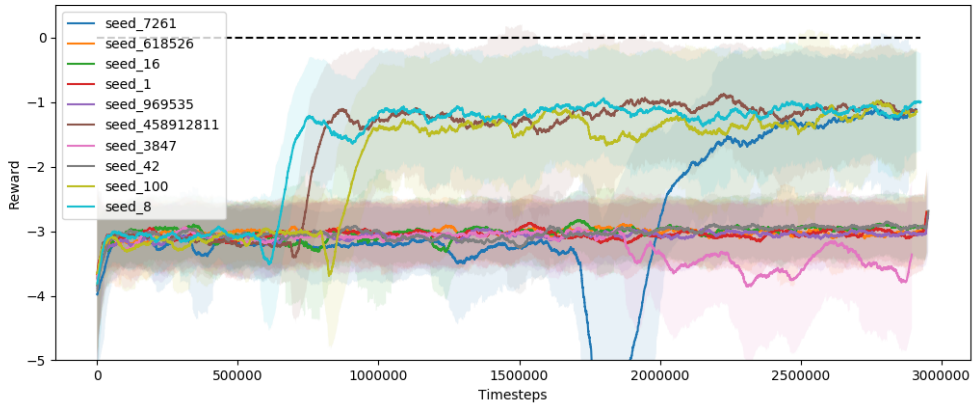

Fig. 5.3: Learning curves (smoothed over 400 timesteps) comparing ACER seeds. Shadowed area represents $\pm 0.5\sigma$ over the smoothed average.

beyond what the numbers tell you. Here we will analyze some of the trajectories in order to get a better understanding of what the model has learnt. We have a sample of good trajectories, trajectories that exceeded the maximum time-step and trajectories that went wrong.

- For the standard case in Fig 5.8, Fig 5.9 and Fig 5.10.
- For the curriculum learning case in Fig 5.11, Fig 5.12 and Fig 5.13.

We can certainly see interesting trajectories from both of the cases, where the agent follows a wall like a bug to reach the objective (this path-finding algorithm exists and it is called Bug-Pathing [Choset,]), avoid the objectives perfectly but fail to reach the target, or just smash into an obstacle without turning or slowing down.

For the failure cases, both seem to struggle with small objects, corners or when they reach a high speed after a long straight line.

Some more interesting cases are the ones of the timeouts. For the curriculum learning case, there were several examples (last 2 from Fig 5.12) where the agent had completed the obstacle avoidance but did not manage to get to the target because the total timesteps run out. Other cases like the 2nd trajectory from Fig 5.12 or the 1st and 4th from Fig 5.9, seem to fail at completing the Navigation problem where they have to backtrack to find the free route. This result was somewhat expected from the different policies because they are implemented without any type of memory (like Recurrence in the NN). So they will face problems in scenarios where the correct solution involves going back on your steps to take an alternative route because when facing the same places again the policy will take the same actions.

Tab. 5.2: Difficulty Levels for Curriculum Learning.

Difficulty Levels		
Level	**Obstacles**	**Distance Threshold**
0	0	60
1	0	40
2	0	20
3	1	40
4	2	40
5	3	40
6	4	40
7	5	40
8	6	40
9	7	40
10	8	40

Tab. 5.3: Averaged Performance of the Policy trained with Curriculum Learning versus the Standard setting.

Episode	**Policy Performance**	
Result	*Standard Setting*	*Curriculum Learning*
Good	72.33%	**82.33%**
Timeout	7.00%	**8.66%**
Fail	20.66%	**9.00%**

In the successful case, both agents learn to avoid simple obstacles and some complex scenarios too. Maybe the curriculum learning one is able to complete more complicated ones, but this might be subjective. The high number of completed episodes is quite impressive for a reactive system and suggests that new navigation techniques could make use of Reinforcement Learning techniques.

5.3 Transfer Learning

Finally, we have the latest experiment in which we test the domain adaptation possibilities of reinforcement learning. This theory of transfer learning is a big promise for robotics, which aims to train in simulators and then deploy to real-world robots. We are going to skip the real-world part but are going to transfer from a simple environment to a more complex and realistic. For this experiment, we created another environment on top of AirSim, where the agent has to perform a similar task but with different dynamics.

We are going to see if:

It is possible to learn a policy in one environment and deploy it in a similar one without losing much performance?

5.3.1 Experiment Details

For this experiment we will get one of the models previously trained in the UAVenv and deploy it in the AirSim environment we created for this task. The agent was trained with curriculum learning for 150M timesteps with a sparse reward function. The agent will have to go from one ball to another selected at random and located around the environment without crashing into the blocks. The environment is detailed in 4.2.1.

For this experiment we can only compare the number of successful episodes in the old environment and in the new one. And qualitatively analyze the drone flight in a video as well as the trajectories.

5.3.2 Results

Here is the table of results Table 5.4 in the old and new environment, we can see a drop of performance as we expected. Looking at some of the trajectories in Fig. 5.14 and Fig. 5.15, we can see that it has lost some of the turning capabilities that showed in Fig. 5.11 due to the new dynamics of the environment. However, it is still able to perform turns in order to avoid the obstacles. For the negative cases, we see that it struggles with getting high speed and losing control. Other negative cases we can appreciate are when getting near the corner, that was a defect that transferred from the old policy. It is not certainly the 85% that we had on the previous policy but this experience provides a brilliant starting point to train in the realistic simulator reducing the amount of training needed. There is also an important point to note, and that is when you work with increasingly realistic environments you start facing the randomness of reality. We could appreciate it when testing the policy, the drone got stuck in the walls of the blocks several times counting as failed episodes until it was able to reset properly.

