POLITECNICO DI TORINO

Master's Degree in Computer Science Engineering

Master's Degree Thesis

AUTONOMOUS LUNAR LANDER
Deep Reinforcement Learning for Control Application

Supervisors

Prof.ssa Elena BARALIS

Dott. Luca ROMANELLI

Dott. Mattia VARILE

Dott. Lorenzo FERUGLIO

Candidate

Matteo STOISA

December 2021

Abstract

This thesis aims to analyze the application of a Deep Reinforcement Learning algorithm to a case study in the field of control. The algorithm chosen is the *Proximal Policy Optimization*, which in recent years has reached the state-of-the-art in various fields of application, the control problem taken under consideration is the powered descent phase of a lander and the subsequent pinpoint landing, the Apollo 11 mission was taken as guideline for some aspects in the creation of the model. The framework *Unity* was used for the modeling and simulation part, the library *ML-Agents* for the management of the DRL part. The implementation of the fidelity of the model has been undertaken with an incremental approach, this has allowed to understand and face gradually the criticalities since the increasing of the physical complexity of the problem grows notably the difficulty in reaching the desired result. The main problems faced and analyzed deal with the interaction between the physical model, the reward function through which the agent learns and the numerous hyperparameters that manage the PPO algorithm. The main features implemented in the first three scenarios carried out are realistic and randomized initial conditions, realistic landing constraints, limited fuel and mass loss caused by its consumption. In these three scenarios, the movement constraint in only three degrees-of-freedom is present as the main simplification; agents with success rates higher than 90% were obtained. In the fourth and last scenario the lander has the possibility to move in six degrees-of-freedom, here the best policy obtained has about 90% of accuracy, the conditions in which the lander fails are half of mild severity and half disastrous. Analyses were also conducted on the inference phase affected by observation noise or physical alterations. The major contribution of this work is considered to be the analysis of the hyperparameters tuning of the PPO algorithm and its benchmarks, the complex structuring of the reward function combined with strategies applied during training that have determined its effectiveness, all related to different and comparable physical conditions. Overall, the results achieved are considered satisfactory, may represent a guideline to implement this methodology in other similar applications, or to continue the development of this case study.

Riassunto

La tesi si prefigge di analizzare l'applicazione di un algoritmo di Deep Reinforcement Learning ad un caso studio nel campo del controllo. L'algoritmo scelto è il *Proximal Policy Optimization*, che negli ultimi anni ha raggiunto lo stato dell'arte in svariati campi di applicazione; il problema di controllo preso in esame è la fase di discesa con propulsione di un lander e il successivo atterraggio in una zona prefissata, la missione Apollo 11 è stata assunta come linea guida per alcuni aspetti nella creazione del modello. È stato utilizzato il framework *Unity* per la parte di modellizzazione e simulazione, la libreria *ML-Agents* per la gestione della parte di DRL. L'implementazione della fedeltá del modello è stata intrapresa con un approccio incrementale, ciò ha permesso di comprendere ed affrontare gradualmente le criticitá dato che al crescere della complessitá fisica del problema cresce notevolmente la difficoltá nel raggiungere il risultato desiderato. I problemi principali affrontati ed analizzati trattano l'interazione tra il modello fisico, la reward function mediante cui l'agente impara ed i numerosi parametri che gestiscono l'algoritmo PPO. Le principali caratteristiche implementate nei primi tre scenari realizzati sono le condizioni iniziali realistiche e randomicizzate, i vincoli di atterraggio realistici, il carburante limitato e la perdita di massa causata dal suo consumo. In questi tre scenari è presente come principale semplificazione il vincolo del movimento in soli tre gradi di libertá; sono stati ottenuti agenti con percentuale di successo maggiore del 90%. Nel quarto ed ultimo scenario realizzato il lander ha la possibilitá di muoversi in sei gradi di libertá, qui la migliore policy ottenuta ha circa il 90% di accuratezza, le condizioni in cui il lander fallisce sono da ritenersi per metá di lieve gravitá e per metá disastrose. Sono inoltre state condotte analisi sulla fase di inferenza affetta da rumori di osservazione o alterazioni fisiche. Si ritiene che il maggior contributo costituito da questo lavoro sia l'analisi del tuning dei parametri dell'algoritmo PPO ed i relativi benchmark, la strutturazione complessa della reward function unita a strategie applicate durante il training che ne hanno determinato l'efficacia, il tutto relazionato a condizioni fisiche differenti e comparabili. Nel complesso, i risultati raggiunti sono considerati soddisfacenti, possono rappresentare una linea guida per implementare questa metologia in altre applicazioni analoghe, o per continuare lo sviluppo di questo caso studio.

Ringraziamenti

Raggiunto questo traguardo della mia Vita, non posso non cogliere l'occasione di esprimere la mia più sentita gratitudine nei confronti di coloro i quali, giorno dopo giorno, hanno contribuito a definire la Persona che sono.

Desidero ringraziare in primo luogo i miei genitori, Rosa e Luigi, i quali mi hanno insegnato i valori dell'impegno e della dedizione, e che costantemente mi accompagnano e sostengono affinché persegua i miei obiettivi, i miei sogni. Insieme a loro, ringrazio Sara ed Emanuele, sorella e fratello maggiori, saldo punto di riferimento su cui posso far sempre affidamento.

Desidero inoltre esprimere la mia gratitudine nei confronti di tutte le persone che hanno fatto o fanno parte della mia vita, ciascuna in misura e maniera peculiare, speciale, ed in particolare nei confronti dei miei Amici, che siano essi di lunga data o novizi, affettuosi o spigolosi, omogenei o irregolari, sporadici o assidui; con loro tutti ho condiviso in questi anni infiniti momenti di spensieratezza e gioia, difficoltà e tristezza, e soddisfazione, frustrazione, ilarità e disperazione, ed odio, Amore, ed infiniti ancora ne condivideremo.

Infine, un sentito ringraziamento va alla mia relatrice e a tutti i formidabili professionisti del team di AIKO, che mi hanno offerto l'opportunità di coronare questo percorso con il connubio delle passioni per l'intelligenza artificiale e per lo spazio.

Table of Contents

List of Tables

List of Figures

Acronyms

A2C Advantage Actor Critic

A3C Asynchronous Advantage Actor Critic

AI Artificial Intelligence

CNN Convolutional Neural Network

DDPG Deep Deterministic Policy Gradients

DOF Degrees of freedom

DL Deep Learning

DQN Deep Q Networks

DRL Deep Reinforcement Learning

DUT Deny Unsuccessful Terminations

FPS Frame Per Second

FSO Forced Successful Observation

GAE Generalized Advantage Estimation

MDP Markov Decision Process

ML Machine Learning

MLP Multi-Layer Perceptron

NN Neural Network

PPO Proximal Policy Optimization

RCS Reaction Control System

RIP Reverse Incremental Progression

RL Reinforcement Learning

SAC Soft Actor-Critic

SDG Stochastic Gradient Descent

SI International System of Units

TRPO Thrust Region Policy Optimization

UI User Interface

Chapter 1

Introduction

This thesis was entirely carried out in collaboration with the company *AIKO - Autonomous Space Missions*, both are a marriage of a passion for deep learning and aerospace.

1.1 Reinforcement Learning for Control

The trial-error approach is the mechanism through which we all have learned to act and pursue our goals within the world around us. Limiting the focus only to the field of the physical world, for example, starting as newborns we learned to control our body and moves understanding their functioning through direct integration with what surrounds us. The artificial intelligence technique called Reinforcement Learning exploits exactly this mechanism as a learning method, these algorithms are able to train entities capable of pursuing objectives starting from the interaction with the environment. The real world is one of the most intuitive and immediate types of environments where one of this agent could live, and indeed it is where many recent studies have attempted to apply the approach. The modeling of the physic, indeed, is a problem that has already extensively studied and implemented, therefore the simulation model used to train the algorithm could be very accurate, enough to potentially deploy the agent in the real world.

In particular, the Reinforcement Learning approach fits perfectly the challenging task of guidance and control of autonomous vehicles. Here the agent choose actions that could control vehicle devices such as engines or wheels on the basis of observations that could be measurements coming from sensors or even frames acquired through cameras. In the aerospace field, a typical example of goal to be pursued is the control of a lander in the landing phase. This is the case study chosen to implement a DRL for control proof-of-concept. Besides being a well known and

established space control problem, on which it is therefore possible to find specific documentation, this choice was inspired by [1].

The landings made so far on the moon had windows of accuracy in the order of magnitude of kilometers, but the pinpoint touchdown achievement will have crucial importance for space applications in the near future. In future missions on Mars, for example, a greater landing accuracy will allow to decide in advance the site that presents the best characteristics thus the risk of running into adverse surfaces will be reduced, moreover landing directly near settlements or sites of specific interest would avoid the journey to reach them, a potential source of complications.

Regarding the methodologies that are currently used to actually solve the landing control problem, [1] explains that the majority of proposed powered decent phase guidance and control systems use two separate and independently optimized systems. Through sensor measurements, the navigation system estimates the spacecraft's state and passes this estimate to the guidance system. This calculates a trajectory that in general specifies the lander's target state as a function of time. The trajectory is then passed to the control system that tracks the trajectory by determining how to control the thrusters. This important study addresses the problem of pinpoint touchdown, and the optimization of the fuel used, exploiting DRL to control the lander so that it follows the trajectory and precalculated characteristics that an optimal landing should have, that is, using reinforcement learning instead of the traditional control system.

In this thesis instead, the learning approach will be exploited leaving the algorithm full freedom of how to find the solution to achieve the goal. In addition to the touchdown achievement, it is possible to pursue the crucial problem of optimizing the use of time and resources. Unlike traditional guidance-control methods, exploiting a DRL agent offers the benefit that once trained the deep neural network provides control at the cost of its access time only, without the need of real-time calculus, so that the action update frequency could potentially be increased. Furthermore, the stochastic approach used by deep learning is preferable than deterministic methods, it has already proved to be more robust to anomalies or noises that are always possible.

1.2 Objectives of the Thesis

The overall goal of this thesis is to analyze the application of a specific Deep Reinforcement Learning algorithm to a specific control problem. The technologies used as starting point are the generic reinforcement learning approach [2], the DRL

state-of-the-art *Proximal Policy Optimization* algorithm [3] implemented in the *ML-Agents* library [4] and the *Unity* framework [5], in which the physical model was created from scratch. The chosen control problem is the powered descent phase and the following pinpoint touchdown of a lunar lander, whose characteristics are inspired by the *Apollo 11* moon landing [6]. The ultimate goal is to obtain an Artificial Intelligence capable of controlling the vehicle in a fully autonomous manner, capable of completing the landing with a reasonable high percentage of success, in a physical simulation of reasonable fidelity that in particular is in six degrees of freedom.

The correct setting of the hyperparameters that determine the proper algorithm learning and the modality in which the RL algorithm operates represent the greatest challenge in dealing with this type of problem. It could be very difficult to deal with them due to the multitude of aspects to be set, a grid search is practically impossible and only the experience in the field can lead to success. For this reason the incremental development and resolution approach was of fundamental importance and it is strongly recommended when facing similar cases. Based on these considerations, given the experimental nature of the application, all the observations that gradually emerged both on the methodology used and on the application of the approach itself could result extremely valuable, and can be obviously considered as part of the objective.

1.3 Structure of the Thesis

The thesis is structured as follows:

- **1. Introduction**: which is this, a generic introduction is set out as well as the main objectives of the work.
- **2. The reinforcement learning approach**: in this chapter the reinforcement learning framework is explained as a generic AI application, first at a high level and then in a formally rigorous way; finally a summary classification of different types of RL algorithms is exposed.
- **3. Deep Reinforcement Learning**: in the first part of this chapter the general operation of deep neural networks is described, then it is explained in detail the functioning of the PPO algorithm and of the parameters that control it, as widely used throughout the entire work.
- **4. Framework, Library and Implementation**: in this chapter the two technologies that have been mainly exploited are described: the framework Unity and the library ML-Agents; subsequently the common characteristics that constitute the base of the implemented physical models will be outlined.

- **5. Autonomous Lunar Lander: 3-DOF scenario Version 1**: this is the first scenario with extremely simplified characteristics (simple collision with the target). The four chapters describing the implementation and resolution of a scenario present first of all the formal description of the problem and of the physical model, then the methodology and the reasoning that emerged applying the RL approach, and finally the solution reached and its analysis.
- **6. Autonomous Lunar Lander: 3-DOF scenario Version 2**: in the second scenario more realistic initial conditions and landing constraints have been implemented. Numerous non-trivial observations emerged already here, these are reported throughout the thesis in appropriate subsections boxed and identified by Greek letters.
- **7. Autonomous Lunar Lander: 3-DOF scenario Version 3**: it is the last scenario in three degrees of freedom, here a system of fuel consumption and relative non-constant mass has been implemented. It was also theorized and tested the first so-called "training strategy", i.e. an unconventional approach that aims to improve performance during the training phase.
- **8. Training Benchmarks Analysis**: here are described and analyzed numerous procedures of training carried out as benchmark purpose, they have the finality to study performances to varying of numerous parameters, in an isolated form that is better analyzable than in the other chapters.
- **9. Autonomous Lunar Lander: 6-DOF scenario**: this is the last scenario implemented, in six degrees of freedom. After the definition of the problem and the description of the approach that led to the resolution, more detailed analysis are carried out for both statistics of success and failure, and also on the robustness of the deep neural network. Several useful observations and strategies emerge in this chapter as well.
- **10. Conclusions**: in the end some final considerations will be presented, together the possible inspirations that could be used as starting points for future works.
- **A. Summary Tables**: here are the summary tables of the characteristics of each of the four scenarios implemented and solved, grouped by physical model, RL prarameters, PPO hyperparameters and reward function.
- **B. External References**: here are some links to external resources.

Chapter 2

The Reinforcement Learning approach

Together with Supervised and Unsupervised Learning, Reinforcement Learning is an Artificial Intelligence learning technique that has the peculiarity of exploiting direct interaction with the environment. In this chapter its fundamental concepts will be presented, as well as the typology of problems to which this approach is applicable. It is made extensive reference to [2] as an established theoretical basis of the argument.

The term "Reinforcement Learning" represents the process of solving a task by progressively improving the choice of decisions that lead to it. Let's imagine a generic entity that lives in a generic environment and receives input from it, this entity has the capability and possibility to make choices that determine changes of the surrounding environment and of itself, it also has the intrinsic will to pursue non-trivial objectives or solve more or less complex tasks. The most natural and intuitive method of understanding which is the best action to achieve these goals is to make repeated attempts and modify the behavior on the basis of the effects they have, the actions that have unsuccessful consequences will be progressively avoided while the actions that lead to benefits will be selected more frequently. At the end of a successful learning process the entity will have consolidated a behavior capable of bringing it to its goals.

2.1 Trial-Error Example Scenarios

The generic scenario briefly summarized above can represent countless situations, here are some significant examples:

- a newborn gradually learns to relieve the feeling of hunger by eating, or that it is preferable to avoid touching an open flame otherwise he feels pain. Hunger and pain are intrinsic sensations that he can not modify or avoid, but he can modify his own behavior to alleviate or avoid them.
- a professional chess player decides how to move his pieces in order to take the lead on the chessboard. He tries to do it both in the short term by trying to capture opposing pieces, and in the long term by following a strategy and predicting possible opponent moves, pursuing the ultimate goal of checkmate.
- a broker buys or sells stocks by predicting the future stock market trend in order to maximize monetary profit, to do this he has to take many factors into consideration.
- an Olympic athlete trains to optimize muscle movements in order to lift as much weight as possible, at the beginning of his career he lifts little weight but gradually improves.
- an adult has learned to suppress the urge to perform acts that would satisfy him in the short term but that would be negative in the long term, such as stealing.

The examples cited above, although very different from each other, share some key characteristics and mechanisms that are the basis of reinforcement learning, they are described below.

2.2 Key Elements of Reinforcement Learning

The following components are the basic actors in a generic reinforcement learning application scenario, as already said they are abstract and applicable to disparate fields, the scenario of the chess player is carried forward as main example:

- The **agent** is the protagonist entity, it takes vision of the state of the environment, on the basis of it decides which action to take and performs it. In our case it is the chess player himself.
- The **environment** is the place where the agent acts, it can be physical or abstract. It is governed by defined rules that determine its evolution spontaneous or consequent to the actions of the agent, the rules of the game of chess in this case.
- In a reinforcement learning scenario we usually speak of an **episode** as the time enclosed between the start of the scenario and the achievement of a

termination condition which can be the goal or other irreversible states. even if less frequent and only mentioned below, Even if less frequent and only mentioned below, it is also possible to have scenarios without termination conditions in which the agent acts to the bitter end.

- Time is usually quantized in **steps**, not necessarily equidistant, at each step the learning loop explained below is carried out.

- The **state** is the input of any kind that the agent gets from the environment, it can also be called observation. In our example it is made up the positions of the pieces on the board, but also the time remaining on the clock of both players, and why not also by additional information such as the body language of the opponent and his elo.

- An **action** is the output performed by the agent that influence the environment, it can be chosen from a limited or continuous set of possibilities. In the case of a chess game, they are the moves that our player can make, or even ask for a draw.

- The **policy** defines the agent's behavior, it can be simplistically thought of as a function that associates an action to be performed to a given state. The objective of the learning process through reinforcement learning is to arrive at defining a policy capable of achieving its objective. In the case of the chess player, the policy is his brain, trained to make choices from all the games previously played and the games studied and observed.

- The **reward** is a signal received by the agent based on the state of the environment and the latest decision made, it reflects the positivity or negativity of the latter. The agent can not change the process that assign him the reward, he can just change behaviour. Indeed he uses the reward to understand if he is acting in the right way, during the learning process he modifies his behaviour in order to maximize the cumulative reward obtained during the episode. The reward function is the mechanism that decides which reward to assign, it is invisible to the agent, its structure is essential for him to learn correctly. In the case of the chess game, obviously the reward given by the final result of the game is fundamental, the periodic reward could be calculated taking into account factors such as the number of pieces captured and the portion of the board controlled.

- The **value function** instead represents the forecast of the rewards that will be accumulated in the future starting from the current state. It is in fact essential that the maximization of the reward is pursued not only in the immediate but also in the long term, as choices that are not excellent in the immediate

future could lead to better situations in the future. For example the chess player could sacrifice a piece following a strategy that will lead him to victory.

Having introduced the methodology approach and the types of problems that can be faced, let's now see two fascinating concepts that are fundamental for learning in general and therefore also for RL. Generally speaking, when you are faced with a choice, you usually want to choose the best of the possibilities to bring yourself into an ever better future situation and so on, choice after choice. Similarly to the real world, in terms of RL previously set out: the agent has the objective of choosing the best action in order to maximize the reward obtained. We must ask ourselves what is meant by *best* choice.

2.2.1 Exploration vs Exploitation

The evaluation of how much an action is positive or not compared to another is made on the basis of an estimate, this estimate derives from the experiences that have been previously lived. However, there is the risk that if we always pick the path that we consider the best, other better conditions than those already known could never be known. Consider that the absolute certainty that a choice is excellent can never exist, i.e. we can not know each outcome deriving from each possible choice (by definition of the stochastic and non-deterministic problem). But obviously this problem is more marked if past experience is limited. In other words, given the fact that we can not know everything, if we always choose what we consider the best we risk foreclosing even better possibilities. This dual concept translates into the need for a trade-off between exploration and exploitation:

- **exploration** consists in deliberately choosing an action that is currently not considered optimal, based on current experience or because it is not known at all. The action taken may be sub-optimal to some extent or even totally random. This behavior has the objective of evaluating his goodness and the hope of finding better conditions than those currently known. This approach is essential especially when the experience is little, that is at the beginning of the learning phase; on the other hand in critical phases it is obviously preferred to avoid random actions because they are much more risky.

- **exploitation**, in the opposite sense, consists in selecting only and only the choices that are considered excellent, fully trusting the own experience; we can call this action *greedy*. This behavior has to be preceded by a learning phase in which a fairly large experience has been accumulated, in this way the risk is minimized.

At the algorithmic level, in the training phase exploration is induced in a decreasing way: at the beginning a lot of exploration is carried out in order to test behaviors in large regions of action, the degree of exploration is gradually decreased since the agent stabilizes behaviors with better defined directions, up to the end of the training in which the exploration is canceled and the agent fully exploits the best behavior discovered. The simplest way to control the degree of exploration is through a coefficient that indicates the probability of making a random choice, this parameter is often indicated as ϵ (in this case we speak of ϵ-*greedy* policy, note that the `epsilon` hyperparameter described in Chapter 3 has a different meaning), usually with size between 0.15 and 0.3 that decays linearly. Obviously in the inference phase, which can correspond to the deployment of the agent in the real environment, only exploitation is used.

2.2.2 Foresight

Speaking of optimal solution in a problem with a certain duration, it is necessary to specify whether it is more important to maximize the gain in the long term or in the short term. In fact, the choice of a certain action could prove to be very advantageous in the immediate future but lead to future negative situations, or on the contrary a not optimal choice in the immediate could lead to much more advantageous situations later on. In general, this duality has to be managed on the basis of the problem that is being faced, i.e. on the basis of the implications that may derive from any specific applications. In the case of an episodic problems with a specific final goal, it makes sense to want to maximize the long-term reward if the final objective is more important than the behavior that leads to it, but there could be the risk of encountering the intermediate unsuccessful states (exactly as it happens in the case study that will be addressed). In the case of endless decision-making processes this sizing could be more difficult, but this type of scenario has not been analyzed here.

Within algorithms this trade-off is usually handled by the γ parameter, which is in charge of weighing the estimations of the rewards that will be obtained in the future. Its operation will be exposed in Section 2.5.

2.3 The Agent-Environment Interaction Loop

Putting these ingredients together we can now define in a more formal way how the interaction between agent and environment takes place. The environment and the agent interact at each discrete time step $t = 0, 1, 2, \ldots$. At each time step t, the agent receives the observation of the environment's state $S_t \in \mathcal{S}$, where $\mathcal{S}$ is the set of possible reachable states. The agent selects an action $A_t \in \mathcal{A}(S_t)$ where

$\mathcal{A}(S_t)$ is the set of actions available in state S_t. The next time step $t+1$ the agent receives a numerical reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$, it is based on the state that was also affected by the previous action. Then he observes the new state S_{t+1} and takes a new decision A_{t+1} and so on, as showed in Figure 2.1.

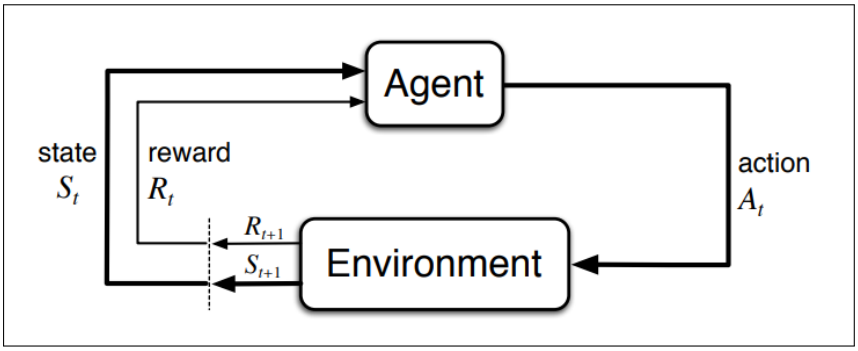


Figure 2.1: Environment-Agent interaction loop

The mapping from observations to probabilities of selecting each possible action is the agent's current policy and is denoted as π_t, where $\pi_t(a|s)$ is the probability that $A_t = a$ if $S_t = s$. It is important to specify that actions are the only way the agent has the possibility to interact, everything that is external cannot be modified in other ways. The agent may know nothing of the environment and its functioning as in the example of the infant, or he may have full knowledge of its behavior and how reward is assigned as in the example of the chess player. Knowing how the environment works does not mean knowing how to solve the task, just think of the famous Rubik's cube, whose easy-to-understand mechanism is extremely complex to solve.

RL methods specify how to modify and improve the policy on the basis of experience, that is on the basis of the rewards obtained. The criterion that is followed is to maximize the sum of all the rewards obtained during the entire episode.

2.4 Goals and Rewards

The purpose of the agent is formalized in terms of a special reward passing from the environment to him: at each time step t, the reward is a simple number given, $R_t \in \mathcal{R}$. Informally, the agent's goal is to maximize the total amount of reward received. This means maximizing not only the immediate reward, but even better

the cumulative reward in the long run. This informal idea can be clearly stated as the reward hypothesis, as reported by [2]:

"That all of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward)."

Thus the use of a reward signal to formalize the concept of a goal is another of the most distinctive characteristics of RL and one of its strengths, indeed translating a task into periodically assigned reward signals is a flexible system that allows to adapt the framework to disparate scenarios. As said, the reward is a generic integer $R \in \mathcal{R} \subset \mathbb{R}$, the logic of the reward function is the heart of a successful learning process: that is to assign a large or small, positive or negative reward based on the current state.

When deciding on the structure of the reward function it is important to keep in mind that the reward should suggest to the agent *what* you want him to achieve, and not *how* he should do it. This difference, that it may seem tiny, allows you to fully exploit the potential of RL, that is to avoid indicating the actions through the reward function but rather to let the algorithm find the best way to reach the goal established. It is often not trivial to structure the reward function so that the agent learns to reach the goal, in fact being able to maximize the rewards obtained may not coincide with reaching the goal, especially when dealing with complex tasks. Thinking back to the game of chess, if capturing a piece gave an excessively positive reward, the algorithm could easily converge to the behavior of capturing as many pieces as possible while ignoring the main goal of victory.

If for example the agent is a robot who has to learn to walk we could assign him a positive score every time he moves forward. If our agent is an entity that has to get out of a labyrinth we could assign him a single large positive reward when he finds the exit, and also a small negative reward at each step so that he learns to find it as quickly as possible. The relative size of the rewards must be correctly sized, indeed larger scores should have a greater impact on the learning process, incorrect sizing could strongly destabilize the learning process. This fundamental aspect however depends on the algorithm in charge of updating the policy, which we will deal with later.

2.5 Returns and Discounting

Let's now formally define what we have so far called the sum of the rewards obtained, that is what the algorithm will try to maximize: if the sequence of

rewards received starting from time step t is denoted $R_{t+1}, R_{t+2}, R_{t+3}, ...$, then we define as **expected reward** G_t the sum:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T \tag{2.1}$$

where T is the last step of the episode. This is true in the case of **episodic tasks**, i.e. episodes that end in a special state called the **terminal state**. We need to distinguish the set of all non-terminal states denoted with $\mathcal{S}$, from the set of all states plus the terminal state denoted with $\mathcal{S}^+$.

In non-episodic tasks instead, called **continuing tasks**, there were no terminal states, $T = \infty$ and consequently easily $G_t = \infty$. We need to add the additional concept of discounting, which consisted in weighing each member of the sum. The **expected discounted return** is defined as:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.2}$$

where γ is a parameter called discount rate, with $0 \leq \gamma \leq 1$. Consequently a reward received k time steps in the future is worth only γ^{k-1} times what it would be worth in the present. If $\gamma < 1$, the infinite sum converges to a finite value, the more γ is close to 1 the more the agent is "farsighted", i.e. it tries to maximize the reward even at a great distance in the future. On the contrary the more γ is close to 0 the more the agent is "myopic", that is, it gives much more importance to the gain in the immediate rather than in the long term; at the limit case of $\gamma = 0$, it aims to optimize only R_{t+1}.

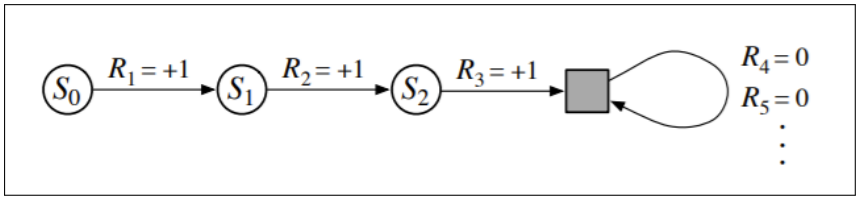


Figure 2.2: Aborting state loop

From here on we will consider the episodic task problem only, since it is mostly used and because it is the application of the case study analyzed. Equation 2.2 can be used in the case of episodic tasks assuming that when the terminal state is reached, a special **absorbing state** is entered where only zero rewards are awarded in it, as showed in Figure 2.2. We can then rewrite equation 2.2 as:

$$G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} \tag{2.3}$$

including the possibility that $T = \infty$ or $\gamma = 1$ (but not both).

2.6 The Markov Property

Now let's talk more deeply about the state, let's formally define a particular property of the environments and their state signals, called the Markov property. By "the state" we mean whatever information is provided to the agent. Since it is given by a processing system that is assumed to be part of the environment, we do not care how it is obtained or computed. The state can be made up of any type of data that can be encoded in a signal, it could include immediate sensations such as sensory measurements or more highly processed versions of original sensations, or even complex structures built up over time from the sequence of sensations.

Ideally, the best type of state signal we can have is a state that summarizes past sensations compactly, so that all relevant past information is summarized within it. This normally requires more than the immediate sensations, but never more than the complete history of all past sensations. An observation of this type is more onerous than the single observation of the present, but less onerous than the sum of all past observations. A state signal that succeeds in retaining all relevant information is said to have the Markov property, or be **Markov**. Returning to the example of the chessboard: the only current knowledge of the pieces would serve as a Markov state because it summarizes everything important about the complete sequence of changes that led to it. Much of the information about the sequence is not reported and cannot be reconstructed, but all that matters for the continuation of the episode is still present. Seen from another point of view, the principle can be formulated as "independence of path", because all that matters is in the current state signal; its meaning is independent of the history or "path" that have led up to it.

Consider how an environment may evolves at time $t + 1$ as result of the action taken at time t, in the most general and causal case this response might depend on everything that has happened earlier. In this case the dynamics can be defined only by specifying the complete probability distribution:

$$Pr\{R_{t+1} = r, S_{t+1} = s' | S_0, A_0, R_1, ..., S_{t-1}, A_{t-1}, R_t, S_t, A_t\} \tag{2.4}$$

for all r, s' and all the possible values of the past events: S_0, A_0, R_1, ..., S_{t-1}, A_{t-1}, R_t, S_t, A_t. On the other hand, if the state signal has the Markov property, then the environment's evolution at the step $t + 1$ depends only on the observation and action at t, in this case the environment's dynamics can be defined by specifying only as:

$$p(s', r|s, a) = Pr\{R_{t+1} = r, S_{t+1} = s' | S_t, A_t\} \tag{2.5}$$

for every r, s', S_t, A_t. Hence the environment and task as a whole own the Markov property if and only if occurs that Equation 2.4 is equal to Equation 2.5. If the environment has the Markov property we can predict the next state and expected next reward based on the only observation in the present, furthermore we can iterate this prediction to all future states and expected rewards.

If the environment does not have the Markov property, we can still approximate it to it, assuming that we can predict the next reward and the action to be selected on the basis of the current state. RL algorithms could be successfully applied even with this approximation, however the considerations made below are assumed to be valid for environments that possess the Markov property, as well as the case study addressed specifically.

2.7 Markov Decision Processes

We call *Markov decision process*, or *MDP*, a reinforcement learning task that has the Markov property, if the action and state spaces are finite then it is called *finite Markov decision processes*. A finite MDP is defined by his state and action spaces and by the dynamics of the environment. Given a state s and an action a, the probability of each possible pair of next state s' and reward r is:

$$p(s', r|s, a) = Pr\{S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a\} \tag{2.6}$$

Starting from the dynamics specified by Equation 2.6, we can compute anything else about the environment:

- the expected rewards for the state–action pairs:

$$r(s, a) = \mathbb{E}\Big[R_{t+1}\Big|S_t = s, A_t = a\Big] = \sum_{r\in\mathcal{R}} r \sum_{s'\in\mathcal{S}} p(s', r|s, a) \tag{2.7}$$

- the state-transition probabilities:

$$p(s'|s, a) = Pr\{S_{t+1} = s'|S_t = s, A_t = a\} = \sum_{r\in\mathcal{R}} p(s', r|s, a) \tag{2.8}$$

- and the expected rewards for the triples state–action–next-state triples:

$$r(s, a, s') = \mathbb{E}\Big[R_{t+1}\Big|S_t = s, A_t = a, S_{t+1} = s'\Big] = \frac{\sum_{r\in\mathcal{R}} rp(s', r|s, a)}{p(s'|s, a)} \tag{2.9}$$

2.8 Value Functions

Almost every RL algorithm involves estimating *value functions*: functions that estimate how good it is for the agent to be in a certain state, or how good it is to choose an action given the state. With "how good" we identify the future rewards that can be expected, i.e. the expected return. Since rewards depend on actions that will be taken, value functions are defined with respect to particular policies π: where the probability $\pi(s, a)$ is the probability to choose the action $a \in \mathcal{A}(s)$ being in state $s \in \mathcal{S}$.

We can informally say that the value of a state s under a certain policy π is the expected return when starting in s and following π thereafter, denoted with $v_\pi(s)$. It is formally defined as:

$$v_\pi(s) = \mathbb{E}\Big[G_t \Big| S_t = s\Big] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right] \tag{2.10}$$

where the notation $\mathbb{E}_\pi[\cdot]$ identifies the expected value of a random variable given that the agent is following policy π, regardless of the time step t. From here on we will call v_π the *state-value function for the policy* π.

Similarly, we now define the value of taking an action a in the state s following a policy π, denoted $q_\pi(s, a)$:

$$q_\pi(s, a) = \mathbb{E}\Big[G_t \Big| S_t = s, A_t = a\Big] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right] \tag{2.11}$$

q_π is the *action-value function for policy* π.

A fundamental property of value functions used throughout RL is that they satisfy particular recursive relationships. For any state s and any policy π, the following consistency condition holds between the value of s and the value of its

possible successor states s':

$$\begin{aligned} v_\pi(s) &= \mathbb{E}\Big[G_t \Big| S_t = s\Big] \\ &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \middle| S_t = s\right] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \middle| S_{t+1} = s'\right]\right] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \Big[r + \gamma v_\pi(s')\Big] \end{aligned} \tag{2.12}$$

The final expression is basically a sum over all values of the variables a, s', r. For each triple, it computes its probability $\pi(a|s)p(s', r|s, a)$, weights the quantity in brackets by that probability, then sums over all possibilities to get an expected value. The result of Equation 2.12 expresses the relationship between the value of a state s and the values of its successor states s', it is called the *Bellman equation for* v_π. Its solution v_π is unique, this equation forms the basis of many ways to compute, approximate, and learn v_π.

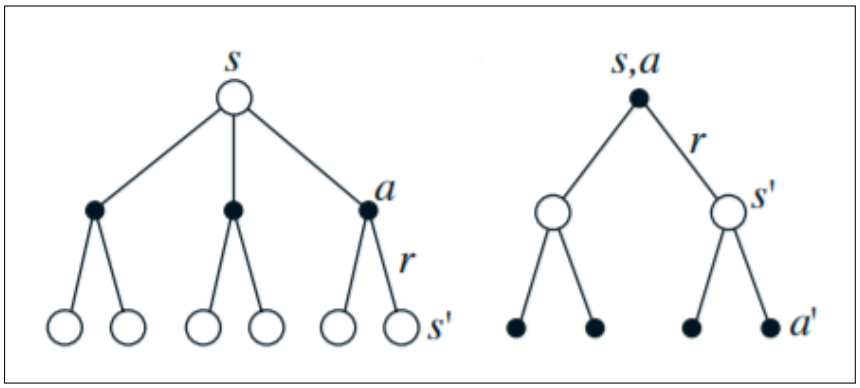


Figure 2.3: Backup diagrams for v_π (left) and q_π (right)

Think of being in a state s and looking ahead to its possible successor states s', as showed by the left Figure 2.3. Open circles represent the states and solid circles represent state–action pairs. Starting from the root node at the top, s, the agent can choose any action a (three possible actions showed in this tree). From each of these, the environment will respond with one of several next states s', and will accordingly assign a reward r. The Bellman equation 2.12 averages over all the

possibilities, weighting each by its probability of occurring. It states that the value of the start state must equal the (discounted) value of the expected next state, plus the reward expected along the way.

Diagrams like those shown in Figure 2.3 are called backup diagrams because they show the relationships that form the basis of the update, or backup, operations that are at the heart of RL algorithms. These methods transfer value information back to a state (or a state–action pair) from its successor states (or state–action pairs).

2.8.1 Optimal Value Functions

The aim of reinforcement learning is, roughly, to find the policy that manages to accumulate the greatest possible amount of reward and to solve the task in charge. In the case of finite MDPs we can precisely define the optimal value function because value functions offer a partial ordering over policies. The policy π_1 is defined to be better than or equal to another policy π_2 if its expected return is greater than or equal to that of π_1, for all the possible states. In other words, $\pi_1 \geq \pi_2$ if and only if $v_{\pi_1} \geq v_{\pi_2}$ for all $s \in \mathcal{S}$. The *optimal policy* is the one that is better than or equal to all other policies, we denote it with with π_*. There may be more than one, but they share the same state-value function, called the optimal state-value function, denoted with v_*, and defined as:

$$v_*(s) = \max_{\pi} v_\pi(s) \tag{2.13}$$

for all $s \in \mathcal{S}$.

