UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS

Willian Marcon Bicaio

Deep Learning for Drone Navigation

São Carlos

2020

Willian Marcon Bicaio

Deep Learning for Drone Navigation

Monografia apresentada ao Curso de Engenharia Aeronáutica, da Escola de Engenharia de São Carlos da Universidade de São Paulo, como parte dos requisitos para obtenção do título de Engenheiro Aeronáutico.

Supervisor: Prof. Dr. Glauco Augusto de Paula Caurin

São Carlos

2020

Ficha catalográfica elaborada pela Biblioteca Prof. Dr. Sérgio Rodrigues Fontes da EESC/USP com os dados inseridos pelo(a) autor(a).

```
       Bicaio, Willian Marcon
B583d     Deep learning for drone navigation / Willian Marcon
       Bicaio; orientador Glauco Augusto de Paula Caurin. São
       Carlos, 2020.


          Monografia (Graduação em Engenharia Aeronáutica)
       -- Escola de Engenharia de São Carlos da Universidade
       de São Paulo, 2020.


          1. Deep drone racing. 2. Autonomous drone racing.
       3. Deep Deterministic Policy Gradient. 4. DDPG. 5.
       Drone racing. 6. Drone navigation. 7. Deep
       reinforcement learning. I. Título.
```

Eduardo Graziosi Silva - CRB - 8/8907

FOLHA DE APROVAÇÃO

Candidato: Willian Marcon Bicaio	
Título do TCC: Deep Learning for Drone Navigation	
Data de defesa: 16/10/2020	

Comissão Julgadora	**Resultado**
Professor Doutor Ricardo Afonso Angélico	Aprovado
Instituição: EESC - SAA	
Pesquisador Flavio Pires Oliva	Aprovado
Instituição: -	

Presidente da Banca: Professor Doutor Ricardo Afonso Angélico

Ricardo A. Angélico

(assinatura)

ACKNOWLEDGEMENTS

Any person who is close enough to me knows how much I am passionate about Aeronautics and Science. My closest friends and family also know the difficulties I had to overcome in order to accomplish this thesis. My most sincere gratitude to them, for helping me hold on to my dreams, for supporting me at all times and for their unconditional help. This really means a lot to me!

I am also grateful to my thesis supervisor, Glauco Augusto de Paula Caurin, for having been so open to my ideas, for the suggestion of this thesis' subject and for all the support and advice he gave me along the way. Having worked with Glauco for the last six months has given me the opportunity to deepen my knowledge on fields such as data science, which will certainly broaden my horizons as an aeronautical engineer.

I would like to thank, as well, all the people who provided me with the knowledge and abilities that I had to apply in order to develop this thesis. Even though from an external point of view my personal achievements are tiny, I feel deeply glad and thankful to them. These achievements would never have been possible if it wasn't for the educators I've had throughout my life, who gave me all the tools I needed along the way. Special thanks to Prof. Ricardo Afonso Angélico, for his inspiring and encouraging teaching methods, his genuine concern for his students and for all the support, advice and friendly conversations we shared during the past few years.

I am also thankful to Caio Augusto Zagria Barbosa, who has shared many of his experiences and background on data science as I worked on this project. His advice and support have saved me a lot of effort in many occasions.

Lastly, I would like to thank anyone else who has dedicated their time to make the accomplishment of this thesis possible.

Thank you for all your encouragement!

ABSTRACT

BICAIO, W. M. **Deep Learning for Drone Navigation**. 2020. 78p. Monografia (Trabalho de Conclusão de Curso) - Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2020.

Deep drone racing and navigation are emerging applications of deep learning which may be used in competitions and potentially to automatize a multitude of tasks accomplished by drones. In this paper, we apply the method Deep Deterministic Policy Gradient (DDPG) to train a neural network whose objective is to direct a simulated quadcopter towards a target, reproducing a simplified drone race environment. The model explored in the paper is not vision-based; it assumes the position and velocity of the drone in relation to the target are known at all times, and these variables are passed as inputs to the model. Based entirely on these variables, the neural network controls the quadcopter's rotors angular speeds, which in turn determine the flight path taken by the drone. DDPG training requires engineering an efficient reward function, which is essential to the convergence of the model. A few different reward functions were tested and are presented in the paper. The results showed that DDPG is a suitable method for training a deep drone racing neural network, as suggested by the fact that, after training, the drone was able to make its way to the target within a certain range of initial distance and regardless of the initial directing vector from the drone to the target. In spite of that, certain minor problems remain to be solved and might be the subject of future works. Among those problems are the facts that the trajectories chosen by the neural network are generally not optimal and that the drone tends to diverge from the target as the former gets closer than a few feet to the latter.

Keywords: Autonomous drone navigation. Deep drone racing. Autonomous drone racing. Deep Deterministic Policy Gradient. DDPG.

RESUMO

BICAIO, W. M. **Deep Learning for Drone Navigation**. 2020. 78p. Monografia (Trabalho de Conclusão de Curso) - Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2020.

Corridas e navegação de drones autônomos são aplicações emergentes da aprendizagem profunda que podem ser utilizadas em competições e potencialmente na automação de uma série de tarefas realizadas por drones. Nesta monografia, aplicamos o método Deep Deterministic Policy Gradient (DDPG) para treinar uma rede neural cujo objetivo é dirigir um quadricóptero simulado rumo a um alvo, reproduzindo um ambiente de corrida de drones simplificado. O modelo explorado na monografia não utiliza nenhum método de visão computacional. Ele assume que a posição e a velocidade do drone em relação ao alvo são conhecidas em qualquer instante, e essas variáveis são passadas como inputs para o modelo. Baseada unicamente nessas variáveis, a rede neural controla as velocidades angulares dos rotores do quadricóptero, que por sua vez determinam a trajetória de voo realizada pelo drone. A aplicação do DDPG para o treinamento de redes neurais requer a construção de uma função de recompensa eficiente, o que é essencial para a convergência do modelo. Algumas opções de funções de recompensa foram testadas e serão apresentadas na monografia. Os resultados demonstraram que o DDPG é um método adequado para o treinamento de redes neurais aplicadas ao problema de deep drone racing, como sugere o fato de que, após o treinamento, o drone foi capaz de se dirigir ao alvo para um certo intervalo de distâncias iniciais e independentemente do vetor diretor inicial do drone ao alvo. Apesar disso, alguns problemas secundários restam a ser solucionados e podem servir como temas de futuros trabalhos. Entre esses problemas estão os fatos de que as trajetórias escolhidas pela rede neural geralmente não são ótimas e de que o drone tende a divergir do alvo quando a distância entre os dois é inferior a cerca de um metro.

Palavras-chave: Navegação autônoma de drones. Deep drone racing. Corrida de drones autônomos. Deep Deterministic Policy Gradient. DDPG.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

AI	artificial intelligence
AIRR	Artificial Intelligence Robotic Racing
CAGR	compound annual growth rate
CNN	convolutional neural network
DDPG	Deep Deterministic Policy Gradient
DPG	Deterministic Policy Gradient
DQN	Deep Q Network
Dr.	Doctor
DRL	Drone Racing League
FPV	first-person view
MDP	Markov decision process
MPC	model-predictive control
ReLU	rectified linear unit
RL	reinforcement learning
UAV	unmanned aerial vehicle
US$	US dollars

LIST OF SYMBOLS

a	action
A	set of all valid actions
$\overrightarrow{a_P}$	quadcopter's acceleration in relation to frame XYZ
$A^{\pi}(\cdot,\cdot)$	advantage function
$a^{\star}$	optimal action
C	center of the quadcopter
c_{M_x}	rolling moment control coefficient
c_{M_y}	pitching moment control coefficient
c_T	thrust control coefficient
f	function governed by the laws of a deterministic environment that determines the next state based on the current state and on the action taken
F_x	x-component of the resulting force acting on the quadcopter (according to frame xyz)
F_y	y-component of the resulting force acting on the quadcopter (according to frame xyz)
F_z	z-component of the resulting force acting on the quadcopter (according to frame xyz)
$\overrightarrow{g}$	acceleration of gravity
$\hat{i}$	unit vector in the direction of x (of frame xyz)
$\hat{I}$	unit vector in the direction of X (of frame XYZ)
I_{xx}	quadcopter's moment of inertia about the x-axis (of frame xyz)
I_{xy}	quadcopter's product of inertia with respect to the x and y axes (of frame xyz)
I_{xz}	quadcopter's product of inertia with respect to the x and z axes (of frame xyz)

I_{yy}	quadcopter's moment of inertia about the y-axis (of frame xyz)
I_{yz}	quadcopter's product of inertia with respect to the y and z axes (of frame xyz)
I_{zz}	quadcopter's moment of inertia about the z-axis (of frame xyz)
$\hat{j}$	unit vector in the direction of y
$\hat{J}$	unit vector in the direction of Y
$J(\pi)$	expected return given a policy π
$\hat{k}$	unit vector in the direction of z
$\hat{K}$	unit vector in the direction of Z
k_{M_T}	thrust moment coefficient
k_T	thrust force coefficient
L	distance from one rotor to the center of the quadcopter
$L(\theta^{PP})$	loss function for estimation of Q^{μ} using a neural network
m	quadcopter's mass
$\overrightarrow{M_{T_1}}$ to $\overrightarrow{M_{T_4}}$	rotors' thrust moments
M_x	x-component of the resulting moment acting on the quadcopter about its center of mass (according to frame xyz)
M_x^{target}	target rolling moment
M_y	y-component of the resulting moment acting on the quadcopter about its center of mass (according to frame xyz)
M_y^{target}	target pitching moment
M_z	z-component of the resulting moment acting on the quadcopter about its center of mass (according to frame xyz)
$\mathcal{N}$	noise
$\mathcal{N}(\mu, \sigma^2)$	normal distribution with mean mu and variance σ^2
P	probability distribution for the next state based on the current state and on the action taken, for a stochastic environment
P^{SV}	quadcopter's roll rate ($\overrightarrow{\omega}$'s x-coordinate)

$\dot{P}^{SV}$	$\partial P^{SV}/\partial t$
Q^{SV}	quadcopter's pitch rate ($\overrightarrow{\omega}$'s y-coordinate)
$\dot{Q}^{SV}$	$\partial Q^{SV}/\partial t$
$Q(s,a\|\theta^{Q})$	critic network
$Q'(s,a\|\theta^{Q'})$	target critic network
$Q^{\pi}(\cdot,\cdot)$	on-policy action-value function given a policy π
$Q^{\star}(\cdot,\cdot)$	optimal action-value function
r	reward
R	reward function
R^{RB}	replay buffer
R^{SV}	quadcopter's yaw rate ($\overrightarrow{\omega}$'s z-coordinate)
$\dot{R}^{SV}$	$\partial R^{SV}/\partial t$
R_1 to R_4	quadcopter's rotors
$R(\tau)$	return over trajectory τ
s	state or observation
S	set of all valid states
t	time
T	total thrust force acting on the quadcopter
$\overrightarrow{T_1}$ to $\overrightarrow{T_4}$	rotors' thrust forces
T^{target}	target total thrust
u	$\overrightarrow{V_P}$'s x-coordinate
$\dot{u}$	$\partial u/\partial t$
v	$\overrightarrow{V_P}$'s y-coordinate
$\dot{v}$	$\partial v/\partial t$
$\overrightarrow{V_P}$	quadcopter's velocity in relation to frame XYZ
$V^{\pi}(\cdot)$	on-policy value function given a policy π

$V^\star(\cdot)$	optimal value function
w	$\overrightarrow{V_P}$'s z-coordinate
$\dot{w}$	$\partial w/\partial t$
$\overrightarrow{W}$	quadcopter's weight
$\hat{x}$	$\overrightarrow{x_P}$'s X-coordinate
$\dot{x}$	$\overrightarrow{V_P}$'s X-coordinate
$\ddot{x}$	$\overrightarrow{a_P}$'s X-coordinate
$\overrightarrow{x_P}$	quadcopter's position in relation to frame XYZ
x_t	Ornstein–Uhlenbeck process
xyz	body-fixed, non-inertial frame of reference fixed to the center of mass of the quadcopter
XYZ	inertial frame of reference fixed to the center of the simulation "scenario"
y	target value of Q^μ
$\hat{y}$	$\overrightarrow{x_P}$'s Y-coordinate
$\dot{y}$	$\overrightarrow{V_P}$'s Y-coordinate
$\ddot{y}$	$\overrightarrow{a_P}$'s Y-coordinate
$\hat{z}$	$\overrightarrow{x_P}$'s Z-coordinate
$\dot{z}$	$\overrightarrow{V_P}$'s Z-coordinate
$\ddot{z}$	$\overrightarrow{a_P}$'s Z-coordinate
α	actor networks learning rate
β	critic networks learning rate
γ	discount factor ($\gamma \in (0,1)$)
θ^{SV}	pitch angle
θ^{OUP}	Ornstein-Uhlenbeck process parameter
θ^{PP}	set of deep reinforcement learning parameters
$\overrightarrow{\dot{\theta}^{SV}}$	$\partial\theta^{SV}/\partial t$

θ^{Q}	critic network parameters
$\theta^{Q'}$	target critic network parameters
θ^{μ}	actor network parameters
$\theta^{\mu'}$	target actor network parameters
μ	deterministic policy
$\mu(s\|\theta^{\mu})$	actor network
$\mu'(s\|\theta^{\mu'})$	target actor network
μ^{OUP}	Ornstein-Uhlenbeck process parameter (drift term)
π	stochastic policy
$\pi^{\star}$	optimal policy
ρ_0	start-state distribution
σ^{OUP}	Ornstein-Uhlenbeck process parameter
τ	trajectory (sequence of states and actions)
τ^{DDPG}	target update rate
ϕ^{SV}	roll angle
$\overrightarrow{\dot{\phi}^{SV}}$	$\partial\phi^{SV}/\partial t$
ψ^{SV}	yaw angle
$\overrightarrow{\dot{\psi}^{SV}}$	$\partial\psi^{SV}/\partial t$
ω	rotor's rotational speed
$\overrightarrow{\omega}$	quadcopter's angular velocity

CONTENTS

1 INTRODUCTION

1.1 Motivation

Despite the fact that the first unmanned aircraft were built more than a century ago, during the First World War, it was only during the past two decades that drones started to be massively adopted for commercial and military applications. Moreover, the popular and global awareness on this technology's uses and its expansions across different types of industries have flourished mainly in the last few years. (WIKIPEDIA, 2020d) (JOSHI, 2019)

In this context, the global drone market is expected to bring US$ 22.5 billion of revenues in 2020 and to grow with a CAGR of 13.8% during the next five years. (WIRE, 2020) These figures demonstrate very well how important and relevant researches in drone technologies are at this moment in time and serve as one of the motivations for this work.

The number-one sector when it comes to importance and frequency of drone applications is the military one, followed by the commercial and the personal sectors.

In military activities, drones are used for combat missions, research and development, supervision, law enforcement and border control surveillance, having the potential to serve as a powerful tool to the resolution of armed conflicts while replacing human pilots and thus saving human lives.

When it comes to the commercial sector, there is a myriad of possible applications, many still under development or developed and popularized during the last decade. Some of the most prominent usages are listed below: (JOSHI, 2019)

- Shipping and delivery
- Aerial photography and filming
- Disaster management through the delivery of supplies and gathering of information
- Search and rescue operations
- Geographic mapping
- Building safety inspection
- Transmission lines inspection
- Precision crop monitoring
- Unmanned cargo transport
- Weather tracking and forecasting of phenomena such as storms, tornadoes and hurricanes

The main applications in personal activities are leisure-related, and include photography and film-making, gaming and use in competitions such as drone races.