Finally to get a better grasp qualitatively of the performance of the drone we can see a video linked from Fig. 5.16 of the policy in action.

Tab. 5.4: Performance of the Transferred Policy

Episode	**Policy Performance**	
Result	*Previous Env.*	*AirSim Env.*
Good	85%	41,3%
Timeout	3%	0%
Fail	12%	58,6%

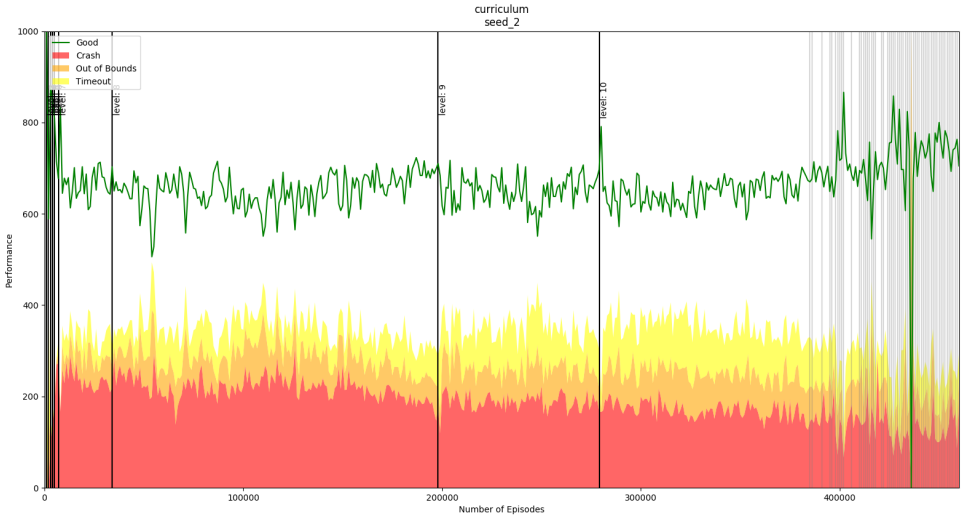

Fig. 5.4: Performance curve for the Curriculum Learning (seed 2) during training. Black bars represent the change of level during training, grey bars represents having passed the last level.

Fig. 5.5: Performance curve for the Standard Setting (seed 0) during training.

Fig. 5.6: Performance curve for the Curriculum Learning (seed 2) during training during the first 60000 timesteps. Black bars represent the change of level during training. Average performance at the end of the 60k timesteps around 650

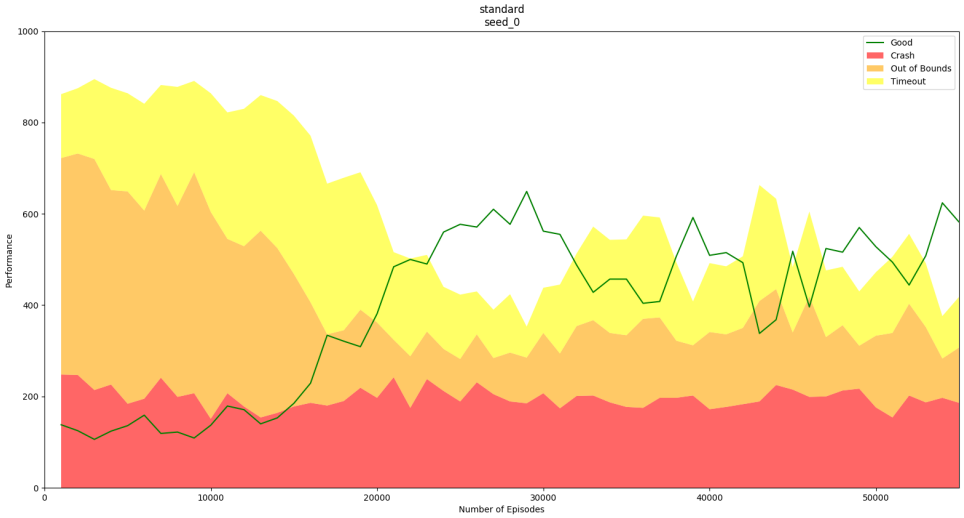

Fig. 5.7: Performance curve for the Standard Setting (seed 0) during training during the first 60000 timesteps. Average performance at the end of the 60k timesteps around 580