Optimal policies also share the same *optimal action-value function*, denoted q_*, and similarly defined as:

$$q_*(s, a) = \max_{\pi} q_\pi(s, a) \tag{2.14}$$

for all $s \in \mathcal{S}$ and for all $a \in \mathcal{A}(s)$. For the state–action pair (s, a), this function represents the expected return for choosing the action a in state s and thereafter following an optimal policy. Thus, v_* can be written in terms of q_* as follows:

$$q_*(s, a) = \mathbb{E}\Big[R_{t+1} + \gamma v_*(S_{t+1}|S_t = s, A_t = a\Big] \tag{2.15}$$

v_* must satisfy the selfconsistency condition expressed by the Bellman equation for state values (Equation 2.12), because it is the value function for a policy.

Furthermore, because it is the optimal value function, v_*'s consistency condition can be written in a special form without reference to any specific policy. In this way we obtain the Bellman equation for s_*, or the *Bellman optimality equation*. Intuitively, it expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state:

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\ &= \max_a \mathbb{E}_{\pi_*}\Big[G_t \Big| S_t, A_t = a\Big] \\ &= \max_a \mathbb{E}_{\pi_*}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t, A_t = a\right] \\ &= \max_a \mathbb{E}_{\pi_*}\left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \middle| S_t, A_t = a\right] \\ &= \max_a \mathbb{E}_{\pi_*}\Big[R_{t+1} + \gamma v_*(S_{t+1}) \Big| S_t, A_t = a\Big] \\ &= \max_{a \in \mathcal{A}(s)} \sum_{s',r} p(s', r|s, a)\Big[r + \gamma v_*(s')\Big] \end{aligned} \tag{2.16}$$

The last two equations are two forms of the Bellman optimality equation for v_*. Similarly, the *Bellman optimality equation for* q_* is:

$$\begin{aligned} q_*(s,a) &= \mathbb{E}\Big[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \Big| S_t, A_t = a\Big] \\ &= \sum_{s',r} p(s', r|s, a)\Big[r + \gamma \max_{a'} q_*(s', a')\Big] \end{aligned} \tag{2.17}$$

Figure 2.4 show graphically through backup diagrams the spans of future states and actions considered in the Bellman optimality equations for both v_* and q_*. The arcs at the agent's choice points to represent that the maximum over that choice is taken rather than the expected value given some policy.

For finite MDPs, the Bellman optimality equation (2.16) has a unique solution that is independent from the policy. Given N possible states, the Bellman optimality equation is actually a system of N equations in N unknowns, one for each state. Furthermore, if the dynamics of the environment are known, i.e. $p(s', r|s, a)$ is known for each state s and action a, then one can solve this system of equations for v_*. One can solve a related set of equations for q_*.

It is relatively easy to determine an optimal policy, once we have v_*. For each state s, there will be one or more actions at which the maximum is obtained in the Bellman optimality equation. Any policy that assigns nonzero probability only to these actions is an optimal policy. It can be seen as a one-step search: if you have

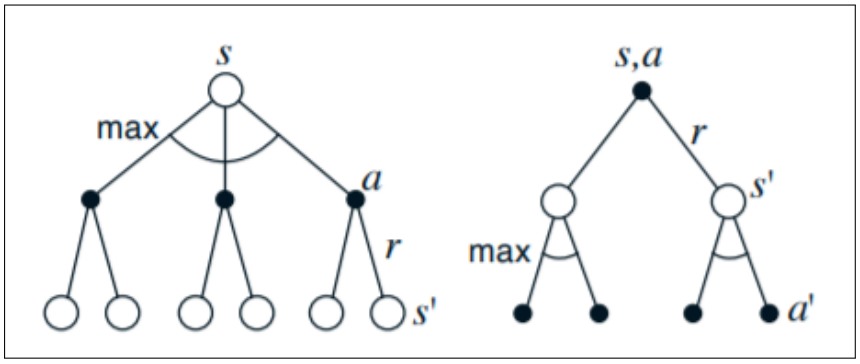


Figure 2.4: Optimization backup diagrams for v_* (left) and q_* (right)

the optimal value function v_* then the actions that appear best after a one-step search will be optimal actions. We can formulate the same concept by introducing the definition of greed: an optimal policy is a policy that is *greedy* with respect to the optimal evaluation function v_*. The term greedy is used in this context to describe policies that select actions based only on their short-term consequences. This means it describes any search or decision procedure that selects alternatives based only on local or immediate considerations, and without considering the possibility that such a selection may prevent future access to even better alternatives. If one uses v_* to evaluate the short-term consequences of actions then a greedy policy is actually optimal also in the long term sense, this is because v_* already takes into account the reward consequences of all possible future behavior. Hence, a one-step-ahead search yields the long-term optimal actions.

Choosing optimal actions is still easier when we know q_*: for any state s, the agent can directly find any action that maximizes $q_*(s, a)$. The action-value function effectively caches the results of all one-step-ahead searches. It provides the optimal expected long-term return as a value that is locally and immediately available for each state–action pair. Hence, the optimal action-value function allows optimal actions to be selected without having to know anything about possible successor states and their values, that is, without having to know anything about the environment's dynamics.

We have defined what optimal value functions and optimal policies are, of course we ideally want our agent to learn an optimal policy. This is very rare, in fact it is almost always impossible to calculate an optimal policy by simply solving the Bellman optimality equation. For the tasks we are usually interested in, optimal policies can be generated only with extreme computational cost. The computational power available is a major problem, in particular, the amount of computation that

the agent can perform in a single time step. The memory available is also an important constraint. A large amount of memory is often required to build up approximations of value functions, policies, and models. For this reason functions must be approximated, using some sort of more compact parameterized function representation.

2.9 RL Algorithms Main Distinctions

Now that we have defined the formal approach to MDP problems, let's see a non-exhaustive classification of the main characteristics of different RL algorithms (some of them exploit neural networks, but this topic will be introduced in the next chapter). The most important distinction in the classification is whether the agent has access or not to a *model* of the environment, it could be provided or learned. By a model we mean a function which predicts rewards and state transitions.

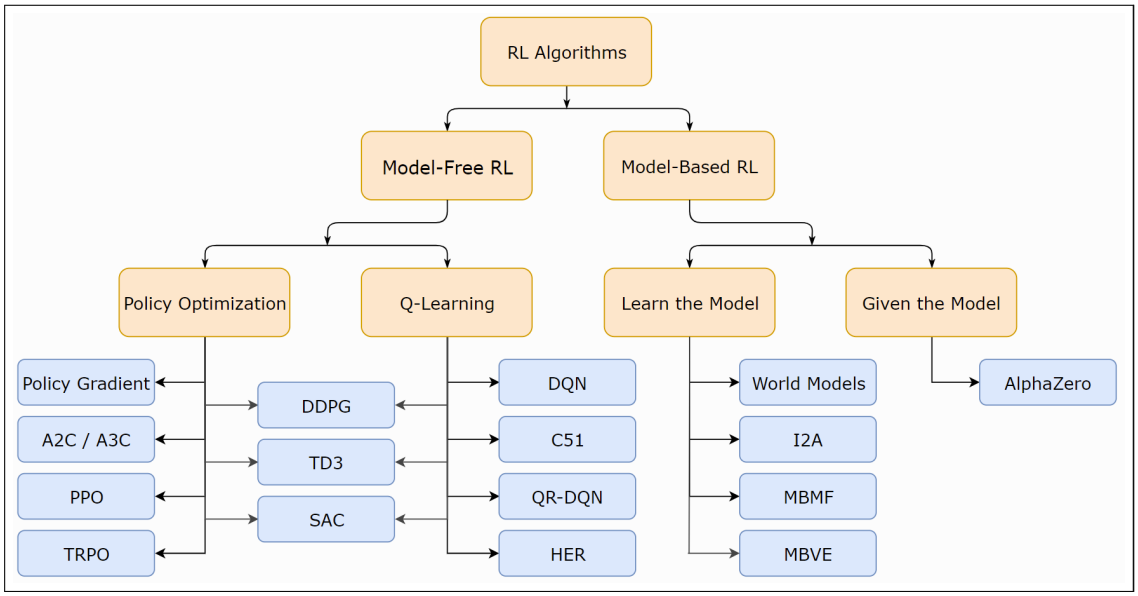


Figure 2.5: Non-exhaustive taxonomy of algorithms in modern RL

The main benefit when having the model is that it allows the agent to plan by thinking ahead, seeing what would happen for a range of possible choices, and explicitly deciding between its options. Agents can then distill the results from planning ahead into a learned policy. However, the main downside is that a ground-truth model of the environment is usually not available, the agent has to learn it purely from experience. Model-learning is fundamentally hard, the biggest

challenge is that bias in the model can be exploited by the agent. This can lead to the risk that the agent performs well with respect to the learned model, but behaves sub-optimally (or even terribly) in the real environment.

The algorithms which use a model are called *model-based methods*, and those that don't are called *model-free methods*. The latter are notoriously more popular and have been more extensively developed and tested since they tend to be easier to implement and tune.

Another fundamental distinction in the types of RL algorithms lies in what is learned, it can be one of the following:

- policies, they can be either stochastic or deterministic
- action-value functions (called Q-functions)
- value functions
- environment models

The main characteristics of the two main clusters that make up the model-free category are summarized below, the description of model-based algorithms is omitted as they are very different from each other and of less interest for this study.

2.9.1 Model-Free Algorithms

For model-free algorithms, the two main branches are:

- **Policy Optimization**. This family of methods represent explicitly a policy as $\pi_\theta(a|s)$, where θ is the set of parameters that characterize it. They indirectly optimize the parameters θ by maximizing local approximations of $L(\pi_\theta)$, or else they optimize it directly by gradient ascent on the performance objective $L(\pi_\theta)$. This optimization is almost always performed *on-policy*, which means only one policy is maintained and improved, and that each update only uses data collected while acting using the most recent version of it. This methods also usually involve learning an approximator function $v_\phi(s)$ for the on-policy value function $v^\pi(s)$, which gets used to understand how to update the policy.

  The most significant and successful example of this category of algorithms is the one called *Proximal Policy Optimization* [3], abbreviated PPO, it will be used as the DRL method for the application case examined and described in detail in the Section 3.2. Another important algorithm is the A2C/A3C [7], which maximize performance directly performing gradient ascent.

- **Q-Learning**. Algorithms of this branch learn an approximator $Q_\theta(s, a)$ for the optimal action-value function, $Q^*(s, a)$. They usually use an objective function based on the Bellman equation. As opposed to Policy Optimization algorithmshis, in this case the optimization is almost always performed off-policy, which means that each update can use data collected at any point during training, regardless of how the agent was choosing to explore the environment when the data was obtained. The corresponding policy is obtained via the connection between Q^* and π^*: the actions taken by the Q-learning agent are given by

$$a(s) = \arg\max_a Q_\theta(s, a) \tag{2.18}$$

  Examples of Q-learning algorithms include the DQN algorithm [8], from which many variations have originated since it is substantially the starting point of DRL, explained in Chapter 3; and the C51 algorithm [9], a variant that learns a distribution over return whose expectation is Q^*.

The main strength of policy optimization algorithms is that they directly optimize the thing you want. This tends to make them stable and reliable. On the other side, Q-learning methods tend to be less stable as they train Q_θ to satisfy a self-consistency equation, optimizing the agent performance only indirectly. But, when they do work Q-learning methods gain the advantage of being substantially more sample efficient, because of reusing data more effectively.

The two approaches are not totally incompatible and there exist a range of algorithms that live in between the two extremes, they able to carefully trade-off between the strengths and weaknesses of either side. Examples include the DDPG algorithm [10], which learns a deterministic policy and a Q-function by using the former to improve the latter; SAC algorithm [11], a variant which uses stochastic policies, entropy regularization, and a other tricks to stabilize learning and score higher than DDPG on standard benchmarks, moreover this is an *off-policy* algorithm which means it can learn from experiences collected at any time during the past.

Chapter 3

Deep Reinforcement Learning

Chapter 2 analyzed the generic structure with which the reinforcement learning framework allows to face any problem that can be modeled as a Markov Decision Process, as extensively described in [2] and [12]. However, the heart of an RL algorithm is the model with which the policy is implemented and how the policy is improved in order to reach an optimal policy approximation. Starting from the early 1980s, in conjunction with the advent of Deep Learning (generally more known for Supervised and Unsupervised Learning) a large number of DRL algorithms have been developed, i.e.RL algorithms which exploit a deep neural network as kernel of the agent.

In the first part of this chapter will be exposed a brief description of DL; in the second part the Proximal Policy Optimization algorithm will be analyzed in detail since widely used throughout the entire work.

3.1 Introduction to Deep Learning

. As resumed in [13], the main concept of Deep Learning relies in exploiting a non-linear function $f : X \to Y$ whose functioning is determined by $\theta \in \mathbb{R}^{n_\theta}$ parameters, with $n_\theta \in \mathbb{N}$:

$$y = f(x, \theta) \tag{3.1}$$

At a very high level we could think of it as a black box that given a vector of input values x provides a vector of output values y determined on the basis of

its current internal configuration θ; the process of updating the internal parameters that gradually lead to better performance represents the "learning" phase, the pure use of an already trained model is the inference phase. Internally, the structure of a deep neural network is composed by the succession of multiple processing layers (Figure 3.1), each of these applies a non-linear transformation and the sequence of these transformations leads to learning different levels of abstraction.

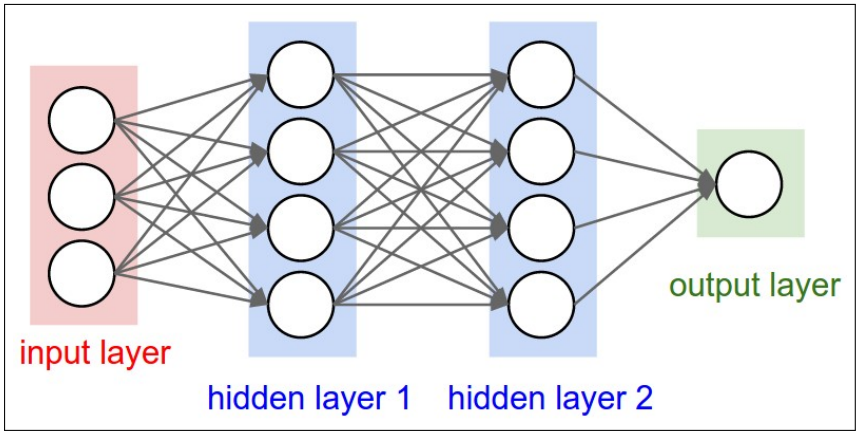


Figure 3.1: Internal representation of a MLP with two hidden layers

Let's examine the structure of a network only composed by *fully connected* layers, it is called **Multi-Layer Perceptron** (MLP), fundamental starting point for various other types on NN. The first layer of the network, the input layer, takes as input the column vector x of size $n_x \in \mathbb{N}$, the following non-linear parametric function is applied to it:

$$h_1 = \phi(W_1 x + b_1) \tag{3.2}$$

here the weight matrix W_1 and the vector b_1 (called *bias*) are parameters, in size respectively $n_{h_1} \cdot n_x$ and n_{h_1}; $n_{h_1} \in \mathbb{N}$ is the size of the first hidden layer h_1 (the first layer after the input layer). ϕ is the *activation function*, i.e. a non-linear function in charge of filtrating the output value, the most commply used are the sigmoid and the ReLu function (Figure 3.2). The transformation of Equation 3.2, also referred to as *Perceptron*, can be applied l times in l similar hidden layers. The output layer, the last one, will finally apply:

$$y = (W_l h_{l-1} + b_l) \tag{3.3}$$

where similarly W_l is of size $n_y \cdot n_{l-1}$ and b_l is of size n_y, i.e the size of the output vector we want for our specific application.

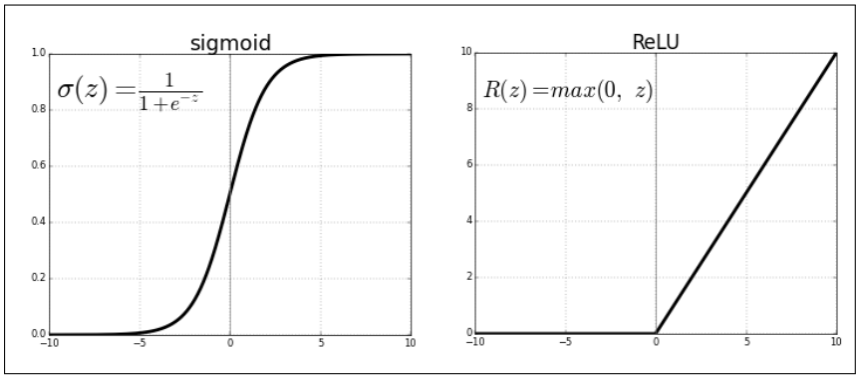


Figure 3.2: Sigmoid and ReLu activation function

In order for the parameters to be improved, indicated in their entirety with θ, the aim is to minimize a certain empirical error $L(\theta)$. The most common method to do so is based on gradient descent via the backpropagation algorithm:

$$\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta) \tag{3.4}$$

where α is the learning rate parameter, that determines the strength of the update.

The **ADAM** method [14] for stochastic optimization is much more used than the simple SDG. By keeping a per-parameter learning rate it improves performance on problems with sparse gradients, moreover per-parameter learning rates are adapted based on the average of recent magnitudes of the gradients for the weight (e.g. how quickly it is changing), this results in better performance on noisy problems. Its formulation is:

$$\theta \leftarrow \theta - \mu \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon} \tag{3.5}$$

with

$$\begin{aligned} \hat{m} &= \frac{m_{t+1}}{1 - \beta_1^{t+1}} \\ \hat{v} &= \frac{m_{t+1}}{1 - \beta_2^{t+1}} \\ m_{t+1} &= \beta_1 m_t + (1 - \beta_1) \nabla L(\theta_t) \\ v_{t+1} &= \beta_2 v_t + (1 - \beta_2)(\nabla L(\theta_t))^2 \end{aligned} \tag{3.6}$$

where ϵ, β_1 and β_2 are hyperparameters (respectively recommended: 10^{-8}, 0.9 and 0.999).

In current applications, beyond the simple feedforward networks introduced above, many different types of neural network layers and techniques have been

developed, each variation can provide specific advantages depending on the application. Among these, we mention for their importance the **convolutional layers**, that are particularly well suited for sequential data and high-dimensional inputs (such as images). Here, the layer's parameters consist of a set of learnable filters called *kernels*, which have a small receptive field and which apply a convolution operation to the input, passing the result to the next layer. As a result, the network learns filters that activate when it detects some specific features. The **pooling layers** reduce the dimensionality of the data carried forward in the network, they can be *max pooling* (which keep only the maximum values within clusters of values) or *average pooling* (which only keep the average calculated on clusters of values). A series of convolutional and pooling layers followed by a series of fully connected layers form **Convolutional Neural Networks** (CNN), powerful and versatile models nowadays widely used in various fields.

3.2 Proximal Policy Optimization Algorithm

The *Proximal Policy Optimization* is a class of algorithms that has been developed and proposed by *OpenAI* in 2017 [3], it has rapidly achieved state-of-the-art results and found wide use in various fields of application, such as robotic control, Atari videogames and more complex videogames such as *Dota 2* [15]. As we have seen in Section 2.9, the PPO algorithm is model-free, i.e. it does not aim to create and use a model that represents the environment. Moreover, it is a policy gradient method (it directly optimizes the policy) that learns **on-line**: it does not keep a replay buffer to store past experiences but it learns directly from them and after being used the experiences are discarded; it also has a system that allows updates not to deviate the policy too much from the current region, making it much more stable.

Let's now describe it in a more formal way all these features, first by analyzing the two fundamental approaches on which it is based:

- **Policy Gradient Method**: they work by computing an estimator of the policy gradient and applying it through a stochastic gradient ascent algorithm, the policy gradient loss $L^{PG}(\theta)$ is defined as:

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t[\log(\pi_\theta(a_t|s_t))\hat{A}_t] \tag{3.7}$$

  Here, the expectation $\hat{\mathbb{E}}_t[...]$ indicates the empirical average over a finite batch of samples, in an algorithm that alternates between sampling and optimization. π_θ is our stochastic policy: the neural network that starting from an observation s_t as input decides the action a_t to take as output; $\log(\pi_\theta(a_t|s_t))$ are the log

probabilities of our policy actions. The A_t term, called *advantage function*, is in charge to estimate what is the relative value of the selected action in the current state, it is calculated as:

$$\hat{A}_t = G_t - v_\pi(s) \tag{3.8}$$

Where G_t is the discounted sum of rewards (weighed sum of all rewards previously got, seen in Section 2.5, Equation 2.3 in particular); as we will see in Algorithm 1, $\hat{A}_t$ is calculated after the episode sequence has been collected, so no guessing is involved since reward are yet obtained. $v_\pi(s)$ is the value function (that basically estimates the discounted sum of rewards from s_t onward, seen in Section 2.8, Equation 2.10 in particular); it is a neural network itself, frequently updated as supervised learning problems on the basis of experiences collected in the environment. So basically the *advantage estimate* gives a measure of how much better the action that has just been took was based on the expectation of what would normally happen in this state. In other words, it answers the question: "was the took action better of worse than the expectation?" The term $\hat{A}_t$ is positive when the chosen actions have a better outcome than the expectation, the gradient is in turn positive and the policy update makes that this actions will be chosen with more probability, when these states will be encountered again in the future. On the other hand, the chosen actions have a worse outcome than the expectation, $\hat{A}_t$ is negative as well as the gradient, the update makes that this actions will be selected less likely.

- **Generalized Advantage Estimation**: proposed in [16], this is a family of policy gradient estimators that significantly reduce variance while maintaining a tolerable level of bias, basically by computing the *advantage estimate* in a more accurate way. As explained in [17]: "the *bias* error is an error from erroneous assumptions in the learning algorithm. High bias can cause an algorithm to miss the relevant relations between features and target outputs (underfitting)"; while "the *variance* is an error from sensitivity to small fluctuations in the training set. High variance may result from an algorithm modeling the random noise in the training data (overfitting)". Let's begin defining $\delta_t^V = r_t + \gamma V(s_{t+1} - V(s_t))$ that is the TD residual of V with discount γ (by definition [2]). Next, consider the sum of k of these δ terms, denoted as $\hat{A}_t^{(k)}$:

$$\hat{A}_t^{(k)} = \sum_{l=0}^{k-1} \gamma^l \delta_{t+1}^V \tag{3.9}$$

Then, similarly taking $k \to \infty$, we get:

$$\hat{A}_t^{(\infty)} = \sum_{l=0}^{\infty} \gamma^l \delta_{t+1}^V \tag{3.10}$$

The generalized advantage estimator is defined as the exponentially-weighted average of these k-step estimators:

$$\hat{A}_t^{GAE(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma, \lambda)^l \delta_{t+l}^V \tag{3.11}$$

γ and λ are both hyperparameters in the range [0, 1]. In the continuation of the analysis of the PPO algorithm we will use for simplicity the notation $\hat{A}_t$ to indicate the advantage function, even if the most recent implementation exploits the method of Equation 3.11 instead of 3.8.

- **Trust Region Methods**: by simply applying the gradient update on every batch of observations, the change of the network parameters would be performed far outside of the range where this data was collected, so that, for example $\hat{A}_t$ would be completely wrong and the updates would destroy the policy. To overcome this problem, the Thrust Region Policy Optimization approach [18], abbreviated TRPO, suggests to put a limit on how much the update of the current policy can move away it from how it is at the moment. To do that, TRPO adds a KL constraint to the optimization objective:

$$\begin{aligned} &\underset{\theta}{\text{maximize}}\ \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t \right] \\ &\text{subject to}\ \hat{\mathbb{E}}_t[\text{KL}[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]] \leq \delta \end{aligned} \tag{3.12}$$

  Note that in this objective function the only change from the vanilla policy gradient (Equation 3.7) is that the log operator has been replaced with a division by $\pi_{\theta_{old}}$, where θ_{old} is the vector of policy parameters before the update. This KL constraint however adds an overhead to the optimization process, the PPO algorithm includes this extra constraint directly into the optimization objective. We can indicate with $r_t(\theta)$ the probability ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$; so given a sequence of sampled states and actions, $r_t(\theta) > 1$ if the action is more likely to be chosen now rather than in the old version of the policy, and $0 < r_t(\theta) < 1$ in the opposite case. TRPO maximizes a "surrogate" objective:

$$L^{CLI}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t \right] = \left[r_t(\theta) \hat{A}_t \right] \tag{3.13}$$

Using the exposed notation, we can now define the main optimization objective function used in the PPO algorithm:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right] \tag{3.14}$$

The objective function is calculated as an expectation operator also in this case, this means that it is computed over batches of trajectories. The expectation is

applied to the minimum between two terms: the first is $L^{CPI} = r_t(\theta)\hat{A}_t$; the second term is a similar version where $r_t(\theta)$ is possibly truncated by the clip function, controlled by the hyperparameter ϵ (let's say $\epsilon = 0.2$). Remember that the operator $\text{clip}(x, x_{min}, x_{max})$ gives x for $x_{min} \leq x \leq x_{max}$, x_{min} for $x < x_{min}$ and x_{max} for $x > x_{max}$.

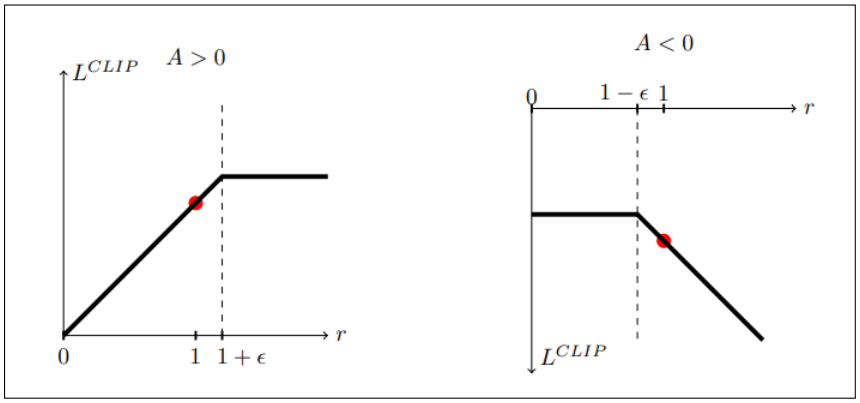


Figure 3.3: L^{CLIP} objective function trend as a function of $r_t(\theta)$, for negative (left) and positive (right) $\hat{A}_t$

The meaning of the min operator is deeply influenced by the fact that $\hat{A}_t$ can be either positive or negative, but it also is very noisy, so the clip system is in charge to ensure that the gradient update does not destroy the network. Figure 3.3 shows the trend of the objective function L^{CLIP} for positive and negative values of the advantage estimate. On the left graph, with $\hat{A}_t > 0$ i.e. when selected actions are better than the expectation, if r gets too high the clip operator flattens it; this means that even if the action is a lot more likely than the expectation, anyhow the gradient update will be done in a limited magnitude. On the right graph, when $\hat{A}_t < 0$ i.e. when selected actions are worse than the expectation, the objective is flattened when goes near zero; this means that even if the action is a lot less likely than the expectation, anyhow the gradient update will not reduce its probability to zero. Lastly, the inclined part of this graph occurs only in the case in which r is largely positive but the choice was wrong ($\hat{A}_t < 0$), the objective function will be negative proportionally to the mistake done; note that this is the only case in which $r_t(\theta)\hat{A}_t < \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t$ on which the min operator will be applied. It can be said that the PPO algorithm bases its strength on the update limitation concept proposed by the TRPO approach, but that the calculation of the objective function is greatly simplified. This means that the performances are excellent compared to other more complex algorithms.

The final loss function used to train an agent is the sum of the L^{CLIP} objective just seen and other two additional terms:

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \right] \tag{3.15}$$

The L_t^{VF} term is in charge of updating the network that has to predict the value function (i.e. to predict how good is it to be in a specific state). It is calculated as squared-error loss $(V_\theta(s_t) - V_t^{targ})^2$. The policy and the value function are two neural network, obviously they perform different tasks but substantially the structure of the input layer and the intermediate layer of both are equal, therefore they are combined into a single network that is updated with a single loss function. The term S, called the *entropy* bonus, is in charge of making sure that the agent performs enough exploration during the training. Entropy for a gaussian distributions is defined as:

$$h = \frac{1}{2} \log 2\pi e \sigma^2 \tag{3.16}$$

where π here is pi, σ^2 is the variance of the gaussian distribution. The $S[\pi_\theta]$ term used in the loss function is the mean of the entropy values calculated with the probability distributions of each possible action under the policy π_θ [19]. c_1 and c_2 are hyperparameters which weigh the contribution of these two terms.

Now that we have analyzed the objective function, we can look at the functioning of the entire algorithm (Algorithm 1); as an actor-critic method, it is made up of two main components. In the inner loop the current policy interacts with the environment, here the advantage function is immediately calculated for each step. Every N episodes, passing in the external loop, all collected experiences are used to run gradient descent using the L^{CLIP} objective function, the old policy is so updated.

Algorithm 1 PPO, Actor-Critic Style

for iteration=1,2,... **do**
  for actor=1,2,...,N **do**
    Run policy $\pi_{\theta_{old}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, ..., \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{old} \leftarrow \theta$
end for

As explained in [15], the implementation of this training system can be decoupled into two separated parts. The interaction with the environment can be

performed in parallel by thousands of *rollout* workers, each of these agents acts independently inside a copy of the environment. Periodically, the *optimizer* node gets the experiences and performs the network update. The local policy used by each worker has to be often refreshed in order to be sure that the latest version is being used.

The neural network used within the algorithm is a MLP, as mentioned it encloses both the policy and the value function estimator. To define its structure it is only necessary to decide how many layers it contains and the size of the layers, the size of the first layer is given by the number of values present in the input vector, that of the output by the number of actions we need, they can be either continuous or discrete. This NN is very effective although simple given the nature of the vector of row observations that we provide as input, which as we will see are values observed by sensors; if images were used as input, it would probably be necessary to use a CNN.

3.3 The PPO Hyperparameters

As for any algorithm of this level of complexity, performances are deeply influenced by the hyperparameters that govern its functioning. As seen in the previous section, they are not a few, but their understanding and tuning is fundamental for success. Below is summarized the specific description of the parameters that here can be tuned, as well as the numerical ranges usually recommended; they are reported with the nomenclature of the *ML-Agents* library [4], it has been used used for the application of the PPO method to the proposed case study, as will be explained in Chapter 4. Extensive reference is made to its documentation [20].

- `batch_size`: the batch size defines the number of experiences (agent state-action-rewards obtained) used for one iteration of a gradient descent update. This should always be a fraction of the `buffer_size`. In a continuous action space, its value should be large, in the order of 1000s (typical range in continuous: [512, 5120]); using a discrete action space, its value should be smaller, in order of 10s (typical range discrete: [32, 512]).

- `buffer_size`: it indicates how many experiences should be collected before doing the updating of the model. It should be a multiple of `batch_size`. Typically a larger `buffer_size` can lead to more stable training updates. The typical value range is [2048, 409600].

- `learning_rate`: this parameter corresponds to the strength of each gradient descent update step. This should typically be decreased if training is unstable,

and the reward does not consistently increase. The typical value range is $[10^{-5}, 10^{-3}]$.

- **gamma**: corresponds to the discount factor for future rewards. This can be thought of as how far into the future the agent should care about possible rewards. In situations when the agent should be acting in the present in order to prepare for rewards in the distant future, this value should be large. In cases when rewards are more immediate, it can be smaller. The typical value range is [0.8, 0.995].

- **epsilon**: it regulates the acceptable threshold of divergence between the old and new policies during gradient descent updating (seen in Equation 3.14). Keeping it small will result in more stable updates, but will consequentially slow the training process. The typical value range is [0.1, 0.3].

- **beta**: it corresponds to the strength of the entropy regularization, which is in charge of making the policy "more random" (seen in Equation 3.15, here it was called c_2). Increasing this parameter will ensure more random actions are taken. This should be adjusted such that the entropy (value measurable from *TensorBoard*) slowly decreases alongside increases in reward. If entropy drops too quickly the agent is probably not exploring enough: increase beta. If entropy drops too slowly: decrease beta since at the end of the training it should be weak. The typical value range is $[10^{-4}, 10^{-2}]$.

- **lambd**: this parameter is the lambda parameter used when calculating the Generalized Advantage Estimate (seen in the previous section). This can be thought of as how much the agent relies on its current value estimate when calculating an updated value estimate. High values correspond to relying more on the actual rewards received in the environment (which can be high variance), low values correspond to relying more on the current value estimate (which can be high bias). The parameter provides a trade-off between the two sides, a higher value can lead to a more stable training process. The typical value range is [0.9, 0.95].

- **num_layers**: this parameter defines how many hidden layers are present in the neural network. For simple problems, fewer layers are likely to train faster and more efficiently. More layers may be necessary for more complex control problems. The typical value range is [1, 3].

- **hidden_units**: it defines how many units are in each fully connected layer of the neural network. For simple problems where the correct action is a straightforward combination of the observation inputs, this should be small. For problems where the action is a very complex interaction between the

observation variables, this should be larger. The typical value range is [32, 512].

- **`normalize`**: this boolean value decides to whether normalization is applied to the vector observation inputs. This normalization is based on the running average and variance of the vector observation. Normalization can be helpful in cases with complex continuous control problems, but may be harmful with simpler discrete control problems.

- **`max_steps`**: it defines how many decision steps of the simulation are run during the training process. This value should be increased for more complex problems. The typical value range is $[5 \cdot 10^5, 10^7]$.

- **`num_epoch`**: the number of epochs represents the number of passes through the experience buffer during gradient descent. The larger the batch_size, the larger it is acceptable to make this. Decreasing this will ensure more stable updates, at the cost of slower learning. The typical value range is [3, 10].

- **`time_horizon`**: it corresponds to how many steps of experience to collect per-agent before adding it to the experience buffer. When this limit is reached before the end of an episode, a value estimate is used to predict the overall expected reward from the agent's current state. As such, this parameter trades off between a more biased, but less varied estimate (short time horizon) and less biased, but higher variance estimate (long time horizon). In cases where there are frequent rewards within an episode, or episodes are prohibitively large, a smaller number can be more ideal. This number should be large enough to capture all the important behavior within a sequence of an agent's actions. The typical value range is [32, 2048].

Chapter 4

Framework, Library and Implementation

In this chapter the technologies that allowed the realization of this study will described. The physical simulations were implemented from scratch using the framework *Unity* [5] (version *2020.3.3f1*), is is a game-engine with *C#* language scripting. Together with it, the *Unity Machine Learning Agents Toolkit* (abbreviated ML-Agents) [4] was exploited, it is an open-source project written in *Python 3*, which allows through RL algorithms to train agents living inside Unity `scenes`; `Package` *Version 2.1.0-exp.1* (released in June 2021) was used. The entire execution of ML-Agents takes place on CPUs, 10 virtual processors of an *AMD Ryzen Threadripper 3970x 32-core* (3.70 GHz frequency) were available and were used for all procedures reported below, it has been used *Windows 10* operating system. In the first part of this chapter we will summarize for both technologies the main features ant the functioning, not in their totality but those that have been mostly used for this work; after that will be exposed the physical characteristics that have been shared basis of the implementation of the Lunar Lander simulations.

4.1 The Unity Framework

Each entity within the simulated world is called **`GameObject`**, for the sake of brevity from now on we will sometimes generically call the GameObjects "objects". Each object owns the base class **`MonoBehaviour`**, that must explicitly be derived when implementing a *C#* script. They are attributed with `components` that define particular properties or behaviors. Each object has the fundamental **`Transform`** component, which defines its position, rotation and scale inside the environment, it also defines the **`Parenting`** (concept that we will analyze better later). Physics

modeling is applied to all objects that possess the **`Rigidbody`** component. Its most important features are: the value of the velocity in the three axial components, the angular velocity on the three axes of rotation, the size of the mass, the local position of the center of mass, whether or not it is subject to the force of gravity; in addition to accessing these values, it is possible to interact with the dynamics of the object through various APIs. The **`Mesh`** componend defines the object's shape. The collisions, and the relative simulation of their consequence, are detected between objects that have the **`Collider`**, it has its own shape which may be different from that of the mesh, in fact it is not visible. Other important components such as `Joints`, `Physics Articulations` and `Character Controller` were not used in this work so their description is omitted. Within the scripts it is possible to access the components of the object itself or of other objects through a search (for example with the method `FindGameObjectsWithTag(tag)`), since this is computationally expensive it is advisable to carry it out only once in the declarative phase, and keep a local reference variable to be accessed when needed.

In Unity two different types of periodic updates are performed: **`Update`** and **`FixedUpdate`**:

- the Update is calculated once per frame, we can define the method `Update()` to execute here our code. Its execution is totally independent from the physical engine, it is important to remember that its execution frequency is subject to variations linked to variations in framerate, for this reason it is good not to include physical interactions in this function, but to use it for minor purposes which can be delayed in case of performance slowdowns. Since only the physical aspects are important in this work rather than graphics and representations, it has not been used. By default, Unity runs at 60 FPS.
- the FixedUpdate instead is called with a regular frequency since here physical calculations are performed, we can add our scripts to be executed here through the function `FixedUpdate()`. By default, the FixedUpdate is performed 50 times per second, i.e. every 0.02 s; this value can be changed through the value `Time.fixedDeltaTime` (expressed in seconds), however, in this work it has not been modified.

Regarding the time, it should also be noted that it is possible to vary the execution speed by changing the value `Time.timeScale` (values less than 1 to slow down, greater than 1 to 100 to speed up), this proved to be very useful in the inference phase to quickly generate large amounts of logs (for example those shown in Section 9.4).

