Universitat de Girona
Escola Politècnica Superior

Treball Final de Màster

Estudi: Màster en Ciència de Dades

Títol: Solving classical astrodynamics problems by means
of Machine Learning approaches

Document: Memòria

Alumne: Isaac de Palau i Viñolas

Tutor: Esther Barrabés Vera
Departament: Departament d'Informàtica, Matemàtica Aplicada i Estadística
Àrea: Matemàtica aplicada

Convocatòria (mes/any): Juny 2023

Solving classical astrodynamics problems by means of Machine Learning approaches

Isaac de Palau i Viñolas
Supervisor: Esther Barrabés Vera

Academic year 2022-2023
Version: 1.0

Universitat de Girona
Escola Politècnica Superior

Departament d'Informàtica, Matemàtica Aplicada i Estadística

MSc Data Science

Solving classical astrodynamics problems by means of Machine Learning approaches

Isaac de Palau i Viñolas
Supervisor: Esther Barrabés Vera

Academic year 2022-2023

Isaac de Palau i Viñolas
Supervisor: Esther Barrabés Vera
Solving classical astrodynamics problems by means of Machine Learning approaches
MSc Data Science, Academic year 2022-2023

Universitat de Girona
Escola Politècnica Superior
Departament d'Informàtica, Matemàtica Aplicada i Estadística
Carrer de Maria Aurèlia Capmany i Farnés, 61
17003, Girona

Abstract

The main goal of this master's thesis is to analize and explore the usefulness of Machine Learning and AI tools applied to two classical problems in the field of Dynamical Systems, more concretely in Celestial Mechanics and Astrodynamics, and classic atomic physics: the spacecraft attitude (i.e. orientation) control problem, and the approximation of the Poincaré map in a dynamical system of a Hydrogen molecule under a microwave field.

The attitude control problem has the goal of finding the optimal sequence of movements (usually in the form of *torques* generated with weak impulse thrusters and/or *magnetorquers*) that allows for the correction of the angular velocity and orientation of a space artifact in order to reach a desired final attitude. In this thesis, this problem has been formulated as a reinforcement learning problem where an agent (the satellite) tries to learn an optimal policy (strategy) that allows it to decide which torque should be applied at each moment in order to maximize a reward function inversely proportional to the attitude error. Several experiments have been performed, using different satellite shapes and with/without environmental perturbations. The results show that our controller is able to stabilize the full attiude with a final average error around ± 2 degrees, and a maximum error less than ± 4 degrees, starting from an arbitrary orientation and a maximum angular velocity of ± 4 rad/s in all three body axis.

The second problem of interest of this thesis is the approximation of the Poincaré map of a dynamical system using a neural network. In the field of dynamical systems, Poincaré maps are a key mathematical tool that helps scientists to study the global dynamics in specific regions of the phase space where the problem is defined. In order to approximate the map using neural networks, we generate a large dataset that contains the initial conditions, and their respective images under the Poincaré map. This dataset has been used to train two regressors, each one consisting of three deep neural-networks. The first regressor has been trained to reproduce the Poincaré map forwards in time, while the second one reproduces it backwards. Unfortunately, results in this second section of the thesis have been poor, and both neural networks have difficulties for correctly learning the map. Nevertheless, the results have been discussed and two future research lines have been proposed that could improve the regressor's performance.

Abstract (català)

El propòsit d'aquesta tesis de màster és analitzar l'utilitat de mètodes i models de Machine Learning i Intel·ligència Artificial per resoldre dos problemes clàssics de l'àmbit de l'astrodinàmica i la mecànica celeste: el problema del control d'actitud (i.e. orientació) de vehícles espacials i el càlcul de mapes de poincaré d'un sistema dinàmic.

Referent al problema del control d'actitud, aquest consisteix en determinar la seqüència de moviments (normalment en forma de *torques* generats per motors d'impuls feble i/o *magnetorques*) que permeten corregir la orientació i velocitat angular d'un vehicle espacial i assolir una actitud final desitjada. Aquest problema s'ha plantejat com un problema d'aprenentatge per reforç en que un agent (el satèl·lit) intenta aprendre una política (estratègia) òptima que li permeti decidir quin torque aplicar per tal de maximitzar una funció de recompensa inversament proporcional a l'error de l'actitud. S'han realitzat varis experiments, amb satèl·lits de diverses formes i amb/sense pertorbacions, i els resultats mostren que el nostre controlador és capaç d'estabilitzar l'actitud amb un error mitjà proper als ± 2 graus, sense arribar mai a superar els ± 4 graus, començant des d'una orientació arbitrària i una velocitat angular màxima de ± 4 rad/s en cada un dels tres eixos del cos.

En el segon problema, l'objectiu és aproximar per mitjà d'una xarxa neuronal el mapa de poincaré d'un sistema dinàmic. En el camp de la Mecànica Celeste, els mapes de Poincaré són una eina matemàtica clau que permet als científics estudiar la dinàmica global de regions específiques de l'espai de fase a on es defineix un determinat problema. Per tal d'aproximar el mapa de Poincaré, hem generat un gran dataset que conté les condicions inicials i les seves respectives imatges per l'aplicació de Poincaré d'un determinat problema. Aquest dataset ha estat després utilitzat per entrenar dos regressors, cada un compost de tres xarxes neuronals profundes. El primer regressor ha estat entrenat per tal d'aprendre el mapa de Poincaré integrant el temps cap endavant, mentre que el segon intenta reproduïr-lo integrant el temps cap enrere. Per desgràcia, els resultats obtinguts en aquesta segona part del treball han estat bastant dolents, i els dos regressors tenen dificultats importants per aprendre correctament el mapa de Poincaré del problema considerat. No obstant, en aquest document es mostren i es discuteixen els resultats, i es proposa una línia d'investigació futura en la qual es podria treballar per intentar millorar la precisió obtinguda.

Agraïments / Acknowledgements

Als meus pares i a en *Salem*, pel seu suport durant aquest últim any.

Als meus companys de master, per haver fet més amenes les hores de classe, d'estudi i de pràctiques.

A l'Esther Barrabés, per la seva (infinita!) paciència i per haver-me guiat i assistit de manera molt amable i propera durant l'elaboració d'aquest treball. I també pels *snacks* durant les tutories.

Finalment, a David Juher per haver-me donat a conèixer aquest màster en ciència de dades, i també per haver ajudat indirectament en l'elaboració d'aquest treball.

Contents

Introduction 0

Celestial Mechanics is the branch of physics concerned with studying the motion and dynamics of objects in outer space, such the orbits of planets, the gravitational fields of stars, or the trajectories of comets, to name a few examples. Astrodynamics can be considered an offspring of celestial mechanics and ballistics that deals with the movement of human-made artifacts in space. The motion and orbits of satellites, rockets and spaceships are typical subjects of study that scientists deal with when working in the field of astrodynamics.

Machine learning, on the other hand, is a sub-field of artificial intelligence concerned with building machines capable of learning from data (and/or from experience) by themselves, with limited or no human intervention. In recent years, machine learning has helped improve and advance numerous scientific fields such as medicine (with algorithms capable of detecting disorders in medical images), biology (automatic folding of proteins), chemistry (automatic identification of chemical compounds), robotics (automatic navigation, learning locomotion gaits), etc.

In this thesis we ask ourselves what tools does machine learning offer that can be beneficial to the fields of both astrodynamics and celestial mechanics. Given the time and scope of this master's thesis, we will focus in two particular, classical problems in the field of astrodynamics: the problem of the attitude control of a spacecraft and the problem of approximating the Poincaré map of a dynamical system.

Next, we summarize the contents of each chapter of this Master Thesis.

Chapter 1 - Machine learning and AI

In the first chapter of this thesis, we will review some basic notions about Artificial Intelligence, Machine Learning and Neural networks. We will study the theoretical foundations of Reinforcement Learning (MDPs, optimal policies, Q-values),we will introduce the Proximal-Policy Optimization method and we will review the inner workings of Artificial Neural Networks and the algorithms that allow them to learn autonomously from data.

Chapter 2 - Attitude control

In Chapter 2 of this thesis, we will study the problem of the attitude control of a spacecraft. The goal of this problem is to find the optimal sequence of movements that allow a human-made, spacefaring object to correct its orientation. The movements that allow this correction are usually in the form of *torques* which can be generated with a wide range of actuators such as weak impulse thrusters, *magnetorquers*, thrust wheels and solar sails, to name a few.