Drone development has historically undergone multiple technology generations, as listed below: (JOSHI, 2019)

- Generation 1: basic remote control aircraft
- Generation 2: static design, fixed camera mount, video recording and still photos, manual piloting control
- Generation 3: static design, two-axis gimbals, HD video, basic safety models, assisted piloting
- Generation 4: transformative designs, three-axis gimbals, 1080P HD video or higher-value instrumentation, improved safety modes, autopilot modes
- Generation 5: transformative designs, 360° gimbals, 4K video or higher-value instrumentation, intelligent piloting modes
- Generation 6: commercial suitability, safety and regulatory standards based design, platform and payload adaptability, automated safety modes, intelligent piloting models and full autonomy, airspace awareness
- Generation 7: complete commercial suitability, fully compliant safety and regulatory standards-based design, platform and payload interchangeability, automated safety modes, enhanced intelligent piloting models and full autonomy, full airspace awareness, auto action (takeoff, landing and mission execution)

Current technologies and developments lie in the fifth, sixth and seventh generations, and important steps have been and are being taken towards the development of enhanced autonomous flight and safety and regulatory standards.

Many of the commercial and some of the personal applications of drones that were listed previously may be very positively impacted by the development of more complex and more reliable autonomous flight operations. In this sense, many types of drones already incorporate self-flying functions such as following modes in which the drone follows predefined waypoints. Yet, these functions still require the action of a pilot to be initiated and stopped, and they do not cover the entire flight duration of a typical drone mission. There is, therefore, an important gap to be filled in the drone industry when it comes to flight and operations automation, and the use of artificial intelligence has been growing in this field. (FEIST, 2020)

One of the vanguard sectors where artificial intelligence is applied to automate drone navigation and control is drone racing. A drone race consists in a competition in which the objective is to navigate a drone through a racecourse within the least amount of time possible. (WIKIPEDIA, 2020a)

The Drone Racing League (DRL) is an international league which organizes races in different locations across the world. Classically, the DRL has been in charge of first-person view (FPV) human-piloted drone races, but since 2019, it also organizes the Artificial

Intelligence Robotic Racing (AIRR), a competition designed for university students and technologists in which the objective is to build an artificial intelligence framework capable of flying high-speed drones through DRL race circuits without any pilot intervention. (DRL, 2019) (WIKIPEDIA, 2020b)

The application of artificial intelligence and, more specifically, neural networks to the problem of autonomous drone racing is better known as ***deep drone racing***.

There are several papers - (KAUFMANN et al., 2018), (MUELLER et al., 2017), (JUNG et al., 2018), (KIM; CHEN, 2015), (ANDERSSON; WZOREK; DOHERTY, 2017), (SMOLYANSKIY et al., 2017), (ROSS et al., 2012), (SADEGHI; LEVINE, 2017), (ZHANG et al., 2016) - related to the subject of deep drone racing, and they exploit different methods and approaches to deal with the objective of generating autonomous aerial vehicle intelligence.

Among these works, there are basically two categories of approach: the first is to use end-to-end machine learning systems and the second is to mix machine learning methods with other state-of-the-art practices, applying, for instance, convolutional neural networks (CNN) for image processing and recognition combined with more traditional path-planning and control systems.

The first category of approach (end-to-end machine learning) is applied in (MUELLER et al., 2017). This paper utilizes a single complex neural network which outputs four stick controls that are to be passed to the drone - throttle, elevator, aileron and rudder - based on a single image of shape 320 x 180 received as input. In this case, the neural network consists of five convolutional and three fully-connected layers, adding up to eight total layers.

The second category of approach is well exemplified by (KAUFMANN et al., 2018). In this paper, a convolutional neural network is combined with a state-of-the-art minimum-jerk-trajectory path planner and with a control scheme which determines the low-level control commands. The CNN receives a single 300 x 200 RGB image as input and outputs a tuple $\{\vec{x}, v\}$, where $\vec{x}$ is a two-dimensional vector that provides the direction to the target (a race gate) in normalized image coordinates and v is a normalized optimal speed to approach the target. The control system then converts the vector $\vec{x}$ to a three-dimensional local reference frame and the normalized speed v to a scalar unnormalized speed. After that, the minimum-jerk-trajectory path planner computes the optimal trajectory for reaching the target and this information is finally translated to low-level executable control commands.

According to (KAUFMANN et al., 2018), it is believed that approaches that mix machine learning and traditional techniques are superior to end-to-end machine learning systems for this type of application because "end-to-end learning of low-level controls requires the network to learn the basic notion of stability from scratch in order to control an unstable platform such as a quadrotor, which leads to high sample complexity, and gives no mathematical guarantees on the platform stability", while the use of classic controllers allows "the network to focus on the main task of robust navigation, which leads to high sample efficiency".

1.2 Objectives

The general objective of this work is to develop a model applicable to the context of deep drone racing.

More specifically, the goal is to build an intelligence which allows for a quadcopter to navigate from any initial position within a bounded region in space to a given (static) target, simulating a drone race environment.

The intelligence must incorporate one or multiple machine learning tools and methods. Considering the resources available for this project, which comprised no drone simulator licenses, the development of an image processing and recognition module for the intelligence was not a viable option. Due to this, the problem of generating a system capable of controlling an aerial vehicle autonomously had to be simplified to one in which the position and velocity of the vehicle in relation to the target are known at all times, and this information is directly passed as inputs to the AI-controller.

Summing up, the main objective of this work is to develop an AI-controller capable of flying a (simulated) quadcopter from an initial arbitrary position - given that this arbitrary position is located within a certain region in space - to a static target representing a typical drone race gate. The target corresponds to a single point in space and the mission is considered as successful if and only if the drone gets close enough to the target (how close is close enough being a settable parameter of the model).

2 BACKGROUND

In this chapter, we will define our problem and objectives more precisely and present a number of approaches and methods that may be applied in order to tackle them. We will also discuss the advantages and disadvantages of each of these methods and introduce the methodology we selected for our mission.

2.1 Problem (re)statement and tools to address it

As stated in Section 1.2, the main objective of this work is to develop an AI-controller capable of flying a (simulated) quadcopter from an initial arbitrary position - given that this arbitrary position is located within a certain region in space - to a static target representing a typical drone race gate.

At this point, as we are trying to select the best approach to achieve our objective, it is essential to define some of our problem's characteristics and specifications in more detail, and that is what we do hereafter:

1. The simulated quadcopter will fly in a continuous space - this is not precisely true, as in a simulation the number of space states that are accessible to the drone is finite, but it is a very good approximation, as the number of possible states is quite large; this means the method we are going to choose to tackle our problem needs to be able to deal with continuous environments.

2. The simulated quadcopter will be modeled in such a way that it will be controlled by three parameters: one related to the total thrust, one related to the pitching moment and one to the rolling moment. There will be no control over the yawing moment, as it is not essential to the guidance of the drone unless tight curves need to be executed. This simplification makes the system simpler and easier for an artificial intelligence to "understand" and learn.

3. Each of the control parameters will vary within a continuous bounded range - again, this is a simplification, as we are dealing with a simulated environment; the selected methodology needs to be applicable to continuous action spaces.

4. The performance of the model will be evaluated according to two criteria:

   a) The task completion rate - how frequently the agent successfully flies the quadcopter to anywhere in the vicinity of the target;

b) The time the quadcopter takes, in average, to reach the target - since we are interested in simulating a drone race environment, it makes sense to optimize how long the agent needs to complete its task.

The key takeaways of this section is that we need a model capable of dealing with both continuous environments and continuous action spaces (as opposed to discrete environments and action spaces) and that is able to learn an optimal replicable "behavior" that optimizes the metrics we are evaluating (task completion rate and time to complete the task).

As we will see later, some methods categorized as reinforcement learning - which is a type of machine learning - can handle this type of problem very well.

2.2 Context and background

Before we dive into the methods that are suitable for tackling our problem, let us understand the general context and areas of which these methods are part and explain some concepts and definitions in a more formal way. This section is specially oriented to beginners in artificial intelligence and data science, and may be skipped by anyone who is already familiarized with these disciplines.

As mentioned earlier, one of the main requirements of this work is to bring one or multiple artificial intelligence methods into our solution to the autonomous drone racing problem.

The artificial intelligence, as a field of studies, is linked to several other disciplines that have been popularized during the last decade. Among these adjacent disciplines, the most important ones are data science and data mining, machine learning and deep learning. All these disciplines are somehow related to the methods we employ in this work, so the next subsections are dedicated to explaining what each of them is about and how they are connected to this project.

This section is largely based on information provided by (KULINA et al., 2020).

Figure 1 shows a Venn diagram illustrating the relationships between the four areas.

2.2.1 Data science and data mining

Data science is the scientific discipline that studies everything related to data, from technical aspects such as data acquisition and data storage to more analytical ones such as data interpretation, visualization and decision-making based on data.

Many artificial intelligence and machine learning methods may be categorized as applications of data science, and that is why data science is a relevant topic for this project.

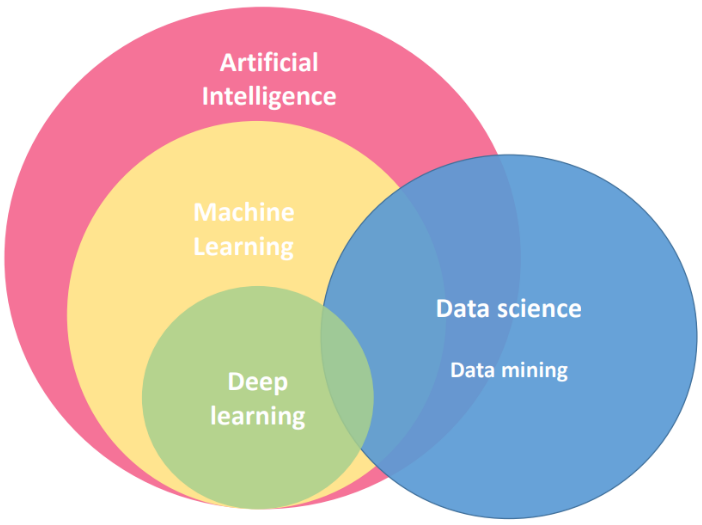


Figure 1 – Venn diagram illustrating the relationships between data science, artificial intelligence, machine learning and deep learning. (KULINA et al., 2020)

Data mining consists in extracting insights from data and thus may be categorized as one of the many branches that derive from data science.

2.2.2 Artificial intelligence

Artificial intelligence is the field of studies that aims to develop human-like behaviors and reasoning in machines. In other words, artificial intelligence methods attempt to provide machines with qualities and competences that are inherent to human-beings, such as the abilities of understanding, learning, reasoning and self-correction from data, as well as the perception and high-level processing of visual, auditory and tactile information.

Artificial intelligence has multiple different branches and Figure 2 shows one way of summarizing them, as suggested by (BOHNHOFF, 2019).

2.2.3 Machine learning

Machine learning is a branch of artificial intelligence which aims at developing algorithms that are capable of learning from historical data and improving its performance solely by the means of experience (i.e., training).

Machine learning and data mining have, under a certain perspective, the same primary objective: to use data in order to grasp the optimal behavior/response/decision/estimation for any given problem. They differ, however, in what means they apply in order to do that: while data mining is based, for the most part, on empirical observations, made by a human being who applies some methods and tools in order to do that, machine learning employs algorithms which are able to learn autonomously from the data, without the need of a human being directly analyzing the data. As the name suggests, in machine learning,

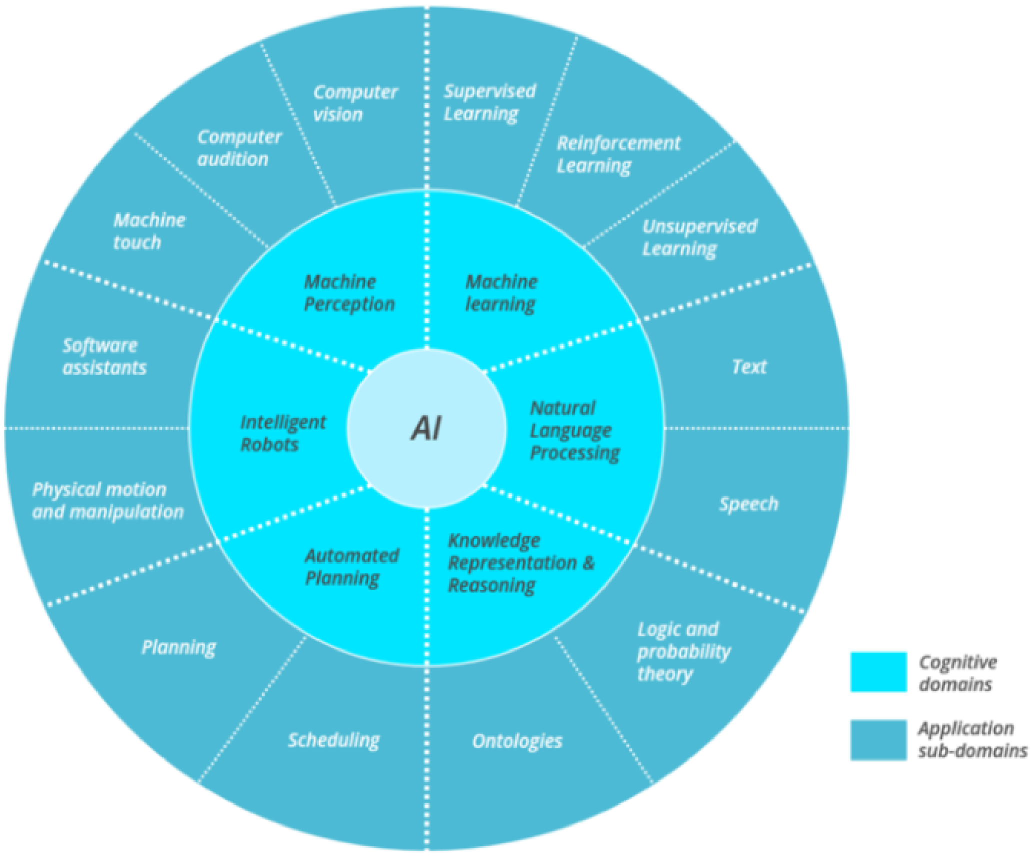


Figure 2 – Artificial intelligence domains summarized, as proposed by (BOHNHOFF, 2019).

the machine is expected to learn on its own - humans are only necessary for correcting machine deviations every now and then, when things get off track.

One of the main advantages of machine learning over more traditional approaches is that machine learning methods need little to no specific information about the model it is trying to understand; in fact, the model is generally learned by the data processing alone. This is convenient because for really complex models, for example, a lot of programming work is saved, as the machine learning approach doesn't require any implementation regarding the model of the phenomenon being analyzed - and this would not be true for traditional approaches. Besides the load and complexity of work needed, the fact that no specific model implementation is required decreases the likelihood of modeling errors. Taking this example to an even more extreme level, machine learning methods can, in principle, learn patterns that humans cannot yet formalize, describe or understand. This fact portrays how powerful this type of approach really is.

Some of the disadvantages of machine learning is that it may require representative amounts and quality of data, as well as good computational processing and storing capabilities.

Machine learning composes the core of the methods used in this work, so it will be discussed in more detail later in the text.

2.2.4 Deep learning

Deep learning is a subset of machine learning. It is characterized by the use of *neural networks* to learn representations of data with multiple levels of abstraction.

Neural networks are multi-layer data-processing networks that apply non-linear transformations in order to generate numeric output from structured variable inputs.

The advantage of deep learning over other machine learning techniques is that it has a great capacity of extracting high-level features from complex data, which is not always the case when it comes to other techniques.

A major disadvantage is that neural networks behave more or less like black boxes: it is difficult to understand what happens to input data as it traverses the network and eventually become an output. Hand in hand with that, neural networks require fine tuning of hyper-parameters, and finding a network layout and hyper-parameters setting that result in optimal or sometimes even acceptable results may be a hard and time-consuming task.

The machine learning method selected in this work is a deep learning method.

2.3 Types of machine learning

As Figure 2 shows, artificial intelligence has multiple domains. It is easy to deduce that, among these domains, only machine learning and machine perception may bring critical contributions to the autonomous drone racing problem.