Each object's **`Start()`** function is called when the simulation, starts but only after that `Awake()` function of each object has been executed; therefore in the first

we have the only certainty that the object itself already exists, while in the second we can refer to any other object because we are sure that it has already been allocated. So both can be used in initialization phase keeping this mechanism in mind.

As said, each object preserves the Parenting status in its Transform component, this concept defines that each object can have a `Child`, or more than one, and at most one `Parent`. In the case of stratified relatives we have a hierarchy, the Parent of all is called `root`. In this way it is possible to group clusters of objects into a single entity. It is important to keep in mind that the characteristics of the Transform of any Child are expressed in a manner relative to those of the Parent, the root only uses coordinates defined in absolute system reference. For the implementation of the scenarios of this work the Lander, the Surface, the Target, the vacuum (the flyable space) are Child of the Environment root (an `Empty` object: non-physical), in turn engine and thrusters have the lander as Parent. Single objects, as well as objects composed of hierarchies of objects, can be used as a definition of a **`Prefab`**: that is an object that can be allocated in a replicated manner while maintaining its own characteristics. This makes it easy to edit multiple identical objects at once by changing their Prefab definition. For the parallelization in the training phase, described in Section 8.5, the scenario was replicated starting from the root exploiting this concept; Figure 4.1 shows the 6-DOF scenario and its constituent objects parallelized four times, on the left we can see the objects present in the form of Prefab and their hierarchy.

Among the methods offered by the Rigidbody component it was of particular importance **`AddForceAtPosition`**`(Vector3, Vector3)` that, obviously, allows to apply a force in a certain position). It takes as parameters a three-dimensional vector as force vector and another three-dimensional vector as point of application, both expressed in world coordinates. Given an object's Transform, it may be useful to use the values `transform.up`, `transform.down`, `transform.left`, `transform.right`, `transform.forward` and `transform.backward` that provide the object's six fundamental directions expressed in the global reference system, each can be serialized with `.normalized` to obtain a versor. For the case of the lander, this function was used to implement all thrusters, for example the ignition of the main engine is implemented as a constant force applied in the center of the cube base:

```
1 landerRigidbody.AddForceAtPosition(
2     thrustMagnitude * transform.up.normalized,
3     transform.localPosition - (transform.up.normalized * (sideSize /
    2f))
4 );
```

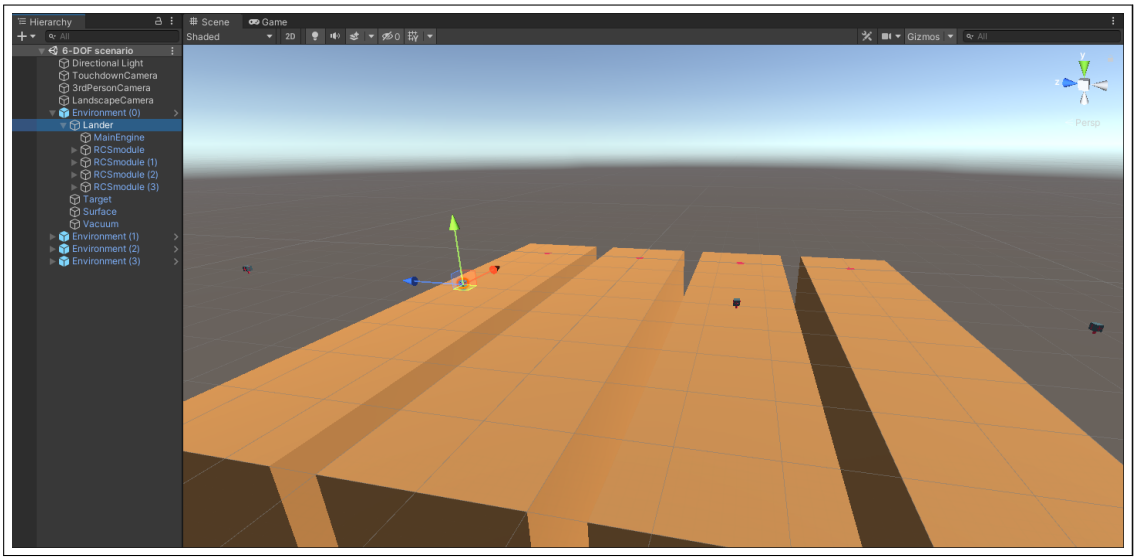


Figure 4.1: Screenshot of the implementation of the 6-DOF scenario parallelized four times, on the left the hierarchy tab shows the objects present in the form of Prefabs and they Parenting.

where `landerRigidbody` is a reference to the lander's Rigidbody component, `thrustMagnitude` is the magnitude of the force, `localPosition` the coordinates of the center of the cube in the Parent reference system (that of the environment then) and the `sideSize` is the size of the cube's faces.

Regarding collisions, detected as mentioned among objects having the Collider component, of particular importance is its `bool` parameter **`isTrigger`**: if set, when a collision occurs (or more in general interactions between colliders) instead of the normal physical effect special functions are immidiately called up. In particular, this three functions whose parameter is assigned the reference to the other object:

- `Collider.OnTriggerEnter(Collider)`: it happens on the FixedUpdate function when two GameObjects collide. It was used in case the lander collides with the ground or the target.
- `Collider.OnTriggerExit(Collider)`: this function is called in the FixedUpdate when two objects that had their own Colliders previously interpolated are now separated. This mechanism has been used to identify the lander exit from the vacuum, that has only the Collider component with isTrigger enabled but no Rigidbody (since it must not have physical interactions).
- `Collider.OnTriggerStay(Collider)`: it is called every FixedUpdate in

which two trigger-objects are interpolated. It was never used for the lander.

Note that this functions to be called both GameObjects must contain a Collider component and one must have Collider.isTrigger enabled, and contain a Rigidbody. If both GameObjects have Collider.isTrigger enabled, no physical collision happens as if both GameObjects do not have a Rigidbody component. In the absence of the isTrigger parameter active, the three functions `OnCollisionEnter`, `OnCollisionExit` and `OnCollisionStay` are called similarly in addition to the normal execution of collision physics. In the implementation of the lander landing scenario, however, the collisions represent termination cases of the episodes, so their physics is not relevant but the trigger mechanism is exploited in three cases above mentioned.

4.2 The ML-Agents Library

As said, ML-Agents allows to use Unity simulations as environments in which we can train artificial intelligent agents, in particular exploiting the reinforcement learning mechanism explained in Chapter 2. At a high level, ML-Agents is constituted by five major components represented in Figure 4.2; as described in official documentation [21], they are:

- **Learning Environment**: the Unity simulation `scene` and all the characters within it. This is the environment in which agents will observe, act, and learn.
- **External Communicator**: it is a piece of code that resides in Unity once the package is installed, it connects the Learning Environment with the Python Low-Level API.
- **Python Low-Level API**: which contains a low-level Python interface for interacting and manipulating a learning environment. Note that the Python API is not part of Unity, but lives outside and communicates with Unity through the Communicator. This API is contained in a dedicated `mlagents_envs` Python package and is used by the Python training process to communicate with and control the `Academy` during training.
- **Python Trainers**: the machine learning algorithms that allows training agents. The package exposes a single command-line utility `mlagents-learn` that supports all the training methods and options. The Python Trainers interface solely with the Python Low-Level API through a port (by default `5004`).

- **Gym Wrapper**: (not represented in the schema) a common way in which machine learning researchers interact with simulation environments is via a wrapper provided by OpenAI called `gym` [22], ML-Agent provides a gym wrapper in a dedicated `gym-unity` Python package and instructions for using it with existing machine learning algorithms which utilize gym.

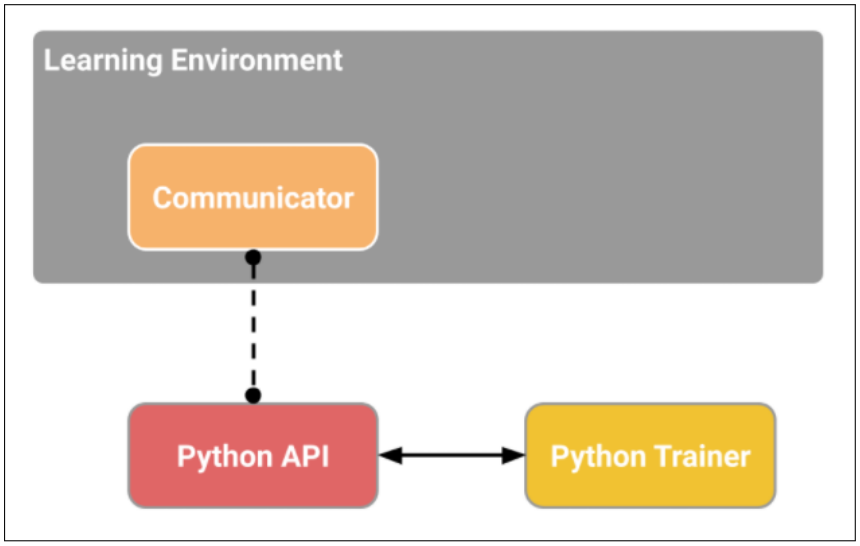


Figure 4.2: High-level component diagram of ML-Agents [21].

Regarding Trainers, currently ML-Agents offers to be used the implementation of two state-of-the-art DRL algorithms: SAC and PPO, the first was previously mentioned in Subsection 2.9.1, the second has been deeply described in Chapter 3 since it has been fully utilized in this work, given its excellent performance. Their implementation is based the OpenAI Baselines [23], using *PyTorch*. After the deep neural networks have been trained, i.e. the policies, they are saved and made available with `.onnx` format (*Open Neural Network Exchange* [24]). Note also that it is possible to `build` Unity scenes containing agents that use policies in inference mode, the generated executable can be run directly on various platforms as well as any Unity project.

This package exposes a command **`mlagents-learn`** that is the single entry point for all training workflows, to start a training procedure the basic command is:

```
mlagents-learn <trainer-config-file> --run-id=<run-identifier>
```

where `<trainer-config-file>` is the file in which resides the training configuration in `.yaml` format, identified by a unique `BehaviorName`. `<run-identifier>` is the unique name with which the outputs of the procedure will be identified, they will be placed in the */results/<trainer-config-file>* folder within the Unity project, they are basically the deep neural network (the policy) and log files that can be read in *TensorBoard*. Other useful options for the command are:

- `-resume`: to resume an interrupted training
- `-initialize-from=<run-identifier>`: in order to begin a procedure of training beginning from an already existing neural net, this mechanism will be analyzed in the Section 9.2
- `-seed`: to specify the basis used to calculate randomness, in order to obtain meaningful comparisons in this work we always used seed equal to 303
- `-force`: to force the eventual overwriting of a homonymous training already present

Regarding the training configuration, we report as an example the default configuration within the yaml file, in order to show how the explanation of the parameters used above and the summary tables (PPO hyperparameters in Appendix A) are effectively structured:

```yaml
behaviors:
    PPO_default:
        trainer_type: ppo
        hyperparameters:
            batch_size: 10
            buffer_size: 100
            learning_rate: 3.0e-4
            beta: 5.0e-4
            epsilon: 0.2
            lambd: 0.99
            num_epoch: 3
            learning_rate_schedule: linear
        network_settings:
            normalize: false
            hidden_units: 128
            num_layers: 2
        reward_signals:
            extrinsic:
                gamma: 0.99
                strength: 1.0
        max_steps: 500000
        time_horizon: 64
```

```
23          summary_freq: 1000
```

4.2.1 Classes, Methods and Fields

Regarding the C# implementation, let's now see the classes with which we build and manage the reinforcement learning framework. The cadence of the RL mechanism is governed by the singleton class **`Academy`**, which is automatically allocated, it has the fundamental task of performing the `step` of the environment: that is to make agents within it take observations, choose actions and make eventual updates (if and only if their decision period requires it). By default the step is perfromed automatically with the same frequency as the physical step(since `AutomaticSteppingEnabled` is enabled, otherwise we can manually call `EnvironmentStep()`). Through the Academy we can also access in reading important values, we mention the most relevant:

- `EpisodeCount`: the number of the current episode
- `InferenceSeed`: the `int` value used as basis for randomness in the inference phase
- `StepCount`: the number of the steps performed within the current episode
- `TotalStepCount`: the number of the steps performed within the entire simulation

Any Unity object can become a RL agent: we have to assign to it a script that implements the **`Agent`** interface class, the following core values reside in it:

- `MaxStep`: the maximum number of decision steps that the agent experiences, it has to be set, if this value is reached the episode is automatically reset
- `StepCount`: the current (decision) step counter within the episode
- `CompletedEpisodes`: the number of episodes that the agent has completed

The following methods control the assignment of the reward and the progress of the episode, they can be recalled at will inside the code but it is good practice to use them inside the FixedUpdate():

- `AddReward(Single)`: it assigns a reward to the agent, it can be a positive or negative `float` value

- `GetCumulativeReward()`: it returns the current cumulative value of the reward accumulated by the agent, that is the sum of all the rewards that have been assigned during the current episode, obviously it can be positive or negative
- `SetReward(Single)`: it overrides the current step reward replacing it with the specified value
- `EndEpisode()`: this method causes the instantaneous termination of the current episode

Furthermore in this script we have to provide our overriding implementation of the following methods:

- `OnEpisodeBegin()`: this method is called before each episode begins, similarly to the Start() method in general, here we can define all the initializations we need, such as reset of both physical and parametric scenario parameters. In the case of the lander scenario, we define here the random initial conditions with which the flight phase begins, below is the piece of code of the lander angular velocity initialization as an example:

```
landerRigidbody.angularVelocity = new Vector3(
    UnityEngine.Random.Range(
        -1f * randomAngularVelocityPerAxis,
        randomAngularVelocityPerAxis
    ),
    UnityEngine.Random.Range(
        -1f * randomAngularVelocityPerAxis,
        randomAngularVelocityPerAxis
    ),
    UnityEngine.Random.Range(
        -1f * randomAngularVelocityPerAxis,
        randomAngularVelocityPerAxis
    )
);
```

  where `UnityEngine.Random.Range(float, float)` randomly provides a number in the given range with uniform distribution, and `randomAngularVelocityPerAxis` is the maximum angular velocity in absolute value that the lander can have at the beginning of the episode on each axis of rotation. In addition the logs obtained in the previous episode are finalized here and the logs of the new episode are initialized.

- `CollectObservations(VectorSensor)`: through this function we provide to the agent the observation state of the environment, that it will use as input

for the step. The following piece of code shows the passing of rotation values as an example:

```
sensor.AddObservation(landerRigidbody.angularVelocity[0]);
sensor.AddObservation(landerRigidbody.angularVelocity[1]);
sensor.AddObservation(landerRigidbody.angularVelocity[2]);
```

`VectorSensor` is an array of generic values, used when exploiting raw-data observations such as in our case; ML-Agents would also allow to use `CameraSensor` for visual observations and `RayPerceptionSensor` for raycast observations. Note that any preprocessing calculations can be performed before data is passed, such as normalizations or scaling or filters e.g. for the reasons set out in Subsection 5.2.1.

- `OnActionReceived(ActionBuffers)`: it is the function with which we obtain the actions that the policy has chosen, or better the output that represents its decisions, and it is our task to translate them into real actions within the simulation. `ActionBuffers` is a `struct` which basically contains two structures we can access as vectors:
  - `DiscreteActions`: in which the discrete decisions reside, has variable length according to the number of decisions desired; each value is an `int` between 0 and $N - 1$, where N is the number of discrete values that can be assumed by the choice, this can be variable and specified for each choice.
  - `ContinuousActions`: in which the continuous decisions reside, has variable length according to the number of decisions desired; each value is a `float` between -1 and 1, this can be scaled to any value range in order to obtain any continuous value decision, to do this the function is provided `ScaleAction(Single, Single, Single)` (takes as input the value and the extremes of the output range).

  As will be explained later, in the case of the implementation of the lander scenario only discrete decisions were used; an example of the access to the actions vector is reported below:

```
switch (actions.DiscreteActions[1])
{
    case (0):
        turnOffRcsThrustersRoll();
        break;
    case (1):
```

```
            turnOnRcsThrusterRollPositive();
            break;
        case (2):
            turnOnRcsThrusterRollNegative();
            break;
        default:
            throw new Exception("indefinite action");
}
```

Here we are talking specifically about the decision that controls the thrusters of the roll rotation, the `switch` statement calls different functions based on the decision made (turn off or turn on and which thrusters, implemented respectively as previously mentioned). Note that the correspondence between the digit representing the decision and the concrete action is completely arbitrary since the agent sees only pure numbers (e.g. here `actions.DiscreteActions[1]` equal to 0 corresponds to roll-thrusters turned off), the important thing is that this correspondence remains the same throughout the entire training and inference phases.

- `Heuristic(ActionBuffers)`: this method allows direct mapping of decisions provided to the agent with input of peripheral devices such as mouse and keyboard. In this way, choosing the *Behaviour Type Heuristic* we will have personally the ability to command the agent, that will ignore the policy. This mode is very useful in the implementation phase to verify that the functions that translate decisions into concrete actions are working correctly. In the following example, pressing the spacebar is mapped to a binary decision (which will be translated into the main engine control):

```
if (Input.GetKey(KeyCode.Space))
    actions.DiscreteActions[0] = 1;
else
    actions.DiscreteActions[0] = 0;
```

The **`BehaviorParameters`** component has to be assigned to the agent to specify the characteristics of the behavior that will control it. It possesses the following fields:

- `BehaviorName`: the unique name of the policy in use, it should not be modified at runtime except with the appropriate Agent method `SetModel(String, NNModel, InferenceDevice)`
- `BehaviorType`: it can be *Inference* (only if a model is currently specified) or Heuristic

- `Model`: the neural network model used as policy, it should not be modified at runtime too except with the appropriate Agent method `SetModel(String, NNModel, InferenceDevice)`
- `BrainParameters`: it specifies the characteristics of the neural network, it in turn contains substructures with the following fields:
  - `VectorObservationSize`: the size of the observation vector, used as input of the neural network
  - `NumStackedVectorObservations`: the number of consecutive observation vectors serialized (mechanism better described in the Section 5.2)
  - `NumContinuousActions`: the number of continuous actions provided as output by the policy
  - `NumDiscreteActions`: the number of discrete actions provided as output by the policy
  - `BranchSizes`: vector of size *NumDiscreteActions* that contains the maximum discrete values that each discrete action can assume

Note that in the version of ML-Agents used it is possible to use continuous and discrete actions simultaneously as desired.

An agent furthermore needs the **`DecisionRequester`** component, it causes the agent to trigger the decision making process with a certain cadence compared to the Academy step. The `DecisionPeriod` field sets this cadence; for instance if it is set to 5, every decision will be taken every 5 Acedemy steps. The `bool` parameter `TakeActionsBetweenDecisions` specifies if actions are to be performed during the Academy steps that are not decision steps, in this work it has always been kept enabled.

Before continuing, it is considered useful to finally summarize the four types of time steps encountered to avoid misunderstandings:

- *step* (basic concept of Unity): corresponding to the framerate, accessible through the Update() method, by default performed 60 times per second but not fixed
- *physical step* (basic concept of Unity): corresponding to the calculation and updating of simulation physics, accessible through the FixedUpdate() method, by default performed 50 times per second
- *Academy step* (introduced by ML-Agents): the RL mechanism step performed within the environment, it is executed simultaneously with the physical step (50 times per second), therefore it will be assimilated to it for brevity

- *decision step* (introduced by ML-Agents): a step in which the agent carries out the observation-action process (and updating in the training phase), with the minimum value 1 it coincides with the Academy step (50 times per second)

From here on the concepts of decision step and physical step (sometimes called step with misuse of language) will be mainly used.

4.3 Physical Models Main Features

Let's now see the main physical characteristics implemented in the Unity simulations, these are shared between the various versions of the scenarios that, as anticipated, have been developed in incremental way in accuracy and complexity. In coordinates (X, Y, Z) used from here on out, Y axis represents the vertical axis with positive sign upwards, as Unity is used to do. Positive rotations are intended using right-hand rule. Units of measurement are expressed using the SI, they are often omitted with abuse of notation.

4.3.1 Environment Model Characteristics

Regarding the environment, these are the characteristics held constant in all the different implementations of the scenario:

- There is a constant **gravitational acceleration** of (0, -1.62, 0) m/s^2, equal to the lunar one. The vertical direction is distributed uniformly throughout the space, it ignores the curvature of the surface, a legitimate simplification given the order of magnitude of the distances involved.

- There are no atmospheric **frictions** or **turbulences**. This characteristic agrees with the lack of atmosphere on the moon, but a more accurate model would be needed in the case of adaptation of the case study to other satellites or planets.

- The lunar **surface** is implemented as a smooth plane with, ideally, infinite size. The presence of any depressions or protuberances in the ground would be irrelevant, but making the assumption that the reliefs do not constitute a risk of impact during low-altitude flight.

- The landing **target** is a smooth circle, with height not different from the surface, its center is the point of reference $(0, 0, 0)$ with respect to the environment. Its dimensions vary in the scenarios and will be specified.

4.3.2 Lunar Lander Model Characteristics

The following features about the lander are kept constant in all the scenario modelings implemented:

- The lander model is made up of a single **rigid body** of cubic shape. Although the dimensions are realistic in almost any scenario, this simplification could be too simplistic when the lander touches the ground, in fact the flat surface of the cube would react differently than the feet on which a real lander rests.

- Its **density** is homogeneous, the center of mass is located in the center of the cube and its position does not change. The center of mass position is used hereinafter as reference point of coordinates $(0, 0, 0)$ relative to the lander.

  Certainly to implement a more accurate physical model the center of mass would not be fixed, but influenced by factors such as the fuel consumption and the fuel position in the tanks and from its movement. The center of mass position in this case should be periodically located and taken into account for the control and the consequent application of forces. This could be a good point of improvement for future and more accurate implementations.

- All the **engines** are modeled as constant forces, both the main one and the secondary ones in the cases of 3-DOF scenarios and those of the RCS in the case of the 6-DOF scenario. When the control commands a thruster to be switched on it immediately gives one hundred percent of its strength and keep applying it constantly as long as it's on, when the control commands a thruster to be switched off it switches off immediately. Therefore there are no intermediate transition values of forces applied, nor delays. Furthermore, both the point of application and the direction are fixed for all forces. Force magnitude, point of application an directions are specified below for all types of thruster and lander model.

  Real thrusters are obviously much more complex, they do not have immediate activation and shutdown times but times to reach full capacity, they cannot be turned off and on at will but must be kept within certain power ranges. Moreover the thrust is not applied constantrly in ane is in itself a very difficult task, surely this can be a large margin improvement.

- The **frequency** of the lander control system is 50 Hz, which means that every 0.02 s the control system imposes new commands on all available thrusters. This is the maximum frequency available, as we will see later, using a lower refresh rate is often preferable as it involves more stable behaviors.

4.3.3 Episode Characteristics

The definition of a scenario, in addition to the physical characteristics of the environment and the lander, includes the definition of three other important components:

- The definition of the **initial conditions** in which the lander is at the beginning of each episode. They are made up by the position and the velocity in environments reference system, the rotation and the angular velocity in the 6-DOF case. The ranges and the type of randomization must also be specified, which initiates each episode differently. Since they have been changed in the gradual implementation of the scenarios they will be specified later.

- The physical **constraints** that the lander must respect during the entire flight phase. The lander must not touch the lunar surface during the flight phase, this would lead to failure, or to a crash in the real world. With the exception of scenario Version 1, the lander must not even touch the target since the success condition is obtained by flying over it, but this will be specified in the Section 6.1. There is also a constraint on the angular velocity in the case of the 6-DOF scenario, it will be specified in the Section 9.1.

  There are no constraints to the distance nor to the speed that can be reached, nor to the possible rotations that can be assumed during an episode. But keeping in mind that each episode has a limited duration in terms of time steps, it is therefore presumably impossible for an episode to end successfully after disproportionately large distances or speeds have been reached by the lander. Some of these constraints have been added within the simulation environment in order to avoid computational waste and decrease the training time, they will be specified later.

- The **final success conditions** of the scenario, i.e. the conditions in which the lander must be in order for the touchdown to take place. In other words, they are the conditions that the agent aspires to achieve. They will be specified later too since they are specific to each scenario.

Starting from the common guidelines set out above, four different scenarios have been implemented, they will be respectively exposed and analyzed in Chapters 5, 6, 7 and 9; at the beginning of each of them a specific section will describe the characteristics of the physical model and the control problem regardless of the application of reinforcement learning, tables in Appendix A also summarize them.

Chapter 5

Autonomous Lunar Lander: 3-DOF scenario Version 1

In the following three chapters will be exposed the mechanisms that led to the "resolution" of the lunar lander control problem, in these the three degrees of freedom constraint is present. As already mentioned, with "solve" we mean in this specific study case the obtaining of a neural network capable of landing successfully, with a certain accuracy and success rate, it is not fixed since specific considerations are made for each case. In all cases the PPO algorithm, deeply described in Chapter 3, was used through the framework consisting of Unity and ML-Agents, described in Chapter 4. The first section of this and the next three chapters the physical modeling of the problem is exposed, it is important to remember that it is formalized regardless of the algorithmic application.

5.1 Physical Model

As said, the major physical simplification adopted in the first three scenarios is the 3-degrees-of-freedom limitation. It offers a notable reduction in the complexity of the control since actions taken independently on the three axes do not influence each other.

The lander has a **mass** of 1500 kg, the cube that models it has faces of 1 m side, it has five **thrusters**. The main engine has a positive vertical direction, it applies its force to the center of the base of the cube, it is present in all the scenarios implemented. The other four thrusters are positioned two per axis with the application point in the center of the cube faces, they are anagonists two by two. In this case all five thrusters can impose the same force of 15 kN.

The control system imposes three commands simultaneously, they can ideally be grouped one for each axis:

- Y axis (vertical): the main engine can be switched on or off
- X axis: the thruster that imposes the force in the positive direction can be turned on, or the thruster that imposes the force in the negative direction can be turned on, or both thrusters can be kept off
- Z axis: similarly to the X axis, the thruster that imposes the force in the positive direction can be turned on, or the thruster that imposes the force in the negative direction can be turned on, or both thrusters can be kept off

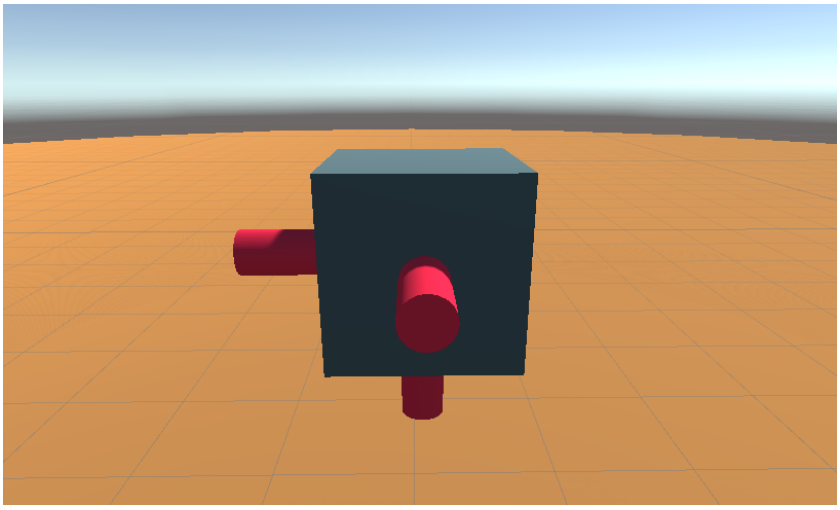


Figure 5.1: Screenshot of the lunar lander model in a 3-DOF scenario, implemented in Unity. The red cylinders have no physical function since they are for display purpose, each cylinder is displayed when the corresponding thruster is turned on.

The mobility in only three degrees of freedom is ensured by the fact that all the forces are applied to the center of the cube faces, i.e. in correspondence only to the center of mass, therefore rotations have no way of occurring. It should also be noted that antagonistic thrusters on the same axis cannot be turned on at the same time, this would result in a resultant of zero forces and therefore a waste of resources. The ignition of a thruster has no other implication apart from the constant application of force, there are no limits on time of use or ignitions.

The **position** of the lander with respect to the target at the beginning of each episode is placed randomly with uniform distribution in the following ranges: between 20 and 30 meters along the Y axis, between 95 and 105 meters on the X axis and between -5 and 5 meters along the Z axis. For simplicity in this scenario the initial **velocity** of the lander is not random, it is set as $(2, -2, 0)$ m/s.

The **target** is a circle with a diameter of 10 m. The condition of successful end of episode in this scenario consists simply in touching the target, the successful end-of-episode condition in this scenario consists simply of touching the target, avoiding touching the surrounding ground but without other constraints.

The characteristics described above formally describe Version 1 of the scenario in 3-DOF, they are summarized in Table A.1. This model is obviously very simplifying and far from the characteristics of Apollo 11, however fundamental for many initial arguments exposed below.

5.2 Reinforcement Learning Application

The first fundamental aspect to be defined for the application of a reinforcement learning algorithm is the composition of the **state** observed. In this first case the state is simply constituted by the position and the velocity. Both are calculated with respect to the center of the target and the center of the cube, both are divided into the three axial components. Remember that the values passed as observations are pure numbers, the algorithm has no knowledge of what they represent but simply learns about them through experience during the training. Therefore the state vector in this case is represented by a vector of 6 floating point values. They are then scaled directly by the pre-processing process carried out by the algorithm implemented in ML-Agents, this normalization is done on the basis of the running average and variance of the state vector.

Each state vector obtained at the time t can be passed to the agent chained with other $n-1$ state vectors obtained in the previous steps $t-1, t-2, ..., t-n+1$. The constant n is called **stacked vectors** in the ML-Agents framework [4], this strategy makes the agent have a vision of the previous states as well. Here, this parameter can assume an integer value in the interval $[1, 20]$. In this scenario, a stacked vectors of value 20 was used.

α

5.2.1 Limit unbounded states

Although the physical formalization of the control problem does not impose explicit limitations on the maximum reachable distance and on the maximum reachable velocity, it is advisable to impose boundaries in the training phase. Indeed, in the specific case of the lander landing, it is easy to imagine that if the lander moves far away from the target or reaches great speeds, the episode will most likely have a negative outcome. Although the agent is left with the possibility of affirming an arbitrary behavior, it is in fact easy to understand that many of the behaviors it tests are completely unsuccessful, especially in the initial phase of training. In the training phase it is essential to limit as much as possible the waste of resources in fruitless episodes, resources intended with computational use and above all training time in terms of decision steps. The time limitation of each episode bounds these potential wastes, but adding other reasonable limits allows you to further save resources.

There is another important aspect for which it is advisable to limit the size of the observations passed to the agent: the scaler system applied to the state vector is calculated on the basis of the maximum values ever recorded for each component of the vector. Therefore, if even just one episode records much greater quantities than those usually recorded, the approximation in the next scaling could result in small quantities being equal.

For this reasons, in this specific scenario, the size of the region above the ground in which the lander can fly has been limited to a parallelepiped of dimensions $(400, 200, 400)$ m, leaving this parallelepiped involves the termination of the episode with relative reward. The maximum speed value that can be provided is limited to 25 m/s.

The **actions** available that the algorithm can impose as output correspond exactly to those made available by the control problem. The output is represented by a vector of three integer values, the first value can assume the values 0 or 1 that will respectively force the main thruster to be turned off or on; the second and third values can be -1 or 0 or 1, respectively to turn the thruster on in the negative direction, turn it off or turn it on in the positive direction, for both horizontal axes X and Z.

Other fundamental parameters needed to define the agent's training are: the agent's **decision period**, set in this case to 10 which it responds to one decision step every 10 physic update steps, i.e. one decision every 0.2 s; and the number of **maximum steps** per episode, set to 1000 decision steps for this scenario. When

we talk about the maximum number of steps for an episode we are talking about physics update steps, as mentioned they are performed every 0.02 s, a maximum time of 1000 steps therefore corresponds to a maximum total of 20 s. The decision period is a parameter of fundamental importance in control applications, the relationship between its sizing and the training of an RL algorithm is taken up several times in this work, and a special benchmark is exposed in the Section 8.3, here we want to focus on the following reasoning:

β

5.2.2 Decision period in training and inference

In the case of control algorithms that interact with the physical world in general, the higher the update frequency, the better. In fact, high control frequency corresponds to greater accuracy in movements and greater reliability in response to external perturbations.

Let's hypothesize to train an agent that operates at a decision frequency f' with $f' < f_{max}$, where f_{max} is the maximum usable update frequency, and to obtain a policy π' capable of achieving the pre-established objective. When we use π' in inference mode we might be tempted to set the update frequency to f_{max}, thinking we will benefit from having the highest possible frequency. However, in most cases, doing this is seriously wrong as π' used to make decisions that were valid for a time $d' = 1/f'$. If instead they are maintained for a time $d_{max} = 1/f_{max} < d'$ the actions will have a different effect than expected, consequently the agent's overall behavior will be very different from what he had in the final training phase, with possible disastrous consequences as inference could work in the real world.

Therefore even if the decision period can be changed at will by passing from training to inference, it is recommended not to do so; for the state and action vectors the problem does not arise as they must remain the same. Although this reasoning could seem trivial, this error was initially made working on this scenario and therefore is reported.

The training **duration** for this agent is one million of decision steps, this size has to be be dimensioned on the base of the complexity of the scenario, even if initially it is not easy to understand how much complex it is, the only way is to try and do some reasoning, we will see that later on this size will be increased.

γ

5.2.3 Maintain training duration

During a training process, whether it seems successful or unsuccessful, it may come naturally to think of lengthening the process in order to obtain a better result, or to reduce it to save resources. It should not be forgotten that the behavior that the agent will have in the inference phase is visible only in the final part of the training. As explained in Chapter 3, this happens because various values that control the PPO algorithm are not constant but decay during the entire duration of the process. We are talking in particular of beta, epsilon and the learning rate values. Changing the total duration of the training in progress causes an anomalous variation of them, which results in a difficult interpretation of the results. Instead, it is preferable to terminate the current process normally and possibly use the neural network obtained as a starting point for another training process, as described in Subsection 9.2.6.

With regard to the hyperparameters of the PPO algorithm used to solve this scenario, they are similar to the standard configuration provided by the ML-Agents library [20], their meaning has been explained in Section 3.3 and they are entirely shown in Table A.3. Given the simplicity of the scenario, no particular tuning were necessary and no particularly significant observations emerged. In Chapter 8 benchmarks and general considerations regarding them will be presented.

Let's now analyze the reward function used in the most successful training, which allowed the resolution of this scenario. The reward function is the set of possible rewards that can be assigned to the agent according to the conditions in which he finds himself. The assignment of rewards is divided into periodic rewards and in terminal type rewards. The first are assigned at each time step of the framework update, i.e. in the flight phase for the case of the lunar lander; remember that it can occur with a greater frequency than the control update time, in Unity it occurs 50 times per second. The latter are assigned at the end of an episode, whether it is successful or unsuccessful.

The possible periodic rewards R_t and the method with which they were calculated are the following:

- **Fixed approach success**: at every time step t the current distance D_t from the center of the target and the base of the lander (i.e. the center of the base face of the cube) is calculated. A step of *minimal approach* is achieved if and only if:

$$D_t < D_{t-1} \tag{5.1}$$

A *fixed approach* is achieved if and only if:

$$D_t < D_{t-1} - a_{min} \tag{5.2}$$

where a_{min} is a consant distance to be dimensioned.

The reward attributed in case of achievement of this condition is a positive floating point value $R_t \in [0, \alpha]$ obtained with the following calculation:

$$R_t = \alpha \left(1 - \frac{D_t}{D_{max}}\right) = \alpha \left(1 - \frac{D_t}{\sqrt{\left(\frac{L_{env,X}}{2}\right)^2 + (L_{env,Y})^2 + \left(\frac{L_{env,Z}}{2}\right)^2}}\right) \tag{5.3}$$

where α is a positive value not necessarily constant and D_{max} represents the maximum distance reachable by the lander within the simulation environment. In fact even if the ground plane and the space above it are ideally infinite they are implemented with limits, the reasons for this choice is similar to the one described in Subsection 5.2.1. In this case the space in which it is possible to fly is modeled by a parallelepiped of dimensions $(D_{env,X}, D_{env,Y}, D_{env,Z})$ where the center of the base coincides with the center of the target. Therefore it is not difficult to understand that the reward R_t is a value between 0 and α mapped in an inversely proportional way with respect to the distance between lander and target.

We can summarize this generic strategy of reward calculation as:

$$\forall\, t,\ D_t < D_{t-1} - a_{min} \Rightarrow R_t = \alpha \left(1 - \frac{D_t}{D_{max}}\right) \tag{5.4}$$

Let us explain the two limit examples to completely clarify the functioning of this calculation: if the lander is very close to the target and as a time step elapses its distance from it decreases by a measure greater than a_{min} then it will receive a reward R_t close to α; if, on the other hand, the lander is at a great distance from the target and as a time step passes it decreases its distance from it by a measure greater than a_{min} then it will receive a reward R_t close to 0 (positive). We could say that the function applies an inverse linear mapping of the distance into a positive reward range.

In the specific case of resolution of this scenario, it have been used $a_{min} = 0.1$ constant, $\alpha = 1$ constant and $D_{max} \simeq 346$ since the environment has

dimensions $(400, 200, 400)$. Therefore Equation 5.4 becomes:

$$\forall\, t,\ D_t < D_{t-1} - 0.1 \Rightarrow R_t \in [0, +1] \mid R_t \simeq 1 - \frac{D_t}{347} \tag{5.5}$$

Note that D_{max} rounded up ensures that R_t is never negative.