We will begin this chapter by reviewing some theoretical notions about rotations, attitude representation and rigid body dynamics. Then, we will present the attitude control problem in terms of a reinforcement learning problem, where an agent (in our case, the control "brain" of a simulated satellite) has to learn by trial and error an optimal strategy to maximize a reward function inversely proportional to its current attitude error. The chapter will end with a discussion of three experiments that have been performed on a satellite running on a custom virtual simulator that show that reinforcement learning approaches are adequate to face the problem of attitude control - although further research would be needed in order to obtain a controller that could be used in an actual mission.

Chapter 3 - Poincaré map approximation

In Chapter 3 of this thesis, we will study the usefulness of neural networks when trying to aproximate the poincaré map of an arbitrary dynamical system.

In the field Dynamical Systems (in particular, Celestial Mechanics or Astrodynamics, poincaré maps are a key mathematical tool that helps to study the dynamics around a specific region of the phase space, in particular, around stable periodic orbits. For example, when considering an orbital system generated by two large mass objects (such as the system created by our planet and the Moon), there are five positions in which a small object placed there would remain stationary with respect to the large objects. These positions are known as *Lagrange points* or *Libration points*, and are usually surrounded by periodic orbits that are useful for those missions that require the deployment of an stationary artifact (such as an antenna or a radiotelescope). To have a good knowledge of these regions is crucial for the design of successful space missions.

In order to aproximate the map using neural networks, we generated a large dataset that contains the initial conditions of a dynamical system [1], and their respective intersections with a predefined poincaré section, integrated using the Runge-Kutta-Fehlberg 78 method. This generated dataset has been employed to train two regressors, each one consisting of three deep neural-networks of several layers. The first regressor has been trained in such a way that, when given an initial condition, it is able to output the intersection point with the poincaré section. The second regressor does the inverse; when given a final intersection point, it predicts the initial condition. Unfortunately, results in this second section of the thesis have been poor, and both neural networks have difficulties when learning the map and giving accurate predictions. Nevertheless, the results have been discussed and two future research lines have been suggested that could improve the performance.

GitHub code repository

In addition to the contents of this document, an external GitHub repository has been created to store all the code and programs developed during this thesis. This repository can be accessed and reviewed in the following link:

`github.com/RecursiveMagus/AstroIA_MasterThesis`

[1] The original idea was to work with the dynamical system of the restricted three body problem, but we decided to begin our first tests with the CP problem. The poor results obtained with the prediction have prevented us to progress further.

The software for this thesis has been developed in MATLAB r2022b and r2023a for Linux, running on Ubuntu 20.04 . The specifications of the hardware used to train the machine learning models are as follows:

- NVIDIA GeForce RTX 3080.
- Ryzen 9 3900X Processor.
- 32GB RAM.

1 AI, Machine learning and Reinforcement learning

In this first chapter we will review some theoretical concepts about AI, Machine Learning and especially Reinforcement Learning, that we will use to solve the two problems of interest of this thesis.

Although the terms are often mistakenly used interchangeably, Artificial Intelligence (AI) and Machine Learning (ML) are not the same concept (see Figure 1.1). Even though Artificial Intelligence lacks a consensual, precise definition (see [13], Section 1.1), it can be considered a scientific discipline concerned in building machines/algorithms capable of procesing information, reasoning and solving problems in a similar way that humans and/or other living beings do, often by mimicking problem-solving strategies employed by living organisms. Machine Learning is a sub-field of AI dedicated to build algorithms capable of learning (in the broadest of senses) from data or experience. Reinforcement Learning, in particular, is a paradigm of Machine Learning that is concerned about how to obtain intelligent agents capable of learning in an autonomous way, without supervision or labeled data, by means of rewards and punishments obtained through experience.

1.1 Reinforcement Learning

Suppose we need to build an agent capable of playing (and winning) in the game of checkers. One way to build such an agent would be not to use a Machine Learning approach, but to employ a deterministic search algorithm such as minimax, capable of finding the optimal set of movements. Although this would be perfectly suited for a simple game such as checkers or even chess, it would be totally impractical for some of the games that current AIs are capable to play (see [8]). Another option would be to take a supervised learning approach; this would require that a dataset is build with thousands of posible board configurations, each one labeled with the most optimal move. Again, this could be feasible for a game of checkers, but impractical when playing more complex games. The third option would be to teach the agent by giving it a reward only when certain events happen during the game; for example, a negative reward when it loses a token, a positive reward when the rival loses a token, and a huge positive reward when the agent wins a game.

However, the best actions do not always yield an immediate reward. In chess or checkers, it is not uncommon to sacrifice a token if this places the oponent in an unadvantageous position or opens some window for attacking; the short term loss of having lost a token can be mitigated for the long-term reward winning the game. Therefore, the problem that Reinforcement Learning tries to solve is: is there any way to build the agent in such a way that it is able to maximize the sum of these rewards *in the long term*?

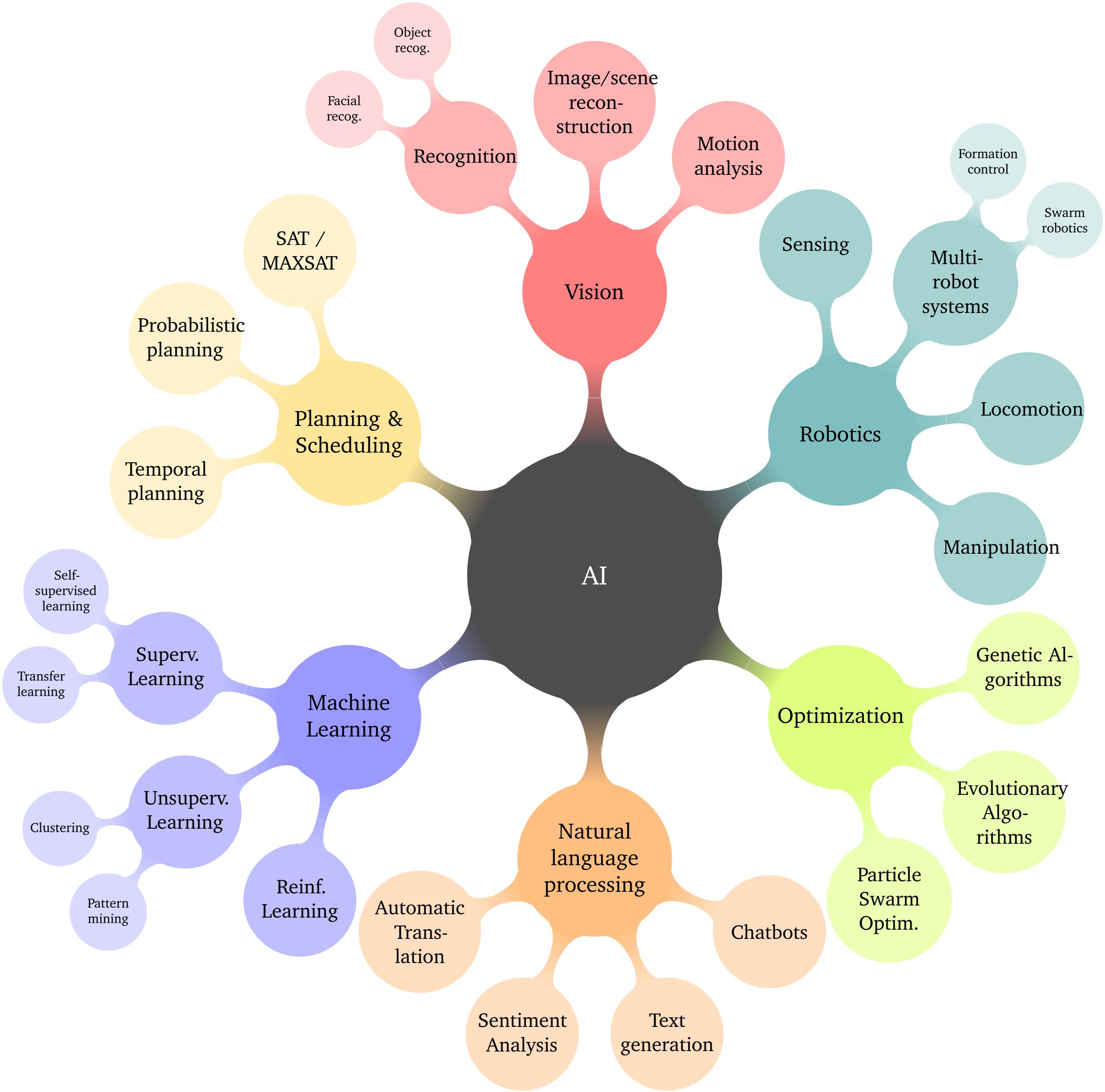


Fig. 1.1: AI and (a few of) its subfields and subdisciplines. Although it is not shown in the figure, these sub-fields often intersect and combine.