As stated in Section 1.1, machine perception, mainly through the field of computer vision, may act as a crucial part of the solution to the problem, providing target orientations as the drone navigates through the racecourse. Even though there are alternative solutions that do not necessarily rely on artificial intelligence to accomplish the same task (localization and orientation), machine perception methods present some robust advantages over more traditional solutions, as argued by (KAUFMANN et al., 2018).

Despite its importance, machine perception is not a major topic on this work, as already mentioned. Due to that, we will focus on machine learning concepts and approaches from now on.

There are basically four types of machine learning: supervised, unsupervised, semi-supervised and reinforcement learning. (BURKOV, 2019) These four types are introduced in the subsections that follow.

2.3.1 Supervised learning

The general objective of a supervised learning model is to predict an output given an input vector. The input vector of such a model encapsulates one or more features that are available to the observer. These features are generally numeric values and translate into mathematical language the characteristics that are associated with some object the model tries to understand - a person, a thing, a phenomenon, a situation, etc. The output of the model is any information that needs to be predicted - it could be a classification of the object under analysis (in which case we would have a *classification model*), a numeric value (in this case the model would be said a *regression model*) or a vector combining multiple classes and/or numeric values.

To learn how to make this type of prediction, supervised models require labeled data sets - large ones more often than not. These data sets must contain associations between input information and output information that are known beforehand. It is based on these associations that the model notices patterns between input and output data and ultimately learns how to make right predictions. The term *labeled* in *labeled data sets* means that, for every input vector available in the data set, there will be an output information associated to it, and this output information is what is called the *label.* In other words, a labeled data set is basically a collection of pairs $\{(x_i, y_i)\}$, where the x_i are feature vectors (the input of the model, as described above) and y_i are the labels associated to them.

The process in which the model "reads" the data and gradually learns from it is called *training.*

A typical example of classification supervised learning model is tumor classification between benign and malignant: based on tumor characteristics such as diameter and form, this model is capable of predicting, with a good level of accuracy, the type of the tumor.

An example of regression supervised learning application is the pricing of a product: based on information like demand, units on stock and competitors' prices, this type of model can determine the optimal price for that product.

2.3.2 Unsupervised learning

Unsupervised learning models, as supervised ones, require data sets to be trained. One of the main differences between these two types of learning is that unsupervised models use unlabeled data sets, as opposed to labeled ones. That implies that unsupervised models cannot make definite predictions as supervised ones do. That might sound a bit strange at first, but even so this type of learning can provide very useful insights based on the available data set.

Clustering, **dimensionality reduction** and **outlier detection** are some applications of unsupervised learning. **Clustering** provides segmentations of the available data based on their features. **Dimensionality reduction** returns output vectors that have fewer dimensions than the input ones, retaining the essential information associated to them. **Outlier detection** consists in the identification of input elements that have features that are very different from those typical of other elements.

2.3.3 Semi-supervised learning

Semi-supervised learning mixes some aspects of both supervised and unsupervised learning, starting by the fact that it requires mixed data sets - partly labeled, partly unlabeled. Typically, unlabeled data form the majority of this type of data sets.

Apart from that, semi-supervised learning models function in a pretty similar way to supervised learning ones. They try to predict an information based on a given input. The advantage of semi-supervised learning is that it allows either the data set to contain fewer labeled data or the data set to contain a larger number of entries overall. This is a positive quality because labeled data are the most difficult to get or to produce, so if training with a lesser amount of them can still provide good results, this might be a good option to consider.

2.3.4 Reinforcement learning

Unlike the other three types of machine learning, reinforcement learning (RL) models do not require any data set in order to be trained. The data is generated and evaluated by the model itself, and this makes this kind of approach attractive for countless applications.

Reinforcement learning is particularly interesting for problems where decision making is sequential and should target long-term goals, like in video games, resource management, logistics, robotics and competitions.

An abstract description of how reinforcement learning works is the following: the machine (also called the *agent* in this context) "lives" in an *environment* and is aware of its *state*, which changes dynamically as the simulation/phenomenon occurs. The state may change for two different reasons: either due to environment-related causes or due to actions taken by the agent itself. In any given state, the agent can execute a finite number of types of *action*, and different outcomes can be expected if different actions are taken. Even though the *types of action* available for the agent to choose is necessarily finite, the number of possible *actions* may be considered infinite, depending on the reinforcement learning model. This is true because some types of action may exist in continuous spectra, in which case the number of possible ways to execute those types of action could be considered infinite.

Every time the agent chooses a set of actions - which is equivalent to every time step of the simulation/application of the model -, it receives a *reward*. This reward varies according to the set of actions taken, and the objective of the agent is to maximize the total reward on the long run. By doing that, the agent learns a *policy*: a function that takes the current state of the agent as an input and outputs a set of actions that it believes will maximize the total reward on the long run. The total, or cumulative reward is formally called *return*. So formally, agents try to optimize their policies as to maximize their return over time.

Figure 3 illustrates some of the concepts presented above and a few basic interactions they share.

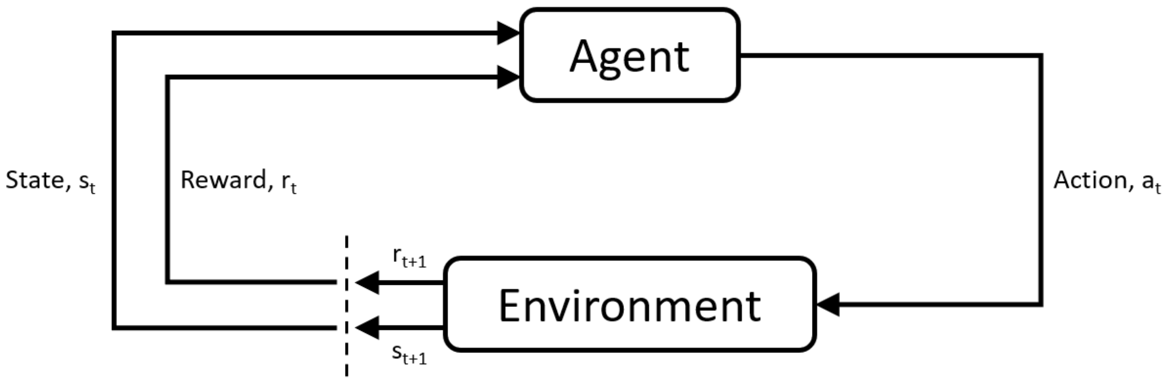


Figure 3 – Some basic concepts in reinforcement learning and a few interactions among them.

Reinforcement learning will be presented in more detail in the sections that follow.

2.3.5 Types of machine learning and the deep drone racing problem

Of the four types of machine learning, two are suitable to the deep drone racing problem: supervised learning and reinforcement learning.

Supervised learning has been applied by (SMOLYANSKIY et al., 2017) and (ROSS et al., 2012), but it requires the generation of a data set, which is not an easy task to do and demands some resources that were not available for our project.

Reinforcement learning was used by (SADEGHI; LEVINE, 2017) and (ZHANG et al., 2016), and one of its greatest advantages over supervised learning is that it does not need the use of data sets. It also has disadvantages over supervised learning. One of them is that it requires the engineering of a reward function, and the success of the model in this case strongly depends on how well this reward function is chosen. In spite of that, due to the limitations of our project, reinforcement learning was one of the only available options, so we stood by it.

(MUELLER et al., 2017) and (ANDERSSON; WZOREK; DOHERTY, 2017) used approaches which mixed supervised and reinforcement learning, and this is a third available option when dealing with deep drone racing.

2.4 Reinforcement learning

Since the methods we apply to achieve the goals of this project are reinforcement learning methods, a more detailed overview of the concepts and main classes and methods of this type of machine learning is desirable. This is what we do in this section.

OpenAI has a great introduction on this subject, and the information provided below is mostly based on their content. (ACHIAM, 2018c)

2.4.1 Key concepts and terminology

In Subsection 2.3.4 the basic RL concepts - agent, environment, state, action, reward, policy and return - have been introduced. Now we present some more advanced ideas:

- An *observation* is a partial description of a state. Both states and observations are generally represented through vectors, matrices or higher-order tensors. When the agent is able to observe the complete state of an environment, the environment is said to be *fully observed*; when the agent can observe only some of the state's features, it is said that the environment is *partially observed*.
- The set of all valid actions an agent can take in a given environment is called the *action space*. Actions spaces can either be discrete - when there is a finite number of actions the agent can take - or continuous - when the agent can choose an action represented by a real number. This distinction between discrete and continuous action spaces is very important, as many RL methods only work for one of them. As mentioned in Section 2.1, the deep drone racing problem deals with continuous action spaces.
- As already mentioned, a *policy* is the function an agent uses to decide what actions to take. Policies can be deterministic, in which case they are usually represented by μ, as follows:

  $$a_t = \mu(s_t) \tag{2.1}$$

  or stochastic, in which case they are often denoted by π:

  $$a_t \sim \pi(\cdot|s_t) \tag{2.2}$$

  Here, a_t represents the action chosen by the policy and s_t represents the input state (or observation).

  In deep reinforcement learning, policies are said to be *parameterized*, as they depend on a set of parameters (like neural networks weights and biases) that can be adjusted over time. The set of parameters that affect this type of policy is usually denoted by θ^{PP}. The superscript PP stands for *policy parameters*. In this case, Equations 2.1 and 2.2 become:

$$a_t = \mu_{\theta^{PP}}(s_t) \tag{2.3}$$

$$a_t \sim \pi_{\theta^{PP}}(\cdot|s_t) \tag{2.4}$$

In this work, we are going to be concerned only with deterministic policies.

- There are two important types of strategy in reinforcement learning that should be balanced during training: *exploration* and *exploitation*. *Exploration* is any strategy the agent applies to discover actions that lead to victory and eventually outperform the typical actions that are output by the current policy. By exploring, the agent avoids getting stuck in local optimum policies, searching for alternative solutions that may generate better-suited behaviors. *Exploitation* corresponds to the application of an already discovered strategy in order to refine it and get the most out of it. Both exploration and exploitation are important to improve the policy constantly. Exploration is most important at the beginning of the training, when the agent still has not made many discoveries about the environment. As the training evolves, exploitation becomes gradually more important.
- A *trajectory* or *episode* is a sequence of states and actions and is usually denoted by τ:

$$\tau = (s_0, a_0, s_1, a_1, ...) \tag{2.5}$$

A transition between two moments in a trajectory is called a *state transition*. State transitions are governed by the laws of the environment and depend uniquely on the current state and on the action taken. The laws of the environment, like policies, can be deterministic or stochastic. For deterministic laws, we have the following relation:

$$s_{t+1} = f(s_t, a_t) \tag{2.6}$$

For stochastic laws, the relation is the following:

$$s_{t+1} \sim P(\cdot|s_t, a_t) \tag{2.7}$$

The initial state (s_0) is generally randomly sampled from the *start-state distribution*, denoted by ρ_0:

$$s_0 \sim \rho_0(\cdot) \tag{2.8}$$

- A *reward function* R is a function that returns the reward given the current and next state and the action taken:

$$r_t = R(s_t, a_t, s_{t+1}) \tag{2.9}$$

- As already mentioned, but now in a more formal way, the *return* is the cumulative reward over a trajectory, and is denoted by $R(\tau)$. There are two main types of return:

  - The *finite-horizon undiscounted return*, which is the sum of the rewards obtained in a finite number of steps:
$$R(\tau) = \sum_{t=0}^{T} r_t \tag{2.10}$$
  - The *infinite-horizon discounted return*, which is the sum of all the rewards ever obtained by the agent, discounted by how far off in the future they will be obtained. In this case, a *discount factor* $\gamma \in (0,1)$ is applied:
$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t \tag{2.11}$$

- We already know that the goal of an agent in RL is to choose a policy that maximizes the expected return. Expressing this in a mathematical way and supposing that both the environment and the policy are stochastic, we have:
$$\pi^\star = \arg\max_{\pi} J(\pi), \tag{2.12}$$
where $\pi^\star$ is the optimal policy and $J(\pi)$ is the expected return given a policy π. The expected value $J(\pi)$ can be calculated as:
$$J(\pi) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)] = \int_{\tau} \mathbb{P}(\tau|\pi)R(\tau)d\tau \tag{2.13}$$
$\mathbb{P}(\tau|\pi)$ is the probability of occurrence of a given trajectory τ with T steps and can be calculated as follows:
$$\mathbb{P}(\tau|\pi) = \rho_0(s_0)\prod_{t=0}^{T-1} P(s_{t+1}|s_t,a_t)\pi(a_t|s_t) \tag{2.14}$$

- A *value function* is a function that outputs the expected return for an agent with any given initial state s or initial state-action pair (s,a) and that acts according to a particular policy forever after the initial state or initial state and action.
There are four main types of value functions:
  1. The *on-policy value function*, $V^\pi(s)$, which gives the expected return when the agent starts in state s and always acts according to policy π:
$$V^\pi(s) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s] \tag{2.15}$$
  2. The *on-policy action-value function*, $Q^\pi(s,a)$, which gives the expected return when the agent starts in state s, takes an arbitrary initial action a - not necessarily from the policy - and then afterwards always acts according to policy π:
$$Q^\pi(s,a) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s, a_0 = a] \tag{2.16}$$
  3. The *optimal value function*, $V^\star(s)$, which gives the expected return when the agent starts in state s and always acts according to the optimal policy in the environment:
$$V^\star(s) = \max_{\pi} \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s] \tag{2.17}$$

4. The *optimal action-value function*, $Q^\star(s,a)$, which gives the expected return when the agent starts in state s, takes an arbitrary initial action a and then afterwards always acts according to the optimal policy in the environment:

$$Q^\star(s,a) = \max_\pi \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s, a_0 = a] \tag{2.18}$$

The following relations hold between value functions and action-value functions:

$$V^\pi(s) = \mathop{\mathbb{E}}_{a\sim\pi}[Q^\pi(s,a)] \tag{2.19}$$

$$V^\star(s) = \max_a Q^\star(s,a) \tag{2.20}$$

- Given an optimal action-value function $Q_\star(s,a)$, the optimal initial action, $a^\star$, may be determined by the following relation:

$$a^\star(s) = \arg\max_a Q^\star(s,a) \tag{2.21}$$

- The *Bellman equations* are very important relations in reinforcement learning that formalize the following basic idea:

  The value of a starting state is the expected reward for that state, plus the value of the next state.

  The Bellman equations for on-policy value functions and discounted returns are:

$$V^\pi(s) = \mathop{\mathbb{E}}_{\substack{a\sim\pi \\ s'\sim P}}[r(s,a) + \gamma V^\pi(s')] \tag{2.22}$$

$$Q^\pi(s,a) = \mathop{\mathbb{E}}_{s'\sim P}\left[r(s,a) + \gamma \mathop{\mathbb{E}}_{a'\sim\pi}[Q^\pi(s',a')]\right] \tag{2.23}$$

  For optimal value functions, the Bellman equations are:

$$V^\star(s) = \max_a \mathop{\mathbb{E}}_{s'\sim P}[r(s,a) + \gamma V^\star(s')] \tag{2.24}$$

$$Q^\star(s,a) = \mathop{\mathbb{E}}_{s'\sim P}\left[r(s,a) + \gamma \max_{a'} Q^\star(s',a')\right] \tag{2.25}$$

  In the four previous equations, s' denotes the next state and a' denotes the next action.

- The *advantage function* $A^\pi(s,a)$ of an action is a function that describes how much better it is to take an action a in state s than following a given policy π in that state, assuming policy π will be followed forever after in both cases. Mathematically, it is given by:

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s) \tag{2.26}$$

- A *Markov decision process* (MDP) is a process that obeys the *Markov property*:

  Transitions only depend on the most recent state and action, and no prior history.