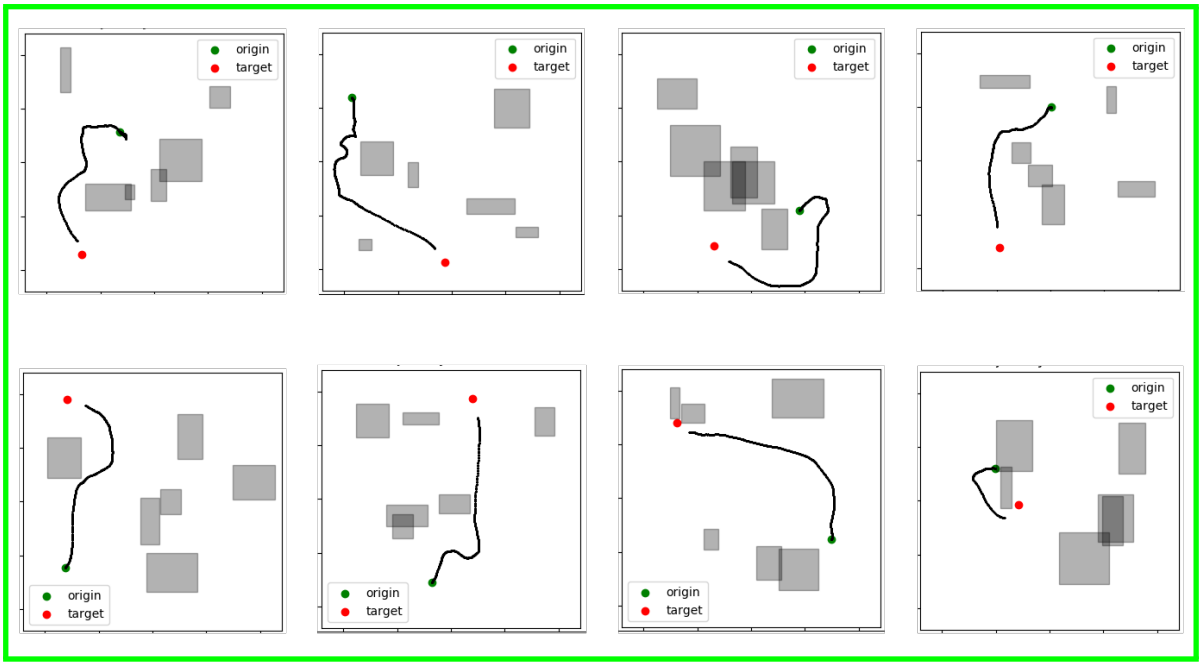

Fig. 5.8: 8 Successful trajectories from the Standard Policy.

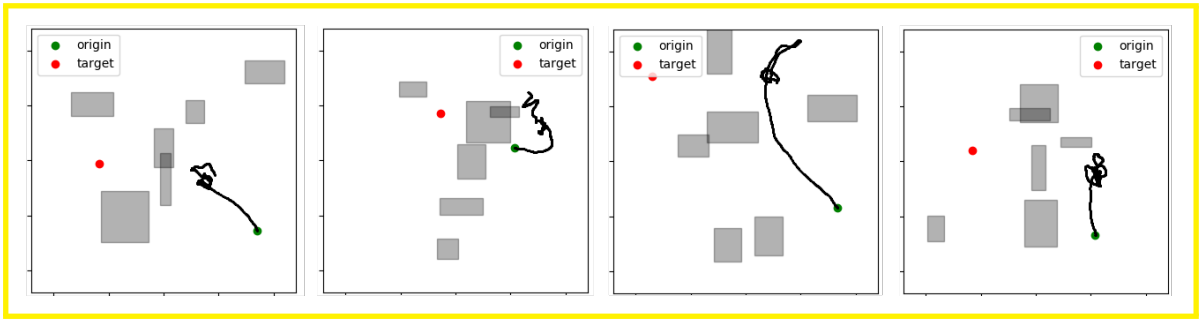

Fig. 5.9: 4 Timeout trajectories from the Standard Policy.

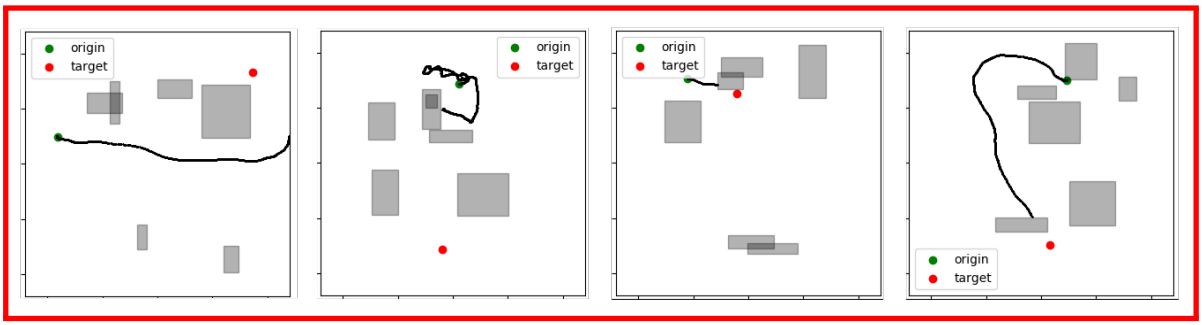

Fig. 5.10: 4 Fail trajectories from the Standard Policy.