1.1.1 Intelligent Agents

The core concept behind most of the perspectives and approaches in the field of AI is the concept of the **rational agent**, or simply **agent**. The precise definition of an agent will vary depending on the context and the particular problem to solve, but generally speaking an agent is an entity (a robot, a person, an algorithm, a *softbot*, etc) capable of sensing an enviroment (real or simulated, abstract or otherwise) and acting upon it using actuators in order to achieve a goal (see Figure 1.2).

For example, a physical robot is an agent that exists in a real environment. It uses its sensors (cameras, IMUs, antennas, etc) to sense its environment, and acts upon it by using its wheels, joints or communication devices. Its goal could be to explore a region, follow a path or deliver a package. In contrast, the satellite we will train during this thesis exists in a virtual environment; its sensors and actuators are just software functions that read or change some variables of the simulation. Still, it lives inside an environment that is a reflection of the real world. Other agents, however, can live in purely abstract environments that do not represent any aspect of the real world.

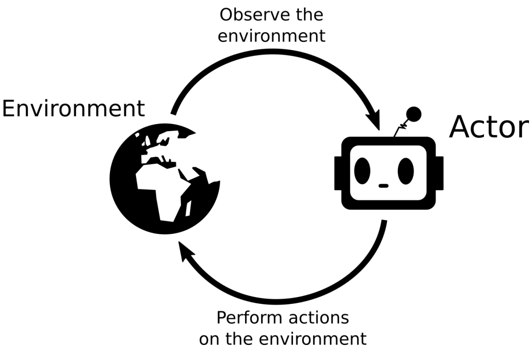


Fig. 1.2: The actor perceives the environment and performs actions on it in order to achieve some goal.

1.1.2 Markov Decision Process

Most problems in Reinforcement Learning are modelled as Markov Decision Process. Informally, a Markov Decision Process can be thought as a random process that can adopt different discrete states, and where we have some degree of control over the probabilities that the current state changes into another.

In more formal terms, suppose we have an agent acting on an environment that can be modelled by a discrete set of states $\mathcal{S}$, and the actions available to the agent can be codified with the elements of a non-empty discrete set $\mathcal{A}$. We will denote the state of the agent at time t as X_t, and the action performed at time t as A_t. Consider also a probability space $(\Omega, \mathcal{F}, \mathbb{P})$, where Ω is the sample space, $\mathcal{F}$ is the set of possible events, and $\mathbb{P}$ is the probability function.

Definition 1.1.1 (Markov Decision Process)**.** A **Markov Decision Process**, or MDP for short, is a 4-tuple $(\mathcal{S}, \mathcal{A}, T, R)$ where:

- $\mathcal{S}$ is a discrete, non empty set called the **set of states**. The situation of the agent in the environment at any given time $t \geq 0$ will be codified by a random process $\{X_t\}_{t\geq 0}$ that takes its values in $\mathcal{S}$. Some states can be considered *terminal*, and finish the process.
- $\mathcal{A}$ is a discrete, non-empty set called the **set of actions**. It defines the actions that the agent is able to perform in a given moment.

- $T(s, a, s') := \mathbb{P}(X_t = s'|X_t = s, A_t = a)$ is a stochastic function called the **transition function**. It defines the probability to change the state of the system to s' when performing the action a on state s.
- $R(s, a) : \mathcal{S} \times \mathcal{A} \longrightarrow \mathbb{R}$ is the **reward function**, which determines the expected reward received when performing the action a on state s.

The transition function must follow the *markovian property*: the next state of the agent must *only* depend on the current state and the chosen action. Formally, $T(s_t, a_t, s_{t+1}) = \mathbb{P}(X_{t+1} = s_{t+1}|A_t = a_t, X_t = s_t) = \mathbb{P}(X_{t+1} = s_{t+1}|A_t = a_t, X_t = s_t, X_{t-1} = s_{t-1}, ..., X_0 = s_0)$.

Definition 1.1.2 (Trajectory)**.** The historic sequence of previous states, actions that led to them and received rewards is called a **trajectory** and is often written as $s_0, a_0, s_1, r_1, a_1, s_2, r_2, a_2, ..., a_{n-1}, s_n, r_n$, where s_t denote the states, a_t denote the actions and r_t denote the rewards obtained when transitioning from state s_{t-1} to state s_t.

Note that in an MDP, due to the markovian property, each action a_t does not depend on the previous trajectory up to that time t.

The goal of an MDP is finding a way to maximize the cumulative sum of expected rewards in the long run.

Definition 1.1.3 (Policy and optimal policy)**.** A **policy** is a function $\pi : \mathcal{S} \longrightarrow \mathcal{A}$ that assigns to each state an action to perform; by choosing a sensible function, this policy can be followed by the agent in order to pick the action to perform at any given time.

Ideally, an **optimal policy**, denoted as $\pi*$, is a policy that maximizes the cumulative sum of expected rewards

$$\sum_{t\geq 0} R(s_t, a_t) \ .$$

Unfortunately, this sum diverges in most practical cases, thus forcing us to find an alternative optimal criteria. A possible alternative is to discount future rewards by a multiplicative factor $\beta \in (0, 1)$. Thus, we want to find a policy that maximizes

$$\sum_{t\geq 0} \beta^t R(s_t, a_t)$$

for each state.

1.1.3 Q-values

The quality of a policy π can be represented through a *value function* that measures the expected sum of rewards if the agent follows the policy. Value functions can be calculated for each state (**state value functions**), or for each action-state pair (**state-action value functions**, sometimes called Q-value functions). We will say that state value functions measure the *utility* or *value* of a state, while state-action value functions measure the utility or value of an action-state pair.

The *state-value function* for a state s under a policy π, denoted as $V^{\pi}(s)$ is the expected return when the agent starts at state s and follows π afterwards:

$$V^{\pi}(s) = \mathbb{E}\left(\sum_{t\geq 0} \beta^{t} R(s_t, \pi(s_t)) \mid s_0 = s\right),$$

where $\mathbb{E}$ is the expected value associated to the probability space $(\Omega, \mathcal{F}, \mathbb{P})$. In contrast, the **state-action value function** for a state s and an action a under a policy π is defined as the expected return of performing action a in state s and following policy π afterwards:

$$Q^{\pi}(s,a) = \mathbb{E}\left(R(s,a) + \beta \sum_{t\geq 1} R(s_t, \pi(s_t)) \mid s_0 = s, a_0 = a\right).$$

Theorem 1.1.1 (Bellman equation)**.** The action-value function Q can be re-written as

$$Q^{\pi}(s,a) = R(s,a) + \beta \sum_{s' \in \mathcal{S}} T(s,a,s') Q^{\pi}(s', \pi(s')).$$

We call this expression the **Bellman Equation**.

Proof. See [14], Section 3.6 . □

The Bellman Equation is a crucial result upon which most classical reinforcement learning methods are built upon. Its importance lies in the fact that it allows us to express the Q-value of a state-action pair in terms of the Q-values of other pairs.

Reinforcement Learning can be considered a particular case of an MDP where the agent does not have *a priori* direct knowledge of the transition function nor the reward function. In order to find an optimal policy that maximizes the expected reward, the agent will have to find a way to estimate this two functions or, at the very least, to *estimate* the utility of a state.

Suppose that for each state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$, the agent has learnt the utilities $Q(s,a)$ (known as the Q-values). In this case, in order to maximize its expected reward the agent only has to calculate these Q-values for each new state and possible action, and then pick the action that yields the maximum value. However, given that the agent does not have knowledge of T nor R, the only way to determine the Q-values $Q(s,a)$ for each state-action pair (s,a) is by learning from experience. One way to do this is by using the **Q-value iteration method**, which turns the Bellman equation into an iterative method:

Algorithm 1 | Q-value iteration

```
Require: A table $Q$ with an entry for each possible state-action pair $(s, a)$ (we will
    denote each entry as $Q(s, a)$). Max number of episodes $E > 0$, a max. number
    of training steps $M > 0$, relaxation parameter $\alpha$, exploration probability $\epsilon$ and
    discount factor $\beta$ with $\alpha, \epsilon, \beta \in (0, 1)$.

    Begin:
    Initialize each element of $Q$ to 0.
    for i $= 1, 2, ..., E$ do
        Set initial state as $s$.
        for i $= 1, 2, ..., M$ do
            Pick random action $a$ with probability $\epsilon$, otherwise pick action $a$ that
            maximizes $Q(s, a)$.
            Perform $a$, observe new state $s'$ and reward $r$.
            Update:

                $Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(r + \beta max_{a'} Q(s', a'))$.
                $s \leftarrow s'$.

            If $s$ is a terminal state, exit loop.
        end for
    end for
```

1.1.4 Policy gradient and actor-critic methods

The Q-value iteration methods is one of many of the so called Q-learning methods. In Q-learning, the agent tries to find an optimal strategy not by learning an actual policy *per se*, but by trying to approximate the utility of each state-action pair. There exists, however, another family of reinforcement learning methods, called the **policy gradient** methods, that seek to construct an *actual* policy from past experiences. In these methods, policies are defined by a tunable set of parameters (such as the constants of a PID filter, the weights of a neural network, etc), and the agent will try to adjust this parameters in a way that maximizes its long-term rewards. This means that policy gradient methods are *directly* modelling a policy.