- **Fixed approach fail**: similarly but on the contrary, a case of *approach fail* occurs if:

  $$D_t \geq D_{t-1} \tag{5.6}$$

  And a *fixed approach fail* occurs when:

  $$D_t \geq D_{t-1} - a_{min} \tag{5.7}$$

  In this scenario, in the case of fixed approach fail a constant reward of value -1 has been attributed, obviously keeping the same a_{min}, we can write it as:

  $$\forall\, t,\ D_t \geq D_{t-1} - 0.1 \Rightarrow R_t = -1 \tag{5.8}$$

  Note that the internal value D_t is recalculated at each time step t regardless of whether an approach has occurred in it or not.

The cases that determine the termination of the episode are listed below, in correspondence with them a terminal reward R_{fin} is assigned:

- **Target hit**: this case represents the success within this scenario as a rudimentary form of touchdown, it does not require the fulfillment of any other condition regarding velocity. In this case a reward $R_{fin} = 1000$ is assigned.
- **Ground crash**: understood as touching the ground outside the target in any way, it corresponds to the assignment of a reward $R_{fin} = 0$.
- **Maximum distance exceeded**: the maximum distance that can be reached with respect to the target is not calculated as a radial distance from it but is intended as a limited space in which to fly, as said in this scenario it is a parallelepiped of size (400,200,400) m, this choice is motivated in the Subsection 5.2.1. The method by which this condition is checked is explained in Section 4.1, it is analogous to the method by which collisions are detected. Leaving this space causes the assignment of a reward $R_{fin} = 0$.
- **Maximum step exceeded**: the maximum number of steps for this scenario is set as $t_{fin} = 1000$, upon reaching it a reward $R_{fin} = 0$ is assigned. As they are physic update steps performed once every 0.02 s, t_{fin} correspond to 20 s.

The fixed approach reward assignment has proved effective but implies a strong influence on the agent on how to act, in Section 6.2 and Subsection 6.2.2 it will be analyzed more in detail and overcome.

5.3 Scenario Solution

All the details that were necessary to define the training procedure that produced an agent able to achieve its purpose were outlined, graphs in Figure 5.2 were produced during it. From here on, the graphs of successful training will be similar to this, so let's analyze it here in detail. The abscissa axis is shared by all three graphs and reports the number of steps performed, remember that the length of a training is expressed in total number of decision steps.

- The first graph at the top shows the trend of the reward obtained by the agent, it can be positive or negative depending on the reward function, it is the goal of the algorithm to maximize this gain over time thus a successful training has a globally increasing trend.
- The central graph shows the duration that the episodes had during the training, here it is expressed in seconds to facilitate the intuitive understanding of the dimension, obviously it can only assume positive values. For these two graphs the raw values recorded are shown in gray color, they are usually very jagged; red and green curves are obtained by applying the exponential smoothing filter [25], with smoothing factor $\alpha = 0.9$.
- The third graph shows the same two smoothed curves scaled in the interval $[0, 1]$, now we have a pure unit of measure at ordinate axis and this makes easier to observe the strong correlation between the variations in the length of the episodes and the reward obtained.

The training process in these conditions let obtain an agent able to successfully reach the touchdown in the 95.5% of episodes, test carried out on 1000 episodes in inference mode; in the remaining 0.5% cases the lander hits the ground instead of the target. In addition it is only specified that in 62.1% of cases the lander touches the target in the inner circle of circumference 5 m. Given the simplicity of the scenario, no further analysis of the results was seemed necessary and it was preferred to proceed with the next implementation.

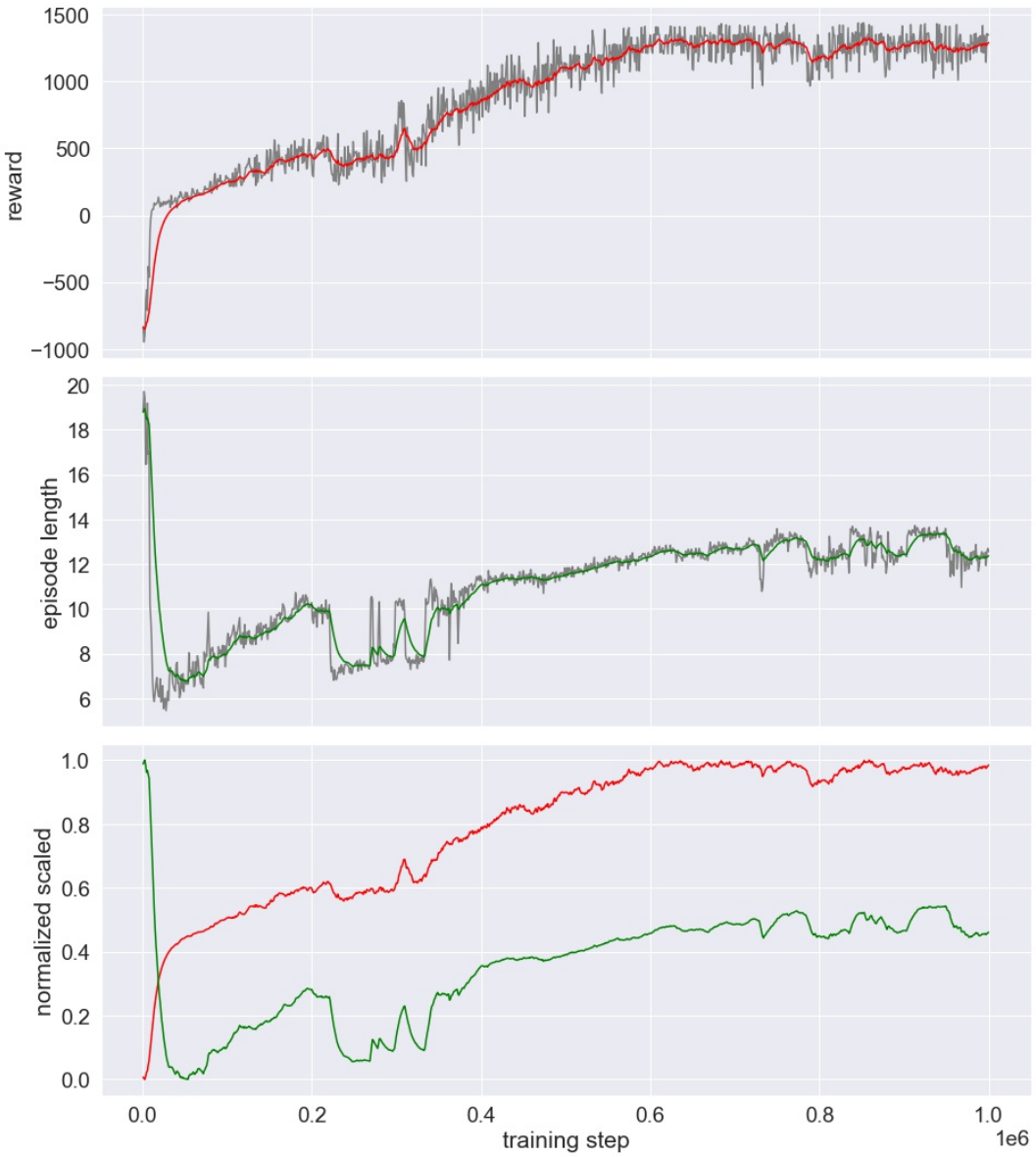


Figure 5.2: Scenario Version 1 training statistics

Chapter 6

Autonomous Lunar Lander: 3-DOF scenario Version 2

The two main features introduced in this scenario are the implementation of more realistic initial conditions, i.e. the conditions in which each episode begins that should resemble those assumed by a lander in the last phase of flight, and final constraints, which are the physical conditions that must be met in order for the desired touchdown to occur. The Apollo 11 mission was chosen as a reference as a source of detailed descriptive material, in particular the official *NASA* reports [6] and [26].

6.1 Physical Model

Both the position and the velocity with which the lander begins the episode are similar to those assumed by the Apollo 11 lunar lander during its landing phase. Both values are randomized within certain values in order to generalize the situation. The analysis of performances under greater randomness ranges is fundamental in this type of applications as it reflects the variability of realistic scenarios and tests the adaptability of deep neural networks. The stochastic approach of neural networks, in fact, is a well established strength rather than approaches based on the pre-calculation of causal situations.

For the **initial constraints**: the starting position is set as $(-700, 150, 0)$ m to which a randomness with uniform distribution of 10% is applied, which leads to the following ranges: $[-630, -770]$ m on the X axis, $[135, 165]$ m on the Y axis and $[-15, 15]$ m on the Z axis. The starting velocity is set as $(18.3, -5, 0)$ m to which a randomness with uniform distribution of 10% is applied, which leads to

the following ranges: $[16.47, 20.13]$ m/s on the X axis, $[-5.5, -4.5]$ m on the Y axis and $[-0.5, 0.5]$ m on the Z axis.

Regarding the **final constraints**, which were obviously necessary to be introduced since Scenario 1 did not treat a real landing, for the landing to be successful the lander must meet the following conditions at the same time:

- The center of mass of the lander must be over the target, i.e. the distance calculated on the only X and Z axes must be minor than 5 m in magnitude.
- The lander base must be at a height less than 1 m.
- The horizontal velocity, that is the velocity calculated on the only X and Z axes must be minor than 1 m/s in magnitude.
- The vertical velocity, that is the velocity calculated on the only Y axis must be minor than 1 m/s in magnitude.

The condition on the lander height is very relevant since if the lander does not have to touch the ground it can continue to fly over the target until the conditions are not all satisfied at the same time. This helps the RL algorithm to achieve its goal, as it will be described below. Touching the target without the maximum speed conditions being met leads to a new unsuccessful termination condition, called *target hit*. Obviously the other unsuccessful terminal cases remain the same.

The **mass** of the lander has been increased to 3000 kg, but it is still not realistic sunce the substantial improvements about it will be made in Version 3. The other parameters are the same as the previous scenario, all the features of this model are specified in the summary Table A.5.

6.2 Reinforcement Learning Application

The input vector of the **observations** was kept the same as the previous scenario: it is composed by the position expressed in the three axes and by the velocity expressed in the three axis; the output vector of the **actions** is the same too, made of three values that represents three decisions: two trivalent for the two horizontal thrusters and one bivalent for the vertical thruster.

The value of the **stacked vectors** parameter was reduced to 1. This decision was taken as a result of a benchmark that showed that, for the same result, the minimum value of stacked vectors involves less consumption of computational

resources, but this aspect will be better analyzed later in Section 8.3.

It was necessary to increase the maximum **duration** of each episode given the greater distance that must be covered, it is set to 90 s corresponding to 4500 steps of physics update. The optimization of the time spent can represent one of the factors of interest in the performance of an agent. In this scenario there are no physical factors that impose time limits, this will happen in the Version 3 with the introduction of the limited fuel available, therefore it is essential to insert a factor that ensures the agent is encouraged not to waste too much time, this aspect is managed by the reward function as explained below (in Subsection 6.2.3 in particular).

The **training duration** has been increased to two millions of decision steps. A previous training procedure with one million steps had led to a similarly positive result, but doubling the duration of the training increased the success statistics; Figure 6.1 shows the graphs obtained during them, clearly show that a million learn the correct behaviour but not enough to fully establish it, in particular more time allows to optimize success as explained in Subsection 6.2.5. In particular, this allowed to increase the accuracy in landing from 81.8% to 97.7% (statistics obtained out of a total of respectively 1000 and 2000 test episodes).

The **decision period** used is 15 which responds to one decision step every 15 physic update steps, i.e. one decision every 0.3 s; it has been increased with respect to the previous scenario as a comparison benchmark led to the following results.

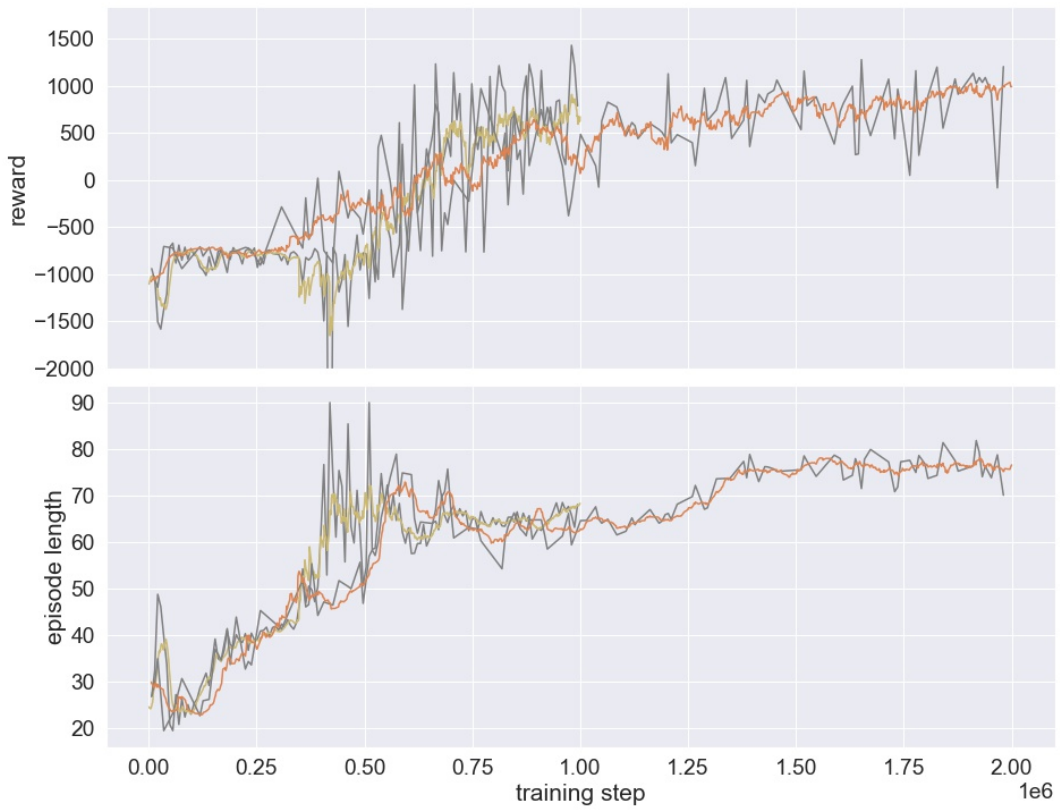


Figure 6.1: Comparison between trainings with 1 million and 2 million decision steps duration, carried out as the same conditions

δ

6.2.1 Counterproductive high control frequency

Generally speaking about the control update frequency, we refer to the number of decision taken in a fixed time, as already mentioned in the Subsection 5.2.2, we would like it to be as high as possible for our real world deployment. Therefore if our model allows us to operate with a maximum frequency f_{max} we are tempted to fully exploit it to obtain the best agent possible; technically this is true, i.e. a fully trailed agent acting at a frequency f_{max} is better than all those acting at any $f < f_{max}$. The problem is that the higher the frequency, the more expensive the training is, just think that the total duration of the training is expressed in number of decision steps: if we double the decision frequency the agent will have half of the episodes available to learn. It is also not taken for granted that by increasing the duration of the training the same final behavior is obtained. On the other hand, obviously, an excessively low frequency would be unable to result in satisfactory behaviors.

Think of the specific case of the lander controls, during the early stages of training the highly random exploration with a high frequency of change would cause antagonistic thrusters to ignite repeatedly with a substantially zero result, the agent would waste time just to learn how to keep the choice of a direction to observe a shift. And probably at the end of the training the control frequency would not be exploited anyway since in order to have significant resulting forces the decisions would have to be maintained for a prolonged time.

For this reason it is strongly advised not to use the maximum available control frequency, but rather to choose it in a way that is commensurate with the specific application.

For this scenario, the **learning rate** has been brought from $3 \cdot 10^{-4}$ to $3.5 \cdot 10^{-4}$, **epsilon** from 0.2 to 0.25 and **lambda** from 0.925 to 0.95 based on benchmarks, however they are not reported as more general and significant comparisons will be presented in Chapter 8.

Regarding the reward function, with the increase of the initial distance and the consequent increase in the duration of each episode it was necessary to radically modify its structure, and notable observations were made. In the first phase of each episode we would like the lander to move in the vicinity of the target, in Version 1 we defined the fixed approach and attributed a positive reward to it, the distance parameter a_{min} was in charge to determine the goodness of the approach. However, the fact that it was constant imposes a major influence on the behavior that the agent had to have, as it was totally inflexible. It basically defines how fast it is thought that the lander should go during the whole episode, furthermore it has been dimensioned on the basis of the distance set to the lander at each episode begin (with very small random ranges in the Version 1 scenario).

ϵ

6.2.2 Between strict and sparse reward functions

As the reinforcement learning mantra dictates, it is good practice to structure the reward function with rules as general as possible, this leaves the agent freedom to generate its own behavior, key strength of the approach. Also consider that in the presence of wide ranges of randomness or wide observation spaces, genericity is more effective in general. So, the assignment of rewards under strict conditions which implicitly impose to the agent to learn a non-arbitrary

behavior are to be avoided.

On the other hand, however, a reward function that is too generic or not very significant rises to what is called *sparse reward environment*, here the agent struggles to learn because it is not stimulated properly. The problem of *sparse reward* is also very significant in applications where it is difficult to find meaningful observation states or feedback related to the actions, due to the inherent structure of the environment.

This important principle is carried out throughout the development of the reward function in the whole project, it is a trade off between the genericity that makes it possible to fully exploit the potential of reinforcement learning, and the imposition of behaviors that allow the agent to learn how to reach the goal.

Therefore, since the periodic reward assignment for fixed approach intrinsically imposes a too strong influence on the agent on how to act (it tells him how fast to go), it is no longer used. However, the minimal approach reward is necessary, the landing task to be learned is quite complex among all the possible state space, without this signal the lander would only learn to fly avoiding terrain and boundaries but it would hardly come close to the final conditions, even with a training of enormous duration. So the minimal approach condition represents a "suggestions", in this scenario it turned out to be sufficient but in the next scenario a compromise had to be found between the generic nature of the minimal approach and the excessive rigidity of the fixed approach.

ζ

6.2.3 Avoid laziness

Since no aspect of the model imposes reasons for being "in a hurry", there is a risk that the behavior of the agent will establish in approaching as slow as possible in order to accumulate small positive rewards at each time step. This type of behavior is called *laziness*, it is a clear example of how, although it may seem that the reward function is well structured, the agent can find a way to exploit it for his own gain while not doing what we would like. Obviously this is a behavior that we want to avoid immediately as it is unacceptable in a real application. The simplest method to stem laziness is to periodically assign a small negative reward, in this way the agent is encouraged to quickly cross the condition in which he finds himself to arrive at a positive condition, wasting as

little time as possible; all the negative reward accumulated will decrease the final positive reward.

On the basis of this considerations, the periodic rewards assigned in this scenario are:

- **Minimal approch success**: similarly to how explained in Section 5.2, the reward R_t is given by an inverse linear mapping of the distance from the target, but this time it is mapped on an interval of negative values. R_t is given by:

$$R_t = -\alpha\left(\frac{D_t}{D_{max}}\right) = -\alpha\left(\frac{D_t}{\sqrt{\left(\frac{L_{env,X}}{2}\right)^2 + (L_{env,Y})^2 + \left(\frac{L_{env,Z}}{2}\right)^2}}\right) \tag{6.1}$$

  where it is decided that $\alpha > 0$ in order to explicit the negative sign in the equation, D_t is the lander base distance from target and D_{max} is the maximum distance reachable similarly to Equation 5.2. We can summarize this generic strategy of reward calculation as:

$$\forall\, t,\ D_t < D_{t-1} \Rightarrow R_t = -\alpha\left(\frac{D_t}{D_{max}}\right) \tag{6.2}$$

  For explanatory example: if the lander is very close to the target and approaches it it will get a negative reward close to zero, if it is at a great distance from the target but decreases its distance it will get reward close to $-\alpha$. For the solution of the scenario $\alpha = 0.5$ was selected, here $D_{max} \simeq 1171$ m.

- **Minimal approach fail**: it is the complementary case to the previous one, a in this case a reward of value -1 is assigned, that is:

$$\forall\, t,\ D_t \geq D_{t-1} \Rightarrow R_t = -1 \tag{6.3}$$

In the light of what is stated in Subsection 6.2.2, wanting to keep the simplicity of the reward function as a guideline, even the assignment for the minimal approach of a reward that is a function of the distance could seem excessive, however, it proved to be fundamental.

η

6.2.4 Non-flat suggestions

The fact that the same behavior leads to a better reward in one condition than in another, is a typical example of a mechanism by which we can provide a

"suggestion" to the agent. In the case of approaching the target the condition we are talking about is the position, but this reasoning can be generalized to any state. We could in words interpret it as "we would like the agent to approach that point, moreover doing it by being in proximity to it is preferable than doing it in the distance". Thanks to this foresight, during a training procedure, the lander slows down considerably in the vicinity of the target to enjoy the reward and in this way it "concentrates" on learning here, i.e. being slow it spends more time here so it does more exploration so it has the chance to find the touchdown. The method of providing rewards that are a function of certain conditions represents a valid method to direct the agent's focus in a more descriptive way than a flat indication, but in any case in a non-constricting way.

On the other hand indeed, if the agent has no vision of the difference in the area it is in, it will fly indiscriminately at no low speed, continuing to approach but not being able to find the landing conditions. This was verified through a specific simulation, using this same reward function but with a flat reward (equal to -0.5) for the minimal approach: initially the agent approaches the target but at great speed, always crashes, for avoiding it begins to approach by keeping its altitude higher and to exit distance boundaries, it is unable to perform even one touchdown so it establishes this unsuccessful behavior. The graph of the reward obtained during the training is omitted as it is almost constant from the first 200 thousand steps to the final 2 million.

As for the terminal rewards, the prize for the landing is a function of how much positive the conditions reached are, failures correspond to negative rewards:

- **Touchdown**: the reward R_t is a positive valued between 500 and 2000. It is composed of a flat part of 500 to which two scores are added, these two values represent the goodness of how much respectively the position and the velocity reached are optimal. The first score s_d is calculated as:

$$s_d = \beta \left(1 - \frac{\sqrt{d_{fin,X}^2 + d_{fin,Z}^2}}{r_{target}} \right) \tag{6.4}$$

  where r_{target} is the radius of the target (5 m) and the numerator of the fraction is the distance of the lander calculated on the X and Z axes; β is a positive value set to 900 for this scenario. The second score s_v is:

$$s_v = \gamma \left[3 - \left(\frac{v_{fin,X}}{\frac{v_{max,XZ}}{2}} + \frac{v_{fin,Y}}{v_{max,Y}} + \frac{v_{fin,Z}}{\frac{v_{max,XZ}}{2}} \right) \right] \tag{6.5}$$

where v_{fin} are the final velocity of the lander in each axis, $v_{max,XZ}$ is the maximum horizontal velocity for the touchdown to be achieved (1 m/s), and $v_{max,Y}$ the vertical (1 m/s); γ is a positive value set to 200 for this scenario.

θ

6.2.5 Achieve it first, then optimize

The conditions that determine the success of an objective are usually formalized as sufficient conditions, but often the achievement of them can be improved with the ambition of learning to always reach an optimal condition as well as sufficient. In the specific case of the lander, we would like it to always land exactly in the center of the target rather than near its edges, and that the final velocities are as close to zero as possible. This would give more security to keep away from failure states, an aspect of fundamental importance in the inference phase. If hypothetically the agent learned to land without distinction between the center and the edge of the target, this would be left to chance, but landing close to the edge would mean running a much greater risk of failing, without him knowing.

Therefore, after it has learned to reach the minimum satisfaction conditions, we are faced with an optimization problem, it is necessary to introduce a mechanism that allows the agent to improve its capabilities. The positive reward for the touchdown introduced in this scenario has exactly this purpose. The flat part of the reward ensures the positive response even if success is achieved with the minimum indispensable conditions; the additional part is the bonus which has the optimum condition as its maximum, obviously the agent wants to maximize it. The optimization process takes place within the training procedure itself, in the first part of the training the agent learns to reach the goal with sufficient conditions, in the second part it will continue to reach it gradually learning to do it in an optimal way. If we don't use this strategy and made, for example, the conditions of success more strict and close to the optimal, we would risk that the agent will have too much difficulty in finding them.

- **Target hit**: the reward assigned is -250, it is smaller than the other two failure cases since less serious, or rather an admissible error in the learning phase and therefore not excessively sanctioned
- **Ground crash**: the reward assigned is -500. It was thought to assign a reward that is function of the distance from the target, the more negative the more the collision occurs far from it (similarly to what was done for the

minimal approach and discussed in Section 5.2); however, this turned out to be superfluous so it was removed in favor of simplicity.

- **Maximum distance exceeded**: the reward assigned is -500. Remember that in this scenario the dimensions of the flight space are limited to $(1600, 300, 1600)$ m.
- **Maximum step exceeded**: no reward is assigned.

6.3 Scenario Solution

The reasoning and conditions explained above led the resolution of this scenario, Figure 6.2 shows the statistics of the training procedure, they are particularly linear and without strange trends. It can be noted that already when 500,000 steps are reached, the landing is achieved (as positive reward is earned), and from here the optimization procedure described in Subsection 6.2.5 acts, in particular from 1.5 to 2 million steps, in which the reward is practically always positive but still constantly increased on average.

The test, of 2000 episodes in total, carried out with the agent in inference mode revealed the following results:

- **Touchdown**: 97.7%
- **Target hit**: 2.2%
- **Ground crash**: 0.1%
- **Maximum step exceeded**: 0.0%
- **Maximum distance exceeded**: 0.0%

The results are excellent and the implementation has been continued.

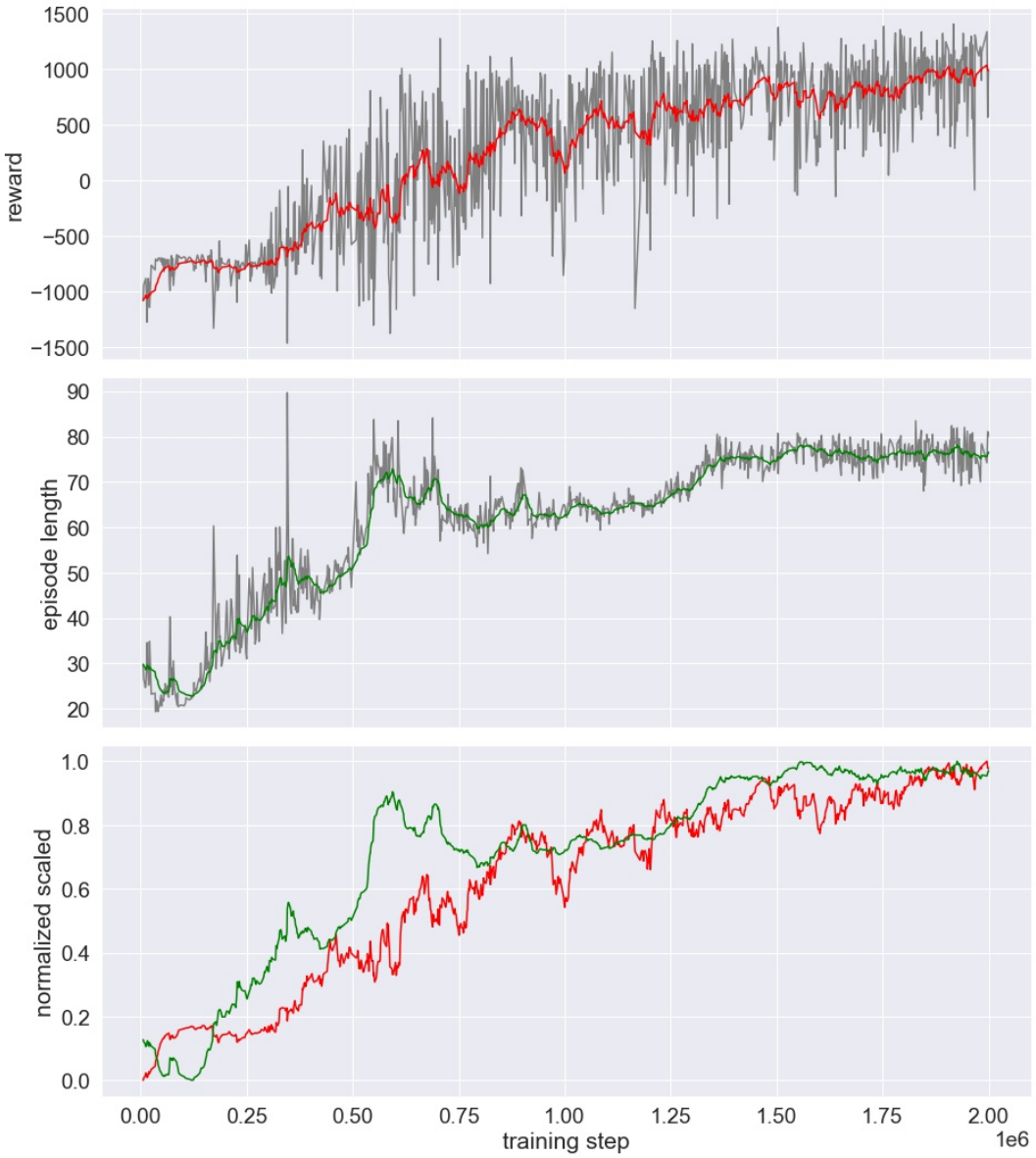


Figure 6.2: Scenario Version 2 training statistics

Chapter 7

Autonomous Lunar Lander: 3-DOF scenario Version 3

After implementing and solving satisfactory initial and final conditions in the Version 2 scenario, it was decided to introduce a fuel consumption system and the relative non-constancy of the mass, since it is one of the most impactful physical aspects in the real operation of a spacecraft. It is an interesting step forward as it represents an optimization problem, the same widely addressed in [1], furthermore non-infinite fuel determines an intrinsic time limit for each episode.

7.1 Physical Model

The **mass** of the lander will now be formed by two different parts: the dry operating mass, that is constantly equal to 7000 kg, and by the mass of fuel on board. At the beginning of each episode the lander has a total mass of fuel of 8000 kg at its disposal; the ignition of each certain thruster involves the consumption of a certain quantity of fuel and this determines the loss of a certain quantity of mass. The ignition of the main engine results in a fuel consumption of 100 kg/s, the ignition of one of the lateral thrusters results in a fuel consumption of 50 kg/s.

The thrusts provided by the **thrusters** have been diversified, the main engine now produces a force of 25 kN, each of the lateral thrusters a force of 12.5 kN, the different fuel consumption is correlated and obviously proportional to the thrust. This represents slightly better the preponderance of vertical thrust than the others, but due to the constraint to 3-DOF the modeling remains very far from realistic.

Note that the quantity of fuel consumption is expressed in mass per second but the decision to turn on or not a thruster is maintained for the entire duration of

each decision step. Therefore in the case of the main engine for example with a decision period equal to 10, if the decision is made to fire it, the following quantity will be subtracted from the mass of fuel:

$$\delta_{m,decision} = \Delta_{m,second} \left(\frac{decision\_period}{physic\_updates\_per\_second} \right) \tag{7.1}$$

where $\Delta_{m,second}$ is the quantity expressed in mass of fuel per second and $\delta_{m,decision}$ is the equivalent spent in the duration of a single decision period. With the parameters used we will have:

$$\delta_{m,decision} = 100 \ \frac{kg}{s} \left(\frac{10}{50 \ Hz} \right) = 20 \ kg \tag{7.2}$$

As for previous implementations, the mass is uniformly distributed over the entire volume of the lander, even if not constant. In the real world the fuel is stored in tanks, usually placed under or around the passenger compartment, this cause the loss of mass to be localized and therefore there would be the displacement of the center of gravity, an extremely influential factor. Moreover, a generic and simplified fuel is used, while in reality it consists of a fuel and a oxidizing with different physical properties. The other aspects of the physical model have been maintained, they are summarized in Table A.9.

ι

7.1.1 Adapt RL to problems, not vice versa

After implementing the model, since this scenario was far more complex than the previous two, looking for the solution of the scenario the question often arose as to whether or not the actions available could have been able to achieve the intended objective given the current physical model. In fact, although the controls and the model are dimensioned on the basis of realistic data, it is not always trivial to be sure about it. Controls are testable with an heuristic method, but with increasing complexity of the problem it is not easy to understand it.

When the solution is hard to be found, driven by uncertainty, there is the temptation to modify the physical model in order to get closer to it. Modifying the model so that the configuration of the current algorithm allows to reach the goal is however conceptually a serious mistake. In fact, it should not be forgotten that our goal is not to build a physical model and set the RL algorithm in order to achieve a desired behavior, but to start from a well-defined and formalized control problem and then apply an algorithm capable of managing it.

This rule can be extended to any field to which the reinforcement learning approach is applied: it is always important to remember that we are using a methodology to deal with a problem, and therefore we do not have to change the problem to ensure that it can be solved by this methodology.

In facing the resolution of this scenario this error was made, failing and thinking that with the simple on-off control of thrusters the landing could not be achieved, a more sophisticated and smoothed thruster system with continuous intermediate values was implemented. However, this radically changed the nature of the problem and realizing this fact, this implementation has been consequently removed. The control problem is independent of the reinforcement learning application as how here it is exposed, this guideline has been maintained in this and the next scenario addressed.

7.2 Reinforcement Learning Application

Starting from the configurations exposed that had worked in the previous scenario, many initial attempts to solve this new implementation did not lead to good results for several reasons. First of all, a serious problem consists in the fact that only negative rewards are assigned, with the exception of the success condition but this is never reached for most of the initial part of the training; this proved necessary in order to avoid *laziness*, as explained in Subsection 6.2.3. In the initial part of the training the episode often goes by in the following way: the agent learns to fly and learns to do it in the direction of the target under the suggestion of the reward function (from the minimal approach part in particular); then, not being able to land yet, it ends the episode mostly by crashing. So at the end of the episode the overall reward is given by the sum of the negative periodic rewards (however optimized) and the final negative reward. If instead it happens that the lander crashes very quickly, which is not rare in the exploration phase, the agent will receive the terminal negative reward but since the episode lasted a short time he will receive little periodic reward. The final balance of the episode will be better than the first case, essentially in the absence of touchdown it is overall more advantageous an episode in which the lander crashes immediately rather than one in which it flies for a certain period and then crashes; we can informally call this trend *suicide*.

Wanting to generalize this fact to any environment, we could say that in the case of positive rewards that are difficult to be earned easily, the training procedure can stabilize on behaviors that end the episode in a failed manner in the shortest possible time. This does not happen constantly, as it is probably caused by the randomness of the first phase of the training procedures, an estimate made on the

numerous training procedures attempted shows a frequency of 30%/40%, obviously when this happens the entire training is thwarted. Figure 7.1 shows the graphs obtained in one of these cases: note how in the first phase of about 50 thousand steps the agent explores while flying, here the episodes have a duration of 50/70 seconds and the reward obtained is very low; after that the behavior stabilizes for a very low duration with a higher reward, as it ends the episode by crashing as soon as possible.

Another serious problem is intrinsic to the duration of the training and the difficulty of the task. In many training simulations the agent fully learned to fly and approach the target into about half of the training steps (initially set to 2 million, as for the scenario Version 1), but when it had to learn to land it still used to spend a lot of time in the first phase states region just because the episode was reset very often. Furthermore at this point of the training both learning rate and epsilon were already low, so much that the exploration and consequent learning are too faint. So the lander could not learn to achieve even a single touchdown because it spent too little time in states "near" the target, and because it was able to explore so little here that it never even tasted one success reward.

The most trivial solution for this problem could be to increase the training time but this is as immediate as inefficient, primarily because facing any increase in difficulty in this way will probably become unsustainable. For this reason, several training strategies has been studied, in order to lighten the computational load and lead to the desired final behaviour of the agent. *Training strategy* means manipulations and rules applied in the training phase, they alter the normal mechanism that the training simulation would have, it can therefore assume trends that do not reflect reality. Their purpose is to make the RL algorithm converge faster to the wanted solution, so the aim remains to achieve an agent able to act in the real world conditions. In other words, the trained agent must be able to act successfully by exploiting the inference without being able to distinguish whether his training took place under realistic or manipulated conditions.