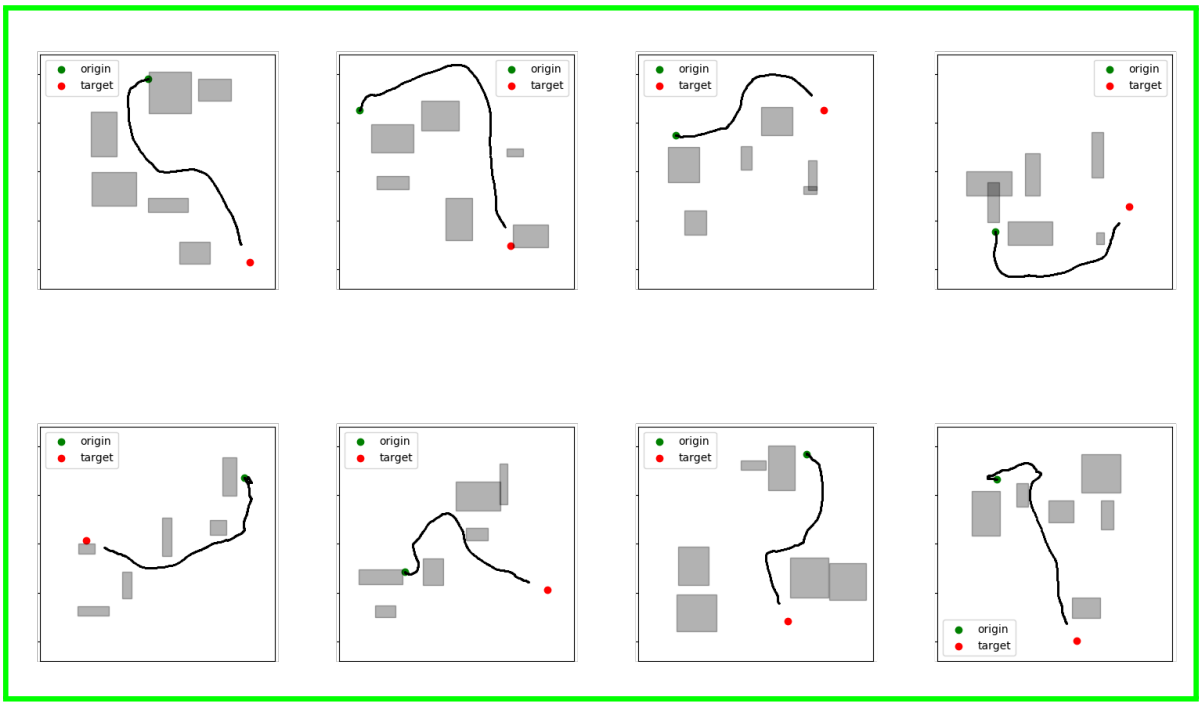

Fig. 5.11: 8 Successful trajectories from the Curriculum Learning Policy.

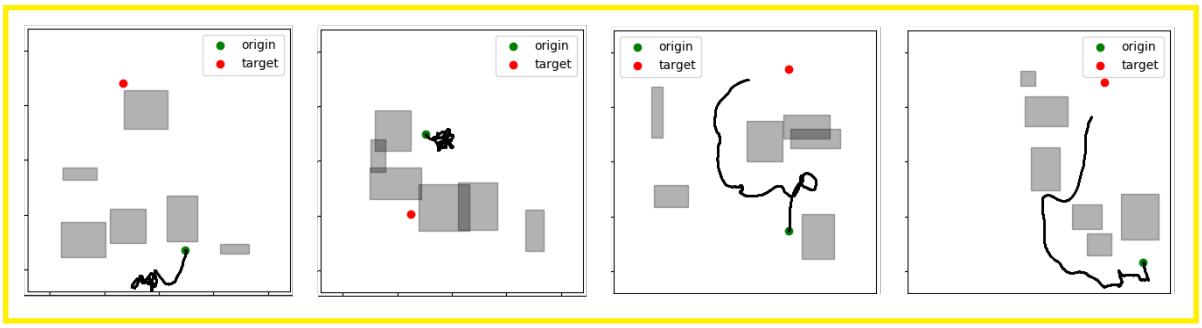

Fig. 5.12: 4 Timeout trajectories from the Curriculum Learning Policy.

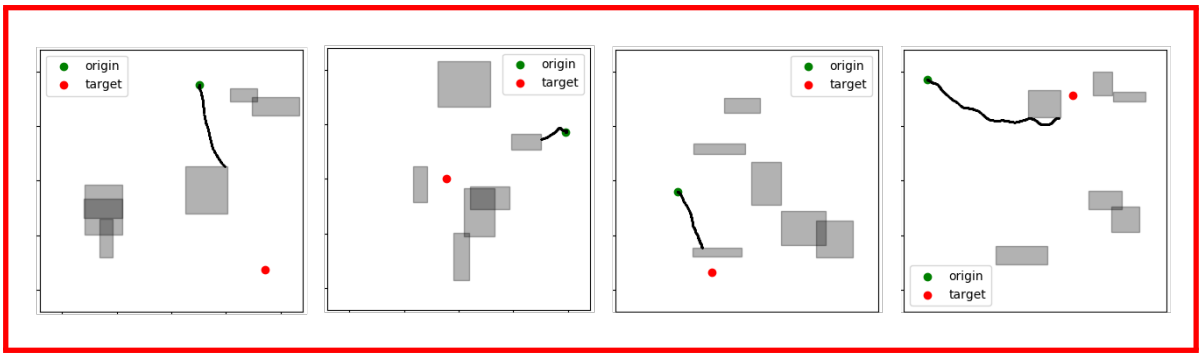

Fig. 5.13: 4 Fail trajectories from the Curriculum Learning Policy.