Reinforcement learning models can be treated as Markov decision processes and formalized by the use of a 5-tuple of the form $\langle S, A, R, P, \rho_0 \rangle$, where:

- S is the set of all valid states,
- A is the set of all valid actions,
- $R : S \times A \times S \to \mathbb{R}$ is the reward function, with $r_t = R(s_t, a_t, s_{t+1})$,
- $P : S \times A \to \mathcal{P}(S)$ is the transition probability function, with $P(s'|s,a)$ being the probability of a transition from state s and action a to state s',
- ρ_0 is the start-state distribution.

2.4.2 Reinforcement learning algorithms

Figure 4 shows a simplified, non-exhaustive taxonomy of algorithms in modern reinforcement learning.

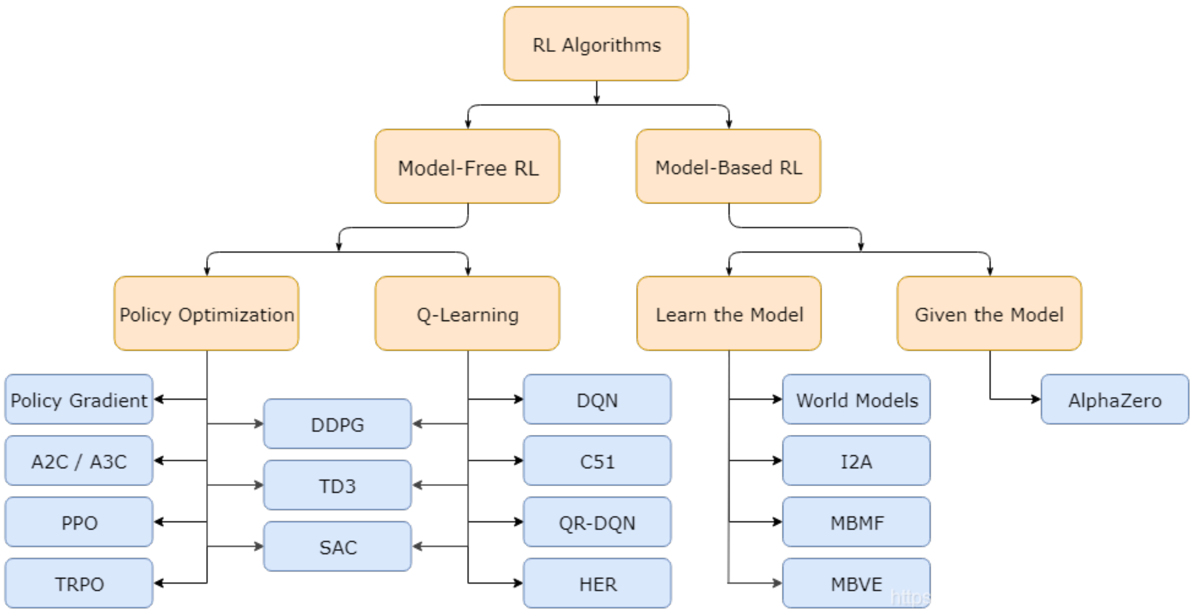


Figure 4 – Non-exhaustive taxonomy tree of modern RL algorithms. (ACHIAM, 2018b)

We dedicate this subsection to the analysis of the different types of RL algorithms currently available, breaking down the taxonomy tree shown in Figure 4.

2.4.2.1 Model-free and model-based algorithms

As the taxonomy tree in Figure 4 shows, one of the most important classification criteria in reinforcement learning is the distinction between *model-free* and *model-based* algorithms.

Model-based methods are those in which the agent knows or learns a model of the environment, i.e., a function that predicts state transitions and rewards. One of the main advantages of model-based methods is that they allow the agent to forecast the results and rewards derived from many possible actions and then to make decisions based on these predictions. AlphaZero is one of the most widespread methods of this category.

When a precise description of the environment is available and can be provided to the agent, model-based algorithms will frequently give superior results to that provided by model-free ones. When a model isn't easily available, on the other hand, learning it from experience can be a very challenging thing to do, and will often result in biases that can jeopardize the performance of the method. In this scenario, model-free algorithms usually outperform model-based ones. Model-free algorithms are also easier to implement and to tune, and these are some of the main advantages of applying them.

2.4.2.2 What to learn

Another classification criterion for RL algorithms is what is learned by the agent. The main possibilities are:

- policies, either stochastic or deterministic,
- action-value functions (Q-functions),
- value functions,
- environment models.

Model-free and model-based algorithms use different approaches when it comes to the choice of what to learn, and they are presented in more detail hereafter.

2.4.2.2.1 What to learn in model-free algorithms

In model-free algorithms, agents usually learn the policy (*policy optimization*), action-value functions (*Q-learning*) or a mixture of both.

In **policy optimization**, the parameters θ^{PP} of the policy $\pi_\theta^{PP}(a|s)$ are optimized either directly, by gradient ascent on the performance objective - the expected return of the policy, $J(\pi_\theta^{PP})$ -, or indirectly, by maximizing local approximations of $J(\pi_\theta^{PP})$. In most cases, an *on-policy* optimization is done - only data collected while acting according to the most recent version of the policy is applied to update it. **Policy gradient**, **A2C/A3C**, **PPO** and **TRPO** are some examples of policy optimization algorithms. The primary advantage of policy optimization methods over Q-learning ones is that they seek to directly optimize the policy, which is the final interest for any RL algorithm. This direct approach makes them more stable and reliable.

In **Q-learning** methods, the agent learns an approximator $Q_\theta^{PP}(s,a)$ for the optimal action-value function, $Q^\star(s,a)$, often using an objective function based on the Bellman equation. Q-learning almost always applies *off-policy* optimization - data collected at any point during training can be applied to update the model, regardless of how the agent was choosing to explore the environment when the data was obtained -, and this is one of its main advantages: the fact that it can reuse data more effectively makes it substantially more sample efficient. The optimal policy in Q-learning methods is indirectly inferred by applying a relation similar to that expressed in Equation 2.21:

$$a(s) = \arg\max_a Q_\theta^{PP}(s, a) \tag{2.27}$$

DQN, **C51**, **QR-DQN** and **HER** are some examples of Q-learning methods.

There are RL algorithms which incorporate characteristics from **both policy optimization and Q-learning**, and they are able to trade-off well between the advantages and disadvantages of each these categories. **DDPG**, **TD3** and **SAC** are examples of this mixed category.

As mentioned before, DDPG is the method we will apply in order to solve the deep drone racing problem. It learns a deterministic policy and a Q-function simultaneously, using each one to improve the other. We will describe this algorithm in more detail later.

2.4.2.2.2 What to learn in model-based algorithms

Model-based RL doesn't have such clear and simple criteria to classify its algorithms according to what is learned as model-free RL does. There are numerous families of methods. Only a few of them are introduced below.

Pure planning is the most approach to model-based RL. It never represents the policy explicitly, and instead, uses techniques such as model-predictive control (MPC) to select actions. The **MBMF** is an example of algorithm of this family.

Expert iteration is an extension of pure planning and it involves using and learning an explicit representation of the policy. **AlphaZero** is an example of this family.

Some model-based algorithms **embed planning loops into policies** as a subroutine while training the output of the policy with any standard model-free algorithm. **I2A** is an example of this family.

2.5 DDPG

In this section, we present how our selected method to solve the deep drone racing problem, the Deep Deterministic Policy Gradient (DDPG), works and is implemented. The main sources of information are the paper which introduced the method (LILLICRAP et al., 2019) and (YOON, 2019).

The selection of DDPG has been made based on the following facts:

- DDPG is compatible with all our problem's characteristics and specifications, listed in Section 2.1 - continuous state space, three-dimensional, continuous action space

and performance optimization based on task completion rate and average time to complete tasks;
- it has a reasonably extensive documentation available online, with introduction essays, articles, tutorials and implementation videos associated to the method;
- it has successfully been applied to multiple physics problems with specifications of the same type as ours.

We list hereafter some of the main characteristics of the method:

- DDPG is a model-free, off-policy algorithm, mixing policy optimization and Q-learning techniques; it requires the policy to be deterministic.
- The algorithm combines features from two previously existing methods: Deep Q Network (DQN) and Deterministic Policy Gradient (DPG); DDPG combines complementary strengths of these methods: like DPG and unlike DQN, it can be applied to high-dimensional, continuous action spaces, and like DQN and unlike DPG, it utilizes neural networks as function approximators.
- Two pairs of actor-critic neural networks are employed; this feature will be described in more detail later.
- Using the exact same learning algorithm, network architecture and hyper-parameters, the methods is capable of solving a multitude of physics tasks; this shows that the method is simple to operate, as, once implemented, there is usually no need to spend time tuning it.
- DDPG can learn policies "end-to-end", i.e., directly from raw pixel inputs.
- In (LILLICRAP et al., 2019), DDPG results in multiple tasks are found to be competitive and sometimes even outperform the results of a model-based method applied to the same tasks.

2.5.1 Background

The background behind DDPG is composed essentially by the concepts presented in Subsection 2.4.1, complemented with some additional information provided in the current subsection.

2.5.1.1 Loss function and target network

DDPG applies the Bellman equation for on-policy action-value functions (Equation 2.23), which we recall here:

$$Q^{\pi}(s,a) = \mathop{\mathbb{E}}_{s'\sim P}\left[r(s,a) + \gamma \mathop{\mathbb{E}}_{a'\sim\pi}[Q^{\pi}(s',a')]\right] \tag{2.28}$$

As mentioned before, in DDPG the policy is required to be deterministic, so we replace the stochastic policy π by a deterministic policy μ. In this case, the expected value

$\mathbb{E}_{a'\sim\pi}[Q^{\pi}(s',a')]$ in Equation 2.28 can be replaced by the Q-value argument itself, so the following equation holds:

$$Q^{\mu}(s,a) = \underset{s'\sim P}{\mathbb{E}}[r(s,a) + \gamma Q^{\mu}(s',\mu(s'))] \tag{2.29}$$

In Equation 2.29, the expected value depends only on the state-transition probability distribution P, given a policy μ. That is the reason why DDPG is an off-policy algorithm: since the expected value depends only on P, it is possible to use transitions achieved by policies different than μ - say an old policy β - to learn the value Q^{μ}.

The value $Q^{\mu}(s,a)$ is estimated by a neural network. Here, it is important to have in mind that the neural network acts in this case as a function estimator - and it is indeed often referred to as a *(non-linear) function estimator*. As already mentioned in Subsection 2.4.1, the outputs of any neural network are controlled by a set of parameters here denoted by θ^{PP}. In addition, as function *estimators*, the values output by the neural network are not the exact Q^{μ}, but rather approximations for Q^{μ}. The quality of these approximations is obviously dependent on θ^{PP} and can thus be maximized by finding an optimum value for θ^{PP}. To do so, the loss function $L(\theta^{PP})$ defined below has to be minimized:

$$L(\theta^{PP}) = \underset{\substack{s\sim P\\ a\sim\beta}}{\mathbb{E}}\left[\left(Q(s,a|\theta^{PP}) - y\right)^2\right], \tag{2.30}$$

In Equation 2.30, y represents the exact value of Q^{μ}. Since there is no way to know the exact value of Q^{μ} - otherwise it wouldn't be necessary to use a neural network to estimate it -, it must be somehow estimated. To do so, a second neural network, called *target network*, is employed.

In short, y is estimated by the target network, and it is given by the following expression:

$$y = r(s,a) + \gamma Q(s',\mu(s')|\theta^{PP}) \tag{2.31}$$

2.5.1.2 Actor-critic networks

DDPG uses four neural networks: a pair of "original" actor-critic and a pair of target actor-critic networks.

The basic idea behind the pair actor-critic is that the actor behaves as the policy of the network, while the critic is a function approximator that estimates the Q-value relative to the action chosen by the actor in a given state.

Putting it in a more detailed way, at each time step the actor receives an observation as input and determines, based on its current parameters, which set of actions the agent should take to maximize the return.

The performance of the actor in choosing an action is then evaluated by the critic, which receives the observation plus the set of actions selected by the actor and outputs the Q value corresponding to that action.

The parameters of the actor network are then updated according to the evaluation made by the critic, following a process that will be described later.

The four networks used in DDPG are denoted according to the functions they approximate, resulting in the following notation:

- The actor network is denoted by $\mu(s|\theta^{\mu})$,
- the critic network is denoted by $Q(s,a|\theta^{Q})$,
- the target actor network is denoted by $\mu'(s|\theta^{\mu'})$,
- and the target critic network is denoted by $Q'(s,a|\theta^{Q'})$,

The variables θ^{μ}, θ^{Q}, $\theta^{\mu'}$ and $\theta^{Q'}$ represent the set of parameters of each network.

2.5.1.3 Actor and critic networks updates

The critic network is updated by minimizing the loss function defined in Equation 2.30, where $Q(s,a|\theta^{PP})$ is obtained from the critic network and y is calculated according to Equation 2.31, with data coming from the target critic network.

When it comes to the actor, the objective is to maximize the expected return, given by Equation 2.13, which, in the case of the actor network, is equivalent to the following:

$$J(\mu) = \mathop{\mathbb{E}}_{s\sim P}[Q(s,a|\theta^{Q})|_{s=s_t,a=\mu(s_t|\theta^{\mu})}] \tag{2.32}$$

To find - or at least get closer to - the parameters of the actor network, θ^{μ}, that issue the maximum value of $J(\mu)$, we can take the gradient of $J(\mu)$ in relation to θ^{μ} and update θ^{μ} by adding this gradient multiplied by a given factor to the old θ^{μ}, as in a gradient descent method. Doing so and applying the chain rule, we obtain:

$$\nabla_{\theta^{\mu}} J(\mu) = \mathop{\mathbb{E}}_{s\sim P}[(\nabla_a Q(s,a|\theta^{Q})|_{s=s_t,a=\mu(s_t)})(\nabla_{\theta^{\mu}}\mu(s|\theta^{\mu})|_{s=s_t})] \tag{2.33}$$

The gradient in Equation 2.33 is called the *policy gradient*, the gradient of the policy's performance which appears in the names of the methods DPG and DDPG.

2.5.1.4 Target networks: generation and updates

We already know that the target networks are used to estimate the actual values of Q. Now we will describe how these networks are generated and updated, as well as why they work.

Initially, when the method is started, the target networks are exact copies of the actor and critic networks. After that, they can be seen as time-delayed copies of the "original" networks - the target parameters are updated by slowly tracking the "original" networks:

$$\theta'_{t+1} \leftarrow \tau^{DDPG}\theta_{t+1} + (1 - \tau^{DDPG})\theta'_t, \tag{2.34}$$

where θ' is the set of target network parameters, θ is the set of "original" network parameters and $\tau^{DDPG} << 1$. We will call τ^{DDPG} the *target update rate.*

The use of target networks and the fact that their parameters updates are "softened" by τ^{DDPG} greatly improves the stability of DDPG, and this is an important contribution the method brought to deterministic policy gradient.

2.5.1.5 Replay buffer

A replay buffer is a finite-sized cache where transitions information sampled from the environment is stored. For each transition, the tuple (s_t, a_t, r_t, s_{t+1}) is saved in the buffer. Then, at each time step, the actor and critic are updated by randomly sampling a mini-batch of transitions from the cache.

This procedure is important because in optimization tasks it is usually necessary that the data be independently distributed. Without the use of a replay buffer, this independent distribution requirement would not be satisfied, as DDPG deals with sequential decision processes and thus with data that is not independent from each other. By storing data in a large replay buffer - which doesn't present any difficulty, since DDPG is an off-policy method- and taking training samples in a random way, this issue is overcome.

When the replay buffer gets full, the oldest samples start being discarded to give place to newer ones.

2.5.1.6 Exploration

In reinforcement learning, for continuous action spaces, exploration is usually accomplished by adding noise to the action selected by the agent, as the following equation shows:

$$\hat{\mu}(s_t) = \mu(s_t|\theta^{\mu}_t) + \mathcal{N} \tag{2.35}$$

In this equation, $\hat{\mu}$ is the exploration policy and $\mathcal{N}$ represents the noise. The type of noise can be selected according to the characteristics of the environment being simulated. (LILLICRAP et al., 2019) uses an Ornstein-Uhlenbeck process (UHLENBECK; ORNSTEIN, 1930) to generate temporally correlated noise, which is a very good option for physical problems. The Ornstein-Uhlenbeck process models the velocity of a Brownian particle with friction, which results in temporally correlated values centered around 0.