In particular, the tested strategies are three:

- **Deny-Unsuccessful-Terminations**, abbreviated DUT, it consists in removing all the termination conditions of the episode except the success one and the time limit, it is described in Subsection 7.2.1
- **Forced-Successful-Observation**, or FSO, it consists of periodically providing the agent with observations of successful states, it has been tested in the 6-DOF scenario and described in Subsection 9.2.1
- **Reverse-Incremental-Progression**, or RIP, it consists in starting the training with simplified initial conditions to gradually reach the desired difficulty

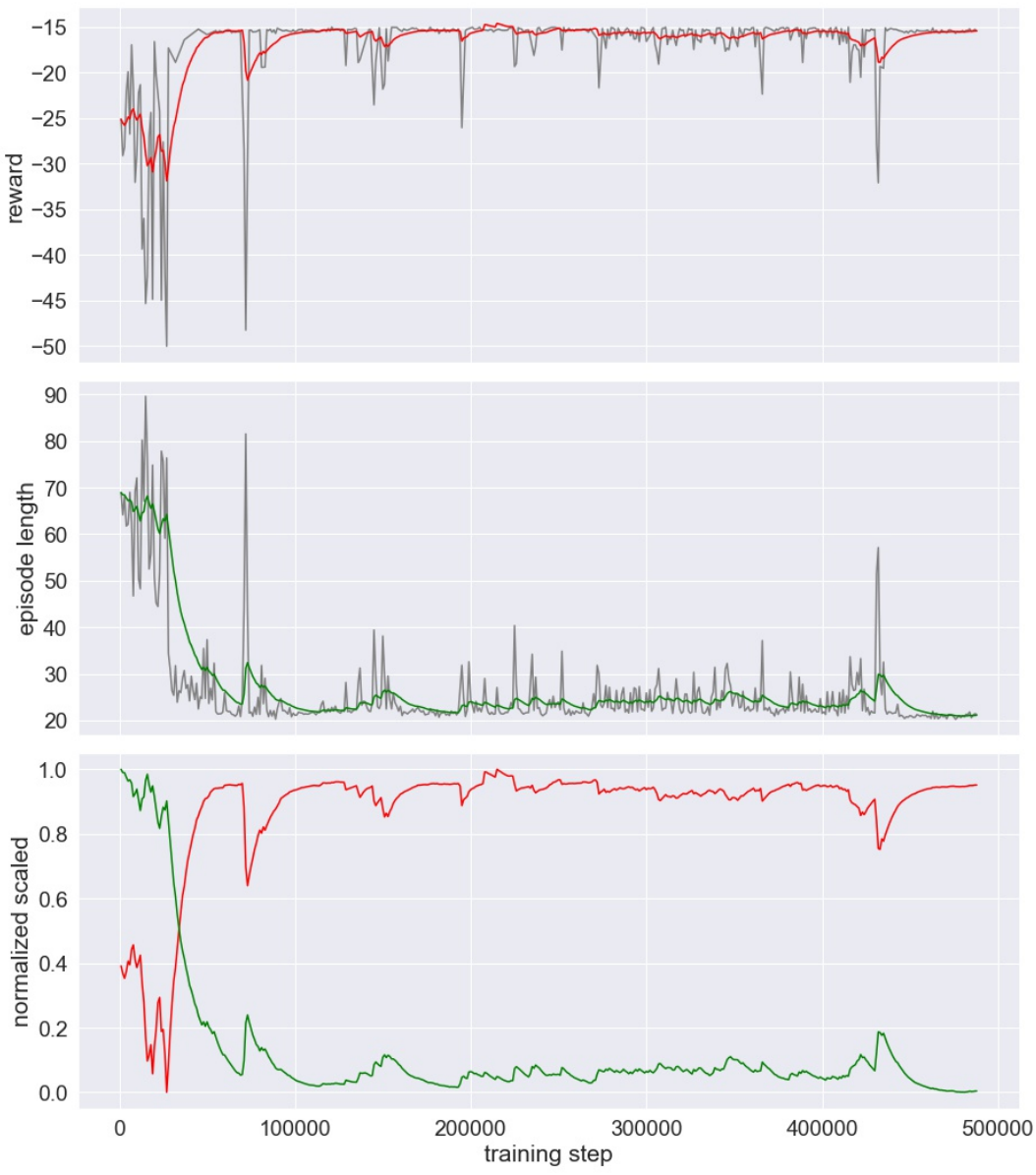


Figure 7.1: Graphs obtained during a training procedure that stabilized on a *suicide* behaviour, note how the two curves are practically mirrored.

at the end of it, it has been applied in the 6-DOF scenario and exposed in Subsection 9.2.3

This type of approach was inspired by the work exposed in [27], in which

autonomous drones are trained to fly across a circuit. Here the problem is that the initial part of the path is covered many more times than the others. So a *Distributed Initialization Strategy* was applied: during a first phase of the simulation the starting position is placed randomly along the way in order to explore it uniformly, in particular in points where difficulty (velocity to be managed in this case) is assumed to be low, therefore: "we retain the benefit of initializing across the whole track while avoiding the negative impact of starting from a hover position in areas of the track that are associated with high speeds."

7.2.1 The DUT Training Strategy

The main idea underlying this strategy is to no longer interrupt and reset the episode when the agent runs into some states that in the real world would terminate it. When any collision occur the agent is brought to a near and safe state and a negative reward is assigned to it, thereafter the episode continues normally. Since the state it is brought into is "near", later the agent will continue to explore and make decisions as it was doing just before it went wrong, as if it was not wrong. But the new state also have to be "safe", i.e. the subsequent choices that the agent will take must allow him to not incur again in the same or other only negative terminations, otherwise it would find himself trapped into a loop of failures. The reward assigned to the occurrence of these situations must be reasonably negative so that at the end of the training they must necessarily be avoided, since in inference phase they still would lead to actually negative situations, as in real world they are.

To analyze the example case of this scenario makes it easier to understand: when the lander meets the conditions of *target hit* or *ground crash* its position is instantly placed 5 m higher of where it is; when it exceeds the maximum distance reachable its position is instantly placed 50 m far from the air limits touched, thereafter the episode continues normally. In both cases a large negative reward is assigned as specified by the reward function in Section 6.2, the distance of reset is a parameter that has to be tuned too. The distances used (5 m and 50 m in this case) and in general the reset conditions obviously have to be properly chosen depending on the scenario.

So now the only two cases that cause the termination of the episode are the exceeding of the time limit of it and the success in landing. Until the success is not achieved, the agent will spend the entire duration of every training episode exploring state-action-rewards, and each collision is handled as described above. At the beginning of the training it explores equally in every region and phase of the scenario, but as it improves its behaviour on the easier initial phase (the target approach), the exploration focuses on the second phase (the landing) and few

resources are still used in the first. It is necessary to introduce the concept of *Dirty Touchdown*: this term will indicate those touchdowns reached in an episode in which failed states had occurred previously, which were restored using the DUT strategy. Obviously a dirty touchdown can only occur in training procedures because, as said, the inference phase remains exactly unchanged: unsuccessful situations remain failures.

For now we have not yet talked about fuel consumption, and in particular the critical condition in which it runs out, this represents an additional negative termination condition introduced in this scenario. Other than the crashes, it was chosen to be avoided as well during the training. So during the simulation, when the fuel runs out, the lander can continue to use the thrusters as if nothing had happened and their use does not cause a further loss of mass, but the theoretically used amount of fuel is tracked. The lander will learn to land having used more fuel than is actually available, thus reaching dirty touchdowns, but the final reward will push it to optimize consumption, until it is able to land with excess fuel. We could summarize this approach as a transformation of an hard constraint, i.e. the limited amount of resources, into an optimization problem that has the minimum objective to be able to advance a positive amount of resources. A sort of union between the DUT strategy and optimization approach, theorized in Subsection 6.2.5.

The use of nonexistent fuel as well as instantaneous position changes are obviously unrealistic events, they are manipulations inserted in the simulation environment but there must be certainty that when using the inference these situations do not occur anymore, i.e. that the agent avoids run into it. Basically we realize it if at the end of the training there are only realistic touchdowns and no more dirty touchdowns. In order to achieve that, in the case of collision the negative reward to be attributed and its size are fundamental, in fact if during the training the punitive reward is not big enough, the agent could accept to suffer it because the change that derives is still advantageous. As regards the use of fuel beyond the limit, various analyzes have revealed that by attributing a negative reward to the achievement of the maximum use of fuel, the agent reacted by deciding not to use it and consequently stabilizing unsatisfactory behaviors, for which it was decided to exploit the negative fuel mechanism. By transforming fuel consumption into an optimization problem, we obtain the desired result without the need for further periodic rewards. To summarize: the negative reward assigned in DUT cases must be large enough for the agent to learn that that situations must be avoided in any case, and not exploited, but in the case of transformation of a hard constraints into an optimization problem the reward can be omitted.

Notate that in this scenario the near and safe state is just assumed as a position

far enough from the spatial limit reached, but the strategy is generic and can be applied to other state values, such as the fuel for the scenario Version 3. No manipulation is applied on the velocity because the lander has the possibility of not incurring the collision again in the following steps using the appropriate thruster, possibly after some other collisions. It was thought to reduce the velocity in the direction of the collision, but this adds the problem that the lander could assume the behavior of colliding and accept the negative reward in order to decrease its speed. As a source of additional complexity rather than advantage, this aspect has been removed.

Let's lastly summarize the two main guideline methodologies that this strategy offers, which in this scenario were applied respectively for the constraints on positions and fuel consumption:

κ

7.2.2 Near-and-safe resets

In scenarios in which the conditions that determine the termination of the episode are multiple and are very frequent compared to success, the exploration of states far from the initial one may prove to be highly unbalanced due to frequent resets. During the training phase it is possible to avoid ending the episode in these situations but to bring the agent into a near-and-safe state (understood in a generic sense, not necessarily in terms of position), obviously in an unrealistic way. The reset in a *near* condition allows to continue the exploration in the region where the fatal error occurred, the *safe* condition ensures to avoid failing loops. In this way the agent can have a better chance of knowing the success condition. These resets must be categorically avoided and not exploited at the end of the training, since in the inference phase they would result in the original case of failure, it is therefore advisable to assign a negative reward of significant magnitude in their correspondence.

λ

7.2.3 Hard constraints as optimization problems

The presence of hard constraints that determine the termination of the episode can cause the limited exploration of the state space, and the consequent difficulty in knowing the success condition. It is possible during the training phase to eliminate these limitations, by implementing unrealistic mechanisms, so that the episode is not terminated and that the agent can learn regardless of them.

Obviously these constraints must be respected in the inference phase, it is possible to translate them into optimization problems, for example by inserting a score (a variable positive reward) in the condition of positive termination. The achievement of optimization down to a minimum threshold ensures that at the end of the training the restrictions will not be violated.

Let's now see the other parameters that led to the solution of this scenario: the value of the residual fuel mass has been added to the vector of the **observations**, or rather the quantity of fuel equivalent to the use of the engines carried out so far. Therefore for the correct functioning of the consumption optimization, it is essential to pass the negative value when the mass of fuel reaches zero, this will be clarified in the description of the reward function. The negative value of fuel used is limited to -8000 kg, for the reason described in Subsection 5.2.1.

The **decision period** has been decreased to 0.2 s, i.e. one decision step every 10 physical update steps. This allows more accuracy in the landing phase, at the expense of a longer **training time**, it has been increased to 10 million decision steps; this is a considerable increase but necessary to allow the optimization phase to complete asymptotically. The number of **maximum steps** per episode has been increased to 5000, corresponding to 100 s, a small increase to allow the DUT strategy to be fully exploited.

The value of **stacked vectors** is mantained 1, value greater than 1 could conceptually seem erroneous if used together with the DUT strategy as the agent would have the memory of an unrealistic event within several state vectors in which a failure condition has occurred. However, this should not lead to a concrete problem assuming that at the end of the training the agent will not run into such situations. The choice of this parameter has been simply chosen as result of better computational performances with the same training result, it will be specified in the next Chapter.

Regarding the hyperparameters that govern the PPO algorithm, compared to the previous scenario **beta** has been doubled from $5 \cdot 10_{-4}$ to 10_{-3} and **epsilon** has been decreased from 0.25 to 0.2. It is reiterated that benchmarks for the choice of these parameters will be set out in Chapter 8.

Let's now analyze the reward function structure, in general compared to the Version 2 scenario, the size of the rewards has been decreased by a factor of 10, this simply determines a greater readability of the results, given that numbers obtained during an episode are more easily commensurable. Let's analyze the periodic rewards:

- **Adaptive approach**: it represents the overcoming of the generic nature of the *minimal approach* and the rigidity of the *fixed approach*, it differs from the latter due to the fact that the a_{min} distance that determines the success of the approach is not constant, denoted with $a_{min,t}$. It is calculated as follows:

$$a_{min,t} = \frac{D_{t-1}}{\max\left(t_{max} - 1000 - t, 1000\right)} \tag{7.3}$$

  Basically this value represents, at any instant t, the minimum advance that the lander must make in order to reach the target within a time limit, if it moved at a constant velocity from now on. The time $t_{max} - 1000$ is the time in which it is thought the lander should reach the target, 4000 time steps in this scenario equivalent to 80 s; it has been dimensioned on the basis of the fact that the fuel limitation already imposes a limit. So $t_{max} - 1000 - t$ is the amount of steps that remain before this time limit; the max operator introduces a limitation that masks values that are too low, this is done because if the time left is too little, a velocity that is too large would be required, but these are to be avoided so it is preferable not to request it at the cost of fail the episode. Rewriting Equation 5.2 with the new $a_{min,t}$, we obtain the condition for the achievement of an adaptive approach:

$$D_t < D_{t-1}\left(1 - \frac{1}{\max\left(t_{max} - 1000 - t, 1000\right)}\right) \tag{7.4}$$

  In the case that this condition is met, the reward R_t assigned is a negative value in the interval $[-0.01, 0]$ linearly inverse to $D_t \in [0, D_{max}]$, as exposed in Section 5.2 using a $\alpha = 0.01$ factor.

- **Adaptive approach fail**: in the case that the previous condition is not met, a flat reward of -0.1 is assigned

Numerous simulations were carried out in which negative rewards were assigned in case of use of thrusters, as a method of optimizing fuel consumption. This approach has always proved very disadvantageous as the agent quickly stabilized on behaviors that avoided using them at all.

μ

7.2.4 Conscious or superficial behaviors

It is crucial to remember that the stratification of a complex reward function is invisible to the agent. The agent only perceives the resulting sum of all rewards and does not understand how much and what contribution each part of the observation gives to it, or rather, it deduces them during the training procedure

by observing millions of state-action-reward instances. If we built a complex reward function in which numerous rewards are assigned, calculated on the basis of numerous different factors of the observation state, it is very likely that the agent first "understands" the simpler parts of it. Consequentially, it optimizes the rewards that can be easily correlated with the actions taken, this can determinate the affirmation of behaviors with actions that can be easily optimized, but which totally ignore the rewards calculated in a more complex way.

This effect was observed by simultaneously assigning a negative reward for the ignition of the thrusters and for the *adaptive approach*. The first condition is easily correlated with the chosen actions, given that if the thrusters are always kept off the fuel is kept and the reward is optimized. The second condition is based on a more complex formulation linked to observations and actions, that is non-trivial, it requires much more exploration and time to be understood and optimized. Also note that the first condition is not even correlated with the observation of the state vector, so exaggeratedly trivial.

The size of the rewards plays an important role: basically, if the more trivial achievements have a smaller prize, they will be optimized later. But again, as said in Subsection 6.2.2, the better solution would be to keep the reward function as simple as possible. In case the complexity of the environment, and consequentially of the reward function, is excessive it is necessary to increase the training time so that the exploration is more extensive, as we will see in the next chapter. It should also be noted that in this scenario, the fuel optimization has been successfully moved within the landing reward, an excellent solution in the case of environments with a single objective and multiple optimization aspects.

In light of this, let's now analyze the other cases, since the DUT strategy is used in this scenario, now some cases are not terminal but non-periodic, remember that in these cases the episode does not end but the lander is brought to a *near and safe* condition which is specified here:

- **Target hit**: the reward assigned is -2.5, the lander is placed at a height of 5 m (calculated with respect to its base) keeping the same X and Z coordinates.
- **Ground crash**: the reward assigned is -5, as for the previous case, the lander is placed at a height of 5 m (calculated with respect to its base) keeping the same X and Z coordinates.
- **Maximum distance exceeded**: the reward assigned is -10, the lander is placed at a distance of 50 m from the point where it collided with the flight space limit.

The only conditions for the termination of each episode are the success and the end of the maximum time (which does not involve the assignment of any reward as well as running out of fuel, as previously said):

- **Touchdown**: the reward assigned in case of successful touchdown is a positive value $R_t \in$ [0,300] composed of three contributions:
  - the score $s_d \in$ [0,90] calculated on the basis of the distance from the exact center of the target using $\beta = 90$, as formulated in Section 6.2
  - the score $s_d \in$ [0,60] obtained on the basis of velocities on each axis using $\gamma = 20$, its formulation is close to the previous one.
  - the score s_f, it is calculated starting from the mass of fuel m_f left at the end of the episode (positive or negative) as follow:

$$s_f = \frac{\delta}{2}\left[1 + \operatorname{sign} m_f \left(\frac{\max\left(|m_f|, 2000\right)}{2000}\right)\right] \tag{7.5}$$

    where $\delta = 150$ has been used. This is basically a mapping similar to the previous two but with the possibility of a negative starting value, it linearly associates values from the interval [−2000,2000] to the interval $[0, 150]$. The upper and lower 2000 kg limits are introduced in order to avoid excessively large rewardswhich would destabilize learning.

7.3 Scenario Solution

We have outlined all the features that were necessary for the training simulation that allowed the resolution of this scenario, let's now

- Figure 7.2 shows three-dimensional graphs obtained by viewing logs taken during progressive throughout the entire training procedure. Six significant episodes are shown, here we can observe the process of change that occurs in the agent's behavior as well as what actually happens with the application of the DUT strategy. As the legends specify, the dimension of the scatters represents the mass of the fuel on board (it is 8000 kg each episode start), remember that in the training phase when it reaches 0 kg it remains constant and the lander is still able to use the thrusters; the color represents the velocity of the lander, on a scale between 0 m/s and 15 m/s. The three spatial axes, of actual size (1600,300,1600) m, are zoomed in on the trajectory for greater readability.
- Figure 7.3, similar to those analyzed in the previous two chapters, represents the trend of the reward obtained and the length of the episodes along the

training as well as the two normalized curves. The graph in Figure 7.4 is a zoomed version of the first (where the large negative rewards are omitted).

- The two graphs in Figure 7.5, moreover, respectively represent the cumulative count of the three different types of termination of the episode (touchdown, dirty touchdown and maximum time exceeded), and the average count of terminal conditions avoided using the DUT mechanism (target hit, ground crash and maximum distance exceeded).
- Lastly, Figure 7.6 shows the trend in fuel consumption, on the abscissa axis we have the time within an episode (expressed in seconds, capped to 100), on the ordinate axis the mass of fuel used in kilograms. The color change of the curves indicates the increase of the episodes tracked, red lines represent the last episodes; one episode every 30 actually occurred is represented here. The value can fall below zero (highlighted by the horizontal black line) according to the application of the DUT strategy, this occurs largely in the first episodes of the training.

Note that the temporal quantities utilized on the abscissa axis of different type of graphs are not exactly the same: with *training step* we mean the total number of decision steps that make up a training procedure (in the order of millions), with *episode* we indicate the single incremental episodes of variable duration (with a maximum of 100 seconds and a minimum reached asymptotically by the convergence of the algorithm, around 45 s). The *time* unit in seconds has a completely different meaning since it indicates the duration of a single episode and not the entire training.

Let's now analyze the training procedure trend in detail, we will use the six main episodes represented in Figure 7.2 as starting point:

- **$\sim$0.5% of episodes** (Figure 7.2.a): at the beginning of the training the lander can not fly, it collides numerous times both on the ground and on the space boundaries (up to 200 and 100 times, out-of-scale values relative graph), it seems to move "bouncing". The velocity is always very high and the fuel runs out quickly, reaching values of -3000 kg (corresponding to 11000 kg used out of 8000 kg available). After each of these collisions the lander is repositioned and the relative negative reward is assigned, indeed in this phase the total rewards are very low (down to less than -1000) but in this way he will soon learn to avoid them. All episodes end when the time limit is reached.
- **$\sim$1% of episodes** (Figure 7.2.b): the lander still advances colliding on the ground many times, but in proximity to the target it slows down and flies over it. It already avoids colliding with the limits of the flyable space, the first good achievement. Since no dirty touchdowns are reached yet, and obviously neither touchdowns, the fuel is still used without restraint.

- **∼3% of episodes** (Figure 7.2.c): the lander clearly slows down in proximity of the target, but it continues to collide with the ground, it moves away from it and then comes back. The decrease in velocity represents a fundamental trend of behavior. Approximately at this phase the first dirty touchdowns are achieved, the episode length starts decreasing and the reward increases. The number of ground crashes remains high and the number of target hits increases as well, this is a clear consequence of the fact that to "find" the landing conditions the lander collides many times near the target, fundamental concept of the DUT strategy.

- **∼30% of episodes** (Figure 7.2.d): we are near to halfway through the overall duration of the training, the behavior now follows a curved trajectory that ends on the target. The ground is still touched sometime (once or less per episode on average) and more rarely the target too. The length of each episode is halved than the time limit, dirty touchdowns and touchdowns are reached almost equally. The fuel optimization phase has begun and will go on until the end, it is obtained by optimizing the reward function in both cases, this is the other main concept of the DUT strategy.

- **∼85% of episodes** (Figure 7.2.e): the trajectory previously observed is more elevated, probably as a consequence of fuel optimization and in order to reduce the risk of collision. Both the reward obtained and the duration of the episodes approach their respective asymptote, the total number of dirty touchdowns reached remains practically constant since now almost only successful touchdowns are performed successfully. The fuel left at the end of each episode is already abundantly greater than zero kilograms.

- **∼100% of episodes** (Figure 7.2.f): the training reaches the end, the final trajectory maintains a curvature near the target but it is much less accentuated than before. It is interesting to note that the velocity is still largely decreased in the last seconds of the episode. Both the reward and the duration have reached their asymptotes, no more collisions occur, the amount of residual fuel is by far positive at the end of each episode. Given all these factors, the result of most of the episodes is successful touchdown.

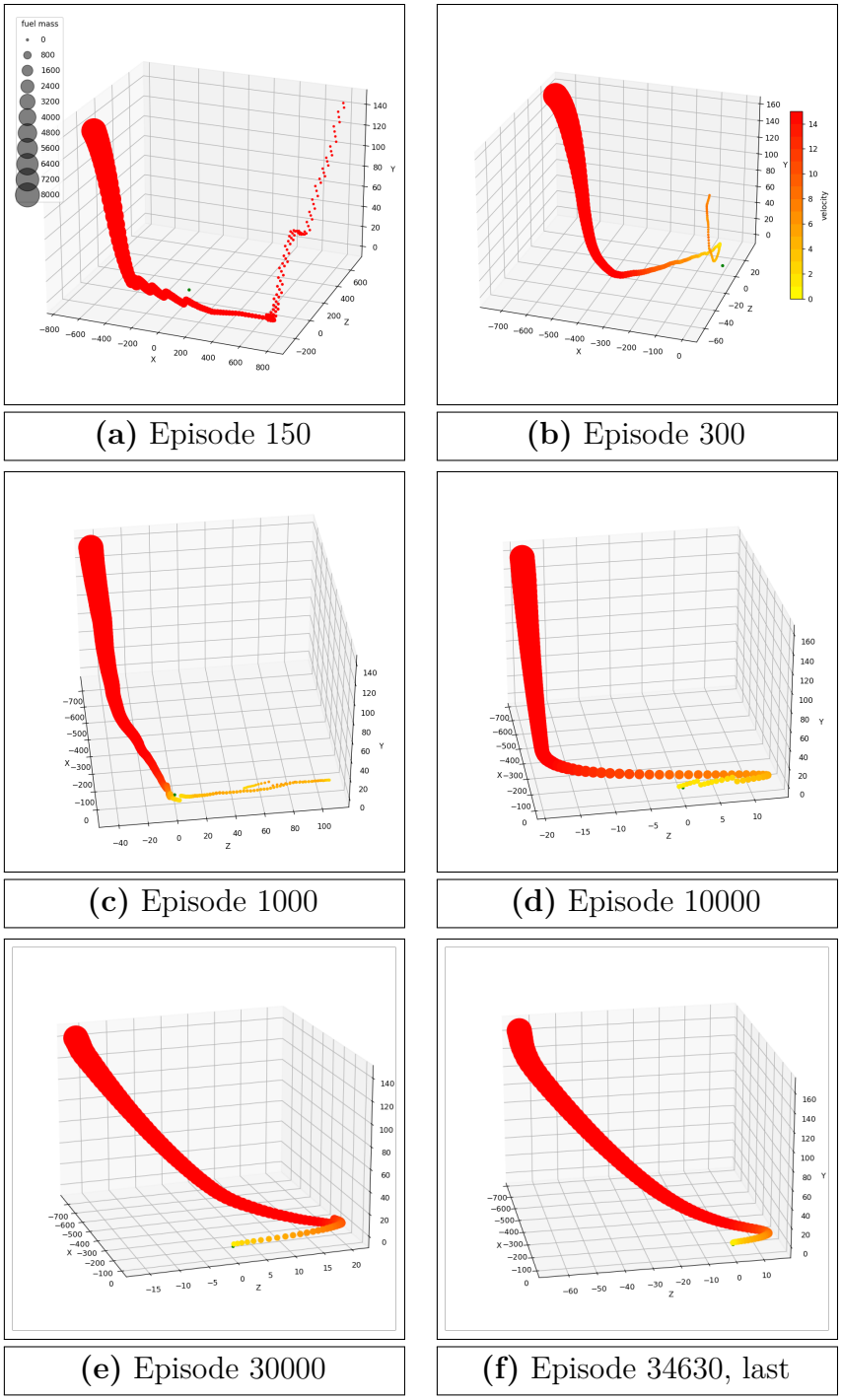
Let's now analyze the statistics obtained by deploying the trained agent in inference mode, over 300 test episodes:

- **Touchdown**: 91.7%
- **Target hit**: 0.3%
- **Ground crash**: 8.0%

- **Maximum step exceeded**: 0.0%
- **Maximum distance exceeded**: 0.0%

A success rate higher than 90% is largely satisfactory, in this scenario the fuel termination is not present as an episode end case as it is not explicitly requested by the physical model, this failure situation results in a ground crash in most cases since without fuel the lander simply falls. In the fourth and final scenario, however, the opposite reasoning will be applied as more specific analysis of the results will be conducted. In the scenario Version 2 the ground crash frequency was practically zero compared to the target hit, in this scenario it is definitely the opposite and from this we can deduce that most of the ground crashes in this scenario are caused by incorrect fuel management. More in-depth analysis of the conditions in the episode endings will be made for the fourth and final scenario.

To sum up, the DUT approach is not applicable a priori, it requires ad hoc considerations and sizing of the reward function, moreover we must be sure that at the end of the training the agent does not exploit unrealistic behaviors. As widely exposed below, in this scenario the DUT strategy has proved effective in achieve a satisfactory result and decreasing the resources necessary to train the agent in achieving success.



(a) Episode 150

(b) Episode 300

(c) Episode 1000

(d) Episode 10000

(e) Episode 30000

(f) Episode 34630, last

Figure 7.2: Log visualization of significant episodes that show the evolution of behaviour during a training procedure that exploits the DUT strategy

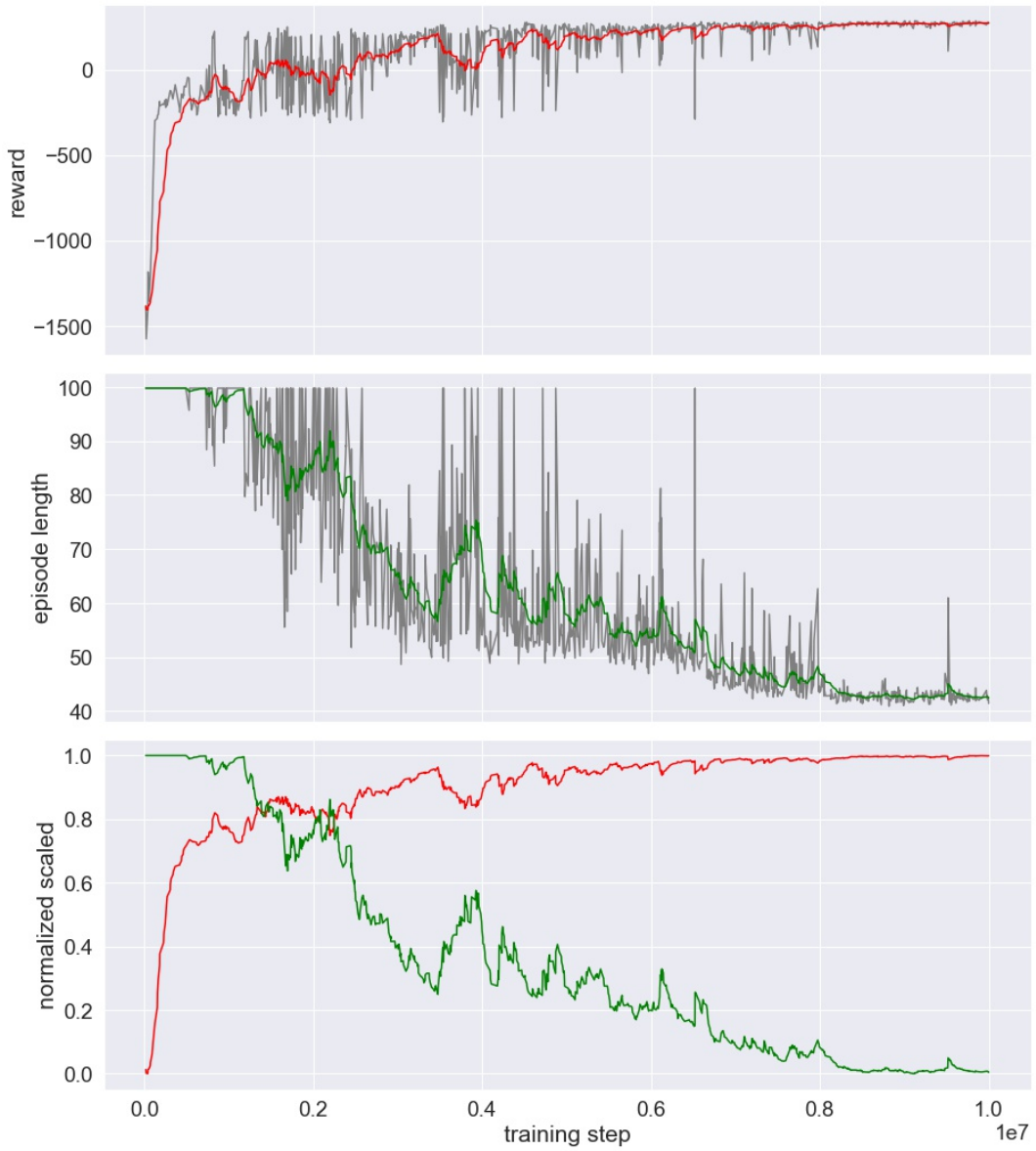


Figure 7.3: Scenario Version 3 training statistics

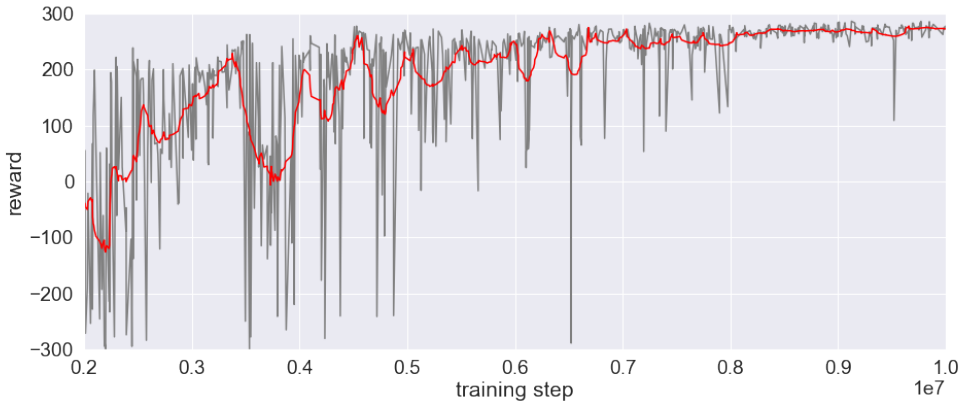


Figure 7.4: Scenario Version 3 statistics of the reward obtained during the training procedure, zoomed in order to remove the great low scores obtained in the initial part of the training.

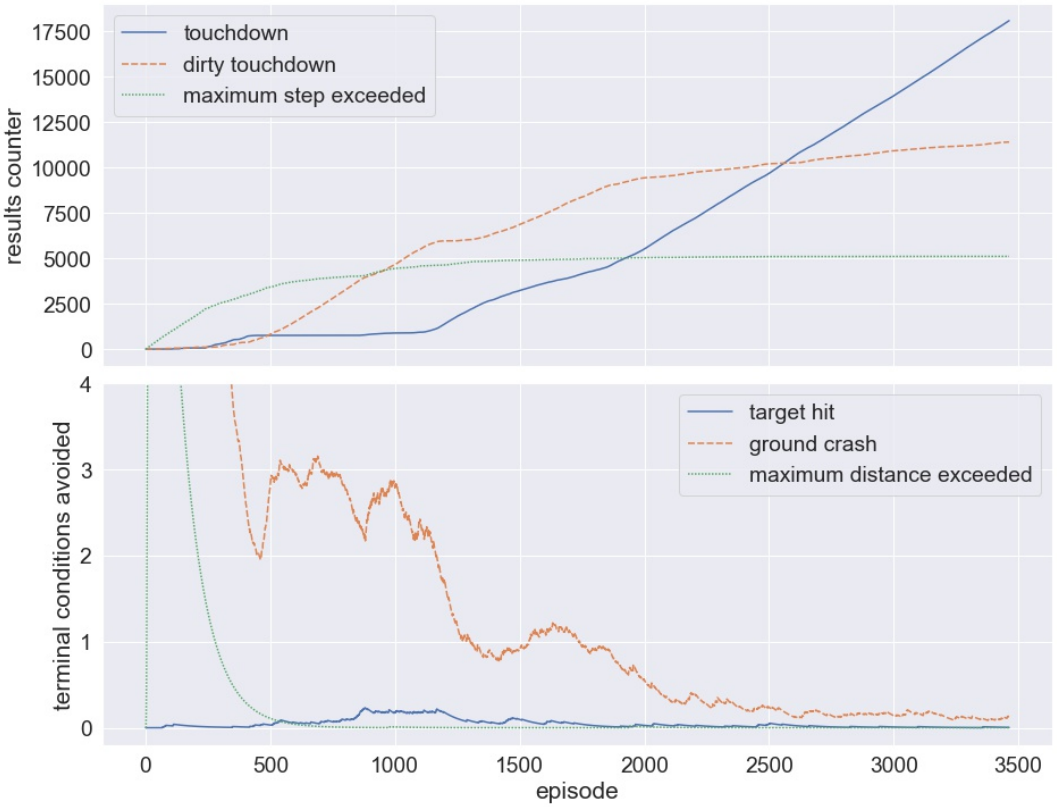


Figure 7.5: Cumulative counter of terminal conditions (above) and counter of the terminal conditions avoided using DUT approach (below).

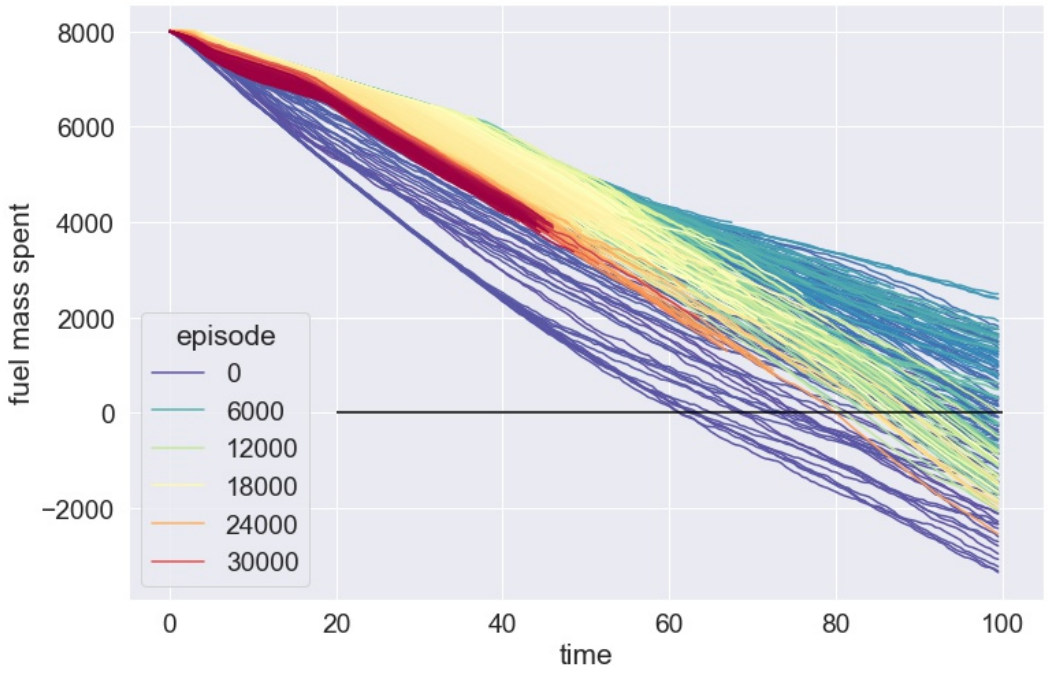


Figure 7.6: Trend of residual fuel mass during an episode, for incremental episodes during the training procedure.

Chapter 8

Training Benchmarks Analysis

The achievement of the solution for the scenario Version 3 was taken as a consolidated result to carry out comparative benchmarks of many aspects. As previously mentioned, in fact, the main problem in this work was the setup and understanding of the numerous variables that can be changed (grouped into the categories RL parameters, PPO hyperparameters and reward function), in this multitude it is difficult to understand what is properly working and what should be changed, consequentially it is not easy to isolate and understand behaviors and trends given their strong interaction. It is believed that keeping a functioning configuration as a reference point and carrying out specific studies can be a good way to draw additional observations. This scenario was chosen because the complexity of the reward function is reasonably low and specially because the training time is short, compared to the 6-DOF scenario that will be addressed in the next chapter.