We use the notation $\theta \in \mathbb{R}^d$, with $d > 0$, to denote the tunable parameters of a policy. We will also write as $\pi(a|s, \theta) = \mathbb{P}(A_t = a|S_t = s, \theta_t = \theta)$ the probability that, at time t, the action a is picked if the current state is s and the current policy parameters are θ. Note that the policy can be parametrized in any way [1], as long as $\pi(a|s, \theta)$ is differentiable with respect to θ.

In order to optimize the parameters, policy gradient methods will try to maximize some scalar performance measure $J_\pi(\theta)$ using *gradient ascent*:

$$\theta \longmapsfrom \theta + \alpha \widehat{\nabla_\theta J_\pi(\theta)},$$

where $\alpha \in \mathbb{R}$ is the *step size* (or *learning rate*) and $\widehat{\nabla_\theta J_\pi(\theta)} \in \mathbb{R}^d$ is an estimate whose expectation approximates the gradient (with respect to θ) of the performance

[1] In this thesis we used neural networks (see 1.2), where the parameters are the weights of the connections between nodes.

measure $J_\pi(\theta)$. Some methods (such as the Proximal-Policy Optimization used in this thesis) try to learn approximations of both the optimal policy and value functions. This family of methods are called **actor-critic methods**.

1.1.5 Proximal-Policy Optimization (PPO)

The challenge we are facing now is to find an adequate $J_\pi(\theta)$ that tells the agent "how well is the current policy doing" and helps it find an optimal policy. In addition, we must find a way to correctly approximate some value function $V_\pi(s)$ that tells the agent the expected long-term reward for reaching each state and following the current policy afterwards.

The algorithm we have chosen to solve this problem is called the Proximal-policy optimization (PPO), published in 2017 by OpenAI researchers and currently considered one of the state-of-the-art reinforcement learning methods (see [7]). Since the code of the algorithms used in this thesis have been developed using Matlab, we will present the PPO algorithm according to the steps presented in the official documentation of this software package (see [12]).

Suppose we have a way to parametrize the policy $\pi(\theta; s)$ and value $V_\pi(\phi; s)$ functions, according to some parameters $\theta = \{\theta_1, \theta_2, ..., \theta_n\}$ and $\phi = \{\phi_1, \phi_2, ..., \phi_m\}$, for example by using neural networks (see next section). We will call the function π the **actor** and the function V the **critic**. The objective of the actor will be to output the probability of taking each action $a \in A$ when the current state is $s \in \mathcal{S}$, and the objective of the critic will be to learn the expected long-term reward of a given state $s \in \mathcal{S}$.

Algorithm 2 PPO algorithm

Set a maximum number of "training" episodes. For each episode, do the following:

Step 1: Initialize the actor $\pi(\theta; s)$ with random parameter values θ and $V(\phi; s)$ with random parameter values ϕ.

Step 2: Generate a trajectory of N steps by following the current policy dictated by the actor. Let's denote the trajectory by

$$s_0, a_0, r_1, s_1, a_1, ..., s_{N-1}, a_{N-1}, r_N, s_N,$$

where s_n is a state observation, a_n is the action taken from this state S_n (as dictated by the actor), S_{n+1} is the next state after taking the action, and R_{n+1} is the reward received from the state transition from S_n to S_{n+1}.

Step 3: for each step $1, 2, 3, ..., N$ of the previous trajectory, calculate the *return* G_t using the *GAE advantage estimator method*:

$$G_t = D_t + V(\phi; s_t)$$

where D_t is the *advantage estimate*:

$$D = \sum_{k=t}^{N-1} (\beta\lambda)^{k-t}\delta_k$$

$$\delta_k = \begin{cases} r_k & \text{if } s_k \text{ is terminal,} \\ r_k + \beta V(\phi; s_k) & \text{otherwise.} \end{cases}$$

β is the discount factor, and λ is a predefined smoothing factor.

Step 4: store the trajectory $s_0, a_0, r_1, s_1, a_1, ..., s_{N-1}, a_{N-1}, r_N, s_N$ in a buffer, along with each calculated advantage D_k and return G_k. We will call this buffer the *experience buffer*.

Step 5: retrieve a mini-batch of M examples from the *experience buffer*.
Step 6: update the critic parameters ϕ by minimizing the loss function L_{critic} across all M samples of the retrieved mini-batch.

$$L_{critic}(\phi) := \frac{1}{2 \cdot M} \sum_{i=1}^{M} (G_i - V(\phi; S_i))^2$$

Intuitively, this loss function can be considered a measure of how well the critic is able to guess the actual value of a state. We will denote the previous parameters as ϕ_{old} and the new updated parameters as ϕ.

Step 7: update the actor parameters θ by minimizing the loss function L_{actor} across all M samples of the retrieved mini-batch.

$$L_{actor}(\theta) := \frac{1}{M} \sum_{i=1}^{M} - \min\{r_i(\theta) \cdot D_i \, , \; c_i(\theta) \cdot D_i\} + w\mathcal{W}_i(\theta, s_i)$$

where:

- $r_i(\theta) := \frac{\pi(\theta; s_i)}{\pi(\theta_{old}; s_i}$, and θ_{old} are the actor parameters from before the last update.
- $c_i(\theta) := \max\{\min\{r_i(\theta), 1 + \epsilon\}, 1 - \epsilon\}$, and ϵ is a predefined parameter, called the *clipping factor*.
- $\mathcal{W}_i(\theta, s_i) := -\sum_{a \in A} \mathbb{P}(\pi(\theta; s_i) = a) \cdot ln(\mathbb{P}(\pi(\theta; s_i) = a))$ is the entropy loss function, and $w > 0$ is a parameter called the *entropy weight*.

Note that this loss function for the actor is the function $J_\pi(\theta)$ we introduced in 1.1.4, and can be considered a measure of "how well" the policy learned by the actor can solve the problem at hand by maximizing the long-term reward.

Repeat **Steps 2** through **7** until reaching the maximum number of training episodes.

Other variants of the PPO method exist, such as methods with different advantage functions or variants that make use of continuous control actions. However, these methods are beyond the scope of this master's thesis.

1.2 Artificial Neural Networks

As the reader might have noticed, in Section 1.1.5 we did not discuss the way in which the policy function π and value function V should be represented, only that these functions should be dependent on some parameters θ and ϕ, and also derivable with respect to them. This omission is deliberate, as there are many different ways in which these functions can be represented. One of such ways, which has been chosen for this master's thesis, is by using Artificial Neural Networks as function approximations for both the actor and the critic.

Artificial Neural Networks (or ANNs, for short) are mathematical objects that mimic the way in which brain cells transmit and process information among them. ANNs are of particular interest in the field of Artificial Intelligence for their ability to compute large amounts of information in parallel, for being noise-resistant and for being capable of learning complex patterns and strategies.

1.2.1 Anatomy of the Artificial Neural Network

In their most usual form (see Figure 1.3), ANNs are typically constituted of one or multiple layers, each on comprised of multiple *nodes* (sometimes also called **units** or simply **neurons**). In most ANN architectures, neurons in one layer are connected to all the neurons in the following layer through directed connections. Consequently, information is propagated from the first layer to the last, but usually not in the opposite direction even through some architectures allow for recurrent information propagation.