The Ornstein–Uhlenbeck process, denoted by x_t, is defined by the following stochastic differential equation: (WIKIPEDIA, 2020e) (TABOR, 2019)

$$dx_t = \theta^{OUP}(\mu^{OUP} - x_t)dt + \sigma^{OUP}\mathcal{N}(0, dt), \tag{2.36}$$

where $\theta^{OUP} > 0$ and $\sigma^{OUP} > 0$ are two parameters, μ^{OUP} is a constant drift term, t is time and $\mathcal{N}(0, dt)$ represents a normal distribution of mean zero and variance dt. Both x_t and μ^{OUP} are vectors whose size is equal to the number of dimensions existing in the action space. The superscript OUP stands for *Ornstein–Uhlenbeck process*.

The implementation of Equation 2.36 in iterative environments - like those where DDPG is applied - can be done by solving the differential equation via finite differences. By doing so, the following equation is obtained: (TABOR, 2019)

$$x_{t+1} = x_t + \theta^{OUP}(\mu^{OUP} - x_t)dt + \sigma^{OUP}\mathcal{N}(0, dt) \tag{2.37}$$

In this case, x_t represents the current noise, x_{t+1} represents the next noise and dt should be seen as the simulation time step rather than a time derivative. Equation 2.37 can be directly applied in order to generate the necessary noise for DDPG exploration.

Some recent results suggest that other kinds of noise, even uncorrelated ones, such as Gaussian noise, also do a good job in DDPG policy exploration. (ACHIAM, 2018a)

In order to balance exploration and exploitation during the duration of the training, it is possible to scale down the magnitude of the noise by multiplying it by a decay factor. (ACHIAM, 2018a)

2.5.1.7 Batch normalization

The data observed by the agent within an environment and/or across environments may have different units and orders of magnitude. This can make it difficult for the neural network to learn effectively and would require repeating the process of tuning the method's hyper-parameters for every new environment being simulated.

To avoid this type of problem, the observation vector is always normalized or standardized before being fed to any neural network. In (LILLICRAP et al., 2019), this is done by adapting a technique called *batch normalization*, from (IOFFE; SZEGEDY, 2015). This technique normalizes each dimension across the samples in a mini-batch to have zero mean and unit variance. In addition, it maintains a running average of the mean and variance to use for normalization during testing.

(LILLICRAP et al., 2019) applies batch normalization on the state input and all layers of the "original" and target actor networks, as well as all layers of the "original" and target critic networks prior to the action input. By doing so, DDPG is able to learn effectively across many different tasks with differing types of units, without the need to manually tune any hyper-parameters when the environment is changed.

2.5.2 Method

Now that all the necessary concepts and terminology have been presented, we can put them all together by showing how DDPG works in practice. We will split this presentation into two parts, the first being about the algorithm and its general structure and the second dedicated to informing the network architecture and hyper-parameters selected and suggested by (LILLICRAP et al., 2019).

2.5.2.1 Algorithm

The algorithm below is taken from the paper which originally introduced DDPG. (LILLICRAP et al., 2019)

Algorithm 1: Deep Deterministic Policy Gradient

```
Initialize critic $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ networks with random parameters $\theta^Q$ and $\theta^\mu$
Initialize target networks $Q'$ and $\mu'$ with parameters $\theta^{Q'} \leftarrow \theta^Q$ and $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R^{RB}$
for episode = 1,M do
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial observation state $s_1$
    for t = 1,T do
        Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
        Execute action $a_t$ and observe reward $r_t$ and new state $s_{t+1}$
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R^{RB}$
        Sample a random mini-batch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R^{RB}$
        Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
        Update critic by minimizing the loss: $L(\theta^Q) = \frac{1}{N}\sum_i (Q(s_i, a_i|\theta^Q) - y_i)^2$
        Update the actor policy using the sampled policy gradient:
            $$\nabla_{\theta^\mu} J(\mu) \approx \frac{1}{N}\sum_i [(\nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)})(\nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s=s_i})]$$
        Update the target networks:
            $$\theta^{Q'}_{t+1} \leftarrow \tau^{DDPG}\theta^Q_{t+1} + (1-\tau^{DDPG})\theta^{Q'}_t$$
            $$\theta^{\mu'}_{t+1} \leftarrow \tau^{DDPG}\theta^\mu_{t+1} + (1-\tau^{DDPG})\theta^{\mu'}_t$$
    end
end
```

Algorithm 1 is used to train the DDPG model. The training is executed through `M` episodes, each episode containing `T` time steps.

The equations for the loss function, $L(\theta^Q)$, and for the policy gradient, $\nabla_{\theta^\mu} J(\mu)$, are adapted versions of Equations 2.30 and 2.33, respectively. They are both obtained by replacing the expected value operator by an arithmetic average of the functions applied to all the N transitions sampled from the replay buffer R^{RB}. Taking the average of the observed data works, in fact, as an estimator of the expected value.

2.5.2.2 Network architecture and hyper-parameters selection

(LILLICRAP et al., 2019) presents the network architecture and hyper-parameters selected for environments with both low-dimensional observation spaces and high-dimensional observation spaces. In most cases, the former refer to non-image inputs and the latter to image inputs. Since in this work we deal exclusively with low-dimensional observation spaces, the information about high-dimensional spaces will be omitted.

(LILLICRAP et al., 2019) uses Adam (KINGMA; BA, 2014) for learning the neural network parameters. Very briefly, *Adam* is an optimization algorithm widely used to update neural network weights based on training data.

The learning rates selected by (LILLICRAP et al., 2019) were $\alpha = 10^{-4}$ for the actor networks and $\beta = 10^{-3}$ for the critic networks. A *learning rate* is a hyper-parameter that controls how fast a given neural network should be optimized. Optimizing a neural network can be understood in an intuitive way as making it learn from the training data. The greater the learning rate, the quicker a neural network tends to be optimized. Despite this seemingly advantageous behavior, as the learning rate grows, the stability of the network optimization is compromised. The process may become unstable when the learning rate exceeds certain boundaries - which cannot be easily predicted. In brief, learning rates should be selected so that they are not so big that an instability in the learning process may arise and at the same time not so small that the learning process becomes too slow.

The discount factor, γ, and the target update rate, τ^{DDPG}, were set to 0.99 and 0.001, respectively.

All four neural networks (two actors and two critics) had two hidden layers with 400 and 300 units, respectively, corresponding to about 130,000 total parameters to be optimized in each network. In the critic networks, action inputs were added after the second hidden layer. All hidden layers were activated by rectified linear units (ReLU function) and the actors' final output layers were activated by the hyperbolic tangent function, in order to bound the actions. This procedure makes the action standardized.

The final layer weights and biases of both the actors and critics were initialized from a uniform distribution taken from the range $[-3 \times 10^{-3}, 3 \times 10^{-3}]$. This aims to ensure that the initial outputs for the policy and value estimates are near zero. The other layers were initialized from uniform distributions taken from the range $\left[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}}\right]$, where f is the number of units of the layer.

The selected values for the mini-batch and the replay buffer sizes were 64 and 10^6, respectively.

For the exploration noise, an Ornstein-Uhlenbeck process with parameters $\theta^{OUP} = 0.15$, $\sigma^{OUP} = 0.2$ and $\mu^{OUP} = 0$ was selected.

3 METHODS

The main objective of this chapter is to describe how the concepts and methods presented in Chapter 2 are applied to our deep drone racing problem.

The application of DDPG was very straightforward for the most part. We reproduced the architecture and hyper-parameters selection presented in Sub-subsection 2.5.2.2 and then developed all the pieces of code that were specific to our problem. These pieces of code compose the environment of the model, essentially.

The environment is responsible for basically two processes within the model: episode initialization and state transitions. They are presented in Sections 3.2 and 3.3, respectively.

The first section of this chapter is a review and refinement of the problem we seek to solve in this work.

3.1 Hypotheses, problem definition and structuring

As already stated in the previous chapters, the goal of this work is to propose a solution to the autonomous drone racing problem using artificial intelligence models dealing with low-dimensional input data.

Sections 1.2 and 2.1 describe some general characteristics of this type of task. In this section, we present the approach we adopted to model the problem, highlighting the simplifying hypothesis we assumed.

In a drone race, be it run by humans or machines, the competing drones must go through a racecourse in the least amount of time possible, passing through a determined set of gates in a specific order.

In order to build a simplified model, which preserved the basic idea of navigation and time optimization requirements in the learning process but would be easier to implement, we substituted the racecourse environment by a single target towards which the drone should fly. By doing so, we are simulating a fraction of a racecourse which is composed basically by a single gate.

The drone starts off at a given point in space and needs to navigate to the target as quickly as possible. During the training, the starting point of the quadcopter is determined randomly and independently at the beginning of each episode. The region in which the drone is allowed to start is the surface of a sphere centered at the target and with a radius set by the user before the training starts. The process of episode initialization will be described in more detail in Section 3.2.

An essential part of the method is the flight dynamics model adopted to simulate the behavior of the drone in the context of the race. This subject is presented in Subsection 3.3.1.

As stated in Section 2.1, in order to navigate towards the target, the drone is controlled by three parameters: one related to the total thrust, one related to the pitching moment and one to the rolling moment, with no control over the yawing moment. These parameters are converted into rotors' rotational speeds - which affect the drone dynamics - by the environment. Subsection 3.3.2 is dedicated to the presentation of the control model.

During the simulation, the agent is rewarded according to three factors. The first is the angle between the quadcopter's flight direction and the direction to the target - the smaller this angle is, the more points the agent gets; this kind of reward is given every time step. The second factor is the final distance to the target; the agent is penalized according to how far it is to the target at the terminal state; this kind of reward is given only once per episode, at the terminal state, and only if the agent is not successful in its mission to navigate towards the target. The third factor is the time the agent needs to get to the target; the quicker the agent completes this task, the better the reward; this kind of reward is given only at the terminal state, and only if the agent is successful in its mission. This reward model will be further explored in Subsection 3.3.3.

The agent is considered successful in its mission if, and only if, it gets close enough to the target before a terminal state is reached. How close "close enough" is can be defined by the user via a parameter.

A terminal state occurs if one of the following situations is found:

1. The quadcopter gets close enough to the target (success).
2. The maximum simulation time is reached; this parameter can be set by the user.
3. After at least 50 time steps, the drone flies towards a direction forming an angle of more than 90 degrees in relation to the target direction, i.e., the drone diverges from the target (failure).

An important task that needs to be accomplished in order to enable the integration between the environment and the DDPG agent is to normalize the state. Subsection 3.3.4 presents this topic in more detail.

We discuss some of these topics further in the next sections.

3.2 Episode initialization

Our code allows two types of episode initialization: user-defined or random.

User-defined initialization is not recommended for training, as if this option is chosen, the agent will learn to navigate to the target from a single starting point - if it starts off anywhere different the starting point it was trained for, it will probably fail. In spite of that, user-defined initialization can be very useful for testing and applying the model, mainly if we are to verify the behavior of the agent when it starts off from a specific position.

Random initialization can be used both during training and testing or application. If this type of initialization is selected, the starting position of the drone will be selected randomly over a spherical surface centered at the target and with radius defined by the user. The initial position (x_0, y_0, z_0) in this case is given by the following equations:

$$\begin{cases} x_0 = r \sin \phi^{RUV} \cos \theta^{RUV} \\ y_0 = r \sin \phi^{RUV} \sin \theta^{RUV} \\ z_0 = r \cos \phi^{RUV} \end{cases} \tag{3.1}$$

where r is the user-defined radius and ϕ^{RUV} and θ^{RUV} are random uniform variables, with $\phi^{RUV} \sim \mathcal{U}(0, \pi)$ and $\theta^{RUV} \sim \mathcal{U}(0, 2\pi)$. The superscript RUV stands for *random uniform variable*.

The motivation for choosing a spherical surface as the initialization space is given by a combination of the two factors below:

1. This choice makes the engineering of a reward function easier, as the quadcopter always starts at an "equivalent" position - the initial distance between the quadcopter and the target is constant.
2. If the agent learns to navigate from any point on a given spherical surface, it is logical to expect that it will learn to navigate from any point inside that sphere as well, so this choice doesn't limit the agent to a small set of possible initial positions.

In random initialization, all state variables except for (x_0, y_0, z_0) are initialized at zero, i.e.:

$$\begin{bmatrix} u(0) & v(0) & w(0) \end{bmatrix}^T = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T \tag{3.2a}$$

$$\begin{bmatrix} \theta^{SV}(0) & \phi^{SV}(0) & \psi^{SV}(0) \end{bmatrix}^T = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T \tag{3.2b}$$

$$\begin{bmatrix} \dot{\theta}^{SV}(0) & \dot{\phi}^{SV}(0) & \dot{\psi}^{SV}(0) \end{bmatrix}^T = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T \tag{3.2c}$$

where $\begin{bmatrix} u(0) & v(0) & w(0) \end{bmatrix}^T$ represents the initial linear velocity, $\begin{bmatrix} \theta^{SV}(0) & \phi^{SV}(0) & \psi^{SV}(0) \end{bmatrix}^T$ represents the initial orientation and $\begin{bmatrix} \dot{\theta}^{SV}(0) & \dot{\phi}^{SV}(0) & \dot{\psi}^{SV}(0) \end{bmatrix}^T$ represents the initial angular velocity. The superscript SV stands for *state variable*.

In user-defined initialization, on the other hand, all state variables can be set to any initial value.

3.3 State transitions

Each state transition is immediately preceded by the agent choosing an action according to its current policy. Given this action, the state transition is carried out by following the steps below:

1. translate the actions command transmitted by the agent into variables with physical meaning - in this case, the four rotors' rotational speeds;

2. simulate the quadcopter behavior in response to this command and obtain the new state of the drone based on this simulation;

3. normalize the new state - the DDPG neural network should only ingest normalized data;

4. calculate some auxiliary variables, such as the new distance from the quadcopter to the target and the angle formed between the drone's linear velocity and the vector from the drone to the target;

5. determine if the current episode should or not be terminated based on the terminal state conditions listed in Section 3.1;

6. calculate the agent's reward given the old and new states and the action.

These steps are further described in the subsections that follow. We present them in a didactic order rather than in the chronological order used above.

3.3.1 Modeling the physical environment: quadcopter flight dynamics

The main references used in this section are (ROSKAM, 1979), (ETKIN; REID, 1996) and (JAISWAL, 2018).

In order to simplify our flight dynamics model, we make some assumptions about the quadcopter:

1. The four rotors of the quadcopter are idealized as points; these points are named R_1, R_2, R_3 and R_4 and they form a square; the distance of each of these four point to the center of the quadcopter (C) - i.e., the semi-diagonal of the square - is denoted by L; these assumptions are illustrated in Figure 5.

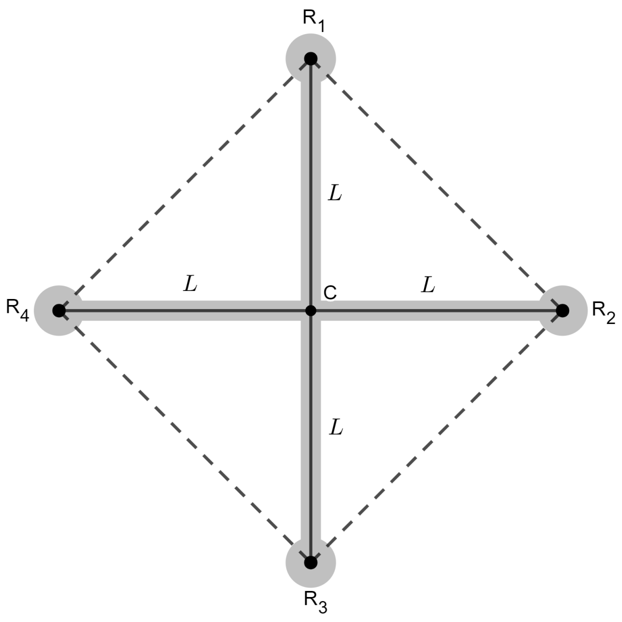


Figure 5 – Quadcopter model with simplification assumptions.