The training that is used as base for the benchmark comparisons is similar to the one previously described as solution for the 3-DOF scenario Version 3 (unless some minor further specifications, gradually made), it will be displayed in all subsequent graphs in red color. It exploits the DUT training strategy too (but not the other two training strategies exposed in the next chapter). For the comparison between training procedures, the reward trend and the final reward value reached are almost always used, they are based on a reward function very similar to that of Chapter 7; remember that in this chapter this score is taken as a measure of goodness of learning regardless of whether the touchdown is reached or not and with what accuracy.

8.1 Learning Rate, Epsilon and Beta

Let's first analyze the fundamental hyperparameters of the PPO algorithm: their meaning was previously exposed is Section 3.3. The learning rate, in charge of weight the updates of the network, has to be sized with a trade-ff between fast but unstable updates and slow but constant updates. Figure 8.1 shows the benchmark for increasing learning rate values, it is evident that the lower one (0.00001) leads to a stable learning but it converges too slowly, on the contrary 0.0005 is too high and its trend is very fluctuating. With an even greater value (0.001) the in training is totally unsuccessful and settles asymptotically on a score in the order of $-1.25 \cdot 10^4$. Values 0.0002 and 0.00035 are the good trade-off, the second in particular is chosen as base for the benchmarks.

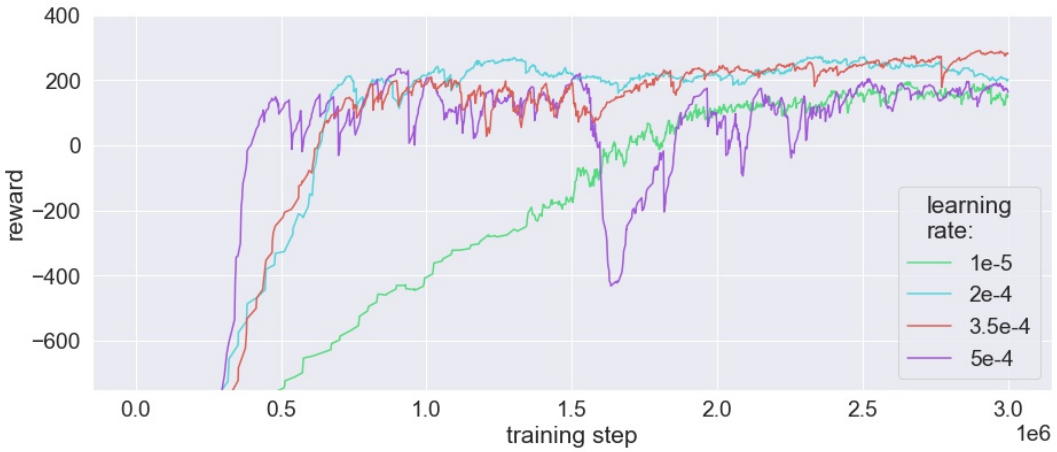


Figure 8.1: Benchmark of different values of the learning rate.

The epsilon parameter determines the rigidity of the algorithm in keeping the policy close to the previous one in the update process. In Figure 8.2 it is evident how a too small value (0.1) significantly slows down the learning to the point that a positive result is not achieved, even if the trend is quite stable. With a larger value (0.3) learning is highly unstable, precisely because the policy makes larger "oscillations". The selected value is the one intermediate.

The correct amount of exploration of states-actions-rewards is fundamental in the training phase, specially at the beginning, but if excessive there is a risk of delaying the convergence and the solution will not be found. This aspect is

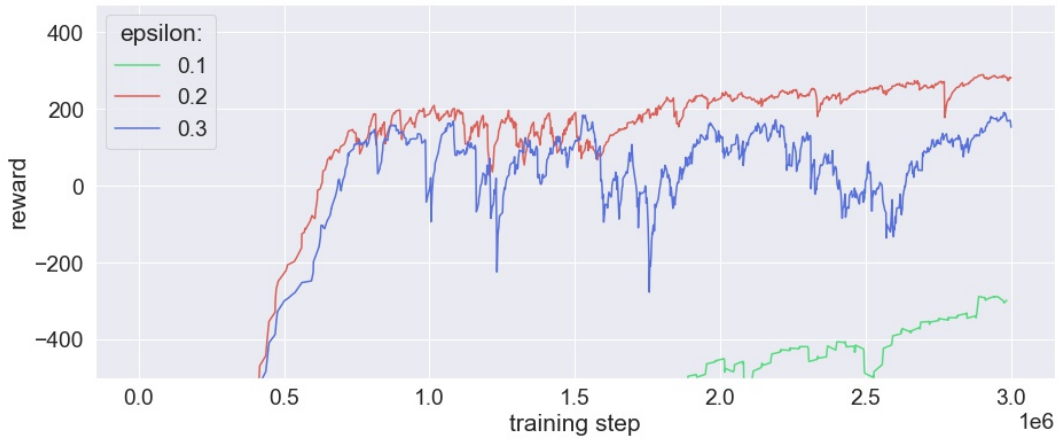


Figure 8.2: Benchmark of different epsilon parameter values.

controlled by the beta parameter, which weighs the contribution of entropy within the loss function. Figure 8.3 shows the comparison for three different values, the graph above shows the reward trend as usual, the graph below shows the value of the entropy which measures the degree of randomness used, remember that it is good that it decays during training neither too quickly nor too slowly. It is evident that the lower value (0.001) results in a premature decay of entropy and a low score, this is because not enough exploration is performed; on the other hand with higher beta parameter (0.1) the entropy value remains elevated for the duration of the training. The intermediate value is considered the best.

8.2 Batch and Buffer Size

Figure 8.4 contains the graphs of several training obtained by varying together the batch size and the buffer size, remember that the first hyperparameter fixes the number of experiences involved during each gradient descent update, the second should be a multiple of it and defines how many experiences have to be accumulated before performing it. As recommended in Section 3.3, since discrete actions are used in this case, non-large values are effective. Any further analysis in the case of using continuous outputs should be carried out.

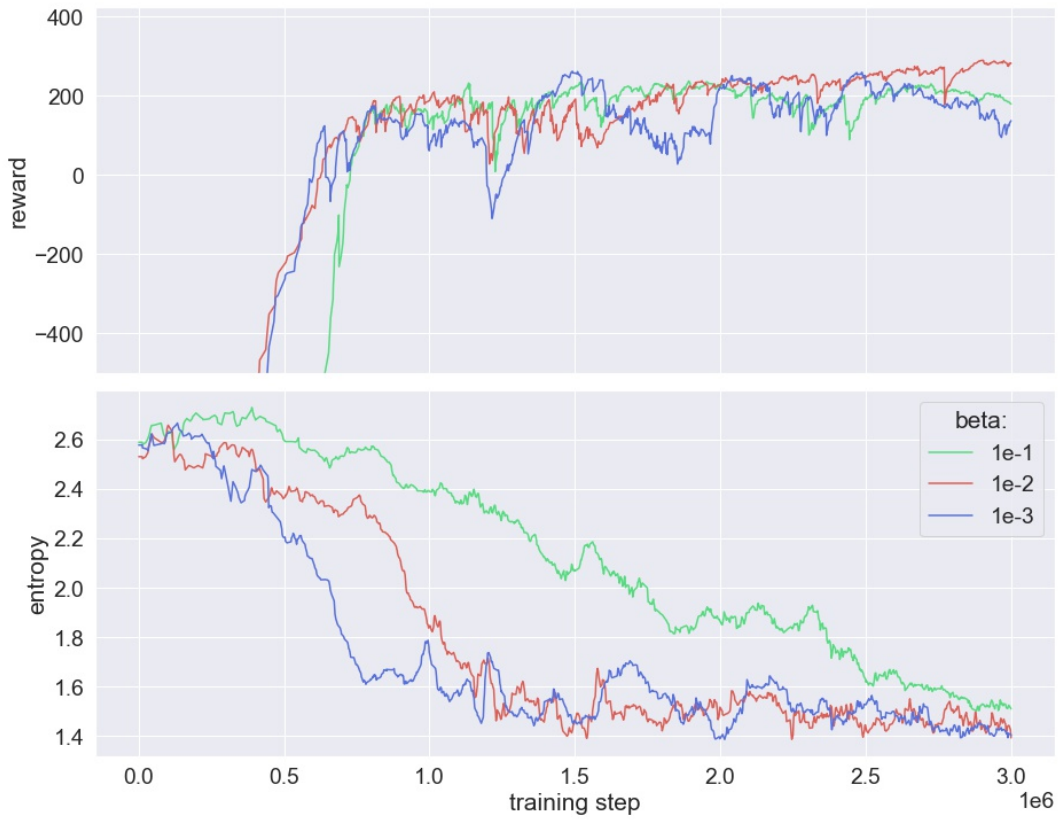


Figure 8.3: Benchmark of different beta parameter values: reward obtained (above) and entropy trend (below).

8.3 Decision Period and Stacked Vectors

Figure 8.5 shows the benchmark for the decision period, remember that the values 1, 5, 10 and 20 correspond respectively to a control frequency of 50, 10, 5 and 2.5 Hz, or even 0.02, 0.1, 0.2 and 0.4 s between one action and another. It is immediately evident that the lower frequency, 2.5 Hz, is not sufficient, in fact the performance is strongly more unstable. The other cases are almost equal, a decision period equal to 10 gets a slightly higher score and good stability, despite having lower update rate.

The benchmark of the value of stacked vectors, i.e. the number of most recent serialized state vectors passed as input to the PPO algorithm, is shown in figure 8.6. The higher value is extremely counterproductive and leads to a negative result. Scores obtained for the other three lower values are similar, 25 stacked vectors

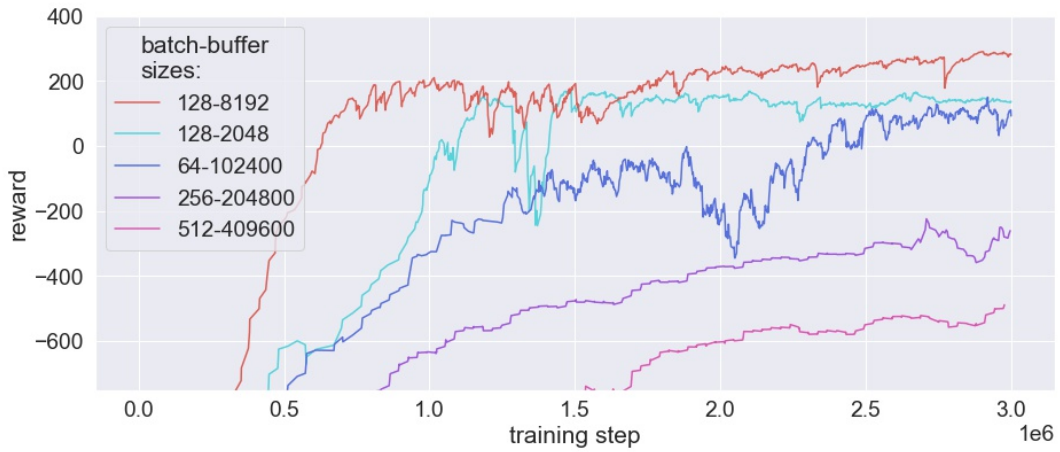


Figure 8.4: Benchmark for different values of the batch size and the buffer size.

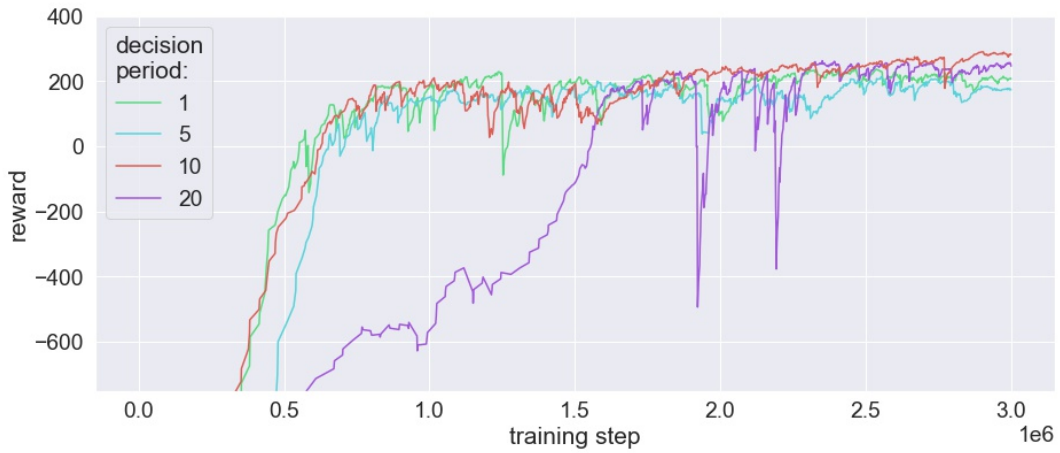


Figure 8.5: Benchmark of different values of the decision period

reach a slightly higher score but with a strongly belated convergence.

The time needed to complete the training increases considerably as this hyperparameter increases, Figure 8.7 shows the relation between relative duration (on the abscissa axis, calculated as the number of hours required to perform one million of

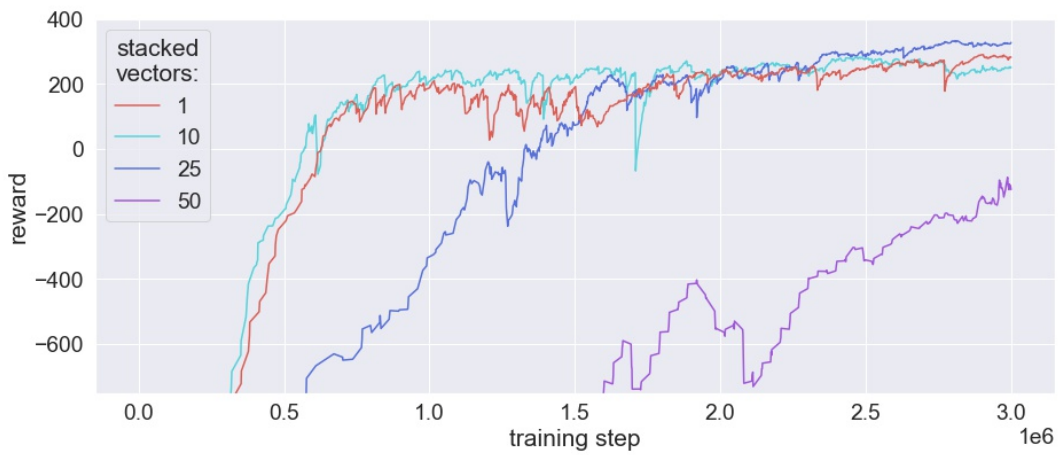


Figure 8.6: Benchmark of different values of the input stacked vectors

decision steps in the training phase), final average reward (on the ordinate axis, mean calculated on the last 25% of rewards obtained) and number of stacked vectors (size and color of the scatters). It should be noted that for the three values below 25, the asymptotic score is almost the same but with a single stacked vector the training time is appreciably reduced, which is why it was chosen for the benchmark base.

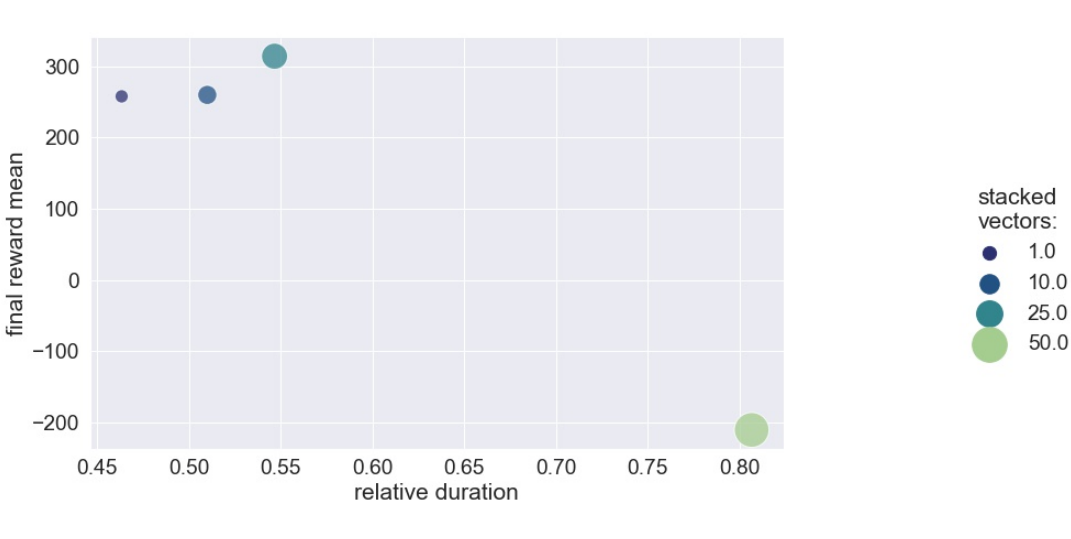


Figure 8.7

8.4 Neural Network Configuration

As exposed in Chapter 3, the neural network used by the PPO algorithm is a MLP, so to define its structure we have to set the number of hidden layers and the number of hidden unit for each of them. The input and output sizes are defined by the application, they are automatically managed by ML-Agents in our case.

Figure 8.8 shows the benchmark for training procedures obtained using one,two or three hidden layers and 64 or 512 hidden units, considered respectively small and large numbers for each layer. It is evident that the performance is lower for networks that use only 64 hidden units, the use of a single large layer (1-512) is totally unsuccessful. The larger network (3-512) demonstrates higher speed in convergence and higher final score. In solving the scenarios in 3-DOF a network with two layers was used; as recommended by the documentation, given the increased complexity for the 6-DOF scenario, we moved to a network of 3 layers and 512 units, the same was assumed as a basis for the benchmarks.

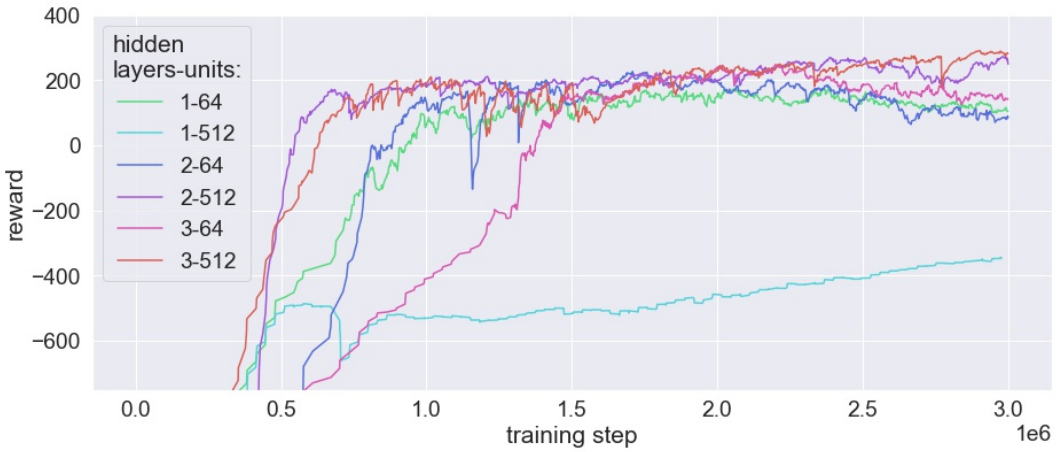


Figure 8.8: Benchmark of the Neural Network configuration, tuning the number of hidden layers and the number of hidden units per layer

As for the duration of the training, with the same number of hidden layers the use of 512 hidden units involves an increase in time required of $\sim 26\%$ compared to the use of only 64; while with the same number of hidden units the use of two layers resulted in an increase of $\sim 6\%$ compared to using one, the use of three layers an increase of $\sim 42\%$ compared to using two (the 1-512 configuration has

been ignored).

8.5 Environment Parallelism

As previously said, the PPO algorithm allows to exploit more than one agent at the same time to experience state-action-rewards and updating a single shared policy (Section 3.2), ML-Agents can work on replicated `prefabs` of the *Unity* environment to take advantage of this. Since the training size is defined as the number of decision steps to be performed, increasing the parallelism decreases the training time. However, it is important to note that with the same decision steps, performances obtained with a parallelized training compared to a single one will not be equivalent, this because the parallelized learning takes place with conditions obtained with the same policy. In other words, having parallel agents increase the number of state-action-rewards experienced simultaneously with the same current policy, but for the behavior to properly evolve enough updates must be made with a yet-evolved behaviour. There is potentially no limit to the degree of parallelism that can be used, but in addition remember that parallel execution time has as bottleneck the strength of the available hardware, which is why (regardless of the final behavior obtained) as the degree of parallelism increases, the training time decrease but not linearly. For this reason it is essential to find the right degree of parallelism in order to obtain the desired behavior in the shortest possible training time, this is of extreme importance especially in the development phase.

Figures 8.9 and 8.10 shows statistics taken from training with incremental degree of parallelism, for trainings of one million and three million decision steps respectively. Figure 8.11 relates the degree of parallelism with the relative duration of the training and with the reward obtained at the end of the training (values obtained and represented similarly to as seen in Figure 8.6), also here separated by total duration of one or three million of decision steps in different columns.

It is evident from the size of the rewards obtained that the training of one million steps is too short, here the asymptotic objective is not reached or is reached too late. With a training of three million the desired reward is achieved, we observe that in the absence of parallelism the training time is much greater ($\sim$ 10 hours to perform one million of training decision steps), with parallelism it is considerably reduced ($\sim$ 1 with 144-parallelism and $\sim$ 2 with 64-parallelism). Between these two, the 64-parallelism has been chosen as base for all benchmarks of this section because it converges much faster and much more stably, 144-parallelism would probably need a longer training. Remember that the parallelism was used only to create the benchmarks, not for the resolution of the four scenarios.

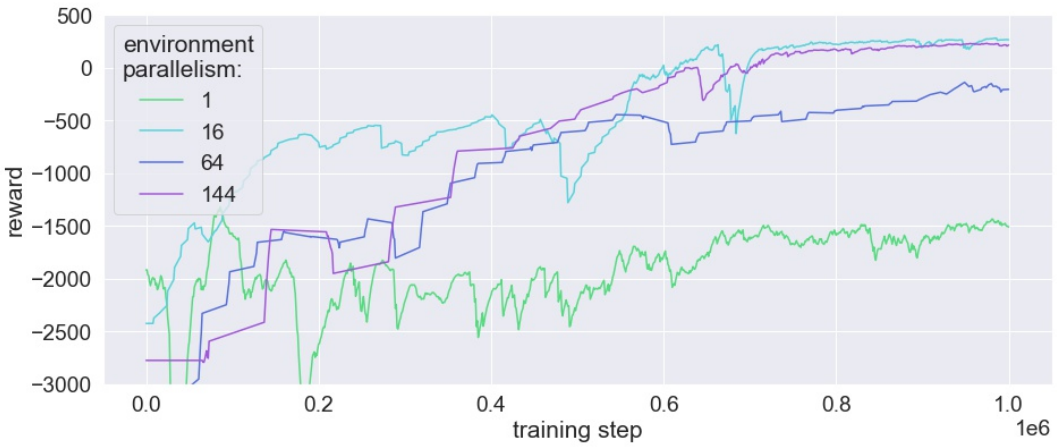


Figure 8.9: Benchmark of the number of parallel environments learning at the same time, within a one million steps training

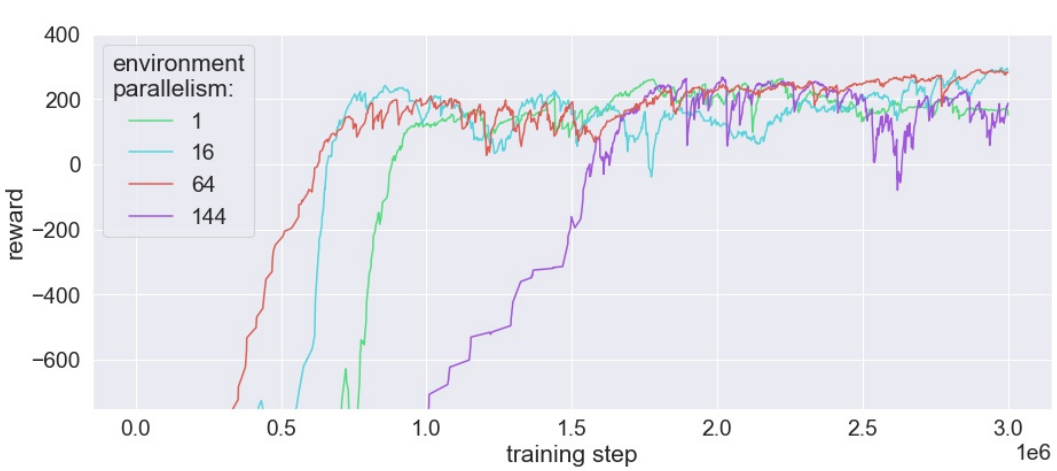


Figure 8.10: Benchmark of the number of parallel environments learning at the same time

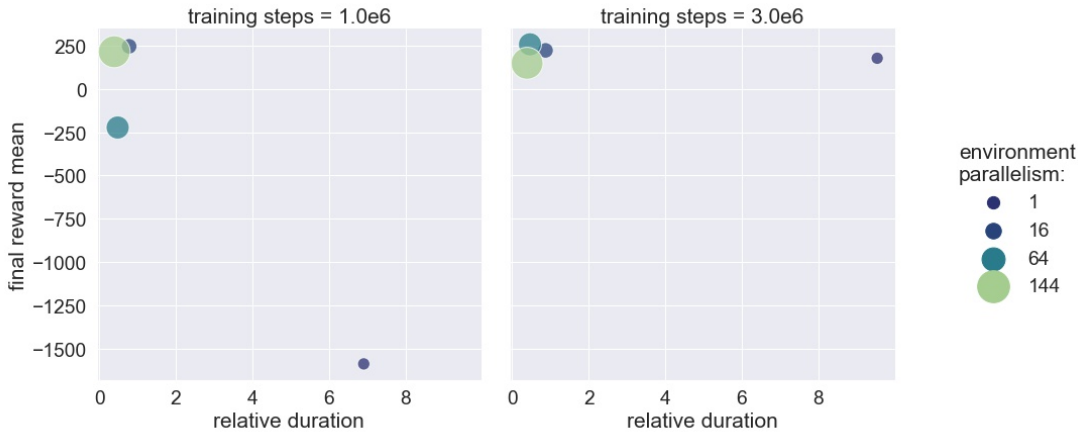


Figure 8.11: Representation of the final reward mean (calculated on the rewards obtained in the last 25% of the training) and of the relative duration of the training (expressed in hours per one million of training steps) as a function of the parallelism of the environment, separated by training length

Chapter 9

Autonomous Lunar Lander: 6-DOF scenario

In this fourth and last implemented scenario, the six degrees of freedom movement is introduced, as the problem addressed in [1], this has greatly increased the complexity of the movement control during the flight phase. Also in this chapter, the purely physical formalization of the problem will be exposed first, then the RL approach will be applied and the related solution found will be analyzed. The failure cases in particular will be analyzed as they are of significant interest, finally considerations on some tests conducted in order to test the robustness of the obtained policy will be carried out.

9.1 Physical Model

In general, the removal of the constraint to the 3-DOF introduces the need of the control of rotation and angular velocity, this results in the implementation of a propulsion system capable of manage both, it replaces the side engines present in previous scenarios, which were strongly not likely. The movement system of most spacecraft consists of a main propulsion system and an RCS. This set of low thrust thrusters is called Reaction Control System, abbreviatedRCS. Compared to the slightly more complex configuration with which this thrusters were positioned in the Apollo 11 lander, the simplest and most intuitive configuration possible was chosen. Being the final implementation and point of arrival of the thesis, the characteristics already present in the previous scenarios are not taken for granted but are described in their entirety, Table A.13 summarizes this model.

The lunar **surface** is represented by a smooth plane with ideally infinite in sizes, the landing **target** is a 10 m radius circle, its center is a point of reference $(0, 0, 0)$

with respect to the environment. There is a constant and uniform gravitational acceleration of $(0, -1.62, 0)\ m/s^2$, there is no atmosphere and not even atmospheric interferences of any kind.

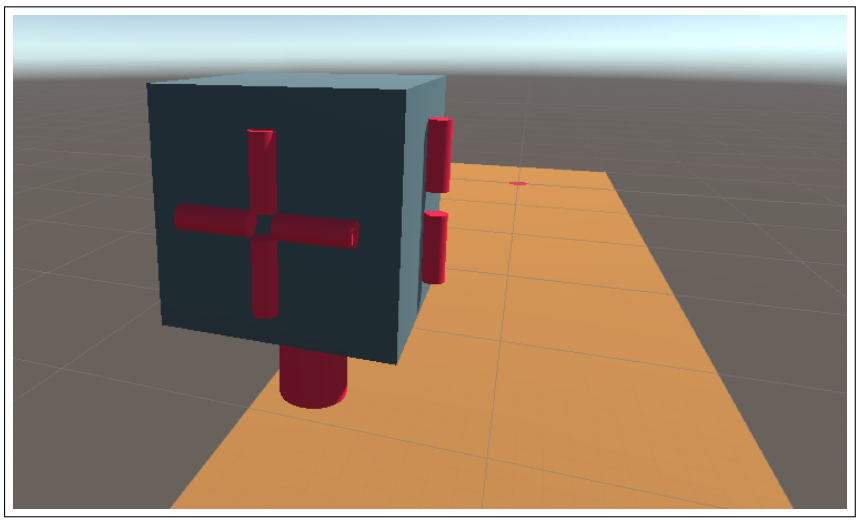


Figure 9.1: Screenshot of the lunar lander model in the 6-DOF scenario, implemented in *Unity*. The large cylinder on the base represents the ignition of the main engine, the 12 cylinders on the side faces (6 visible here) represent the thrusters of the RCS. Remember that during the flight they can only be switched on at the same time in one way for each axis of rotation. The reduced size of the surface identifies the dimension of the flight space, limited to a parallelepiped of dimensions (900, 200, 200) m, with the target in position (350, 0, 100) m relative to its center (visible in the distance).

The lander model is made up of a single **rigid body** of cubic shape, the side of each of its faces measures 5 m in length, the density is homogeneous, the center of mass is located in the center of the cube and its position does not change. The center of mass position is used hereinafter as reference point of coordinates $(0, 0, 0)$ relative to the lander. The dry operating **mass** of the lander is 7000 kg and the total mass of the fuel is 8000 kg, the model has a single typology of fuel. The fuel and consequently its mass does not have a particular location in the lander but is distributed over the entire volume of its rigid body. The lander has one main engine and a set of secondary thrusters that make up the reaction control system, the use of the main engine decreases the quantity of fuel available and consequently its mass with a trend explained afterwards, instead the use of secondary thrusters does not cause fuel consumption.

The ignition of the single **main engine** applies in position $(0, -2.5, 0)$ a single constant force (0, 55000, 0): i.e. 55 kN force with positive vertical direction only with respect to the lander's reference system, applied in the center of its base. The direction of the thrust is fixed vertically to the lander and cannot be angled. The **RCS** is made up secondary thrusters: two pairs of antagonist thrusters for each axis of rotation, so twelve in totale. Each one applies a constant force of 500 N when turned on, forces are imparted in the center of the side faces of the lander, the arm of the forces is therefore 2.5 m. The Table 9.1 resumes the of each of them in the lander's reference system, Figure 9.1 shows the stylized implementation of them. The **loss of mass** caused by the consumption of fuel (for the use of the main engine only) amounts to 300 kg/s, i.e. 6 kg at each ignition command which can be imposed up to 50 times per second. At every control update, the main engine can be turned on if and only if at least 6 kg of fuel are still present. As said, the use of RCS thrusters does not cause fuel consumption and not even loss of mass.

Thruster (Rotation Verse)	**Point of Application**	**Direction**	**Magnitude**
Roll +	(0, 0, -2.5)	(0, 1, 0)	500 N
Roll +	(0, 0, 2.5)	(0, -1, 0)	500 N
Roll -	(0, 0, -2.5)	(0, -1, 0)	500 N
Roll -	(0, 0, 2.5)	(0, 1, 0)	500 N
Yaw +	(-2.5, 0, 0)	(0, 0, 1)	500 N
Yaw +	(2.5, 0, 0)	(0, 0, -1)	500 N
Yaw -	(-2.5, 0, 0)	(0, 0, -1)	500 N
Yaw -	(2.5, 0, 0)	(0, 0, 1)	500 N
Pitch +	(-2.5, 0, 0)	(0, 1, 0)	500 N
Pitch +	(2.5, 0, 0)	(0, -1, 0)	500 N
Pitch -	(-2.5, 0, 0)	(0, -1, 0)	500 N
Pitch -	(2.5, 0, 0)	(0, 1, 0)	500 N

Table 9.1: RCS thrusters configuration.

The control update **frequency** is 50 Hz, at each control update the main engine can be turned off or turned on, at each control update every group of secondary thrusters can be turned off or turned on in positive or else negative direction of rotation (that is, at most two thrusters per group). Each thruster can provide exclusively 100% of its possible thrust (or 0% in case it is off), it is not possible to provide intermediate thrusts, the transitions from switching off to on and vice versa are instantaneous.

At the beginning of each simulation episode the **initial conditions** are determined with uniform distribution randomness within the following ranges:

- values in range [-630, -770] m for the position on the X axis, in range [135, 165] m on the Y axis and in range [-15, 15] m on the Z axis
- values in range [16.47, 20.13] m/s for the velocity on X axis, in range [-4.5, -5.5] m/s on the Y axis and in range [-0.5, 0.5] m/s on Z axis
- values in range $[-\frac{\pi}{16}, \frac{\pi}{16}]$ rad for the rotation separately on the X and Y axes and in range $[\frac{3\pi}{16}, \frac{5\pi}{16}]$ rad on Z axis
- values in range $[-\frac{\pi}{16}, \frac{\pi}{16}]$ rad/s for the angular velocity separately each axis

The mass of fuel available at the beginning of the episode has no random initialization, it is fixed at 8000 kg.

During the **flight phase**, the lander must not touch the lunar surface and not even the target (since landing successfully does not involve touching it), this would result in a crash. Furthermore, the magnitude of the total angular velocity vector must not exceed $\frac{\pi}{4}$ rad/s.

Regarding the **touchdown constraints**, for the lander to successfully land all the conditions set out below must be respected at the same time:

- the sum of distances on X and Z axes between lander's center of mass and target center must be less than 10 m (the lander's center of mass must be above the target)
- the distance on Y axis between lander's center of mass and target center must be less than 3.5 m (in the absence of inclination on the X and Y axes, the base of the lander must be less than 1 m high from the target)
- the sum of velocities on the X and Z axes must be less than 1 m/s in magnitude, velocity on Y axis must be less than 1 m/s in magnitude
- the rotation must be in the range less than $\frac{\pi}{16}$ rad in magnitude on each axis
- the angular velocity must be in the range less than $\frac{\pi}{16}$ rad/s in magnitude on each axis

If and only if all this conditions are met at the same time the landing is considered completed: the control is deactivated and all the thrusters are automatically switched off, at this point gravity will make the lander rest on the target.

9.2 Reinforcement Learning Application

The resolution of this scenario turned out to be far more complex than the previous three, in particular the addition of the rotation and angular velocity components to be controlled both in the flight phase and to achieve the touchdown. This inevitably involves a higher stratification of the reward function, which consequentially is more hardly "understood" by the agent. During the work, the following two training strategies were designed and implemented.

9.2.1 The FSO Training Strategy

Within an episodic scenario with a single successful case, the fact that the agent has difficulty in achieving it even once could be given by this complication: it must learn both what to do and how to do it, and it must learn both at the same time. In other words, the interaction mechanism with the environment and the knowledge of the reward function allows the agent to progressively understand which actions are best in certain situations, however, the agent does not have a global vision of what the ultimate goal is within the episode. Also consider that in the previous scenarios it was noted that the first landings achieved during the training and the relative discovery of the positive reward pushed the agent to carry out a great exploration to find it again. But in the case of a more complex scenario like this, in which the 6-DOF makes even the initial flight phase difficult, the problem is that landing is never reached; the DUT strategy (Subsection 7.2.1) tries to solve the same problem, it proved sufficient to solve the scenario Version 3 but not this last. It would make sense and it would be equally successful if the agent could know what is the objective to be pursued, and only had to find out the way to achieve it.

FSO stands for *Forced Successful Observation*, let's see why. Starting from the idea exposed above, the implementation of this strategy plans to "show" to the agent successful conditions, not as a consequence of the choices made but as if it were imposed from the top. Practically, every F_{FSO} normal training episodes, an *FSO-episode* takes place: instead of in the normal initial conditions of the episode, the agent is placed directly in a successful conditions state; where the hyperparameter F_{FSO} is an integer greater or equal to 1 to be tuned. This episode substantially lasts only one step since the constraints for the touchdown are obviously verified, the related reward is assigned and the episode ends; after that other F normal episodes are carried out, and so on. The idea is that in this way the agent observes the desired state and tastes the associated reward (that is very positive), so that after having known about it, it will be encouraged to search for it again during normal training episodes. The success conditions imposed in the *FSO-episode* are not constant, they are decided randomly for each one, so that the vision of the

states of interest is covered in its entirety; here too, uniform distribution has been chosen for all ranges that define the successful constraints.