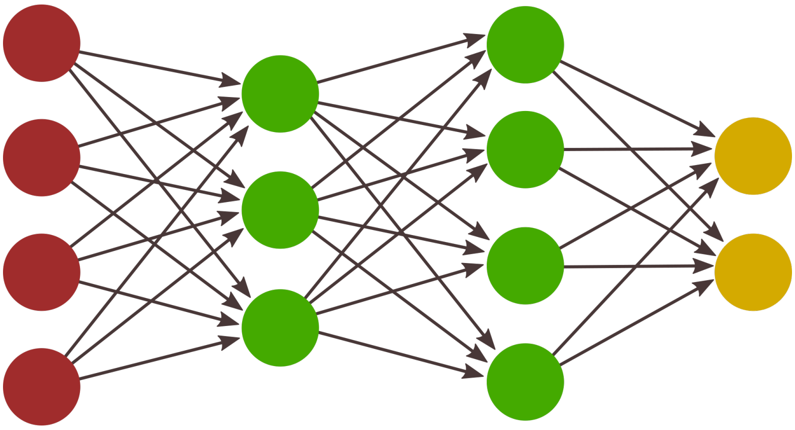


Fig. 1.3: Example of an Artificial Neural Network, consisting of an input layer of 4 units (red), two hidden layers of 4 and 3 units each (green) and one output layer of 2 units (yellow)

A node is composed of several parts (see Figure 1.4): the synaptic weights (or simply weights), the bias, the activation function and the output signal. The neuron collects the input signals, referred here as $x_1, x_2, ..., x_n$, and multiplies each one for its synaptic weight $w_1, w_2, ..., w_n$. These synaptic weights can be considered the strength of the connections between the neuron and the ones in the previous layer. These weighted inputs are then added together (sometimes a bias b is also added to the sum) and a non-linear function f, called the *activation function*, is then applied to this sum to produce the output signal.

$$\text{Output} = f\left(b + \sum_{i=1}^{n} w_i \cdot x_1\right).$$

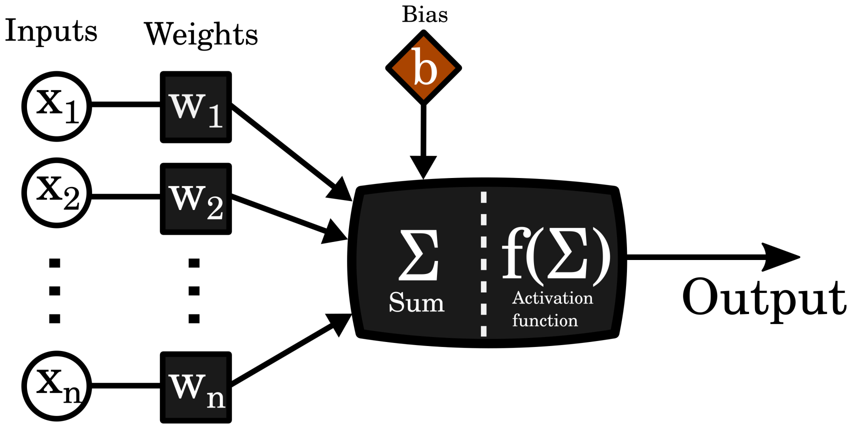


Fig. 1.4: Mathematical model for a single neuron/node.

1.2.2 Activation functions

Activation functions serve two purposes. First, they determine whether a neuron should be activated or not based on its input. If an input encodes some feature or information that the neuron deems "important", the function should activate the neuron. Secondly, activation functions add non-linear complexity to the full network and allow it to learn more complex patterns.

Some of the most commonly used activation functions are the **sigmoid**, **threshold** and **ReLU**.

1. Sigmoid function: also known as an S-shape function, and is used mainly to represent probabilities in binary classification tasks (Supervised Learning). Figure 1.5 shows the graph of this activation function.

$$f(x) = \frac{1}{1+e^x}.$$

2. Threshold function: acts like a logical boolean threshold gate. It outputs 0 if the sum is less than a predefined threshold T, or 1 if the output exceeds the threshold.

$$f(x) = \begin{cases} 0 & \text{if} \quad x < T, \\ 1 & \text{otherwise.} \end{cases}$$

3. ReLU function: the *Rectified Linear Unit* (or ReLU) is currently the most widely used activation function, and one that yields the best results in most practical ap-

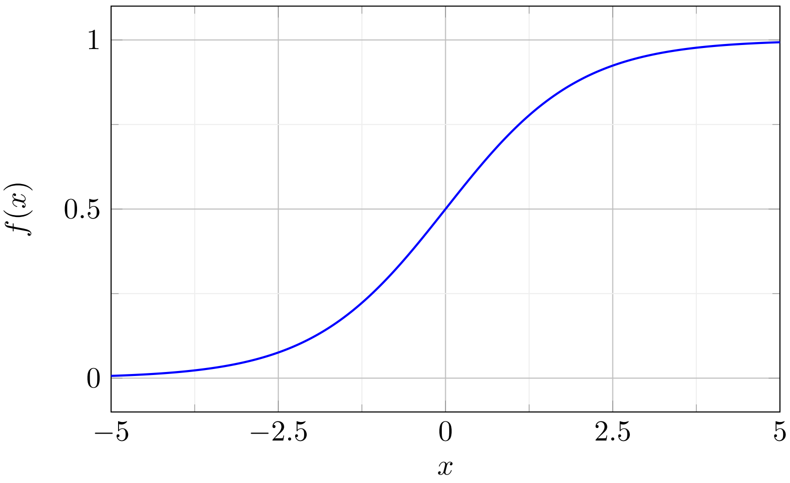


Fig. 1.5: Graph of the sigmoid activation function.

plications. It simply replaces negative values with zero, and leaves positive values unchanged.

$$f(x) = \max\{0, x\}.$$

1.2.3 Backpropagation

By carefully adjusting the weights of each node, an ANN with enough layers and connections could, in theory, approximate any real-valued function. The backpropagation algorithm (see [6]), originally published in 1986, offers a simple yet elegant way to repeatedly adjust the weights of an multi layered ANN in order to minimize a measure of difference (or *Loss*) between the actual output of the network and the desired output given a particular set of examples.

The intuitive idea behind the algorithm is the following: suppose that we have a set of labeled examples $\{e_1 = (x_1, y_1), e_2 = (x_2, y_2), ..., e_n = (x_n, y_n)\}$ of the function we want to approximate, where x_i are the inputs of the network and y_i the desired outputs. The algorithm iterates through each example e_i, feeds the input vector x_i and observes the output y_i'. Then, it calculates the difference between the desired output y_i and the actual output y_i'. This difference is the error produced by the last layer of the network; then, the backpropagation algorithm propagates back this error to the previous layers (hence the name of the algorithm) and, for each layer, calculates the "fraction" of the final error it is responsible for. When the error of each layer has been calculated, an optimization method (such as gradient descent) can be applied to adjust the weight values with respect to the error.

Algorithm 3 describes the backpropagation algorithm using a simple gradient descent algorithm as a weight optimizer. In the programs developed during this master's thesis we used the ADAM optimizer since it's faster and usually converges to better solutions (see [11]).

Algorithm 3 | Backpropagation algorithm

Inputs: a set of n labeled examples (x_n, y_n), a multilayer ANN with M layers, weights w_{ji}, activation function $f(*)$ and its derivative $f'(*)$, loss function $Loss(y_i, a_i)$, learning rate $\alpha \in (0, 1)$.

```
Repeat, for each training example $(x_i, y_i)$:

Step 1: for each node j in the input layer, calculate its output $a_j$ as $a_j \leftarrow x_j$.

Step 2: for each layer $l = 2, ..., M - 1$ do:

    $a_{lj} \leftarrow f\left(b_j + \sum_j w_{jl} a_{l-1}\right),$    for each neuron $j$ in layer $l$.

Step 3: for each node $j$ in the output layer, do:

    $\Delta_j \leftarrow f'(a_{M-1}j) \cdot Loss(y_i, a_{ji}).$

Step 4: for each layer $l = M - 1, ..., 1$ do:

    Step 4.1: for each node $j$ in layer $l$ do:

        $\Delta_j \leftarrow f'(a_j) \cdot \sum_j w_{jl} a_{l-1}.$

        Step 4.1.1: for each node $i$ in layer $l + 1$ do:

            $w_{ji} \leftarrow w_{ji} + \alpha \cdot a_j \cdot \Delta_i$    (Gradient descent).

Stop when some stopping criterion is reached (such as having an average Loss less than some threshold).
```

Attitude control 2

In this second chapter we want to study attitude control procedures of artificial satellites by means of modern machine learning techniques. The main goal to build a controller capable of learning by itself, through trial and error, how to stabilize a spacecraft by applying small torque corrections.

We will begin this chapter by reviewing some basic theoretical concepts about attitude representation and how to mathematically represent the orientation of an object in the 3D space with absolute precision. After that, we will introduce Euler's equations, that will allow us to construct a dynamical system to express the changes in the orientation of our satellite when an external torque is applied. We will also study how the attitude control problem can be represented as a Reinforcement Learning problem. Finally, we will present the results obtained during our experiments and simulations.

2.1 Attitude representation

(Except where otherwise indicated, this full Section 2.1 will follow the steps and concepts presented in Chapter 2 of [2] and Sections 2.5 and 2.6 of [5]).

This section is concerned with finding an effective way to mathematically express the attitude (i.e. 3D orientation) of a rigid body in space.