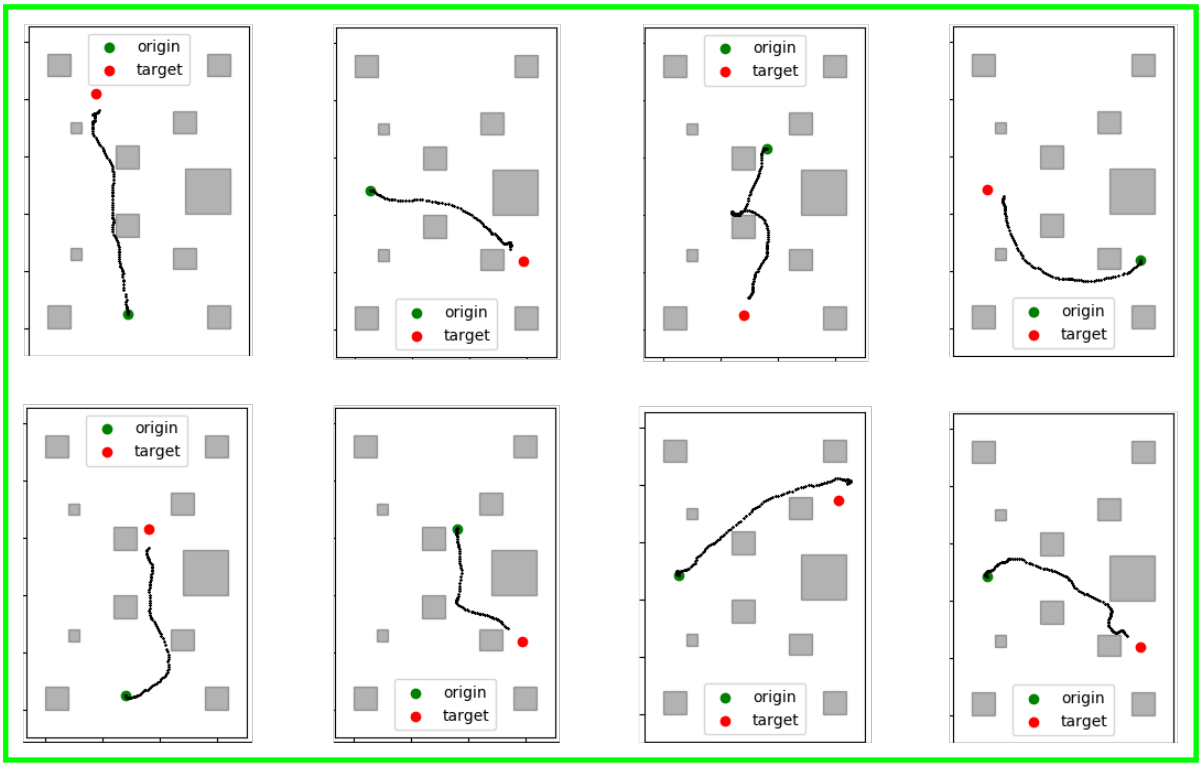

Fig. 5.14: 8 Successful trajectories from the Transferred Policy.

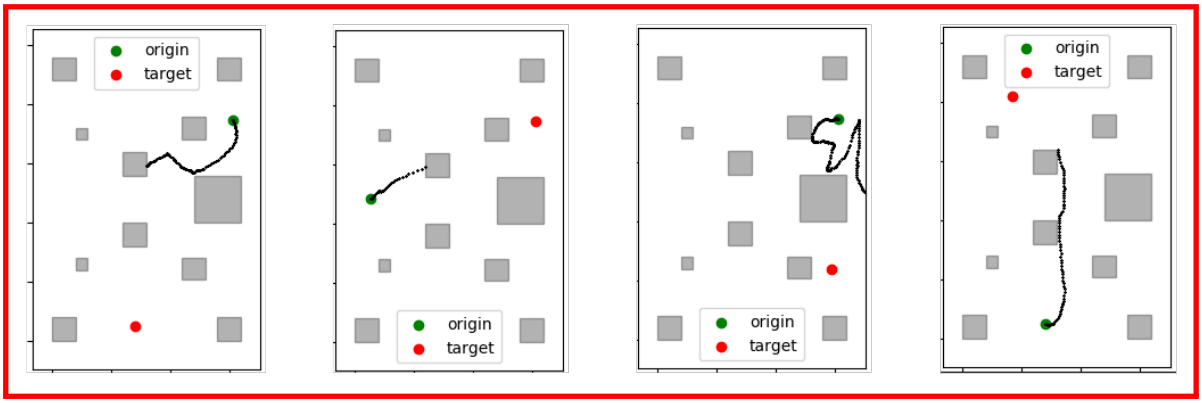

Fig. 5.15: 4 Fail trajectories from the Transferred Policy.

Fig. 5.16: Video of the Transferred policy. To watch it scan the QR code or go to this url: `https://youtu.be/CfLkdpxGZgA`.

6. CONCLUSION AND FUTURE LINES

In this section we are going to reflect on the ideas we learned form this work, what can we take from it and look into the future of how can it be improved.

6.1 Conclusion

The original intent from the work was to see if we could make a UAV fly avoiding obstacles without explicitly telling it how to do it. And we can say that this objective has been completed successfully.

In this work, we approached the problem of partially observable navigation with a reactive system trained by model-free Reinforcement Learning. This approach is attractive because it can reduce engineering effort at the cost of more computing power during training. We designed an agent with a focus on it being able to navigate independently of the map. And we developed an environment with a reward function that captures the problem we are trying to solve. We used well-tested RL algorithms without any hyper-parameter tuning and achieved promising results. Going from agents moving aimlessly, to reaching the target more than 85% of the episodes. Despite having promising results, there is still work to be done with this approach and other ones to achieve industry-ready navigation systems.

We had to face a learning problem and dealt with the limitations of RL and the ever-present Sample Inefficiency, which most of the solutions were centred around mitigating it. If we had to take away some ideas from Reinforcement learning they would be:

- Although it has some limitations RL provides a very useful tool for solving problems in the future. Given its generality, there will be a lot of applications that make use of it in the future
- RL excels in settings where it is easy to generate huge amounts of experience, like simulators; The problem can be simplified, like ours with the smaller environment; There is a clear way to define a reward function and if it is possible that it be rich.
- When implementing it, it should be kept always in mind the list of limitations and how can they be hindering your performance. Even tough it is a very powerful tool, the implementation and debugging processes are very hard and complex.