Obviously this differs from the normal development of the training episodes, as happens by definition for the proposed training strategies, this is the least invasive strategy as it occurs in separate and very short episodes, technically the agent is not provided with evidence of unreal events. However, it should be noted that the use of stacked vectors greater than 1 could cause problems as if the episode ends in a number of steps lower than the number of sequenced state vectors, a fill of zeros would be performed: an impossible condition in realistic episodes. In this scenario, a stacked vectors value of 1 has always been used, so the question should eventually be investigated further. It should also be noted that the reading of the reward and episode length logs taken during the training is very staggered: indeed very high rewards are periodically assigned from the beginning and an episode each F_{FSO} lasts only one step. An accurate manipulation of the log pipeline could eliminate this noise and bring the visualization back to normal, but this implementation has not been carried out because it is not considered essential. Let's summarize how this strategy works:

ν

9.2.2 Provide focused state-rewards

In scenarios where the positive termination condition is very specific and it is difficult to be explored, the agent may never know it and settle for failed behaviors. It could be useful to forcibly show to the agent the desired states and their related rewards, regardless of the action that the agent takes. This can be done, for example, periodically with a certain cadence alternating with normal episodes. The hope is that in this way the agent will be pushed to find these favorable conditions using the means at his disposal.

This strategy was exploited to achieve the first effective policy in resolving the 6-DOF scenario. However, it is necessary to point out that several trainings carried out in this and the previous 3-DOF benchmark scenario have presented an extremely anomalous trend. In particular in this cases the learning process converges much less quickly than in the normal case, obtaining the opposite of the desired result, but not in all cases. This probably happens because the introduction of extremely short episodes has greater implications on the functioning of the PPO algorithm than expected, this should be investigated further but we report anyway the theorization of the approach as a possible basis for future analysis or implementations.

9.2.3 The RIP Training Strategy

If success is not achieved due to its high complexity, one would intuitively think of initially decreasing the degree of complexity to begin reaching the goal and then gradually re-introducing it back until the desired configuration is restored. The idea behind this strategy is precisely this: to start the training with simplified conditions and as the learning improves, increase the complexity, until success is achieved in the final-normal configuration. The manipulation of complexity during the training will be controlled by two factors, $C_{1,RIP}$ and $C_{2,RIP}$, both are hyperparameters between 0 and 1 by definition. The first has the function of weighing the degree of complexity that is used within an episode, degree 1 represents the total complexity, values close to zero represent highly simplified conditions. This parameter must be set at the beginning of the training, during the training it increases until it reaches the value 1, when it is reached the complexity of the scenario corresponds with the normal complexity; the growth was chosen linear but could be implemented with other trends. The $C_{2,RIP}$ parameter is fixed and represents the point within the training in which $C_{1,RIP}$ reaches the value 1, with $C_{2,RIP} = 1$ we will have that $C_{1,RIP}$ grows during the entire duration of the training until it reaches the normal complexity at its end, with $C_{2,RIP} = 0.5$ we will have that the complexity grows up to half of the training and for the whole second half of the training it will be normal.

In particular in this scenario, the strategy was applied to manipulate the episode phase that comes before the fulfillment of the final conditions, the flight phase in this case study, since the landing itself has already been successfully manipulated by the the approach set out in the Subsection 6.2.5 for example. In any case, this approach can be applied in a general sense to any aspect identifiable as a carrier of complexity. But let's see what is meant by complexity reduction in this specific case. The introduction of the movement in six degrees-of-freedom brings with it an intrinsic complexity that can not be reduced, however we can act on the conditions that we arbitrarily impose to the agent at the beginning of each episode. These have been sized in the model definition to be plausible in reflecting a flight phase, but in all their range now already represent a first considerable obstacle as it is required the lander to fly from distance in a controlled manner up to near the target.

So in this scenario we implement the RIP training strategy by applying the $C_{1,RIP}$ weight factor to the initial conditions of position, velocity, rotation and angular velocity, in order to simplify the control of the lander flight phase. The amount of available fuel is reduced too by the factor because it is assumed that the lander could only be in these conditions having already consumed part of it. Let's assume to use $C_{1,RIP} = 0.1$ and $C_{1,RIP} = 0.5$: at the beginning of the training the episode will take place with initial conditions multiplied by 0.1, that

is, the initial position will be very close to the target, the velocity will be very low, rotations and angular speeds will be almost nil; given the simpler conditions the agent should be more likely able to explore and potentially taste the success. As the training continues, $C_{1,RIP}$ increases linearly and with it the complexity of the scenario approaches the normal one. Exactly in the middle of the training (since $C_{2,RIP} = 0.5$) $C_{1,RIP}$ reaches 1 and from here on it will remain constant, the training will continue normally until the end. Pay attention that for low $C_{1,RIP}$ values there may occur serious problems, for example by obtaining a position of very small value the lander could be positioned already in contact with the ground or the target; furthermore too low values of $C_{2,RIP}$ are considered ineffective.

ξ

9.2.4 Temporary complexity reduction

In scenarios where there is a phase with a numerically quantifiable complexity, a mechanism can be defined whereby this phase is presented with reduced complexity at the beginning of the training procedure. In this way the agent should be able more easily to find the right behavior; as the training continues the complexity must be introduced gradually until it reaches its full amount at the end of the training or, better, at a certain distance from the end. In this way the agent should be able to arrive to face the problem in its full difficulty. This mechanism of increase of complexity can be governed by two parameters, the first determines the weight of the initial complexity reduction and gradually grows, the second indicates the point in the training in which the growth of the complexity reduction ends; the decay can occur linearly or with other trends.

This strategy has proven its worth and it has been used for the first training of the 6-DOF scenario final solution, even if, again, it sometimes leads to great instability. In some training procedure during the attempts, the lander can actually reach some touchdowns in the initial phase of the training thanks to the reduced complexity conditions, but as it increases the performance worsens enormously until the final result is unsatisfactory. This is probably due to the fact that the simple linear increase of the values that constitute the physical conditions of the lander within an episode do not reflect faithfully enough the conditions that will actually be encountered at maximum complexity. In other words, if for instance the lander at some point in the training has a certain $C_{1,RIP} = 0.5$, it would actually have conditions that are simplified, but there are different from those that would be encountered in the realistic scenario at this point. Various values for the two hyperparameters were tested: $C_{1,RIP} = 0.1, 0.15, 0.2$ and of $C_{2,RIP} = 0.5, 0.75, 1$ but an absolute stable trend has not been found; again, the theorization of the

strategy is reported as a possible starting point for future improvements.

We now come to the approach with which the problem was faced, the three values of the rotation and the three values of the angular velocity, expressed for each axis of rotation, were added to the observation vector. As for the actions, the one that controls the main engine remains unchanged, the lateral thrusters have been replaced by three groups of RCS thrusters that to be controlled need 3 discrete actions each (off, on in positive direction, on in negative direction). It was decided to also reduce the size of the flight space, since such large trajectories could not lead to success mainly due to the reduced availability of fuel, the boundaries are increased to (900, 200, 200) m with the target position relative to the center of this parallelepiped: (350, 0, 100) m. Moreover, it was initially immediately clear that the frequency of update of the control had to be necessarily increased to achieve a stable flight phase, attempts made with decision step around 10 were unsuccessful, it was decreased to 3, equivalent to an update of the control with period 0.06 s. In addition, the size of the neural network was increased to three hidden layers of 512 hidden units each.

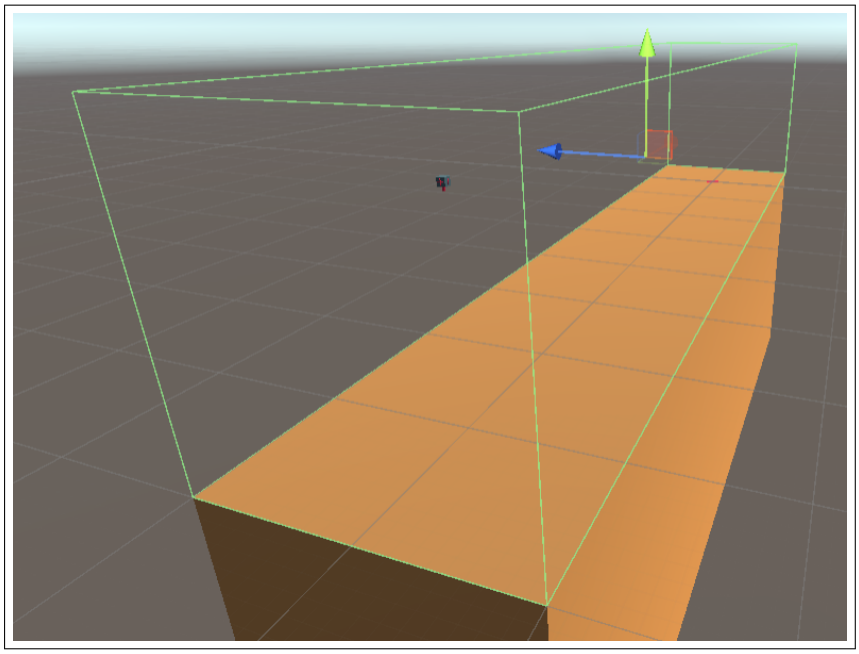


Figure 9.2: Screenshot of the lunar lander model in the 6-DOF scenario implemented in Unity, here it can be seen the configuration of the available flight zone limits. Suspended in the center of the random initialization zone we can glimpse the lander, in the distance we can barely see the target.

9.2.5 First Training Procedure

As mentioned, the introduction of the six degrees of freedom brought a greater number of variables to control and a relative stratification of the rewards assigned to the agent on the basis of them. With "stratification of the reward function" we mean the simultaneous assignment of different reward signals based on different state-variables, but of which the agent sees only the sum so it is less and less trivial for it to understand the cause-effect link. This obstacle has been overcome basically exploiting two mechanisms: the first is the use of successive training procedures performed on the same policy with different specific conditions, the second, more trivial and less elegant, is the considerable increase of the training time. Applying the first methodology, it has been sought primarily to make the policy learn the first productive behavior of flying in a controlled way, being able to maneuver RCS thrusters and the main engine in order to avoid the boundaries of the space and simply maintain a stable flight configuration; once that was done, the part that pushes the lander closer to the target was added to the reward function, in order to reach the touchdown. That was the idea, but using training durations of less than 10 million, the first phase of pure controlled flight was learned very well, but adding the second task, however, was too complex and touchdown was not achieved stably.

At this point, it was decided to try increasing the training time considerably, performing a 40 million step procedure that lasted just under 5 days. This has led to a good result: the agent has learned to fly in a controlled manner and to move in proximity of the target, bumping into it practically every episode, moreover the fuel consumption is far excessive than the availability (also here the possibility to use more than the maximum threshold has been provided). The hyperparameters of the PPO algorithm have been kept constant with respect to the previous configuration of the scenario already valid, so let's analyze the structure of the reward function that led to this first result, first of all the DUT training strategy has been used in a hybrid way, i.e. not for all terminal cases (later we will specify which ones), the FSO strategy with period $F_{FSO} = 2$ has been used and the RIP strategy with hyperparameters $C_{1,RIP} = 0.1$, $C_{2,RIP} = 0.5$. Periodic rewards are:

- **Adaptive approach**: the reward is [0,1] calculated linearly from the inverse of the target distance, as exposed in Section 7.2, but here the denominator for the calculation of $a_{min,t}$ is $\max(3500 - t, 1000)$ since in this scenario t_{max} is considerably high.
- **Adaptive approach fail**: a flat reward of -1 is assigned.
- **Stable angular velocity**: this reward has been added with the objective of teaching the agent to avoid reaching angular velocities that are too high, as failure is practically certain in this type of condition. A limit angular velocity

value ω_{stable} has been defined, below which the lander is considered stable: it is one third of the maximum angular velocity reachable by definition of the physical problem, i.e. $\omega_{stable} = \pm\pi/12$, calculated as the resulting magnitude of the three axes of rotation.

- **Stable angular velocity fail**: this occurs if the angular velocity magnitude is in the range $[\pi/12, \pi/4]$, it is assigned a reward in the range [-1, 0] calculated in linear inverse way.

- **Fly over the target**: this flat reward equal to 0.5 is awarded if the lander is found to be flying above the target, this is to encourage the behavior of maintaining a stable position in this space with the hope that touchdown conditions will be found. Specifically, this flight space is a parallelepiped of size (4, 11, 4) m, the position calculation is done relative to the center of the lander's base.

Regarding the non-periodic rewards:

- **Target hit**: this is the only case where the DUT training strategy is applied, in addition to the -50 flat reward, the position is reset to a height of 7.5 m, the velocity is reduced by a factor of 0.5, the rotation is reset to 0 rad on each axis and finally the angular velocity is reduced by a factor of 0.5. Related considerations will be made later.

- **Ground crash**: a flat reward equal -100 is assigned, the episode is then termined.

- **Maximum distance exceeded**: a flat reward equal -200 is assigned, the episode is then termined. The boundaries of the flight space have been reduced to (900, 200, 200) m as described above.

- **Maximum rotation exceeded**: this condition is not imposed by the physical modeling of the problem, however it is believed that assuming large inclinations represents a point of no return in the piloting of the lander and that successful behaviors do not involve such movements. Therefore, a $\theta_{lim} = \pm\pi/4\ rad$ limit angle calculated on each rotation axis has been chosen, in this reward function the exceeding of this value implies the termination of the episode and the assignment of a negative reward equal to -100.

- **Maximum angular velocity exceeded**: when the limit value is exceeded ($\pm\pi/4\ rad/s$), a negative reward of -100 is assigned, the episode is not terminated according to the DUT strategy but the angular velocity is decreased by a factor of 0.5.

- **Touchdown**: similar to that of the previous scenario, the reward assigned in case of successful touchdown is a positive value $R_t \in [0,230]$ composed of three contributions:
  - the score $s_d \in [0,90]$ calculated on the basis of the distance from the exact center of the target using $\beta = 90$.
  - the score $s_v \in [0,60]$ obtained on the basis of velocities on each axis using $\gamma = 20$.
  - the score $s_f \in [0, 80]$ calculated exactly as in the previous scenario (here with $\delta = 80$).

Consider first of all the massive use of the DUT training strategy only in the case of the target hit, which offers to the agent an easy way to stabilize its conditions in exchange for a relatively low negative reward, not by chance in fact the behavior of the lander is stabilized on arriving to collide on the target. It will be the task of subsequent training to refine this behavior. Starting from previous training attempts, it has been observed that the periodic reward assigned in order to keep low the angular velocity is fundamental, specially during the training of a blanc network, in fact this represents the first real behavior that must be learned. Moreover, it is not trivial to find that the suicidal behavior is not undertaken (despite the many possible cases of negative termination), this could be a success to be attributed to the FSO and RIP strategies, but there is no certainty. Given the length of the training and the potential amount of data generated, no logs or statistics were generated as in the previous scenarios that would allow an in-depth analysis of the evolution of the behavior, this is a potential aspect to be improved in the project pipeline. Note that the reward in case of touchdown and dirty touchdown does not even take into account the conditions of rotation and angular velocity introduced in this scenario, precisely because we knew that this would not be the final configuration; in fact as anticipated, this policy is a good starting point for further learning, even if unsuccessful per se. Before going any further, let's summarize this approach here, which, as we'll see, turned out to be appropriate:

o

9.2.6 Successive incremental trainings

In cases where the training objective is particularly complex, it is common for the reward function to be significantly stratified, in the sense that it opaquely encompasses many signals calculated on the basis of many different variables. This obfuscates the cause-and-effect mechanism observed by the agent, contributing to the failure to achieve success. If it is possible to fragment

the problem into subproblems the training can be split into multiple successive trainings, in which updates with different conditions are applied to the same policy in order to arrive at the desired objective. It is advisable to start with the subtasks that are initially encountered in the scenario and are that fundamental to setting the right behavior, and then move on to more complex aspects. A policy can prove to be an excellent starting point for subsequent training even if it is a failure. Naturally, the techniques and strategies outlined above can be applied to each of the training procedures according to appropriate reasoning.

9.2.7 Second Training Procedure

A second training procedure was therefore applied to the previously obtained policy (some intermediate unsuccessful attempts are omitted). The general RL configuration (generically enclosed in this work into the RL parameters) and the algorithm hyperparameters have been kept unchanged, remember that also for this scenario all the configurations are summarized in specific tables in Appendix A.4. The first training procedure succeeded in teaching the agent to fly in a controlled manner and to approach the target, the objective of the second training is to learn how to achieve the conditions that allow the touchdown. First note that all training strategies were disabled, since they were effective in the phases learned during the first training in order to let the agent know the right direction, understood in the broad sense of behavior. Regarding the reward assigned for the touchdown, it has been resized and contributions given by the conditions of rotation and angular velocity have been added, let's see its composition in full for completeness (for the calculation of the values refer to the previous chapters):

- the score $s_d \in [0,50]$ calculated on the basis of the distance from the exact center of the target using $\beta = 50$.
- the score $s_v \in [0,90]$ obtained on the basis of velocities on each axis using $\gamma = 30$.
- the score $s_\theta \in [0,30]$ obtained on the basis of rotations on each axis with the calculus:
$$s_\theta = \epsilon \left[3 - \left(\frac{|\theta_{fin,X}| + |\theta_{fin,Y}| + |\theta_{fin,Z}|}{\theta_{max}} \right) \right] \tag{9.1}$$
where θ_{fin} are the inclinations on each axis at the time of landing, $\epsilon = 10$ the weight hyperparameter.
- the score $s_\omega \in [0, 30]$ calculated exactly as in the previous:
$$s_\omega = \zeta \left[3 - \left(\frac{|\omega_{fin,X}| + |\omega_{fin,Y}| + |\omega_{fin,Z}|}{\omega_{max}} \right) \right] \tag{9.2}$$

where ω_{fin} are the angular velocities on each axis at the time of landing, $\zeta = 10$ is the weight hyperparameter.

- the score $s_f \in [0, 60]$ calculated as the previous with $\delta = 60$, but this time using a maximum perceived fuel range of [-4000, 4000] kg.

The DUT approach has been removed by the occurrence of the target hit, now in this situation the episode ends and the agent is assigned a reward calculated exactly as the reward of the touchdown but decreased by a 10 factor. This is the actual method by which the behavior obtained in the first training (that collided with impunity on the target benefiting from the DUT conditions), is turned into a behavior in which the agent begins to taste the reward of the touchdown even if in a reduced way. By maximizing and optimizing this reward the agent will arrive at the touchdown itself, as also theorized in Subsection 6.2.5. The magnitude of all the others non-periodic rewards has been increased by a factor of 5.

The periodic positive or negative reward assigned in case of adaptive approach was maintained, but to it has been the following condition: the positive reward for adaptive approach is assigned if and only if the currently available fuel is greater than zero, otherwise the negative reward of adaptive approach fail is assigned regardless. In order to streamline the reward function, the periodic reward for stable angular velocity has been removed, since it is assumed that the lander is now able to fly, as well as the reward for the flight over the target. In some other training attempts carried out starting from the previous policy, all periodic rewards were disabled, thinking that since the agent had already learned to arrive near the target, it would continue to do so without signals during the flight phase and optimize it at will. Probably because of the randomness in the initial phase of training, however, in these simulations the agent completely unlearned the previously achieved behavior; let's point out this non-trivial observation:

π

9.2.8 Need for reward regardless

Using the strategy of successive incremental training procedures, working on a policy with already stabilized behavior, it has been observed that the complete cancellation of the periodic reward involves a decay of performance and a degeneration of the previous behavior, probably due to randomness in the first phase of the new training. Although we would like to leave completely to the agent the constitution of the behavior, it seems necessary to maintain

a periodic reward that keeps the policy stable, it should be always as little invasive as possible.

The total duration for this second training procedure was 20 million of decision steps. It successfully modified the previous policy by making it learn to achieve the desired touchdown with an accuracy of 74.09%, score obtained with more than 15000 test episodes in inference mode. However, this result could be improved because a serious bug was found in the implementation of touchdown constraints and on the related reward function calculus): contrary to what was previously reported, the rotation contraint that must be respected to achieve the landing (angle less than $\pi/16$ in magnitude) was applied also to the vertical rotation axis, moreover the calculation of the score s_θ for the accuracy of the assumed rotation at the final instant was also applied to the Y rotation axis (for the touchdown and for the target hit in the second training as well). In substance to land the lander had to respect a rotation constraint along the vertical axis obviously unreasonable, therefore a third training with the due corrections has been carried out.

9.2.9 Third Training Procedure

A duration of 20 million decision steps was also chosen for this last training. In order to increase the accuracy of the final position reached in the landing, a target radius decrease factor of value $r_{safe} = 0.75$ was applied. In practice during the training phase the size of the target is reduced by 25% so that the lander is accustomed to be more accurate, in doing inference landings "at the edge" will still be considered victorious. Note that this scaling has to be also applied to the range by which the touchdown score based on positional accuracy is calculated. Let's define this technique more generically:

ρ

9.2.10 Harder training, safer result

Within the simulation in which we train our agents it may be convenient to define the constraints that define success in the real world in a more stringent rather than exact way. By doing so the policy will become accustomed to reaching its goal in an optimal way, and in the case that in the real world it should slightly exceed the constraints for which it was trained, it would still be successful. On the other hand, obviously, if the constraints are defined in an exaggeratedly strict way, the training will be too difficult.

In case of target collision the episode is now terminated with a score of -100,

this is because by now the agent has learned to land and the drastic approach is preferred to the incremental one in order to improve accuracy. The other parts of the reward function have been maintained equal in how much considered effective, some changes have been made in the weights of the scores that compose the reward for the touchdown, they are summarized in the Table A.4.3.

To succeed in this last effort of optimization applied to a network already trained, hyperparameters of the PPO algorithm have been properly changed. We can in fact speak in this context of *fine-tuning* of the policy, the general behavior is in fact already learned, we only want to refine it and increase the accuracy; to do this we must avoid that the updates of the network are coarse but conservative, moreover it can be advantageous to reduce the exploration that to a large extent is now superfluous. In particular the learning rate has been decreased from $3.5 \cdot 10^{-4}$ to $2.5 \cdot 10^{-4}$ and epsilon has been halved to 0.1 in order to have more stable and slower updates, beta was decreased by a 10 factor to reduce exploration (0.0001). Let's summarize this remarkable concept:

σ

9.2.11 Policy fine-tuning

Exploiting the strategy of successive incremental training procedures, once the goal is roughly reached we can define a final training in order to better optimize the accuracy and the optimization tasks within the scenario. In this type of procedure it is fundamental not to degrade the behaviour previously reached, it is therefore advised to use low values for hyperparameters such as learning rate and epsilon in order to ensure not destructive updates of the network. It is also recommended to decrease the degree of exploration since the desired overall behavior has already been found, this corresponds to a low beta value in the case of the PPO algorithm.

9.3 Scenario Solution

The Figure 9.4 shows the trend of the three successive trainings that led to the obtaining of the resolving policy of this scenario. It should be noted that the value of perceived reward is considerably discontinuous between one training and another because changes have been made in the rewards assigned, also at the beginning of each training a certain randomness is present so it is normal that the performance is slightly lower, this is clearly visible in the trend of the length of the episodes. In the case of subsequent training, however, this type of graph is not considered very significant since, as already mentioned, the change in reward function reduces the

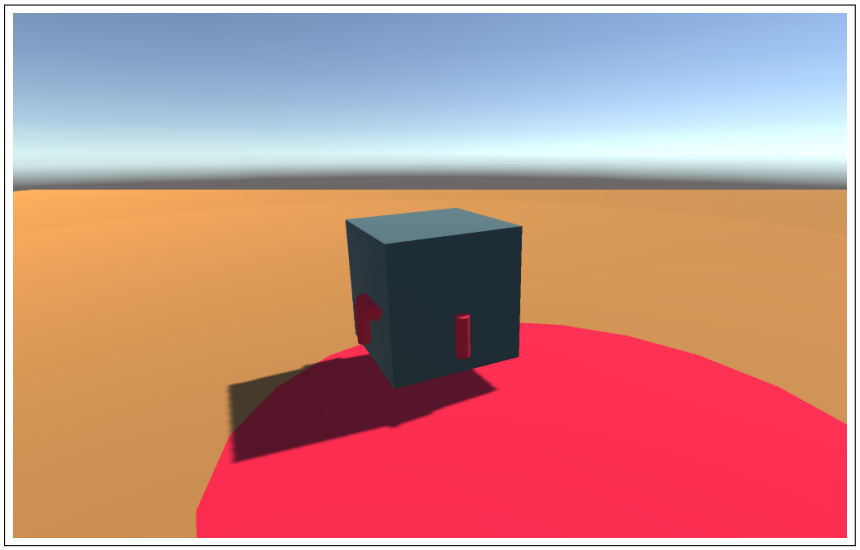


Figure 9.3: Screenshot of the lunar lander model in the 6-DOF scenario while performing the touchdown phase.

validity of the comparison of the rewards obtained.

The final success rate in landing is equal to 89.34%, obtained on a total of 7000 test episodes in inference mode. The third training procedure, fine-tuning, therefore achieved an excellent improvement by increasing the achievement of success by ∼15 percentage points. Figure 9.5 shows several visualization graphs (similar to those seen in Chapter 7) of the trajectory assumed by the lander in different successful episodes, we can observe that despite the randomized starting point of each episode the trajectories are similar, they indeed present a recurrent curvature also in this scenario.

9.4 Failure Analysis

It is extremely useful to take a closer look at only those cases of failure. In the real world, in fact, they are not all the same, but they can occur in the form of conditions slightly different from those desired or in the form of irreparable catastrophes. Indeed, especially in the case of very expensive and delicate applications such as space applications, it's important to understand how badly the found solution works in these cases. In Figure 9.6 we can see the percentage of the types of failure on a sample of 500 episodes run in inference mode, note that some failure cases

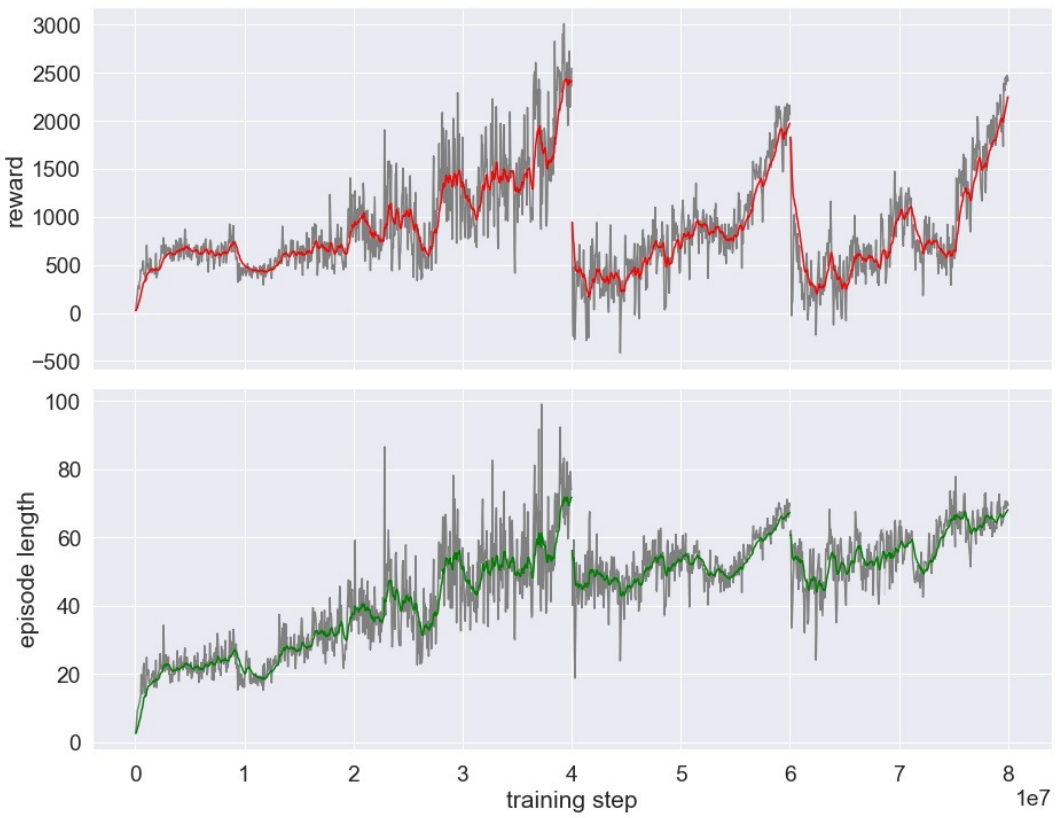


Figure 9.4: 6-DOF scenario training statistics

present during the training phases are not possible here (maximum time, maximum distance or maximum rotation exceeded). Consider that in the physical model the fuel termination was not a specific constraint, in the real world this would correspond to falling to the ground and crashing, but in order to better understand the type of failure it was considered reasonable to distinguish actual crashes with fuel terminations. Failed episodes ending in a target collision are the 20%, this is the least worst condition as it means that at least the predetermined position was reached despite some of the other constraints not being met; ground crashed are almost the 50% of cases, this does not look promising, but let's analyze more deeply the final conditions that occurred. The Figures 9.7, 9.8, 9.9, 9.10 and 9.11 represent the statistics reported in the last physical step logged before the fatal condition occurred, representing respectively distance to target, velocity, rotation on X and Z axes, angular velocity (all calculated as magnitude sum of the axial values) and fuel mass. The colors identifying the type of failure are the same as in Figure 9.6.

Regarding collisions with the target (the blue scatters), the variable that seems to be the cause of failure is the angular velocity, which in any case is not very high. Regarding the episodes in which the lander runs out of fuel (the green scatters), we see that they are all concentrated at a very short distance from the target and with conditions close to those of landing, we can therefore deduce that the fuel runs out practically always while trying to land. These two conditions together occurred the 36.6% of negative cases, it would seem fair to say that in a real-world scenario they represent a non-catastrophic failure, this is a quite good result. Regarding collisions with the ground (the orange scatters), a distinction must be made between two types of failure: one less serious and one much more. The cases that occur at short distance from the target with low velocity, low rotations and low angular velocity are assimilated to target hit cases that occur outside the boundaries of the target; they are not catastrophic if we assume that the pinpoint touchdown can be made in a larger area to some extent. On the other hand, collisions with terrain that occur at great distances from the target and especially with other values far in excess of those expected at landing are to be considered catastrophic. The rough count of the size of these two clusters is about half and half. Finally, with regard to the cases where the angular velocity limit is exceeded, they are always catastrophic as they occur in the flight phase, especially together with a high velocity.

Based on this analysis, we can state that about 60% of the failure cases are not disastrous (about 20% of target hits, 17% of fuel ending and the half of 48% of ground crashes), they occur due to the simultaneous non-match of all touchdown constraints but during the controlled landing phase, not during the flight phase. So, if there were less stringent constraints, the success statistic could be greatly enlarged; note also that it could be a good strategy to define tighter constraints than in the real world, as mentioned in Subsection 9.2.10. As for the remaining 40% of cases (about 16% of angular velocity exceeded and the worst half of ground crashes) they are to be considered extremely unsuccessful. Figure 9.12 shows the success and failure statistics separated into catastrophic and non-catastrophic cases.

9.5 Policy Resilience

Theoretically, one of the strengths of using deep neural networks is their ability to react optimally in unforeseen cases, given the nature of the stochastic rather than deterministic approach. Using the optimally trained policy described in this chapter, tests have been performed in inference mode with physical characteristics different from those used in all training phases; in particular, changes in the mass of the lander, the force applied by the thrusters and the main engine. The graph

in Figure 9.13 shows the accuracy recorded in these scenarios, the percentages of the tests with manipulated physics are obtained on a sample of 1000 episodes each. Note how an increase in mass and a decrease in force of the thrusters corresponds to a loss of performance; the decrease in mass instead corresponds to a greater number of successes, probably because in this case the fuel is in surplus.

Another significant analysis that can be conducted is on the resilience of the observations noise. In fact within the implementation framework observation values are exact, but in the real world they derive from estimates or calculations made on the basis of sensors that are affected by uncertainty and potential measurement errors. In a realistic scenario, it is therefore fundamental that the decision making process takes into account these issues and is resilient to them. The tests conducted consist of providing the agent "dirty" observations for the position and the velocity, per each axis. The true values are randomized with a Gaussian distribution that has as mean the value itself and as standard deviation the value multiplied by a weight factor (calculated exploiting an approximation formula). The fact that the standard deviation is not constant but related to the size of the observation is considered necessary as well as realistic because for distances and velocities of high values there is a greater error, while for smaller values, during the landing phase in this case, there is more precision. Note how the application of noise degrades the performance in a more than linear way; coincidentally with a factor of 0.05 the performance increases, probably caused by the fact that the tests with noise are made on a sample of only 1000 episodes while the vanilla one with 7000, so more accurate.

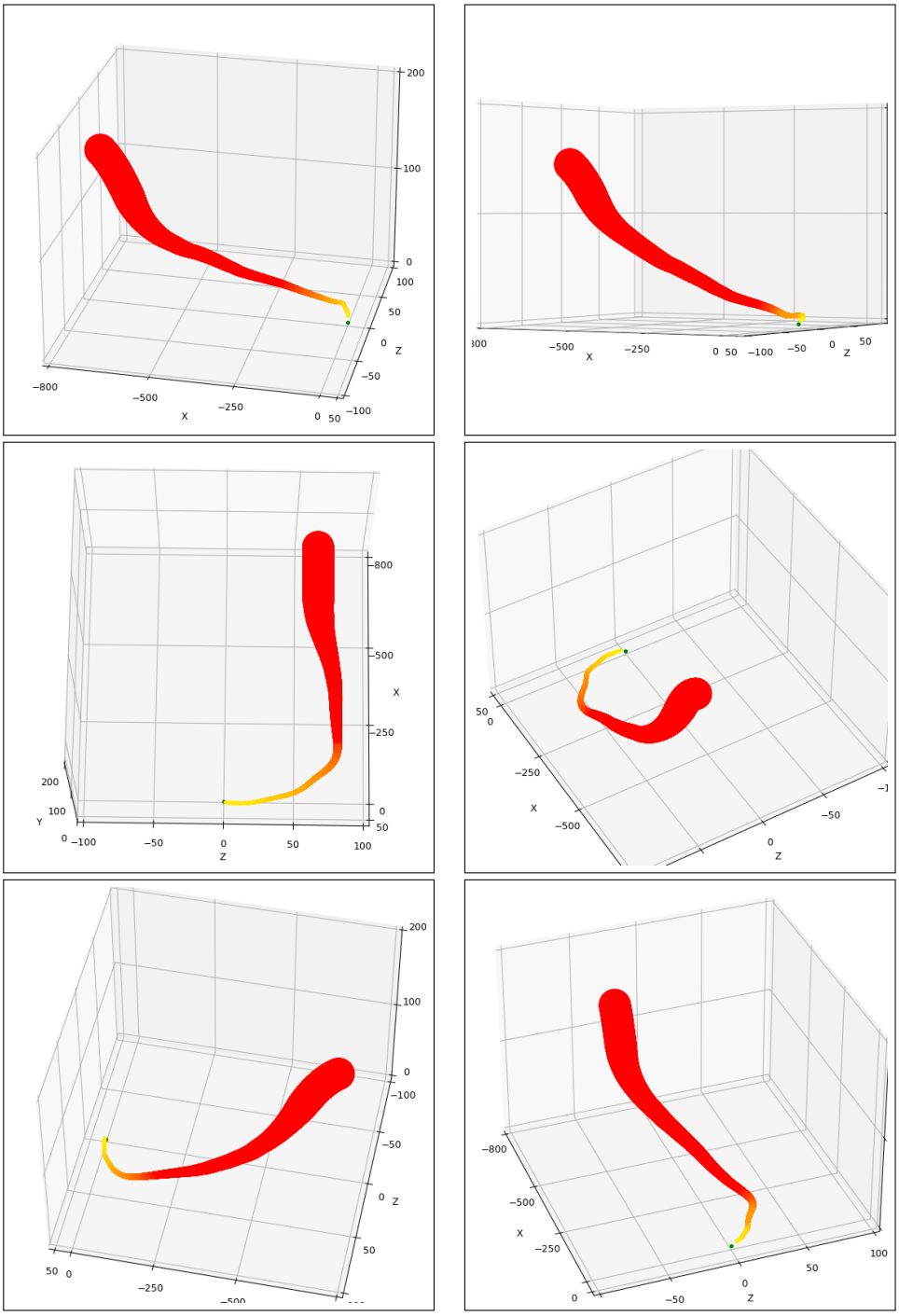


Figure 9.5: Representation of the trajectories recorded in six different episodes leading up to the touchdown, the size and color of the scatters is equivalent to the graphs shown in Figure 7.2

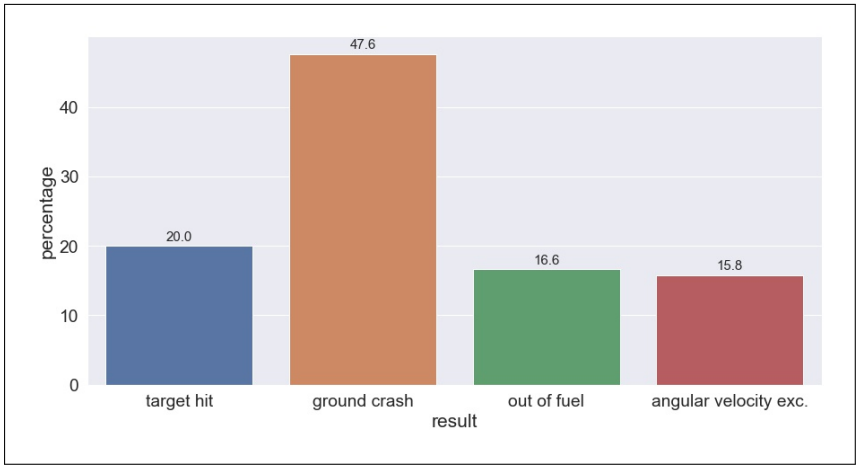


Figure 9.6: Counter representation of the types of failures occurred in the 6-DOF scenario, over 500 episodes in inference mode.