A rigid body is an object that cannot be deformed by the action of external forces. Even though from a philosophically point of view it could be argued that such bodies cannot exist in reality -every object is deformable to some extent, we will consider that our satellite is a rigid body from a purely practical point of view, since it will greatly simplify our calculations.

The mathematical representation of an object's orientation is indicated as *attitude*. Several ways to represent the attitude exist, with *Direct Cosine Matrices*, Quaternions and Euler Angles being the most usual ones. Each of these representations have their practical advantages and drawbacks. In this section, we will review some of these representations and justify why Quaternions are the most appropiate for the problem at hand.

2.1.1 Reference Frames

Let us consider the real space $\mathbb{R}^3$. A *frame of reference* $\mathcal{E} = \{O; \mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3\}$ (called a *Cartesian frame* or simply a *frame*) is a set containing three orthonormal vectors $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3 \in \mathbb{R}^3$, and an origin point $O \in \mathbb{R}^3$.

Given a three dimensional point P we define its *motion trajectory* as the coordinate vector $\mathbf{r}(t)$ of $r = \overline{OP}$ in a frame $\mathcal{E}$ during a time interval $0 \leq t < T$.

A frame $\mathcal{I} = \{O; \mathbf{i}_1, \mathbf{i}_2, \mathbf{i}_3\}$ is said to be *inertial* if its origin O is not accelerating and the axis $\mathbf{i}_j, j = 1, 2, 3$ are not rotating. This is

$$d\mathbf{v}_O/dt = 0,$$
$$d\,\mathbf{i}_k/dt = 0\,,\ k = 1, 2, 3,$$

where $\mathbf{v}_O$ denotes the velocity of the origin point O.

A *rigid body* is a continuum distribution of point masses located at a position $\mathbf{r}$ with respect to an inertial frame $\mathcal{I}$. The main property that defines a rigid body is that the relative positions between any pair of points remain constant regardless of the external forces applied to the body. That is, given two particles P_a and P_b located at $\mathbf{r}_a$ and $\mathbf{r}_b$ at time t,

$$||\mathbf{r}_a(t) - \mathbf{r}_b(t)|| = \mathit{constant},$$

for every vector norm and metric $|| \cdot ||$.

In order to study the attitude of a rigid body we need to consider two reference frames. The first one is an inertial reference frame $\mathcal{I}$, already introduced. The second frame is the *body frame* $\mathcal{B} = \{C; \mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3\}$. The choice of the center C is arbitrary, although the center of mass of the rigid body is often used.

In order to obtain the three orthonormal vectors $\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3$ we suppose that the rigid body contains three non-aligned particles with coordinates $\mathbf{r}_1, \mathbf{r}_2$ and $\mathbf{r}_3$. By taking the non-colinear vectors

$$\mathbf{u}_1 = \mathbf{r}_1 - \mathbf{r}_2, \quad \mathbf{u}_2 = \mathbf{r}_3 - \mathbf{r}_1,$$

an orthonormal basis can be build using the Gram-Schmidt orthonormalization process:

$$\begin{aligned}
\mathbf{b}_1 &= \frac{\mathbf{u}_1}{||\mathbf{u}_1||}, \\
\mathbf{b}_2 &= \frac{\mathbf{u}_2 - < \mathbf{b}_1, \mathbf{u}_1 > \mathbf{b}_1}{||\mathbf{u}_2 - < \mathbf{b}_1, \mathbf{u}_1 > \mathbf{b}_1||}, \\
\mathbf{b}_3 &= \mathbf{b}_1 \times \mathbf{b}_2.
\end{aligned}$$

The *attitude* of a rigid body is defined as the set $\mathcal{R}$ of possible representations $\mathbf{R}(\mathcal{B}, \mathcal{E})$ of the body frame $\mathcal{B} = \{C; \mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3\}$ with respect to an observer's frame $\mathcal{O} = \{C; \mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3\}$, with both frames sharing the same origin C. The choice of the observer's frame is arbitrary. Note that, since the choice of the body frame is also arbitrary, the attitude representation is uniquely defined only by the particular choice of $\{\mathcal{O}, \mathcal{B}\}$.

In the following subsections we will study three different possible representations for the attitude of a rigid body.

2.1.2 Rotation Matrices

Let P be a point with coordinates expressed in a reference frame $\{O_\mathcal{I}; \mathbf{i}_1, \mathbf{i}_2, \mathbf{i}_3\}$, and suppose that we want to express it in the reference frame $\{O_\mathcal{F}; \mathbf{f}_1, \mathbf{f}_2, \mathbf{f}_3\}$. In order to achieve this, two transformations have to be performed:

1. A *translation* of the origin $O_\mathcal{I}$ (if necessary), given by the relationship $\overline{O_\mathcal{I}\,P} = \overline{O_\mathcal{I}\,O_\mathcal{F}} + \overline{O_\mathcal{F}\,P}$.

2. A *change of basis* that allows us to represent the vectors of $\mathcal{I} = \{\mathbf{i}_1, \mathbf{i}_2, \mathbf{i}_3\}$ as a linear combination of the basis $\mathcal{F} = \{\mathbf{f}_1, \mathbf{f}_2, \mathbf{f}_3\}$.

We will call a change between two orthonormal basis of $\mathbb{R}^3$ with the same orientation a *rotation of the reference frame*, or simply a *rotation*. Therefore, the attitude of a rigid body can be represented by the rotation which transforms the body reference frame $\mathcal{B}$ into the observer frame $\mathcal{O}$.

Consider now a vector $\mathbf{x} \in \mathbb{R}^3$, and suppose its coordinates in the basis $\mathcal{I}$ and $\mathcal{F}$ are

$$\mathbf{x} = x_1\,\mathbf{i}_1 + x_2\,\mathbf{i}_2 + x_3\,\mathbf{i}_3,$$
$$\mathbf{x} = X_1\,\mathbf{f}_1 + X_2\,\mathbf{f}_2 + X_3\,\mathbf{f}_3,$$

for some $x_1, x_2, x_3, X_1, X_2, X_3 \in \mathbb{R}$. Any change of basis between $\mathcal{I}$ and $\mathcal{F}$ can be represented as an orthogonal matrix R that allows us to relate the coordinates of $\mathbf{x}$ in both basis:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = R \cdot \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \cdot \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix},$$

$$\begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} = R^{-1} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{21} & r_{31} \\ r_{12} & r_{22} & r_{32} \\ r_{13} & r_{23} & r_{33} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}.$$

This matrix R is called the *rotation matrix* between frames $\mathcal{I}$ and $\mathcal{F}$, and is sometimes denoted as $R_{\mathcal{I}\mathcal{F}}$. Rotation matrices are sometimes called *attitude matrices* when used to denote the attitude of a rigid body.

It can be proven that any rotation $R_{\mathcal{I}\mathcal{F}}$ is equivalent to a rotation of angle θ around an axis $\mathbf{e}$ with coordinates expressed in $\mathcal{I}$. Therefore, it is often common to denote a rotation matrix as $R(\theta, \mathbf{e})$. This result is known in literature as *Euler's rotation theorem* (see [5] pg. 61-62 and 68-71 for its precise statement and proof).

2.1.3 Euler Angles

An *Euler elemental rotation* of axis $\mathbf{e}_j$ and angle θ in any frame $\mathcal{O} = \{C; \mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3\}$, denoted by $R(\theta, \mathbf{e}_j)$ represents a rotation of angle θ around the axis $\mathbf{e}_j$. For any given reference frame $\mathcal{O}$, only three Euler elemental rotations can exist, whose attitude matrices are denoted by

$$R_1(\theta, \mathbf{e}_1) = X(\theta) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{pmatrix},$$

$$R_2(\theta, \mathbf{e}_2) = Y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{pmatrix},$$

$$R_3(\theta, \mathbf{e}_3) = Z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

It can be proven ([5], Section 2.6.3) that any rotation between two arbitrary frames can be expressed as a compositon of three elementary rotation matrices $X(\phi)Y(\theta)Z(\psi)$. The elements of the triad (ϕ, θ, ψ) are called *Euler Angles*.

Euler Angles are subject to a singularity known as *Gimbal Lock* whenever the rotation angle θ is either zero or $\pm\pi$, which results in the loss of at least one *Degree of Freedom* on which to rotate (see [5], Section 2.6, Lemma 1).

2.1.4 Quaternions

In order to avoid the Gimbal Lock singularity, we need to find an alternative representation to Euler Angles. Although rotation matrices could be useful since they lack singularities, quaternions are often preferred as they present a more compact representation of rotations.