2. Each rotor produces a force and a moment, both normal to the plan formed by the four rotors.

3. Two of the rotors spin in clockwise direction and two of them spin in counterclockwise direction; here we assume that rotors R_1 and R_3 spin in the counterclockwise direction and R_2 and R_4 spin in the clockwise direction.

4. The quadcopter has two planes of symmetry, each one of them containing the line that passes through one of the two pairs of opposing rotors and perpendicular to the line that passes through the other pair of opposing rotors.

3.3.1.1 Frames of reference

Two frames of reference are employed to develop the flight dynamics equations that describe the physical environment of the problem. The first of them is an inertial frame XYZ fixed to a point in space that can be seen as the center of the simulation "scenario". This frame's Z-axis has the same orientation as the gravitational acceleration, i.e., it points downwards in relation to the Earth. X and Y are each parallel to the surface of the Earth. Frame XYZ forms a positive orthonormal basis and will be called the *global frame*.

The second frame of reference is a non-inertial frame xyz with its origin fixed to the center of mass of the quadcopter. The x-axis of this frame points in the same direction as the vector $\overrightarrow{CR_1}$ and the y-vector points in the direction of vector $\overrightarrow{CR_2}$. The z-axis is such that xyz forms a positive orthonormal basis - this implies that the z-axis points in the opposite direction of the lift forces generated by the four rotors of the quadcopter. xyz is said a *body-fixed* frame of reference. That means it is rigidly attached to the quadcopter and thus moves with the quadcopter.

3.3.1.2 External forces and moments

In our model, we consider the following external forces and moments acting on the quadcopter:

- the quadcopter's weight, $\overrightarrow{W}$;
- four thrust forces, $\overrightarrow{T_1}$, $\overrightarrow{T_2}$, $\overrightarrow{T_3}$ and $\overrightarrow{T_4}$, one for each rotor;
- and four thrust moments, $\overrightarrow{M_{T_1}}$, $\overrightarrow{M_{T_2}}$, $\overrightarrow{M_{T_3}}$ and $\overrightarrow{M_{T_4}}$, one for each rotor.

The quadcopter's weight is calculated by the classic equation given below:

$$\overrightarrow{W} = m\overrightarrow{g}, \tag{3.3}$$

where m is the quadcopter's mass and $\overrightarrow{g}$ is the acceleration of gravity.

The thrust forces are calculated by the following equation:

$$T_i = k_T \omega_i^2, \tag{3.4}$$

where the subscript i can assume any value in the set $1, 2, 3, 4$ and denotes the rotor the associated variable refers to, k_T is an empirical coefficient that depends on multiple factors, like air density and propeller diameter, and ω is the rotational speed of the rotor. We will refer to the coefficient k_T as the *thrust force coefficient.*

Similarly, the thrust moments are calculated as follows:

$$M_{T_i} = k_{M_T} \omega_i^2, \tag{3.5}$$

where k_{M_T} has a similar description as k_T and will be referred to as the *thrust moment coefficient.*

The aerodynamic forces are only indirectly considered in the model, by limiting the speed of the drone in each direction to certain maximum values.

3.3.1.3 Kinematic variables

The position of the quadcopter in relation to the inertial frame of reference XYZ is given by:

$$\overrightarrow{x_P} \doteq \hat{x}\hat{I} + \hat{y}\hat{J} + \hat{z}\hat{K}, \tag{3.6}$$

where $\hat{I}$, $\hat{J}$ and $\hat{K}$ are unit vector in the directions of X, Y and Z, respectively.

The quadcopter's velocity in relation to frame XYZ is calculated as the time derivative of $\overrightarrow{x_P}$:

$$\overrightarrow{V_P} = \frac{d\overrightarrow{x_P}}{dt} \doteq \dot{x}\hat{I} + \dot{y}\hat{J} + \dot{z}\hat{K} \tag{3.7}$$

The acceleration, in turn, is the time derivative of the velocity:

$$\overrightarrow{a_P} = \frac{d\overrightarrow{V_P}}{dt} \doteq \ddot{x}\hat{I} + \ddot{y}\hat{J} + \ddot{z}\hat{K} \tag{3.8}$$

Velocity $\overrightarrow{V_P}$ can just as well be decomposed according to frame xyz's axes. In that case, we would have:

$$\overrightarrow{V_P} \doteq u\hat{\imath} + v\hat{\jmath} + w\hat{k}, \tag{3.9}$$

where $\hat{\imath}$, $\hat{\jmath}$ and $\hat{k}$ are unit vectors in the directions of x, y and z, respectively.

The time derivatives of u, v e w are given by the following equations:

$$\dot{u} \doteq \frac{\partial u}{\partial t} \tag{3.10a}$$

$$\dot{v} \doteq \frac{\partial v}{\partial t} \tag{3.10b}$$

$$\dot{w} \doteq \frac{\partial w}{\partial t} \tag{3.10c}$$

The orientation of the drone (in relation to the global frame XYZ) is described by a set of Euler angles (θ^{SV}, ϕ^{SV}, ψ^{SV}), which are defined below. XYZ's axes are referred to as X, Y and Z and xyz's axes as x, y and z.

- *Attitude* or *pitch* angle (θ^{SV}): angle formed from X to x', where x' is the projection of x on the plane XZ; θ^{SV} is positive if it has clockwise direction when observed with Y pointing into plane XZ.
- *Bank* or *roll* angle (ϕ^{SV}): angle formed from Y to y', where y' is the projection of y on the plane YZ; ϕ^{SV} is positive if it has clockwise direction when observed with X pointing into plane YZ.
- *Heading* or *yaw* angle (ψ^{SV}): angle formed from X to x', where x' is the projection of x on the plane XY; ψ^{SV} is positive if it has clockwise direction when observed with Z pointing into plane XY.

The quadcopter's angular velocity, $\overrightarrow{\omega}$, is decomposed according to xyz's axes:

$$\overrightarrow{\omega} \doteq P^{SV}\hat{\imath} + Q^{SV}\hat{\jmath} + R^{SV}\hat{k}, \tag{3.11}$$

The time derivatives of P^{SV}, Q^{SV} and R^{SV} are given by:

$$\dot{P}^{SV} \doteq \frac{\partial P^{SV}}{\partial t} \tag{3.12a}$$

$$\dot{Q}^{SV} \doteq \frac{\partial Q^{SV}}{\partial t} \tag{3.12b}$$

$$\dot{R}^{SV} \doteq \frac{\partial R^{SV}}{\partial t} \tag{3.12c}$$

At the same time, the following relation applies:

$$\overrightarrow{\omega} = \dot{\phi}^{SV}\hat{\imath}^{RA} + \dot{\theta}^{SV}\hat{\jmath}^{RA} + \dot{\psi}^{SV}\hat{k}^{RA}, \tag{3.13}$$

where $\hat{\imath}^{RA}$, $\hat{\jmath}^{RA}$ and $\hat{k}^{RA}$ are unit vectors resulting from a set of rotations applied to $\hat{\imath}$, $\hat{\jmath}$ and $\hat{k}$ (see (ROSKAM, 1979) for more detailed information) and:

$$\dot{\psi}^{SV} \doteq \frac{\partial \psi^{SV}}{\partial t} \tag{3.14a}$$

$$\dot{\theta}^{SV} \doteq \frac{\partial \theta^{SV}}{\partial t} \tag{3.14b}$$

$$\dot{\phi}^{SV} \doteq \frac{\partial \phi^{SV}}{\partial t} \tag{3.14c}$$

3.3.1.4 Dynamic variables

The external forces and moments acting on the quadcopter are decomposed according to frame xyz's axes:

$$\overrightarrow{W} \doteq W_x\hat{\imath} + W_y\hat{\jmath} + W_z\hat{k} \tag{3.15}$$

$$\overrightarrow{T_i} \doteq T_{i_z}\hat{k} \tag{3.16}$$

$$\overrightarrow{M_{T_i}} \doteq M_{T_{i_z}}\hat{k} \tag{3.17}$$

Here, it is worth recalling, from the beginning of this section, that the thrust forces and moments produced by the rotors are normal to the plan formed by the four rotors. That is why $\overrightarrow{T_i}$ and $\overrightarrow{M_{T_i}}$ have a single component z.

3.3.1.5 Equations of motion

The derivation of all the equations applied in this subsection can be found in (ROSKAM, 1979).

The application of the conservation of linear and angular momentum to the flight dynamics of a quadcopter results in the six scalar equations that follow. The three equations associated to the conservation of angular momentum are not general, but they apply to our problem. They lack all the terms that are proportional to any product of inertia. Since the drone is bisymmetric, all products of inertia are zero, i.e., $I_{xy} = 0$, $I_{xz} = 0$ and $I_{yz} = 0$, and that is the reason why we can consider the simplified equations shown below.

$$m(\dot{u} - vR^{SV} + wQ^{SV}) = F_x = W_x \tag{3.18a}$$
$$m(\dot{v} - wP^{SV} + uR^{SV}) = F_y = W_y \tag{3.18b}$$
$$m(\dot{w} - uQ^{SV} + vP^{SV}) = F_z = W_z + \sum_{i=1}^{4} T_{i_z} \tag{3.18c}$$

$$I_{xx}\dot{P}^{SV} + (I_{zz} - I_{yy})R^{SV}Q^{SV} = M_x = L(T_{4_z} - T_{2_z}) \tag{3.19a}$$
$$I_{yy}\dot{Q}^{SV} + (I_{xx} - I_{zz})P^{SV}R^{SV} = M_y = L(T_{1_z} - T_{3_z}) \tag{3.19b}$$
$$I_{zz}\dot{R}^{SV} + (I_{yy} - I_{xx})P^{SV}Q^{SV} = M_z = \sum_{i=1}^{4} M_{T_{i_z}} \tag{3.19c}$$

In these equations:

- F_x, F_y and F_z are the resulting forces along axes x, y and z of frame xyz;
- M_x, M_y and M_z are the resulting moments about the quadcopter's center of mass along axes x, y and z of frame xyz;
- I_{xx}, I_{yy} and I_{zz} are the moments of inertia around axes x, y and z of frame xyz.

The forces and moments components that appear in Equations 3.18 and 3.19 can be calculated as follows:

$$W_x = -W \sin\theta^{SV} \tag{3.20}$$
$$W_y = W \cos\theta^{SV} \sin\phi^{SV} \tag{3.21}$$
$$W_z = W \cos\theta^{SV} \cos\phi^{SV} \tag{3.22}$$
$$T_{i_z} = -T_i = -k_T\omega_i^2 \tag{3.23}$$
$$M_{T_{i_z}} = \begin{cases} M_{T_i} = k_{M_T}\omega_i^2, & \text{if } i \in \{1,3\} \\ -M_{T_i} = -k_{M_T}\omega_i^2, & \text{if } i \in \{2,4\} \end{cases} \tag{3.24}$$

The different signs in Equation 3.24 are due to the spinning directions of the rotors: those that spin in the counterclockwise directions generate positive moments and those that spin in the clockwise direction generate negative moments.

The velocity components $\dot{x}$, $\dot{y}$ and $\dot{z}$ are related to u, v and w according to the following relation:

$$\begin{Bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{Bmatrix} = \begin{bmatrix} \cos\psi^{SV} & -\sin\psi^{SV} & 0 \\ \sin\psi^{SV} & \cos\psi^{SV} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta^{SV} & 0 & \sin\theta^{SV} \\ 0 & 1 & 0 \\ -\sin\theta^{SV} & 0 & \cos\theta^{SV} \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi^{SV} & -\sin\phi^{SV} \\ 0 & \sin\phi^{SV} & \cos\phi^{SV} \end{bmatrix} \begin{Bmatrix} u \\ v \\ w \end{Bmatrix} \tag{3.25}$$

Equation 3.25 can be inverted in order to calculate u, v and w from $\dot{x}$, $\dot{y}$ and $\dot{z}$, as the relation below shows:

$$\begin{Bmatrix} u \\ v \\ w \end{Bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi^{SV} & \sin\phi^{SV} \\ 0 & -\sin\phi^{SV} & \cos\phi^{SV} \end{bmatrix} \begin{bmatrix} \cos\theta^{SV} & 0 & -\sin\theta^{SV} \\ 0 & 1 & 0 \\ \sin\theta^{SV} & 0 & \cos\theta^{SV} \end{bmatrix} \begin{bmatrix} \cos\psi^{SV} & \sin\psi^{SV} & 0 \\ -\sin\psi^{SV} & \cos\psi^{SV} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{Bmatrix} \tag{3.26}$$

Lastly, the angular velocity components P^{SV}, Q^{SV} and R^{SV} are related to $\dot{\psi}^{SV}$, $\dot{\theta}^{SV}$ and $\dot{\phi}^{SV}$ according to the following equations:

$$P^{SV} = \dot{\phi}^{SV} - \dot{\psi}^{SV}\sin\theta^{SV} \tag{3.27a}$$

$$Q^{SV} = \dot{\theta}^{SV}\cos\phi^{SV} + \dot{\psi}^{SV}\cos\theta^{SV}\sin\phi^{SV} \tag{3.27b}$$

$$R^{SV} = \dot{\psi}^{SV}\cos\theta^{SV}\cos\phi^{SV} - \dot{\theta}^{SV}\sin\phi^{SV} \tag{3.27c}$$

Inverting Equations 3.27, we obtain:

$$\dot{\theta}^{SV} = Q^{SV}\cos\phi^{SV} - R^{SV}\sin\phi^{SV} \tag{3.28a}$$

$$\dot{\phi}^{SV} = P^{SV} + (Q^{SV}\sin\phi^{SV} + R^{SV}\cos\phi^{SV})\tan\theta^{SV} \tag{3.28b}$$

$$\dot{\psi}^{SV} = \frac{Q^{SV}\sin\phi^{SV} + R^{SV}\cos\phi^{SV}}{\cos\theta^{SV}} \tag{3.28c}$$

Equations 3.28 hold as long as $\theta^{SV} \neq (\pi/2 + k\pi)$ rad, with $k \in \mathbb{Z}$. For $\theta^{SV} = (\pi/2 + k\pi)$ rad, Equations 3.27 cannot be inverted.

3.3.1.6 Solving procedure

In order to obtain the temporal evolution of the quadcopter's position, orientation and linear and angular velocities - which is the objective of the simulation of the quadcopter's flight dynamics -, the following procedure needs to be followed:

- **Initialization:**
  1. Set the initial position, orientation, linear and angular velocities (decomposed according to the global frame) according to any criteria defined by the problem or the user:

$$\overrightarrow{x_P}(0) = \overrightarrow{x_{P_0}} \tag{3.29}$$

$$\overrightarrow{V_P}(0) = \overrightarrow{V_{P_0}} \tag{3.30}$$

$$\begin{bmatrix}\theta^{SV}(0) & \phi^{SV}(0) & \psi^{SV}(0)\end{bmatrix}^T = \begin{bmatrix}\theta_0^{SV} & \phi_0^{SV} & \psi_0^{SV}\end{bmatrix}^T \tag{3.31}$$

$$\begin{bmatrix}\dot{\theta}^{SV}(0) & \dot{\phi}^{SV}(0) & \dot{\psi}^{SV}(0)\end{bmatrix}^T = \begin{bmatrix}\dot{\theta}_0^{SV} & \dot{\phi}_0^{SV} & \dot{\psi}_0^{SV}\end{bmatrix}^T \tag{3.32}$$

where $\overrightarrow{x_{P_0}}$, $\overrightarrow{V_{P_0}}$, $\begin{bmatrix}\theta_0^{SV} & \phi_0^{SV} & \psi_0^{SV}\end{bmatrix}^T$ and $\begin{bmatrix}\dot{\theta}_0^{SV} & \dot{\phi}_0^{SV} & \dot{\psi}_0^{SV}\end{bmatrix}^T$ are initial conditions set by the problem or the user.