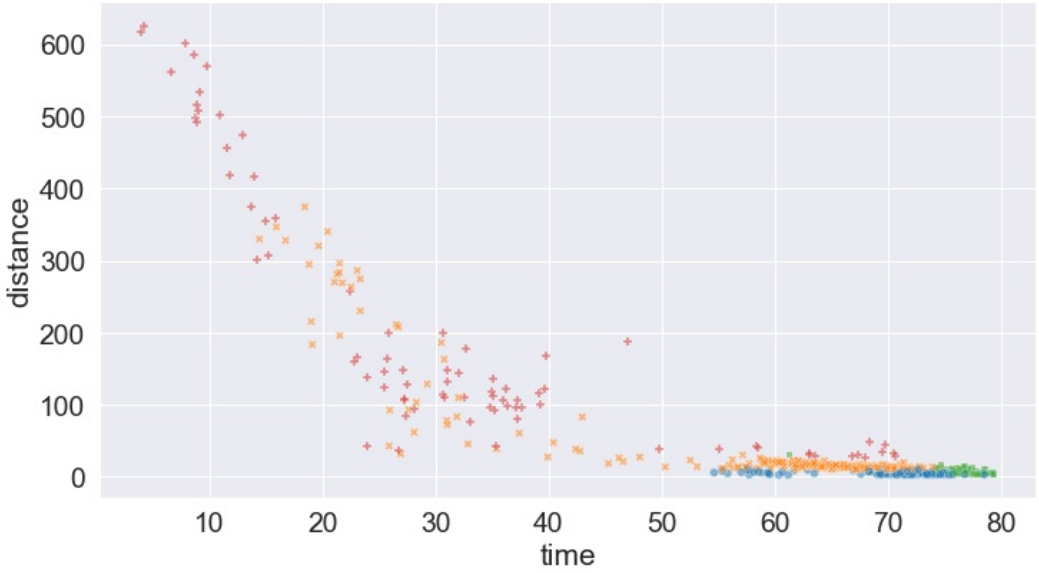


Figure 9.7: Statistics of the distance from target at the instant of failure (color legend equivalent to Figure 9.6).

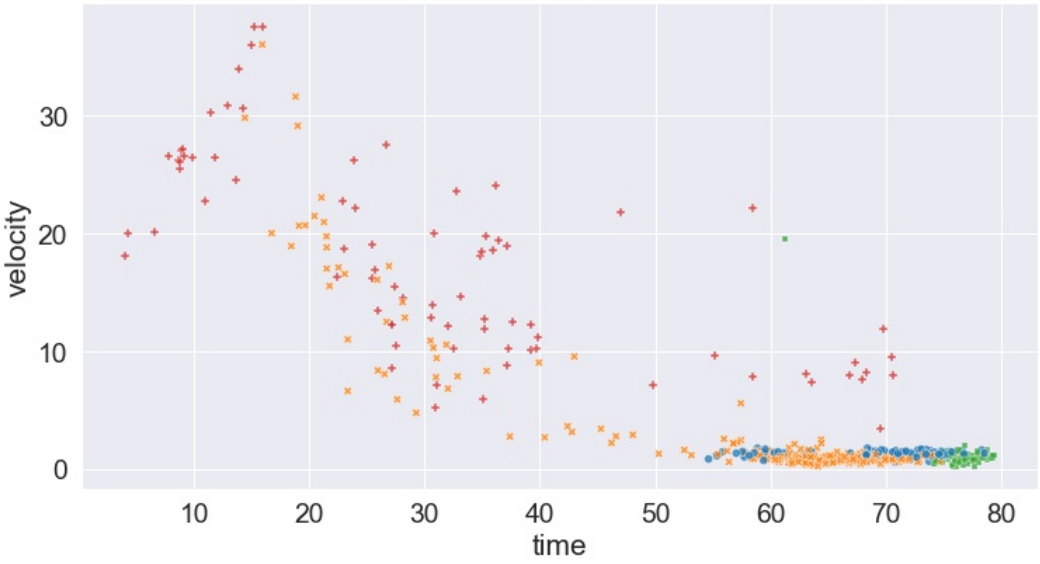


Figure 9.8: Statistics of the velocity magnitude at the instant of failure (color legend equivalent to Figure 9.6).

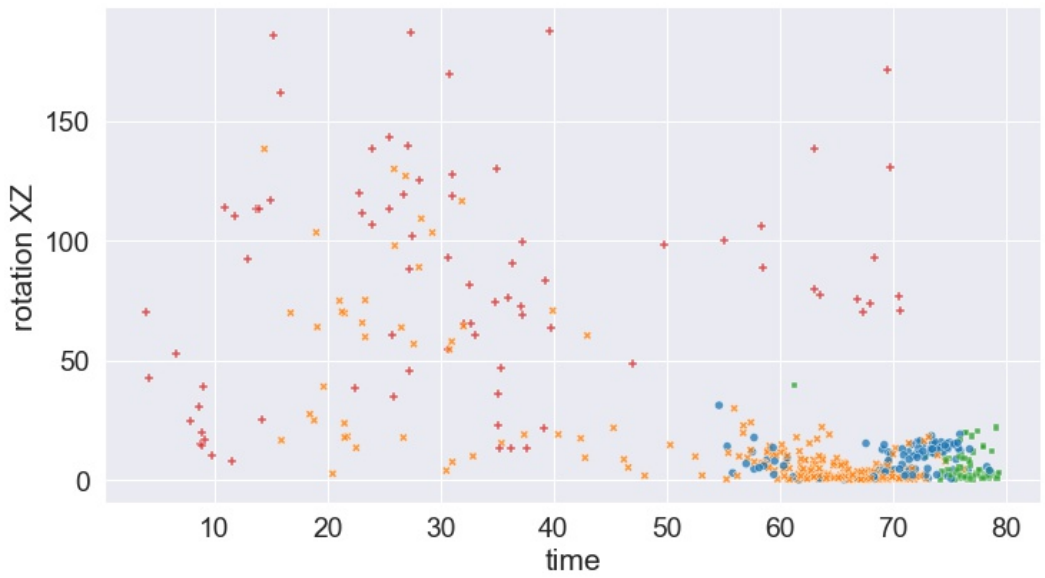


Figure 9.9: Statistics of the rotation magnitude on the X and Z axes, at the instant of failure (color legend equivalent to Figure 9.6).

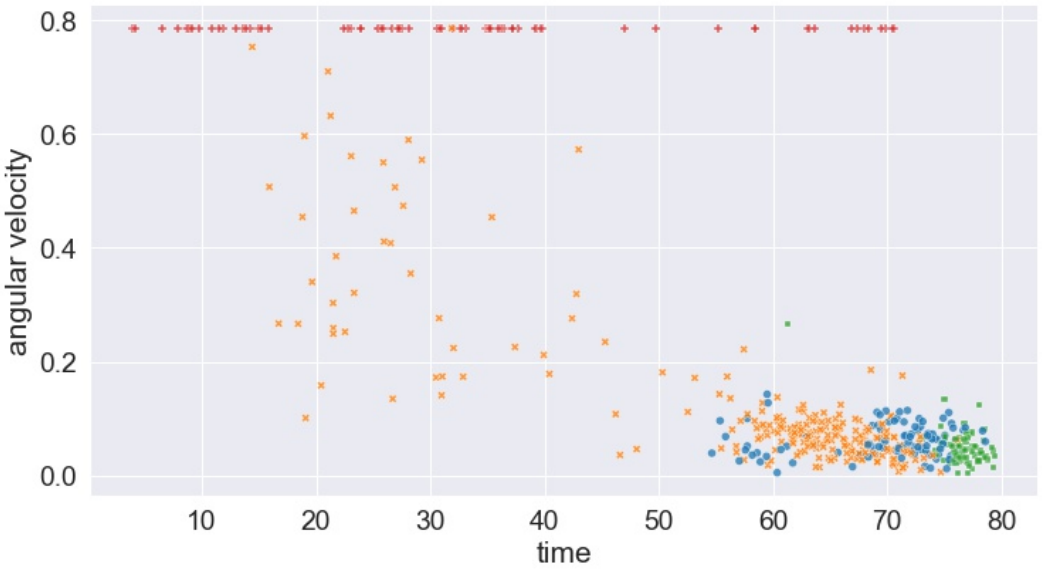


Figure 9.10: Statistics of the angular velocity at the instant of failure (color legend equivalent to Figure 9.6).

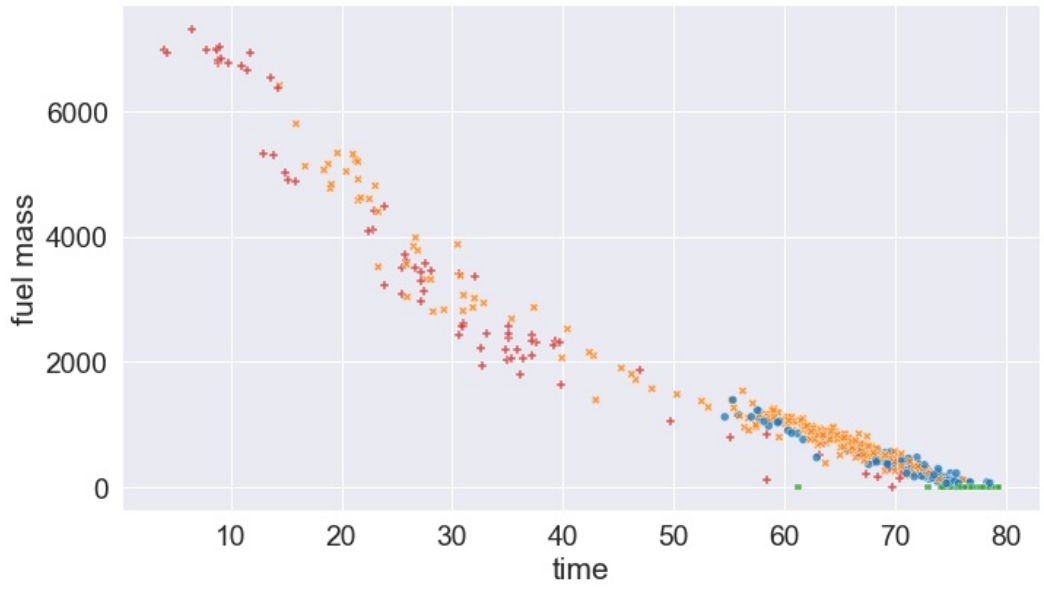


Figure 9.11: Statistics of the fuel mass left at the instant of failure (color legend equivalent to Figure 9.6).

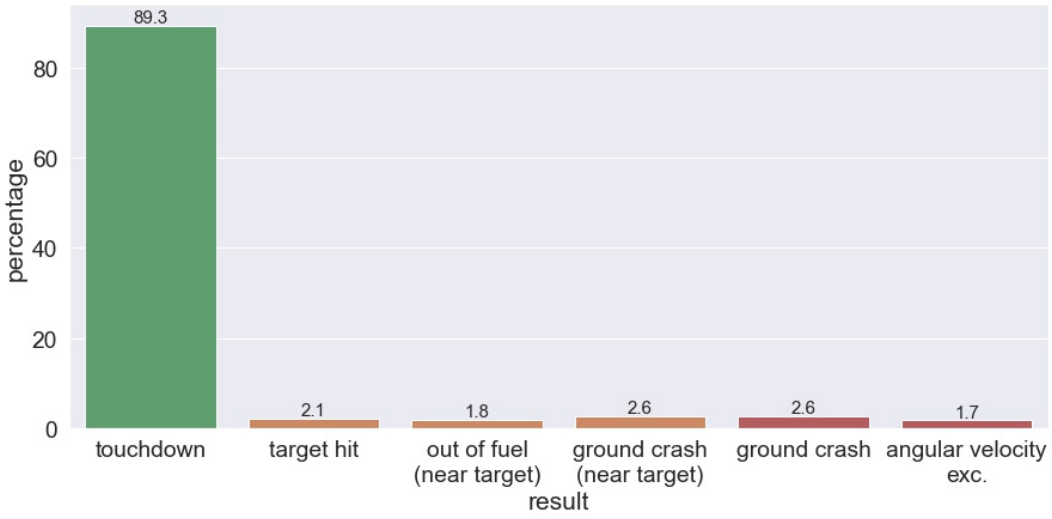


Figure 9.12: Result statistics taking into account separately non-catastrophic failures (in orange) and totally catastrophic cases (in red).

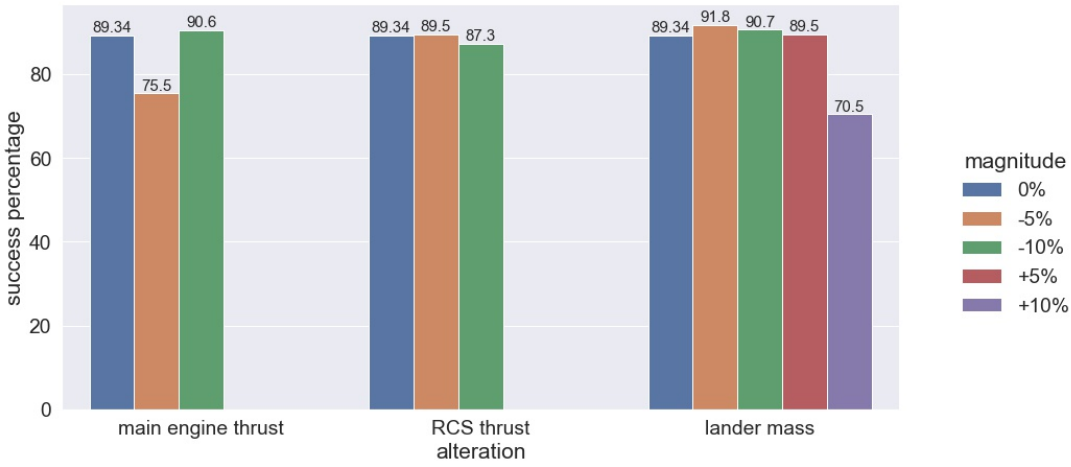


Figure 9.13: Statistics of success percentage for tests performed with alterations of the lander's physical model, with different magnitude of alteration.

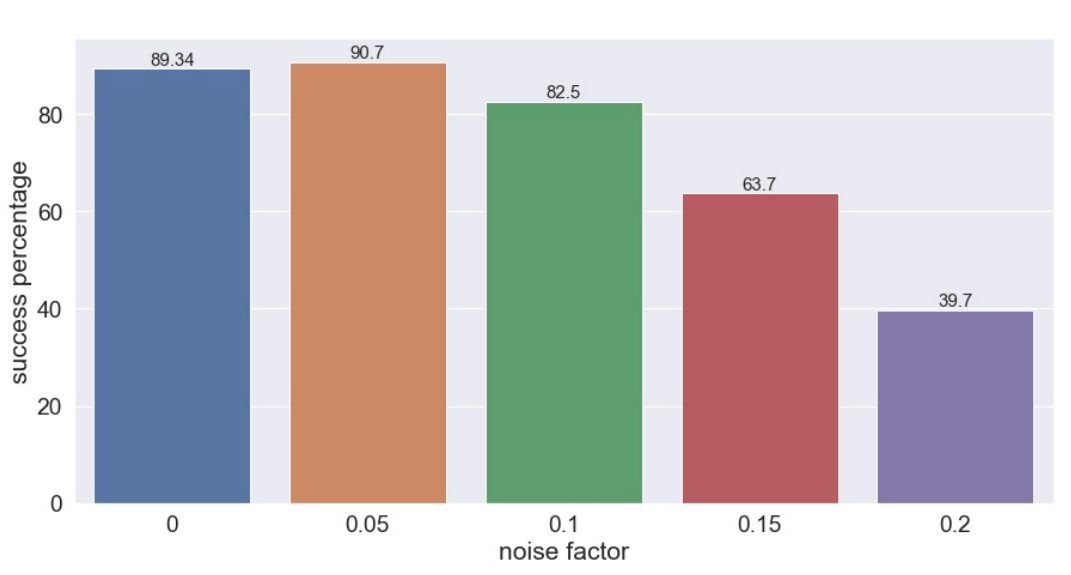


Figure 9.14: Statistics of success percentage for tests performed with Gaussian-noisy-observations, with different magnitude of noise.

Chapter 10

Conclusions

Reached the end of the work carried out, based on the objectives that initially had been set, it's safe to say that the results met expectations. Some final considerations will now be made before defining some possible guidelines for the continuation of the project.

10.1 Final Considerations

First of all, we report that the control scenario was well posed and non-trivial, the fact that it represents a problem of considerable interest for future real-world applications constitutes a fact of considerable importance. The results achieved by artificial intelligences trained in this project are remarkable, but even more so the reasoning that emerged from the experimental approach applied represent the real resource; indeed it is believed that this is the actual result that increases the value of experimental research in this field of innovative applications. In addition, the validity of the application of a state-of-the-art DRL algorithm to a control problem is confirmed, at least within simulations, for now.

Obviously, the approach is far from being applied in the real world, the physical model in fact has gross approximations that can not be ignored. The crudest simplification that was left out throughout the discussion is the modeling of the thrusters, they were modeled as instantaneously on or off forces. A real thruster is a very complicated system, in which one must take into account regimes of action, reaction times, and variations in the application of forces. Discrete output control is inadequate to describe and control them. We also report as physical inaccuracies exploited the cubic shape, the homogeneous density, the fixed center of mass, the single type of fuel, the absence of delays in control, the exactness of all the observation values (only summarily analyzed in the last chapter).

10.1.1 Future Works

All of the physical features listed above are potential implementations that can be pursued, with the goal of arriving at a model that is realistic enough to potentially release the agent into the real world. Moreover, although the pre-established result has been evidently achieved, an accurate comparison with other types approaches would be necessary. Many other DRL algorithms are already available, with peculiar features and potentially very effective, and many are being developed as the research in this area is extremely current. But even with more priority, the classical analytical methods of calculation should be compared, this is the only way to actually quantify the goodness of the results achieved.

Much work remains to be done, but surely Deep Reinforcement Learning research and application will find more and more space in our future.

Appendix A

Summary Tables

A.1 3-DOF scenario Version 1

Environment (parallelepiped)	
Gravitational Acceleration	(0, -1.62, 0) m/s^2
Target (circle)	
Diameter	10 m
Lunar Lander (cube)	
Dimensions	(1, 1, 1) m
Mass	1500 kg
Engine Y Force	15000 N
Engine X+ Force	15000 N
Engine X- Force	15000 N
Engine Z+ Force	15000 N
Engine Z- Force	15000 N
Initial Conditions	
Position X Range	[-95, -105] m
Position Y Range	[20, 30] m
Position Z Range	[-5, 5] m
Velocity	(2, -2, 0) m/s
Final Constraints	
Target hit	

Table A.1: Physical Model

State Vector	Position (X, Y, Z)
	Velocity (X, Y, Z) bounded to 25 per axis
Stacked Vectors	20 vectors
Action Vector	Engines X (3 discrete)
	Engine Y (2 discrete)
	Engines Z (3 discrete)
Decision Period	0.2 s (1 decision step every 10 update steps)
Maximum Step	20 s (1000 update steps)
Training Duration	1 M (decision steps)

Table A.2: RL Parameters

Algorithm	
batch size	128
buffer size	2048
learning rate	0.0003
beta	0.0005
epsilon	0.2
lambda	0.925
epoch number	3
learning rate schedule	linear
Neural Network	
normalize	true
hidden units	256
number layers	2

Table A.3: PPO Hyperparameters

Periodic Reward	
Fixed approach	[0, 1] linearly inverse to target distance [0, D_{max}] m with $a_{min} = 0{,}1$
Fixed approach fail	{-1}
Terminal-Success Reward	
Target Hit (Touchdown)	{1000}
Ground Crash	{0}
Maximum distance exceeded	{0} with boundaries (400, 200, 400) m

Table A.4: Reward Function

A.2 3-DOF scenario Version 2

Environment (parallelepiped)	
Gravitational Acceleration	(0, -1.62, 0) m/s^2
Target (circle)	
Diameter	10 m
Lunar Lander (cube)	
Dimensions	(1, 1, 1) m
Mass	3000 kg
Engine Y Force	15000 N
Engine X+ Force	15000 N
Engine X- Force	15000 N
Engine Z+ Force	15000 N
Engine Z- Force	15000 N
Initial Conditions	
Position X Range	[-630, -770] m
Position Y Range	[135, 165] m
Position Z Range	[-15, 15] m
Velocity X Range	[16.47, 20.13] m/s
Velocity Y Range	[-5.5, -4.5] m/s
Velocity Z Range	[-0.5, 0.5] m/s
Final Constraints	
Target Distance X Z	$< \pm$ 5 m
Target Distance Y	$< \pm$ 1 m
Velocity X Z	$< \pm$ 1 m/s
Velocity Y	$< \pm$ 1 m/s

Table A.5: Physical Model

State Vector	Position (X, Y, Z)
	Velocity (X, Y, Z) bounded to 25 per axis
Stacked Vectors	1 vector
Action Vector	Engines X (3 discrete)
	Engine Y (2 discrete)
	Engines Z (3 discrete)
Decision Period	0.3 s (1 decision step every 15 update steps)
Maximum Step	90 s (4500 update steps)
Training Duration	2 M (decision steps)

Table A.6: RL Parameters

Algorithm	
batch size	128
buffer size	2048
learning rate	0.00035
beta	0.0005
epsilon	0.25
lambda	0.95
epoch number	3
learning rate schedule	linear
Neural Network	
normalize	true
hidden units	256
number layers	2

Table A.7: PPO Hyperparameters

Periodic Reward	
Minimal approach	[-0.5, 0] linearly inverse to target distance [0, D_{max}] m
Minimal approach fail	{-1}
Terminal-Success Reward	
Touchdown	[500, 2000] : {500} + [0, 900] linearly inverse to target horizontal distance [0, 5] m + [0, 200] lineraly inverse to velocity [0, 1] m/s per axis
Target Hit	{-250}
Ground Crash	{-500}
Maximum distance exceeded	{-500} with boundaries (1600, 300, 1600) m

Table A.8: Reward Function

A.3 3-DOF scenario Version 3

Environment (parallelepiped)	
Gravitational Acceleration	(0, -1.62, 0) m/s^2
Target (circle)	
Diameter	10 m
Lunar Lander (cube)	
Dimensions	(2, 2, 2) m
Dry Operating Mass	7000 kg
Initial Fuel Mass	8000 kg
Engine Y Force	25 kN
Engine Y Fuel Consumption	100 kg/s
Engine X+ Force	12.5 kN
Engine X+ Fuel Consumption	50 kg/s
Engine X- Force	12.5 kN
Engine X- Fuel Consumption	50 kg/s
Engine Z+ Force	12.5 kN
Engine Z+ Fuel Consumption	50 kg/s
Engine Z- Force	12.5 kN
Engine Z- Fuel Consumption	50 kg/s
Initial Conditions	
Position X Range	[-630, -770] m
Position Y Range	[135, 165] m
Position Z Range	[-15, 15] m
Velocity X Range	[16.47, 20.13] m/s
Velocity Y Range	[-5.5, -4.5] m/s
Velocity Z Range	[-0.5, 0.5] m/s
Final Constraints	
Target Distance X Z	$< \pm$ 5 m
Target Distance Y	$< \pm$ 1 m
Velocity X Z	$< \pm$ 1 m/s
Velocity Y	$< \pm$ 1 m/s

Table A.9: Physical Model

State Vector	Position (X, Y, Z)
	Velocity (X, Y, Z) bounded to 25 per axis
	Fuel Mass bounded to -8000
Stacked Vectors	1 vector
Action Vector	Engines X (3 discrete)
	Engine Y (2 discrete)
	Engines Z (3 discrete)
Decision Period	0.2 s (1 decision step every 10 update steps)
Maximum Step	100 s (5000 update steps)
Training Duration	10 M (decision steps)

Table A.10: 3-DOF scenario Version 3: RL Parameters

Algorithm	
batch size	128
buffer size	2048
learning rate	0.00035
beta	0.001
epsilon	0.2
lambda	0.95
epoch number	3
learning rate schedule	linear
Neural Network	
normalize	true
hidden units	256
number layers	2

Table A.11: 3-DOF scenario Version 3: PPO Hyperparameters

Training Strategy	
DUT Strategy	Enabled
Periodic Reward	
Adaptive approach	[-0.01, 0] linearly inverse to target distance [0, D_{max}] m with $a_{min,t} = \frac{D_{t-1}}{\max(t_{max}-1000-t,1000)}$
Adaptive approach fail	{-0.1}
Non-periodic Reward	
Target Hit	{-2.5} R_{DUT} reset position: 5 m
Ground Crash	{-5} R_{DUT} reset position: 5 m
Maximum distance exceeded	{-10} with boundaries (1600, 300, 1600) m R_{DUT} reset position: 50 m
Terminal-Success Reward	
Touchdown	[0, 300] : { [0, 90] linearly inverse to target horizontal distance [0, 5] m + [0, 20] lineraly inverse to velocity [0, 1] m/s per axis + [0, 150] linear to fuel mass left [-2000, 2000] kg }

Table A.12: Reward Function

A.4 6-DOF scenario

Environment (parallelepiped)	
Gravitational Acceleration	(0, -1.62, 0) m/s^2
Target (circle)	
Diameter	20 m
Lunar Lander (cube)	
Dimensions	(5, 5, 5) m
Dry Operating Mass	7000 kg
Initial Fuel Mass	8000 kg
Engine Y Force	55 kN
Engine Y Fuel Consumption	300 kg/s
RCS Thruster Force (two each verse)	500 N
RCS Thruster Lever Arm	2.5 m
Initial Conditions	
Position X Range	[-630, -770] m
Position Y Range	[135, 165] m
Position Z Range	[-15, 15] m
Velocity X Range	[16.47, 20.13] m/s
Velocity Y Range	[-5.5, -4.5] m/s
Velocity Z Range	[-0.5, 0.5] m/s
Roration X Range	[$-\pi/16$, $\pi/16$] rad
Roration Y Range	[$-\pi/16$, $\pi/16$] rad
Roration Z Range	[$3\pi/16$, $5\pi/16$] rad
Angular Velocity Range (per axis)	[$-\pi/16$, $\pi/16$] rad/s
Flight Phase Constraints	
Angular Velocity (total)	$< \pm\ \pi/4$ rad/s
Final Constraints	
Target Distance X Z	$< \pm$ 10 m
Target Distance Y	$< \pm$ 1 m
Velocity X Z	$< \pm$ 1 m/s
Velocity Y	$< \pm$ 1 m/s
Rotation (per axis)	$< \pm\ \pi/16$ rad
Angular Velocity (per axis)	$< \pm\ \pi/16$ rad/s

Table A.13: 6-DOF scenario: Physical Model

State Vector	Position (X, Y, Z)
	Velocity (X, Y, Z) bounded to 25 per axis
	Rotation (X, Y, Z)
	Angular Velocity (X, Y, Z)
	Fuel Mass bounded to -8000
Stacked Vectors	1 vector
Action Vector	Main Engine (2 discrete)
	RCS Thrusters Roll (3 discrete)
	RCS Thrusters Yaw (3 discrete)
	RCS Thrusters Pitch (3 discrete)
Decision Period	0.06 s (1 decision step every 3 update steps)
Maximum Step	150 s (7500 update steps)
Training Duration	40 M Training 1 (decision steps)
	20 M Training 2 (decision steps)
	20 M Training 3 (decision steps)

Table A.14: 6-DOF scenario: RL Parameters

Algorithm	
batch size	128
buffer size	2048
learning rate	0.00035 Training 1,2
	0.00025 Training 3
beta	0.001 Training 1,2
	0.0001 Training 3
epsilon	0.2 Training 1,2
	0.1 Training 3
lambda	0.95
epoch number	3
learning rate schedule	linear
Neural Network	
normalize	true
hidden units	512
number layers	3

Table A.15: 6-DOF scenario: PPO Hyperparameters

A.4.1 First Training Procedure

Training Strategy	
DUT Strategy	Hybrid
FSO Strategy	$F_{FSO} = 2$
RIP Strategy	$C_{1,RIP} = 0.1$ $C_{2,RIP} = 0.5$
Periodic Reward	
Adaptive approach	[0,1] linearly inverse to target distance [0, D_{max}] m with $a_{min,t} = \frac{D_{t-1}}{\max(3500-t,1000)}$
Adaptive approach fail	{-1}
Stable angular velocity	{0.01} with $\omega_{stable} = \pm\pi/12$ rad/s
Stable angular velocity fail	[-1, 0] linearly inverse to angular velocity $[\omega_{stable}, \pi/4]$ rad/s
Fly over the target	{0.5} with boundaries (4, 11, 4) m
Non-periodic Reward	
Target Hit	{-50} R_{DUT} reset position: 7.5 m R_{DUT} factor velocity : 0.5 R_{DUT} reset rotation : (0, 0, 0) rad R_{DUT} factor angular velocity : 0.5
Ground Crash	{-100}
Maximum distance exceeded	{-200} with boundaries (900, 200, 200) m
Maximum rotation exceeded	{-100} with $\theta_{lim} = \pm\pi/4$ rad per axis
Maximum angular velocity exceeded	{-100} R_{DUT} factor angular velocity: 0.5
Terminal-Success Reward	
Touchdown	[0, 230] : { [0, 90] linearly inverse to target horizontal distance [0, 10] m + [0, 20] lineraly inverse to velocity [0, 1] m/s per axis + [0, 80] linear to fuel mass left [-2000, 2000] kg }

Table A.16: 6-DOF scenario: Reward Function - Training 1

A.4.2 Second Training Procedure

Periodic Reward	
Adaptive approach with available fuel	[0,1] linearly inverse to target distance [0, D_{max}] m with $a_{min,t} = \frac{D_{t-1}}{\max(3500-t, 1000)}$ having fuel left > 0
Adaptive approach fail	{-1}
Non-periodic Reward	
Target Hit	[0, 26] : { [0, 5] linearly inverse to target horizontal distance [0, 10[m + [0, 3] lineraly inverse to velocity [0, 1] m/s per axis + [0, 1] lineraly inverse to rotation [0, $\pm\pi/16$] rad per axis + [0, 1] lineraly inverse to angular velocity [0, $\pm\pi/16$] rad/s per axis + [0, 6] linear to fuel mass left [-4000, 4000] kg }
Ground Crash	{-500}
Maximum distance exceeded	{-1000} with boundaries (900, 200, 200) m
Maximum rotation exceeded	{-500} with $\theta_{lim} = \pm\pi/4$ rad per axis
Maximum angular velocity exceeded	{-500}
Terminal-Success Reward	
Touchdown	[0, 260] : { [0, 50] linearly inverse to target horizontal distance [0, 10] m + [0, 30] lineraly inverse to velocity [0, 1] m/s per axis + [0, 15] lineraly inverse to rotation [0, $\pm\pi/16$] rad per X Z axis + [0, 10] lineraly inverse to angular velocity [0, $\pm\pi/16$] rad/s per axis + [0, 60] linear to fuel mass left [-4000, 4000] kg }

Table A.17: 6-DOF scenario: Reward Function - Training 2

A.4.3 Third Training Procedure

Training Strategy	
DUT Strategy	Hybrid
Periodic Reward	
Adaptive approach with available fuel	[0,1] linearly inverse to target distance [0, D_{max}] m with $a_{min,t} = \frac{D_{t-1}}{\max(3500-t,1000)}$ having fuel left > 0
Adaptive approach fail	{-1}
Non-periodic Reward	
Target Hit	{-100}
Ground Crash	{-500}
Maximum distance exceeded	{-1000} with boundaries (900, 200, 200) m
Maximum rotation exceeded	{-500} with $\theta_{lim} = \pm\pi/4$ rad per axis
Maximum angular velocity exceeded	{-500}
Terminal-Success Reward	
Touchdown	[0, 600] : { 100 + [0, 60] linearly inverse to target horizontal distance [0, 7.5] m + [0, 30] lineraly inverse to velocity [0, 1] m/s per axis + [0, 30] lineraly inverse to rotation [0, $\pm\pi/16$] rad per X Z axis + [0, 30] lineraly inverse to angular velocity [0, $\pm\pi/16$] rad/s per axis + [0, 200] linear to fuel mass left [-4000, 4000] kg }

Table A.18: 6-DOF scenario: Reward Function - Training 3

Appendix B

External References

At the link https://www.youtube.com/watch?v=g2a_b1DM224 is available the video of a complete episode where you can see the control of the lander in flight and landing, published on the official AIKO YouTube channel. It is realized with more sophisticated graphics and explanatory UI, the policy used is the one trailed in Chapter 9.

Moreover, at the link https://github.com/MatteoStoisa/AutonomousLunarLander-DeepReinforcementLearningForControlApplication it is available a GitHub repository showcasing the project.

Bibliography

[1] Brian Gaudet, Richard Linares, et al. «Deep Reinforcement Learning for Six Degree-of-Freedom Planetary Powered Descent and Landing». In: (2018) (cit. on pp. 2, 70, 100).

[2] Richard S. Sutton and Andrew G. Barto. «Reinforcement Learning: An Introduction». In: (2015) (cit. on pp. 2, 5, 11, 23, 27).

[3] John Schulman, Filip Wolski, et al. «Proximal Policy Optimization Algorithms». In: (2017) (cit. on pp. 3, 21, 26).

[4] Unity-Technologies. *ML-Agents*. 2021. URL: `https://github.com/Unity-Technologies/ml-agents/blob/main/docs/ML-Agents-Overview.md` (cit. on pp. 3, 31, 34, 51).

[5] Unity Technologies. *Unity*. 2021. URL: `https://unity.com/` (cit. on pp. 3, 34).

[6] Nasa. «APOLLO 11 MISSION REPORT». In: (1969) (cit. on pp. 3, 59).

[7] Volodymyr Mnih, Adrià Puigdomènech Badia, and others. «Asynchronous Methods for Deep Reinforcement Learning». In: (2016) (cit. on p. 21).

[8] Volodymyr Mnih, Koray Kavukcuoglu, et al. «Playing Atari with Deep Reinforcement Learning». In: (2013) (cit. on p. 22).

[9] Marc G. Bellemare, Will Dabney, et al. «A Distributional Perspective on Reinforcement Learning». In: (2013) (cit. on p. 22).

[10] Volodymyr Mnih, Adrià Puigdomènech Badia, and others. «Timothy P. Lillicrap and Jonathan J. Hunt and others». In: (2019) (cit. on p. 22).

[11] Tuomas Haarnoja, Aurick Zhou, and others. «Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor». In: (2018) (cit. on p. 22).

[12] Adam White and Marta White. *Reinforcement Learning Specialization*. 2021. URL: `https://www.coursera.org/specializations/reinforcement-learning` (cit. on p. 23).

[13] Vincent François-Lavet, Peter Henderson, et al. «An Introduction to Deep Reinforcement Learning». In: (2018) (cit. on p. 23).

[14] Diederik P. Kingma and Jimmy Lei Ba. «ADAM: A METHOD FOR STOCHAS‡ TIC OPTIMIZATION». In: (2017) (cit. on p. 25).

[15] OpenAI. *OpenAI Five*. 2018. URL: https://openai.com/blog/openai-five/ (cit. on pp. 26, 30).

[16] John Schulman, Philipp Moritz, et al. «HIGH-DIMENSIONAL CONTINUOUS CONTROL USING GENERALIZED ADVANTAGE ESTIMATION». In: (2018) (cit. on p. 27).

[17] Wikipedia. *Bias–variance tradeoff*. 2021. URL: https://en.wikipedia.org/wiki/Bias%5C%E2%5C%80%5C%93variance_tradeoff (cit. on p. 27).

[18] John Schulman, Sergey Levine, et al. «Trust Region Policy Optimization». In: (2017) (cit. on p. 28).

[19] Daniel Ratke. *PPO - a Note on Policy Entropy in Continuous Action Spaces*. 2021. URL: https://blog.xa0.de/post/PPO%20---%20a-Note-on-Policy-Entropy-in-Continuous-Action-Spaces/ (cit. on p. 30).

[20] Unity-Technologies. *Training with Proximal Policy Optimization*. 2018. URL: https://github.com/gzrjzcx/ML-agents/blob/master/docs/Training-PPO.md (cit. on pp. 31, 54).

[21] OpenAI. *OpenAI Gym*. 2021. URL: https://github.com/openai/gym (cit. on pp. 38, 39).

[22] Unity-Technologies. *ML-Agents Toolkit Overview*. 2021. URL: https://github.com/Unity-Technologies/ml-agents (cit. on p. 39).

[23] OpenAI. *Baselines*. 2019. URL: https://github.com/openai/baselines (cit. on p. 39).

[24] The Linux Foundation. *Open Neural Network Exchange*. 2019. URL: https://onnx.ai/ (cit. on p. 39).

[25] Wikipedia. *Exponential smoothing*. 2021. URL: https://en.wikipedia.org/wiki/Exponential_smoothing (cit. on p. 57).

[26] Floyd V. Bennett. «APOLLO EXPERIENCE REPORT - MISSION PLANNING FOR LUNAR MODULE DESCENT AND ASCENT». In: (1972) (cit. on p. 59).

[27] Yunlong Song, Mats Steinweg, et al. «Autonomous Drone Racing with Deep Reinforcement Learning». In: (2021) (cit. on p. 74).