Informally, we can think of quaternions as an extension of complex numbers that are built upon three distinct imaginary units. It can be proven that unitary quaternions (that is, quaternions of modulus equal to one) can be used to perfectly represent rotations in $\mathbb{R}^3$ and are free of singularities.

Formally, we define the ring $\mathbb{H}$ of quaternions as the one generated by a four-element basis which consists of the real unit 1 and three imaginary elements $\mathbf{i}$, $\mathbf{j}$ and $\mathbf{k}$ satisfying the following properties:

$$\begin{aligned} &1 \cdot \mathbf{i} = \mathbf{i}, && \mathbf{i} \cdot \mathbf{j} = -\mathbf{j} \cdot \mathbf{i} = \mathbf{k}, \\ &1 \cdot \mathbf{j} = \mathbf{j}, && \mathbf{j} \cdot \mathbf{k} = -\mathbf{k} \cdot \mathbf{j} = \mathbf{i}, \\ &1 \cdot \mathbf{k} = \mathbf{k}, && \mathbf{k} \cdot \mathbf{i} = -\mathbf{i} \cdot \mathbf{k} = \mathbf{j}, \\ &\mathbf{i} \cdot \mathbf{i} = \mathbf{j} \cdot \mathbf{j} = \mathbf{k} \cdot \mathbf{k} = -1, && \end{aligned}$$

where $\cdot$ denotes the standard product.

The elements $\mathfrak{q} \in \mathbb{H}$ are 4-dimensional vectors represented in the form

$$\mathfrak{q} = q_0 + q_1\mathbf{i} + q_2\mathbf{j} + q_3\mathbf{k},$$

where $q_0, q_1, q_2, q_3 \in \mathbb{R}$. Quaternions are also often represented as four-dimensional coordinate vectors in $\mathbb{R}^4$ as follows:

$$\mathfrak{q} = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} = \begin{bmatrix} q_0 \\ \mathbf{q} \end{bmatrix} = [q_0 \;\; \mathbf{q}]^T,$$

where $\mathbf{q}$ is used to denote the vector of coefficients of the three imaginary units.

Given two quaternions $\mathfrak{a} = [a_0 \; \mathbf{a}]^T$, $\mathfrak{b} = [b_0 \; \mathbf{b}]^T$, their sum and product are defined as follows:

$$\mathfrak{a} + \mathfrak{b} := [a_0 + b_0 \;\; a_1 + b_1 \;\; a_2 + b_2 \;\; a_3 + b_3]^T,$$
$$\mathfrak{a} \otimes \mathfrak{b} := [a_0\, b_0 - \mathbf{a} \cdot \mathbf{b},\; a_0\mathbf{b} + b_0\mathbf{a} + \mathbf{a} \times \mathbf{b}]^T,$$

where $\cdot$ denotes the dot product, and $\times$ denotes the cross product in $\mathbb{R}^3$.

We also define the norm and the conjugate of a quaternion $\mathfrak{q}$ respectively as follows:

$$|\mathfrak{q}| := \sqrt{\mathfrak{q}^T \mathfrak{q}} = \sqrt{q_0^2 + \mathbf{q}^T \cdot \mathbf{q}},$$
$$\mathfrak{q}^{-1} := [q_0 \;\; -\mathbf{q}]^T = [q_0 \;\; -q_1 \;\; -q_2 \;\; -q_3]^T.$$

Quaternions that satisfy $|\mathfrak{q}| = 1$ are called *unitary quaternions*. Given an attitude matrix $R(\theta, \mathbf{v})$, it can be transformed to a unit quaternion by using the following formula:

$$\mathfrak{q} = \begin{bmatrix} \cos(\theta/2) \\ \sin(\theta/2)\mathbf{v} \end{bmatrix}, \quad |\mathfrak{q}| = 1.$$

This result is useful since it allows us to establish a direct relationship between unit quaternions and rotation matrices.

2.2 Rigid Body Dynamics

(This section is based on Section 2.1 of [17]).

In order to properly describe the changes in attitude of an artificial satellite in space, we must find a way to correctly establish its dynamic and kinematics models.

The angular velocity ($\boldsymbol{\omega}$) of our satellite, represented in the body frame, can be described by Euler dynamic equation of a rigid body:

$$\dot{\boldsymbol{\omega}} = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} := I^{-1}(T - \boldsymbol{\omega} \times (I \cdot \boldsymbol{\omega})), \tag{2.1}$$

where $I \in \mathbb{R}^{3\times3}$ represents the inertia matrix of the rigid body and $T = [T_x, T_y, T_z]^T$ is the torque acting on the centroid of the rigid body. In most simulations, this torque T will be the *control torque* produced by the actuators of the spaceship and (in some cases) wil also contain the enviromental perturbations.

The orientation of the satellite can be described as a rotation between an arbitrary inertial frame and the body frame using unit quaternions. The changes in this orientation quaternion can be described with the following formula:

$$\dot{\mathfrak{q}} = \frac{1}{2}\mathfrak{q} \otimes \begin{bmatrix} 0 \\ \omega \end{bmatrix} . \tag{2.2}$$

Therefore, by taking into account Equations 2.1 and 2.2, the full attitude of our satellite will be described by the following system of differntial equations, sometimes called in literature the **state-space equation**:

$$\begin{cases} \dot{\mathfrak{q}} = \frac{1}{2}\mathfrak{q} \otimes \begin{bmatrix} 0 \\ \omega \end{bmatrix} , \\ \dot{\boldsymbol{\omega}} = I^{-1}(T - \boldsymbol{\omega} \times (I \cdot \boldsymbol{\omega})), \end{cases} \tag{2.3}$$

with some initial conditions $\mathfrak{q}_0$, $\boldsymbol{\omega}_0$. The attitude of the satellite can be known at any time by integrating these equations.

The attitude control problem of a rigid body has the goal of, given the current atitude $(\mathfrak{q}, \boldsymbol{\omega})$, find the maneuver (or sequence of maneuvers) in the form of torque vectors T that, when applied, drives the system defined by Equation (2.3) to some desired attitude $(\mathfrak{q}_{desired}, \boldsymbol{\omega}_{desired})$. A controller is an algorithm or procedure that allows us to solve the attitude control problem by dictating the torque that must be produced by the actuators of the spacecraft at any given time.

2.3 Attitude control as a Reinforcement Learning problem

Having introduced the attitude control problem of a rigid body in space, we now ask ourselves how it can be formulated as a reinforcement learning problem.

As seen in Chapter 1, reinforcement learning is a sub-field of machine learning that seeks ways to create intelligent agents capable of learning how to optimally perform tasks in an environment. In our particular case, the environment will be the dynamical system defined by the state-space equation (2.3), and the agent will be the controller of the satellite, which dictates the control torque that should be applied at each time step (see Figure 2.1).

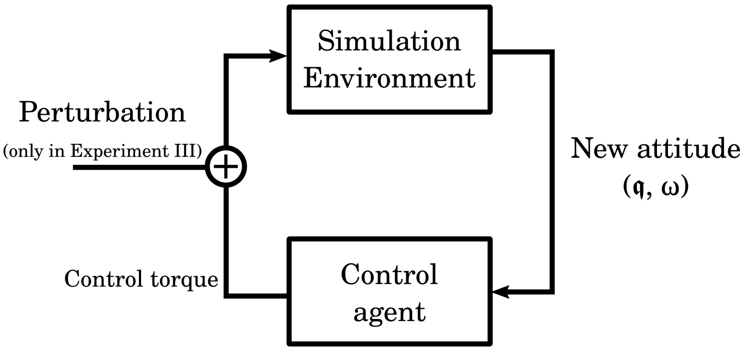


Fig. 2.1: Diagram of the simulator operation flux.

In this section we will review the general design of the reinforcement learning environment used during the experiments in order to train our AI-based attitude controller. Note that in some experiments this design may vary in some minor details that will be highlighted during the discussion in Section 2.4.

2.3.1 Objective

Our main goal will be to obtain an attitude controller capable to conduct the satellite to the attitude $\mathfrak{q}_{desired} = [\pm 1, 0, 0, 0]$ and $\boldsymbol{\omega}_{desired} = [0, 0, 0]$rad/s, starting from an arbitrary attitude.

2.3.2 Representation of the states and actions

The current state of the agent at any time will be encoded by a vector of 7 elements. The first 4 elements will represent the components of the orientation quaternion, and the last three elements will represent the angular velocity (in rad/s) around the three body axis:

$$[\overbrace{q_0, q_1, q_2, q_3}^{\text{orientation (quaternion)}}, \underbrace{\omega_x, \omega_y, \omega_z}_{\text{angular vel.}}] \tag{2.4}$$

The version of the PPO algorithm used during this thesis is a discrete control algorithm. This means that the set of actions that the agent is able to perform must be discrete. Therefore, we must find a way to discretize and encode the control torque into a set of numerable actions. We will define 31 different actions, each one corresponding to a torque value between ±1 and $\pm 10^{-4}\, N \cdot m$ in one of the three body axis (see Table 2.1).