2. Obtain the initial linear and angular velocities decomposed according to the body-fixed frame applying Equations 3.26 and 3.27.

- **At each time step:**

1. Calculate all external forces and moments acting on the quadcopter using Equations 3.20 to 3.24.
2. Calculate the linear and angular accelerations solving Equations 3.18 and 3.19 for the variables $\dot{u}$, $\dot{v}$, $\dot{w}$, $\dot{P}^{SV}$, $\dot{Q}^{SV}$ and $\dot{R}^{SV}$ (all other variables are known).
3. Integrate variables $\dot{u}$, $\dot{v}$, $\dot{w}$, $\dot{P}^{SV}$, $\dot{Q}^{SV}$ and $\dot{R}^{SV}$ over time in order to obtain u, v, w, P^{SV}, Q^{SV} and R^{SV}, respectively.
4. Integrate variables $\dot{x}$, $\dot{y}$, $\dot{z}$, $\dot{\theta}^{SV}$, $\dot{\phi}^{SV}$ and $\dot{\psi}^{SV}$ (with their values at the beginning of the time step) over time in order to obtain x, y, z, θ^{SV}, ϕ^{SV} and ψ^{SV}, respectively.
5. Update the linear and angular velocities decomposed according to the global frame applying Equations 3.25 and 3.28.

3.3.1.7 Implementation

The equations and concepts presented in this section were implemented in Python and resulted in a class we named `Quadcopter`. This class is essentially destined to simulate the behavior of the quadcopter during its flight, which is an essential part of the environment we needed to implement.

The input data needed to instantiate the class `Quadcopter` are basically the quadcopter's parameters (mass, moments of inertia, thrust coefficients, etc.) and some simulation parameters (time interval in one time step, initial position, orientation and velocities).

The implementation of the equations previously presented was very straightforward.

The acceleration and velocity integrations mentioned in Sub-subsection 3.3.1.6 were accomplished via finite difference methods. (WIKIPEDIA, 2020c) Except for the first four iterations of a given simulation, the method that was employed is one with a fifth-order accuracy. In the first four iterations, this method cannot be applied, as it requires the knowledge of the kinematic variables from the four immediately previous time steps of the simulation. In these initial iterations, more simple - and thus less precise - methods are employed. Since each simulation has hundreds of iterations and the time step intervals are generally small, the use of these less precise methods on the first four steps does not impact the overall quality of the simulator.

The equation below shows the main selected method applied to the integration of

$\dot{u}$. The application of this method to any other variable is analogous.

$$u_{i+1} = \frac{60}{137}(\dot{u}_{i+1}\Delta t + 5u_i - 5u_{i-1} + \frac{10}{3}u_{i-2} - \frac{5}{4}u_{i-3} + \frac{1}{5}u_{i-4}) \tag{3.33}$$

3.3.2 Translating agent commands into physical variables: the control model

The control model employed in this work was built in a way that aims to make the agent's learning process less difficult, by establishing actions that correspond exactly to some of the commands a human would use to control a drone: total thrust, pitching moment and rolling moment.

The agent's commands are thus composed by three coefficients, all of which are normalized values that vary between -1 and 1. Let us denote these coefficients by c_T (thrust coefficient), c_{M_y} (pitching moment coefficient) and c_{M_x} (rolling moment coefficient).

The first step in translating these commands into physical variables is to "unnormalize" them. By doing so, the results we obtain are the target total thrust, pitching and rolling moments corresponding to the agent's commands, which we denote by T^{target}, M_y^{target} and M_x^{target}, respectively. The unnormalization operation is carried out according to the following equations:

$$T^{target} = T^{min} + \frac{c_T + 1}{2}(T^{max} - T^{min}) \tag{3.34}$$

$$M_x^{target} = c_{M_x} M_x^{max} \tag{3.35}$$

$$M_y^{target} = c_{M_y} M_y^{max}, \tag{3.36}$$

where T^{min}, T^{max}, M_x^{max} and M_y^{max} are user-defined variables and represent the minimum and maximum total thrust and the maximum rolling and pitching moments, respectively. The setting of these variables determines how aggressively the quadcopter might react to the agent's commands. For the pitching and rolling moments, the minimum allowable values are symmetric to the maximum ones, which justifies why Equations 3.35 and 3.36 have a similar format, but not Equation 3.34.

Once we know the target total thrust and rolling and pitching moments, it becomes easier to determine what the rotors' rotational speeds must be in order to get those target values. As presented along Subsection 3.3.1, all these variables are related according to the following equations:

$$T = k_T(\omega_1^2 + \omega_2^2 + \omega_3^2 + \omega_4^2) \tag{3.37}$$

$$M_x = k_T(\omega_4^2 - \omega_2^2)L \tag{3.38}$$

$$M_y = k_T(\omega_1^2 - \omega_3^2)L \tag{3.39}$$

An additional variable comes from the fact that we want to force the yawing moment to be zero at all times:

$$M_z = k_{M_T}(\omega_1^2 - \omega_2^2 + \omega_3^2 - \omega_4^2) = 0 \tag{3.40}$$

Inverting Equations 3.37 to 3.40, we obtain what each rotor's rotational speed must be:

$$\omega_1 = \sqrt{\frac{T^{target}}{4k_T} + \frac{M_y^{target}}{2k_T L}} \tag{3.41a}$$

$$\omega_2 = \sqrt{\frac{T^{target}}{4k_T} - \frac{M_x^{target}}{2k_T L}} \tag{3.41b}$$

$$\omega_3 = \sqrt{\frac{T^{target}}{4k_T} - \frac{M_y^{target}}{2k_T L}} \tag{3.41c}$$

$$\omega_4 = \sqrt{\frac{T^{target}}{4k_T} + \frac{M_x^{target}}{2k_T L}} \tag{3.41d}$$

Thus, the translation of the agent's commands into rotors' rotational speeds is performed by applying Equations 3.34 to 3.36 and 3.41 sequentially.

3.3.3 Instructing the agent: the reward model

Three different reward models were implemented and tested. We present each of them in this subsection, starting by the one which was selected in the end.

The following variables are used in these models:

- `advance_angle`: angle between the current linear velocity of the quadcopter and the vector from the quadcopter to the target, given in degrees;
- `dt`: user-defined parameter controlling the time interval between any two consecutive time steps in the simulation, given in seconds;
- `failure_reward_constant`: user-defined parameter controlling how aggressively the agent should be penalized for a non-successful episode;
- `n_steps`: number of time steps since the beginning of the episode;
- `reward`: the reward associated to the current time step;
- `success_reward_constant`: user-defined parameter controlling how aggressively the agent should be rewarded for a successful episode;
- `target_distance`: distance from the quadcopter to the target, given in meters;
- `terminal_state`: Boolean which indicates if a terminal state has been reached (`True`) or not (`False`); the conditions that characterize a terminal state are listed in Section 3.1;
- `terminal_target_distance`: user-defined parameter defining how close the quadcopter must get to the target in order for the episode to be considered successful, in meters;
- `time_step`: a time step within an episode.

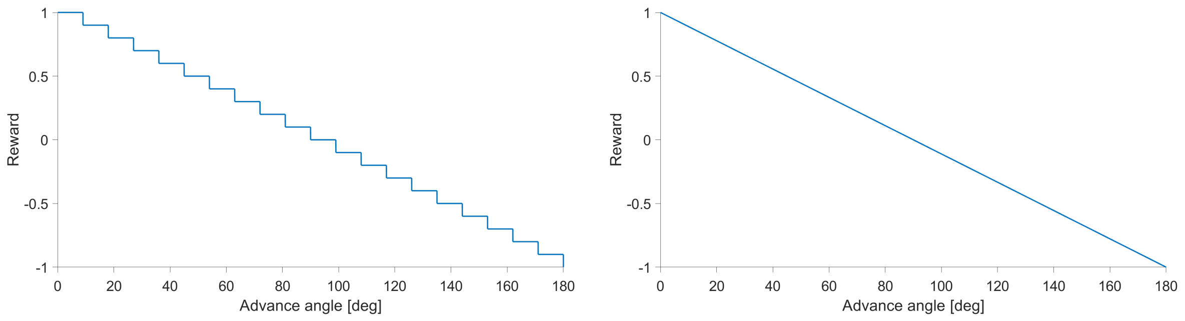
3.3.3.1 Reward model 1: the selected model

Algorithm 2: Reward model 1

```
foreach time_step do
    if terminal_state then
        if target_distance <= terminal_target_distance then
            reward = $\frac{\texttt{success\_reward\_constant}}{\texttt{n\_steps} \times \texttt{dt}}$
        else
            reward = $-$failure_reward_constant $\times$ target_distance
        end
    else
        reward = $1 - \lfloor \frac{\texttt{advance\_angle}}{9} \rfloor / 10$
    end
end
```

As already stated in Section 3.1, in this model the agent is rewarded according to three factors:

- the advance angle, at every non-terminal time step; in this case the reward follows the law given in Algorithm 2 and illustrated in Figure 6a;
- the final distance to the target, at the terminal state, and only if the agent has not been successful in guiding the quadcopter to the target; in this case, the reward is negative - i.e., it is a penalization - and proportional to the final distance to the target;
- the amount of time the quadcopter took to get to the target, at the terminal state, and only if the agent has been successful; in this case, the reward is positive and proportional to the inverse of the time taken; the time taken is given by the multiplication `n_steps` $\times$ `dt` in Algorithm 2.



(a) Model 1 (b) Model 2

Figure 6 – Reward as a function of the advance angle.

3.3.3.2 Reward model 2

Algorithm 3: Reward model 2

```
foreach time_step do
    if terminal_state then
        if target_distance <= terminal_target_distance then
            reward = $\frac{\text{success\_reward\_constant}}{\text{n\_steps} \times \text{dt}}$
        else
            reward = $-$failure_reward_constant $\times$ target_distance
        end
    else
        reward = $1 - \frac{\text{advance\_angle}}{90}$
    end
end
```

This model is very similar to Model 1. The sole difference is that the reward due to the advance angle is a linear function of the angle in this case (Figure 6b), instead of a step function.

3.3.3.3 Reward model 3

Algorithm 4: Reward model 3

```
foreach time_step do
    if terminal_state then
        if target_distance <= terminal_target_distance then
            reward = $\frac{\text{success\_reward\_constant}}{\text{n\_steps} \times \text{dt}}$
        else
            reward = $-$failure_reward_constant $\times$ target_distance
        end
    else
        reward = $0.0888 \times \tan(-0.9436 \times \text{advance}_\text{a}\text{ngle} + 84.923797313630573)$
    end
end
```

Once again, the only difference between this model and Model 1 is the reward function due to the advance angle. In this model, we used a tangent function, shown in Figure 7, which aimed to make the agent choose a as-straight-as-possible trajectory to the target by increasing the reward rapidly as the advance angle approaches zero. The performance of this model during training was not as good as expected, however.

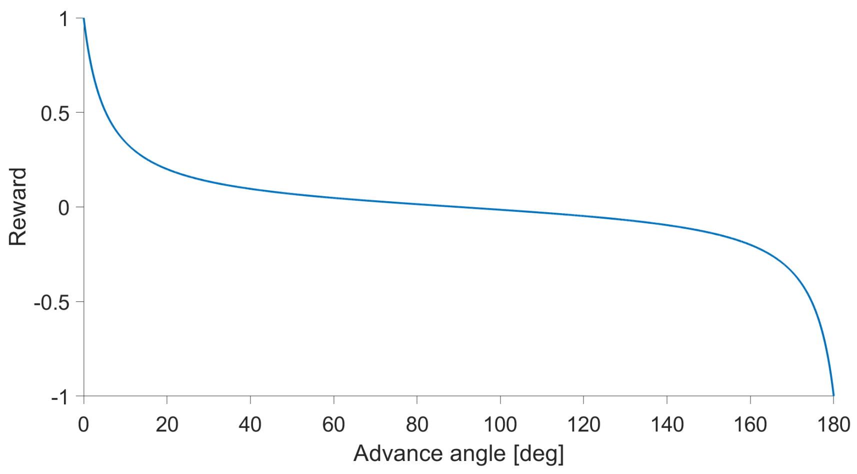


Figure 7 – Reward as a function of the advance angle - Model 3.

3.3.4 Communicating with the agent: state normalization

As stated in Sub-subsection 2.5.1.7, normalizing the data which is used to feed the agent is a very important procedure.

We used a different approach than (LILLICRAP et al., 2019) to do this task: instead of using the suggested standardization procedure - which corresponds to subtracting the average and dividing by the standard deviation -, we applied feature scaling - which is equivalent to mapping the values into a range $[0, 1]$ or other values in the vicinity of this range.

In order to do so, we established two reference values for each state variable, one that we believed would be reasonable to be less than any observed value for that variable and one that would be greater than any observed value. For some variables like the position ones there is no way of guaranteeing that an observed value will never be less or greater than a given reference. This is not a problem however, as long as the values are mapped into the range $[0, 1]$ most of the time.

The advantage of using feature scaling instead of standardization is that the average and the standard deviation of the state variables don't need to be calculated at each time step. Besides that, the average and standard deviation vary with time, so the quality of standardization may be worsened during testing and the application of the model, as these measures are only updated during training.

The main advantage of standardization over feature scaling is that it's more effective than feature scaling. It also doesn't depend on user-defined input.

A secondary distinction between (LILLICRAP et al., 2019)'s and our approaches is that (LILLICRAP et al., 2019) suggests normalizing the data only after they are sampled and transformed into a batch. In our approach, on the other hand, the state variables are already normalized when they are stored in the replay buffer.

4 RESULTS AND DISCUSSION

Figures 8 and 9 illustrate the agent's learning process. Figure 8 shows the moving average of the agent's final score and Figure 9 shows the moving average of the agent's final distance to the target along the training episodes.

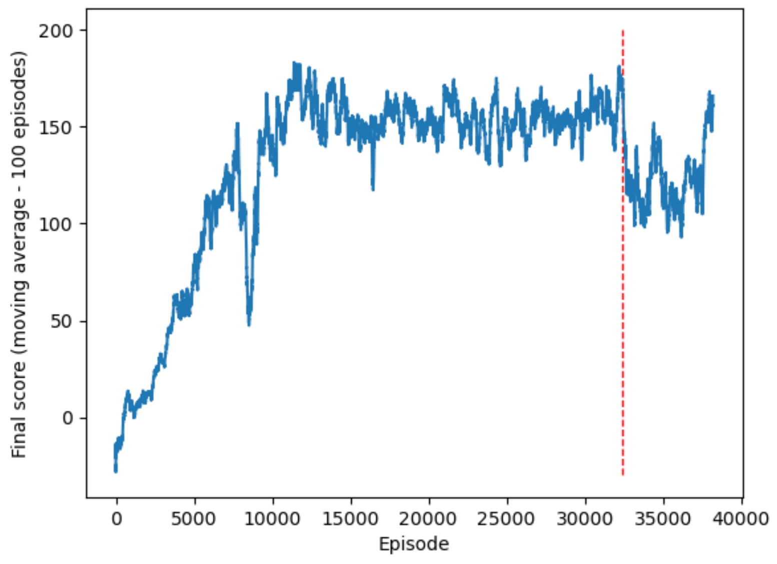


Figure 8 – Agent's final score (moving average) during training.