Then there is the UAV part of the problem, that showed us that flying is no easy task even if it is simplified. However, it will have many applications in the future and be key to a lot of fields like agriculture, maintenance or surveillance. We also saw that Navigation is an essential building block in the future, especially for robotics and autonomous systems. And that for now it is far from solved, and much more from a learning perspective.

Now for the personal takeaway from this work, I would remark some aspects:

In this thesis, we had to work with different disciplines and different skill-sets. There was a lot of theoretical work to understand and put in context the learning algorithms without making them look like ”magic”. The theory was also useful to solve implementation details which could have led to different types of failure.

There has also been a lot of research work, looking for the latest papers and related work facing not only our problem but similar problems like work in robotics or video-games. The field of RL is also a very fast paced growing field nowadays, with more than 3 major conferences each year. During the development of the work, there were new papers, tools and algorithms released (many citations come from 2018).

Then there was a lot of practical work, for the development of the environment there were more than 5000 lines of code written. A lot of the project involved testing and figuring out what was the mistake that blocked progress. Certainly software development skills were essential and they were improved during this work. If the best practices are applied from the beginning, a lot of headaches and problems will be avoided. Proper Version Control, good documentation, Tests for everything and logging tools from the beginning are highly encouraged.

There was also hardware management skills involved too. During the development of this work, we had to set up 3 computers with the hardware and software needed so that we could run the experiments in timescales closer to a day, not a week.

6.2 Future Lines

Looking at the future in the context of this work we could spot several incremental improvements, lots of them include working with features we implemented but didn't have the time to work on:

- Try to work with recurrent policies. Theoretically they should improve the solution [Hausknecht and Stone, 2015], however, hey are really hard to implement properly, tune the hyper-parameters and debug when they don't work.
- Train in the AirSim environment for a longer time. The compute time and time constarins limited us for trying to train in this environment.
- Add more randomization to the UAVenv. It would be interesting to test further the domain randomization hypothesis and its consequences.
- Try the some of the features we implemented like working with continuous actions.
- Another feature would be to work with image inputs.
- Make the system somehow work with the directed dynamics. We found a lot of problems when working with angles, but it should be possible to do somehow.

For the more advanced work which could lead to interesting research directions. Some of this ideas are being investigated in top research laboratories all around the world. It would be interesting to:

- Try to make use of model-based learning methods. Imbuing an algorithm with human knowledge should increase performance.
- Try to find the middle ground where classical methods like SLAM could benefit from the power of RL.
- If better performance is achieved, try to deploy the policy to a real-life drone and see experiment with different tasks.
- Try more limited observations, for our environment we implemented an input system we finally didn't get to use. The collision perception system tried to imitate an already existent system that outputs a probability of a crash and a steering angle to avoid the obstacle. This was inspired by [Loquercio et al., 2018], where they train a computer vision system to get an image and output those two numbers, with the hopes of implementing the real vision system in the simulator. This was convenient because it reduced the collision detection to just 2 numbers keeping the input vector small, and it promised to work with real images. This reduced a lot the dimensionality of the problem, but also limits the observability a lot so it becomes a harder challenge.

BIBLIOGRAPHY

[Abadi et al., 2016] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283.

[Achiam, 2018] Achiam, J. (2018). Spinning up in deep rl. `https://spinningup.openai.com/en/`.

[Anderson et al., 2018] Anderson, P., Chang, A., Chaplot, D. S., Dosovitskiy, A., Gupta, S., Koltun, V., Kosecka, J., Malik, J., Mottaghi, R., Savva, M., et al. (2018). On evaluation of embodied navigation agents. *arXiv preprint arXiv:1807.06757*.

[Andrychowicz et al., 2018] Andrychowicz, M., Baker, B., Chociej, M., Jozefowicz, R., McGrew, B., Pachocki, J., Petron, A., Plappert, M., Powell, G., Ray, A., et al. (2018). Learning dexterous in-hand manipulation. *arXiv preprint arXiv:1808.00177*.

[Bengio et al., 2009] Bengio, Y., Louradour, J., Collobert, R., and Weston, J. (2009). Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM.

[Brockman et al., 2016] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym.

[Cadena et al., 2016] Cadena, C., Carlone, L., Carrillo, H., Latif, Y., Scaramuzza, D., Neira, J., Reid, I., and Leonard, J. J. (2016). Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age. *IEEE Transactions on robotics*, 32(6):1309–1332.

[Chang et al., 2017] Chang, A., Dai, A., Funkhouser, T., Halber, M., Niessner, M., Savva, M., Song, S., Zeng, A., and Zhang, Y. (2017). Matterport3d: Learning from rgb-d data in indoor environments. *arXiv preprint arXiv:1709.06158*.

[Chevalier-Boisvert et al., 2018] Chevalier-Boisvert, M., Willems, L., and Pal, S. (2018). Minimalistic gridworld environment for openai gym. `https://github.com/maximecb/gym-minigrid`.

[Choset,] Choset, H. Robotic motion planning:bug algorithm. `https://www.cs.cmu.edu/~motionplanning/lecture/Chap2-Bug-Alg_howie.pdf`.