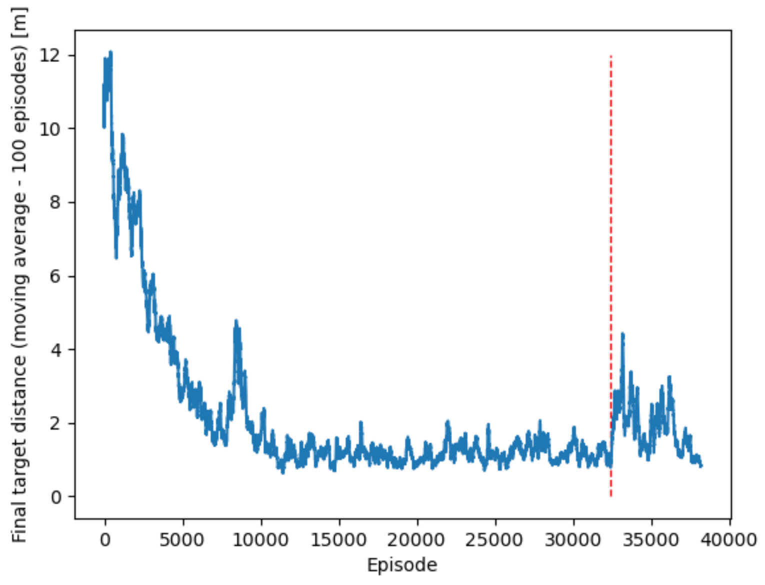


Figure 9 – Agent's final distance to the target (moving average) during training.

The agent that resulted from training had an average final score of 166 and an average final target distance of 0.85, both considering a 100-episode moving average.

Both Figure 8 and Figure 9 contain a red vertical line around episode number 32400 in a region in which the performance of the agent was notably degraded. The reason for this degradation was a modification in the learning settings: until episode 32400, training occurred with the application of exploration noise; after episode 32400, this noise was no longer employed. This modification was put in place for two reasons:

1. By that time, the agent had already discovered a strong policy, so the use of exploration noise was not necessary anymore.

2. During testing and application, no exploration noise is applied; training the agent without noise has shown to yield better results during these phases than training the agent with noise.

The model was tested in 100 randomly-generated episodes, and the results obtained - in terms of the quadcopter's final distance to the target and the task completion time - are shown in Table 1.

Table 1 – Model performance in a sample containing 100 randomly generated episodes, in terms of the quadcopter's final distance to the target and the task completion time.

Aggregate function	**Target distance [m]**	**Completion time [s]**
Minimum	0.188	7.450
Maximum	1.526	19.600
Average	0.587	13.817
Standard deviation	0.267	2.696

Figures 10, 11 and 12 show some examples of trajectories taken by the agent in order to fly the quadcopter from its initial position to the target.

Even though the trajectories chosen by the agent are not optimal, as they are usually slightly curved, the results and the practical examples shown in Figures 10, 11 and 12 demonstrate that the agent has been successful in learning to fly the quadcopter to the target within a reasonable amount of time.

The time taken to complete each episode is not great yet, and it remains to be further optimized.

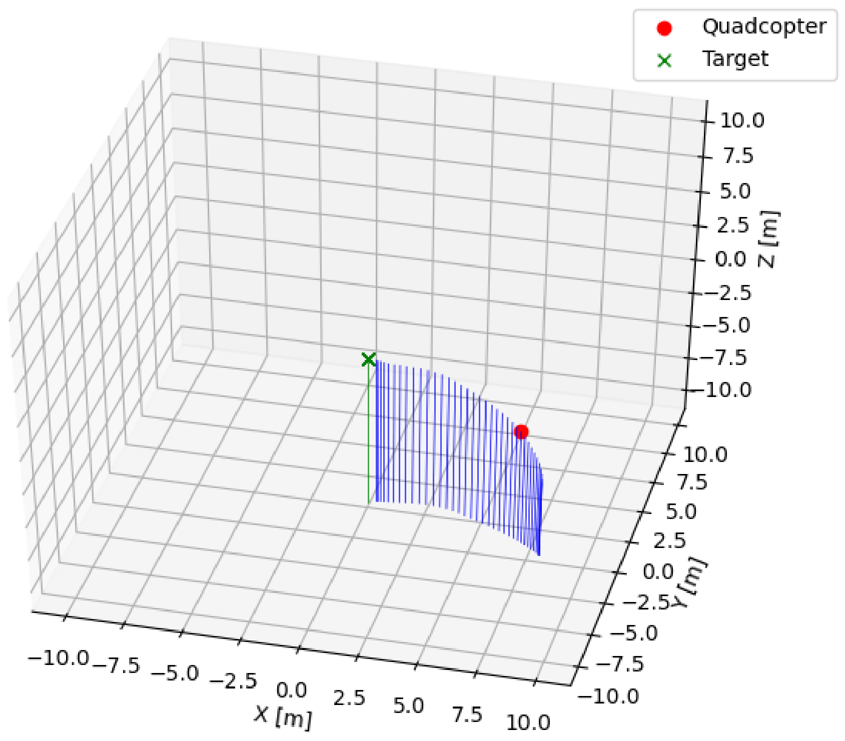


Figure 10 – Trajectory chosen by the agent to fly the quadcopter from its initial position to the target - Example 1.

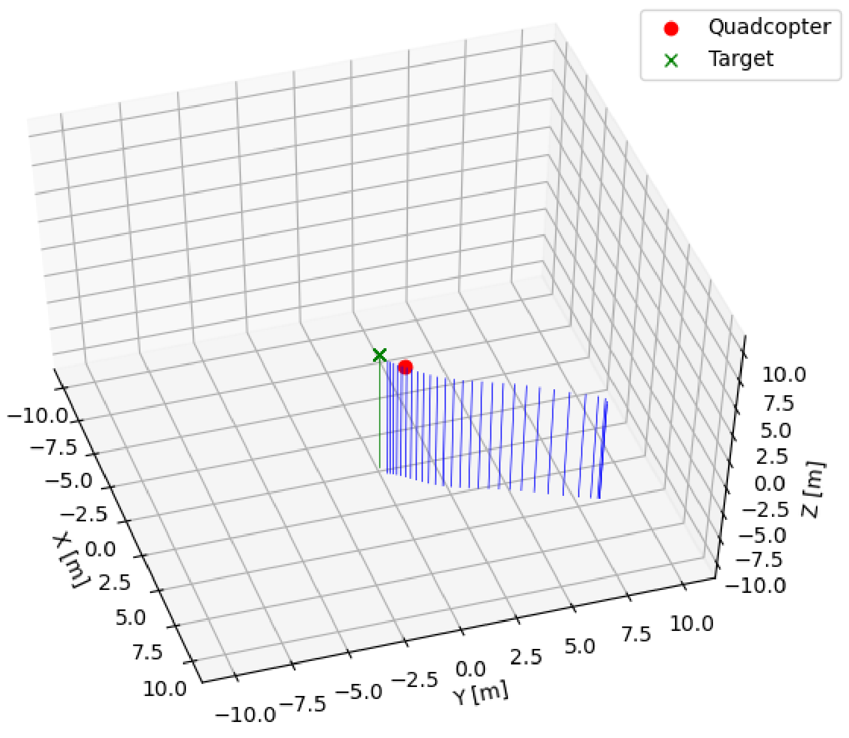


Figure 11 – Trajectory chosen by the agent to fly the quadcopter from its initial position to the target - Example 2.

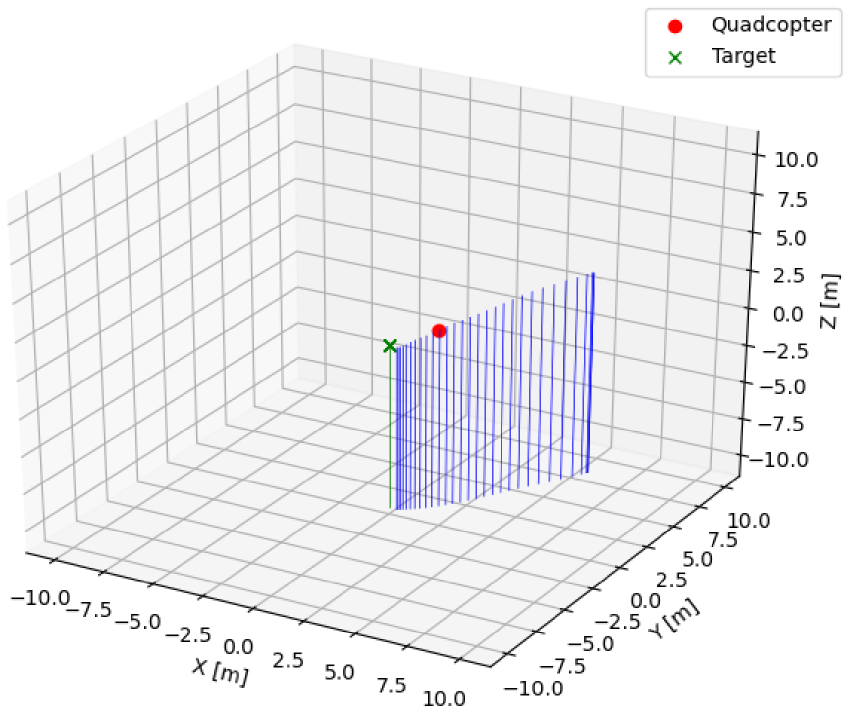


Figure 12 – Trajectory chosen by the agent to fly the quadcopter from its initial position to the target - Example 3.

5 CONCLUSION

This work presents a solution to a simplified version of the deep drone racing problem, applying the method Deep Deterministic Policy Gradient (DDPG) with low-dimensional input.

The exact same architecture, set of parameters and hyper-parameters used in (LILLICRAP et al., 2019) were applied in this work, yielding satisfactory results. This evidences the robustness and simplicity of DDPG, which, as stated in (LILLICRAP et al., 2019), can handle a wide range of continuous physics problems with the exact same set of hyper-parameters, as long as all input variables are normalized.

A few reward models were engineered and tested, and the one with the best performance among them was picked.

Our model output consistently good results in what concerns the task of navigating from an arbitrary initial position to a predefined target. The time the agent takes to complete this task remains to be optimized though.

Some next steps that may be taken in order to develop this work further are: replace the normalization model we applied by the one suggested by (LILLICRAP et al., 2019) and (IOFFE; SZEGEDY, 2015); search for a new reward model capable of improving the agent's performance with regard to the task completion time; improve sample efficiency by applying the method proposed by (GU et al., 2016); generalize the control model to allow for yawing commands; and train the agent to deal with initial orientations and velocities other than zero.

BIBLIOGRAPHY

ACHIAM, J. **Deep Deterministic Policy Gradient**. 2018. Accessed: Oct. 06, 2020. Disponível em: <https://spinningup.openai.com/en/latest/algorithms/ddpg.html>.

_____. **Introduction to RL**. 2018. Accessed: Oct. 04, 2020. Disponível em: <https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html>.

_____. **Introduction to RL**. 2018. Accessed: Oct. 01, 2020. Disponível em: <https://spinningup.openai.com/en/latest/spinningup/rl_intro.html>.

ANDERSSON, O.; WZOREK, M.; DOHERTY, P. Deep learning quadcopter control via risk-aware active learning. 2017.

BOHNHOFF, T. **A Five-Minute Guide to Artificial Intelligence**. 2019. Accessed: Sep. 30, 2020. Disponível em: <https://medium.com/appanion/a-five-minute-guide-to-artificial-intelligence-c4262be85fd3>.

BURKOV, A. **The Hundred-Page Machine Learning Book**. [S.l.]: Andriy Burkov, 2019. ISBN 9781999579500.

DRL. **AI Robotic Racing**. 2019. Accessed: Sep. 29, 2020. Disponível em: <https://thedroneracingleague.com/airr/>.

ETKIN, B.; REID, L. D. **Dynamics of Flight - Stability and Control**. [S.l.]: John Wiley & Sons, Inc., 1996.

FEIST, J. **Autonomous drone vs self-flying drones, what's the difference?** 2020. Accessed: Sep. 29, 2020. Disponível em: <https://dronerush.com/autonomous-drone-vs-self-flying-drones-10653/>.

GU, S. et al. Continuous deep q-learning with model-based acceleration. 2016.

IOFFE, S.; SZEGEDY, C. Batch normalization: Accelerating deep network training byreducing internal covariate shift. 2015.

JAISWAL, P. **Demystifying Drone Dynamics!** 2018. Accessed: Oct. 06, 2020. Disponível em: <https://towardsdatascience.com/demystifying-drone-dynamics-ee98b1ba882f>.

JOSHI, D. **Drone technology uses and applications for commercial, industrial and military drones in 2020 and the future**. 2019. Accessed: Sep. 28, 2020. Disponível em: <https://www.businessinsider.com/drone-technology-uses-applications>.

JUNG, S. et al. Perception, guidance, and navigation for indoor autonomous drone racing using deep learning. **IEEE Robotics and Automation Letters**, v. 3, 2018.

KAUFMANN, E. et al. Deep drone racing: Learning agile flight in dynamic environments. **Conference on Robotic Learning (CoRL)**, 2018.

KIM, D. K.; CHEN, T. Deep neural network for real-time autonomous indoor navigation. 2015.

KINGMA, D.; BA, J. Adam: A method for stochastic optimization. 2014.

KULINA, M. et al. A survey on machine learning-based performance improvement of wireless networks: Phy, mac and network layer. 2020.

LILLICRAP, T. P. et al. Continuous control with deep reinforcement learning. 2019.

MUELLER, M. et al. Teaching uavs to race: End-to-end regression of agile controls in simulation. 2017.

ROSKAM, J. **Airplane Flight Dynamics and Automatic Flight Controls**. [S.l.]: Roskam Aviation and Engineering Corporation, 1979.

ROSS, S. et al. Learning monocular reactive uav control in cluttered natural environments. 2012.

SADEGHI, F.; LEVINE, S. Cad2rl: Real single-image flight without a single real image. 2017.

SMOLYANSKIY, N. et al. Toward low-flying autonomous mav trail navigation using deep neural networks for environmental awareness. 2017.

TABOR, P. **Reinforcement Learning in Continuous Action Spaces | DDPG Tutorial (Tensorflow)**. 2019. Accessed: Oct. 06, 2020. Disponível em: <https://www.youtube.com/watch?v=vcGv5vmOydc>.

UHLENBECK, G. E.; ORNSTEIN, L. S. On the theory of the brownian motion. **Physical review**, 1930.

WIKIPEDIA. **Drone racing**. 2020. Accessed: Sep. 29, 2020. Disponível em: <https://en.wikipedia.org/wiki/Drone_racing>.

______. **Drone Racing League**. 2020. Accessed: Sep. 29, 2020. Disponível em: <https://en.wikipedia.org/wiki/Drone_Racing_League>.

______. **Finite difference coefficient**. 2020. Accessed: Oct. 08, 2020. Disponível em: <https://en.wikipedia.org/wiki/Finite_difference_coefficient>.

______. **History of unmanned aerial vehicles**. 2020. Accessed: Sep. 28, 2020. Disponível em: <https://en.wikipedia.org/wiki/History_of_unmanned_aerial_vehicles>.

______. **Ornstein–Uhlenbeck process**. 2020. Accessed: Oct. 06, 2020. Disponível em: <https://en.wikipedia.org/wiki/Ornstein\T1\textendashUhlenbeck_process>.

WIRE, B. **Drone Market to Grow from $22.5 Billion in 2020 to Over $42.8 Billion by 2025, at a CAGR of 13.8%**. 2020. Accessed: Sep. 28, 2020. Disponível em: <https://www.businesswire.com/news/home/20200721005593/en/Drone-Market-to-Grow-from-22.5-Billion-in-2020-to-Over-42.8-Billion-by-2025-at-a-CAGR-of-13.8---ResearchAndMarkets.com>.

YOON, C. **Deep Deterministic Policy Gradients Explained**. 2019. Accessed: Oct. 05, 2020. Disponível em: <https://towardsdatascience.com/deep-deterministic-policy-gradients-explained-2d94655a9b7b>.

ZHANG, T. et al. Learning deep control policies for autonomous aerial vehicles with mpc-guided policy search. 2016.