[Daftry et al., 2016] Daftry, S., Bagnell, J. A., and Hebert, M. (2016). Learning transferable policies for monocular reactive mav control. In *International Symposium on Experimental Robotics*, pages 3–11. Springer.

[Dean, 2017] Dean, J. (2017). Machine learning for systems and systems for machine learning. In *Presentation at 2017 Conference on Neural Information Processing Systems*.

[DeepMind, 2016] DeepMind (2016). Deepmind ai reduces google data centre cooling bill by 40%. `https://deepmind.com/blog/deepmind-ai-reduces-google-data-centre-cooling-bill-40/`. Accessed 21/05/19.

[Dey et al., 2016] Dey, D., Shankar, K. S., Zeng, S., Mehta, R., Agcayazi, M. T., Eriksen, C., Daftry, S., Hebert, M., and Bagnell, J. A. (2016). Vision and learning for deliberative monocular cluttered flight. In *Field and Service Robotics*, pages 391–409. Springer.

[Dhariwal et al., 2017] Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., Wu, Y., and Zhokhov, P. (2017). Openai baselines. `https://github.com/openai/baselines`.

[Dosovitskiy and Koltun, 2016] Dosovitskiy, A. and Koltun, V. (2016). Learning to act by predicting the future. *arXiv preprint arXiv:1611.01779*.

[Gauci et al., 2018] Gauci, J., Conti, E., Liang, Y., Virochsiri, K., He, Y., Kaden, Z., Narayanan, V., and Ye, X. (2018). Horizon: Facebook's open source applied reinforcement learning platform. *arXiv preprint arXiv:1811.00260*.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. MIT press.

[Gupta et al., 2017a] Gupta, S., Davidson, J., Levine, S., Sukthankar, R., and Malik, J. (2017a). Cognitive mapping and planning for visual navigation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2616–2625.

[Gupta et al., 2017b] Gupta, S., Fouhey, D., Levine, S., and Malik, J. (2017b). Unifying map and landmark based representations for visual navigation. *arXiv preprint arXiv:1712.08125*.

[Hausknecht and Stone, 2015] Hausknecht, M. and Stone, P. (2015). Deep recurrent q-learning for partially observable mdps. In *2015 AAAI Fall Symposium Series*.

[Henderson et al., 2018] Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., and Meger, D. (2018). Deep reinforcement learning that matters. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

[Hill et al., 2018] Hill, A., Raffin, A., Ernestus, M., Traore, R., Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., and Wu, Y. (2018). Stable baselines. `https://github.com/hill-a/stable-baselines`.

[Irpan, 2018] Irpan, A. (2018). Deep reinforcement learning doesn't work yet. `https://www.alexirpan.com/2018/02/14/rl-hard.html`.

[Jaderberg et al., 2016] Jaderberg, M., Mnih, V., Czarnecki, W. M., Schaul, T., Leibo, J. Z., Silver, D., and Kavukcuoglu, K. (2016). Reinforcement learning with unsupervised auxiliary tasks. *arXiv preprint arXiv:1611.05397*.

[Jiang et al., 2015] Jiang, L., Meng, D., Zhao, Q., Shan, S., and Hauptmann, A. G. (2015). Self-paced curriculum learning. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*.

[Kaelbling et al., 1998] Kaelbling, L. P., Littman, M. L., and Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1-2):99–134.

[Karkus et al., 2018] Karkus, P., Hsu, D., and Lee, W. S. (2018). Integrating algorithmic planning and deep learning for partially observable navigation. *arXiv preprint arXiv:1807.06696*.

[Kolve et al., 2017] Kolve, E., Mottaghi, R., Gordon, D., Zhu, Y., Gupta, A., and Farhadi, A. (2017). Ai2-thor: An interactive 3d environment for visual ai. *arXiv preprint arXiv:1712.05474*.

[LaValle, 2006] LaValle, S. M. (2006). *Planning algorithms*. Cambridge university press.

[Liu et al., 2017] Liu, S., Atanasov, N., Mohta, K., and Kumar, V. (2017). Search-based motion planning for quadrotors using linear quadratic minimum time control. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2872–2879. IEEE.

[Liu et al., 2018] Liu, S., Mohta, K., Atanasov, N., and Kumar, V. (2018). Search-based motion planning for aggressive flight in se (3). *IEEE Robotics and Automation Letters*, 3(3):2439–2446.

[Lopez and How, 2017] Lopez, B. T. and How, J. P. (2017). Aggressive 3-d collision avoidance for high-speed navigation. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5759–5765. IEEE.

[Loquercio et al., 2018] Loquercio, A., Maqueda, A. I., del Blanco, C. R., and Scaramuzza, D. (2018). Dronet: Learning to fly by driving. *IEEE Robotics and Automation Letters*, 3(2):1088–1095.

[Mellinger and Kumar, 2011] Mellinger, D. and Kumar, V. (2011). Minimum snap trajectory generation and control for quadrotors. In *2011 IEEE International Conference on Robotics and Automation*, pages 2520–2525. IEEE.

[MIcrosoft,] MIcrosoft. Neurips conference analytics, microsoft. https://www.microsoft.com/en-us/research/project/academic/articles/neurips-conference-analytics/. Accessed: 21-05-2019.

[Mirowski et al., 2016] Mirowski, P., Pascanu, R., Viola, F., Soyer, H., Ballard, A. J., Banino, A., Denil, M., Goroshin, R., Sifre, L., Kavukcuoglu, K., et al. (2016). Learning to navigate in complex environments. *arXiv preprint arXiv:1611.03673*.

[Mnih et al., 2016] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937.

[Mnih et al., 2015] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529.

[Molchanov et al., 2018] Molchanov, A., Hausman, K., Birchfield, S., and Sukhatme, G. (2018). Region growing curriculum generation for reinforcement learning. *arXiv preprint arXiv:1807.01425*.

[Müller et al., 2018] Müller, M., Dosovitskiy, A., Ghanem, B., and Koltun, V. (2018). Driving policy transfer via modularity and abstraction. *arXiv preprint arXiv:1804.09364*.

[Ng et al., 1999] Ng, A. Y., Harada, D., and Russell, S. (1999). Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 99, pages 278–287.

[Oh et al., 2016] Oh, J., Chockalingam, V., Singh, S., and Lee, H. (2016). Control of memory, active perception, and action in minecraft. *arXiv preprint arXiv:1605.09128*.

[OpenAI, 2018] OpenAI (2018). Openai five. https://blog.openai.com/openai-five/. Accessed 21/05/19.

[Pan and Yang, 2009] Pan, S. J. and Yang, Q. (2009). A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359.

[Parisotto and Salakhutdinov, 2017] Parisotto, E. and Salakhutdinov, R. (2017). Neural map: Structured memory for deep reinforcement learning. *arXiv preprint arXiv:1702.08360*.

[Sadeghi and Levine, 2016] Sadeghi, F. and Levine, S. (2016). Cad2rl: Real single-image flight without a single real image. *arXiv preprint arXiv:1611.04201*.

[Savva et al., 2017] Savva, M., Chang, A. X., Dosovitskiy, A., Funkhouser, T., and Koltun, V. (2017). Minos: Multimodal indoor simulator for navigation in complex environments. *arXiv preprint arXiv:1712.03931*.

[Schulman et al., 2015] Schulman, J., Levine, S., Abbeel, P., Jordan, M., and Moritz, P. (2015). Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897.

[Schulman et al., 2017] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

[Shah et al., 2017] Shah, S., Dey, D., Lovett, C., and Kapoor, A. (2017). Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics*.

[Shannon, 1950] Shannon, C. (1950). "theseus," a life-sized magnetic mouse controlled by relay circuits, learns its way around a maze. https://www.youtube.com/watch?v=nS0luYZd4fs. Accessed 21/05/19.

[Silver et al., 2016] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484.

[Sutton and Barto, 2011] Sutton, R. S. and Barto, A. G. (2011). Reinforcement learning: An introduction.

[Taylor and Stone, 2009] Taylor, M. E. and Stone, P. (2009). Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(Jul):1633–1685.

[Tordesillas et al., 2018] Tordesillas, J., Lopez, B. T., Carter, J., Ware, J., and How, J. P. (2018). Real-time planning with multi-fidelity models for agile flights in unknown environments. *arXiv preprint arXiv:1810.01035*.

[Van Nieuwstadt and Murray, 1998] Van Nieuwstadt, M. J. and Murray, R. M. (1998). Real-time trajectory generation for differentially flat systems. *International Journal of Robust and Nonlinear Control: IFAC-Affiliated Journal*, 8(11):995–1020.

[Vinyals et al., 2019] Vinyals, O., Babuschkin, I., Chung, J., Mathieu, M., Jaderberg, M., Czarnecki, W. M., Dudzik, A., Huang, A., Georgiev, P., Powell, R., Ewalds, T., Horgan, D., Kroiss, M., Danihelka, I., Agapiou, J., Oh, J., Dalibard, V., Choi, D., Sifre, L., Sulsky, Y., Vezhnevets, S., Molloy, J., Cai, T., Budden, D., Paine, T., Gulcehre, C., Wang, Z., Pfaff, T., Pohlen, T., Wu, Y., Yogatama, D., Cohen, J., McKinney, K., Smith, O., Schaul, T., Lillicrap, T., Apps, C., Kavukcuoglu, K., Hassabis, D., and Silver, D. (2019). AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. `https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/`. Accessed 21/05/19.

[Wang et al., 2016] Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K., and de Freitas, N. (2016). Sample efficient actor-critic with experience replay. *arXiv preprint arXiv:1611.01224*.

[Watkins and Dayan, 1992] Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3-4):279–292.

[Weng, 2017] Weng, L. (2017). An overview of deep learning for curious people. `https://lilianweng.github.io/lil-log/2017/06/21/an-overview-of-deep-learning.html`.

[Weng, 2018] Weng, L. (2018). A (long) peek into reinforcement learning. `https://lilianweng.github.io/lil-log/2018/02/19/a-long-peek-into-reinforcement-learning.html#deadly-triad-issue`.

[Weng, 2019] Weng, L. (2019). Domain randomization for sim2real transfer. `https://lilianweng.github.io/lil-log/2019/05/05/domain-randomization.html`.

[Wu et al., 2017] Wu, Y., Mansimov, E., Grosse, R. B., Liao, S., and Ba, J. (2017). Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. In *Advances in neural information processing systems*, pages 5279–5288.

[Zhu et al., 2017] Zhu, Y., Mottaghi, R., Kolve, E., Lim, J. J., Gupta, A., Fei-Fei, L., and Farhadi, A. (2017). Target-driven visual navigation in indoor scenes using deep reinforcement learning. In *2017 IEEE international conference on robotics and automation (ICRA)*, pages 3357–3364. IEEE.

[Zoph and Le, 2016] Zoph, B. and Le, Q. V. (2016). Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*.