Action number	**Torque** $(N \cdot m)$
1	$[0, 0, 0]$
2 and 3	$[\pm 1, 0, 0]$
4 and 5	$[0, \pm 1, 0]$
6 and 7	$[0, 0, \pm 1]$
8 and 9	$[\pm 10^{-1}, 0, 0]$
10 and 11	$[0, \pm 10^{-1}, 0]$
12 and 13	$[0, 0, \pm 10^{-1}]$
$\vdots$	$\vdots$
26 and 27	$[\pm 10^{-4}, 0, 0]$
28 and 29	$[0, \pm 10^{-4}, 0]$
30 and 31	$[0, 0, \pm 10^{-4}]$

Tab. 2.1: Available actions and their respective torque values.

2.3.3 Simulation environment

The simulation environment is the component of the code that simulates the behaviour of the satellite when a torque is applied. It has various tasks: numerically integrate Equation (2.3) in order to solve the satellite's current attitude, manage the initalization of each training episode, communicate the reward of each action to the agent and calculate/apply the environmental perturbations to the satellite.

Numeric integrator (RKF-45)

The Runke-Kutta-Fehlberg 45 method (or RKF45, for short) is a numerical method to numerically propagate the initial conditions of a differential equation in the form $\dot{x} = f(t, x(t))$. In this thesis we used the Matlab implementation of this method, called ode45 (see [9]).

In our particular case, f is the function defined by Equation (2.3), which depends on $\mathfrak{q}$, $\boldsymbol{\omega}$ and, indirectly, t. The inertia tensor I and the torque T are considered constants, even though the former may change at every time step.

Although alternative integration methods have been considered during the course of this thesis (such as the Adams-Bashford method), we have decided to use RKF45 as the integrator method since it is available in most packages and libraries (such as Python and Matlab), its easy to understand, and offers a good balance between precision and velocity.

Training episodes and initial conditions

In order for the agent to learn an optimal policy, we will make it try to stabilize the satellite in various independent simulations (between 2500 and 6000 simulations), each one lasting 300 seconds with a time-step of 0.1 seconds (resulting in a total of 3000 time steps for each episode). Ideally, we should see that the agent performs poorly during the first episodes, but it improves its performance as the number of completed simulations grow.

Each simulation will have different initial conditions. The satellite will begin with a randomized orientation given by a uniform random rotation (see [15]) and an angular velocity with each axis taking a random value from a Normal$(0, 1.5)$ distribution.

The inertia matrix configuration is initialized at the begining of each training episode. In this thesis we used two different inertia matrices during the experiments. The first one corresponds to a microsatellite (considered to be a cube for practical purposes) with Mass$= 5Kg$ and side$= 0.83m$:

$$I = 5 \cdot \frac{0.83^2}{6} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{2.5}$$

The second inertia matrix used during the experiments represents a cubesat (a class of cheap and small cubic satellites) with Mass$= 1.18Kg$ and side$= 0.1m$:

$$I = 1.18 \cdot \frac{0.1^2}{6} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{2.6}$$

Reward function

The objective we want our controller to fulfill is to reduce the angular velocity of the three body axis to zero, and drive the orientation quaternion to $[\pm, 0, 0, 0]$. However, the goal of a reinforcement learning agent is to find a policy that maximizes the sum of rewards. Therefore, we must carefully design a reward function that, when maximized, allows the agent to achieve the goal of driving the satellite's attitude to the desired values.

In all experiments (except the first one, in which the agent only has to control the angular velocity), the reward function will be calculated as follows:

$$Reward = -a \cdot r_q - b \cdot r_w, \tag{2.7}$$

where

$$r_q = |q_1| + |q_2| + |q_3|\,, \tag{2.8}$$
$$r_\omega = |\omega_x| + |\omega_y| + |\omega_z|, a = 3,\ \ b = 1. \tag{2.9}$$

We have tested the reward function 2.8 with different a, b values, although $a = 3$ and $b = 1$ seem to yield the best results. Additional reward functions have also been considered, but this reward function has been the one with the best performance.

Perturbations

When a satellite moves around any orbit, it will encounter constant enviromental perturbations, in the form of external torques, that will produce small, unintentional changes in its attitude. If not adressed, in the long term this will cause unstable flying and rolling. In addition, external collisions with dust particles and debris along with the loss of propellant after long periods of operation might produce changes in the mass and shape of the spacecraft, thus affecting the inertia matrix.

In Experiment III (see Section 2.4.3) perturbations are also taken into account. These perturbations are modelled as a torque vector that is added to the control torque at each integration time step. Each element of this vector is generated from a $\mu = 0, \sigma = 10^{-3}$ normal distribution. Although larger perturbations have been tried, the controller has not been able to correct them.

In addition to this torque, a small perturbation is added to the inertia tensor of the satellite during Experiment III at the beginning of each training episode, in the form of a random symmetric 3x3 matrix with its values taken from a Uniform$(0, 0.1)$ distribution.

2.3.4 Controller

The controller of the virtual satellite is composed of two neural networks, called the *actor* and the *critic*, that will be trained using the PPO method introduced in Section 1.1.5.

The agent and critic are both fully connected 5-layer neural networks of 7, 128, 128 and 64 neurons; the output layer for the actor has 31 neurons (one for each posible action) and the output layer for the critic has a single neuron (since it only has to output the value of V_π). The activation function for each layer is a ReLU, with the exception of the output layers which have no activation functions.

The hyperparameters for the neural networks and their training process are represented in Table 2.2.

Hyperparameter	**Value**
Advantage method	GAE (factor 0.95)
Reward discount	0.99
Clipping factor	0.02
Entropy Loss Weight	0.01
Learning rate (critic & actor)	$10^{-}5$
Experience horizon	1024
Batch size	512
Training epochs (critic & actor)	10

Tab. 2.2: Hyperparameters of the actor/critic networks and the training process.

When observing these states of the enviroment (i.e., when feeding state vector 2.4 into the neural networks), the values of $\omega_{x,y,z}$ are divided by 10 (a value outside the usual range of operation for this problem) in order to scale the angular velocity to values between -1 and 1, which are preferred when training neural networks. The environment still uses the original non-scaled ω values when integrating, making computations and showing the results.

2.4 Experiments and Discussion

In this section we will review, in a general way, the results obtained during 4 experiments performed with the simulation environment and the AI-based controller.

- **Experiment I:** we will train a controller to stabilize the angular velocity of the satellite, without taking into account the orientation.
- **Experiment II:** we will try to control the full attitude of the satellite. We will divide this experiment into two sub-experiments, each one with a different inertia matrix configuration.
- **Experiment III:** we will train a controller capable of stabilizing the full attitude of a satellite in the presence of perturbations. We will add environmental perturbations in the form of small torques, and we will also add small random changes to the inertia matrix at the beginning of each training episode.

In each experiment, the controller is trained through a certain number of episodes (see 2.3.3) and then tested on 25 additional episodes, without further learning (i.e. without modifying the weights of the neural networks).

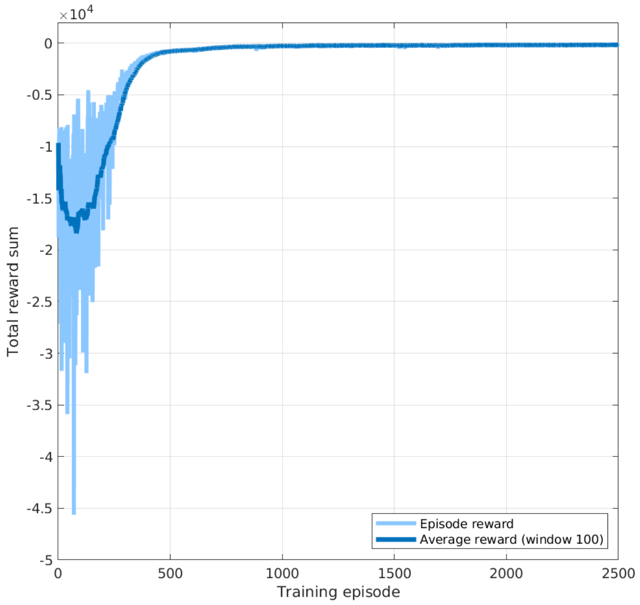


Fig. 2.2: Training process for the angular velocity controller agent used in Experiment I. Light blue lines represent the total reward sum for each training episode, while the dark blue line represents the average reward of the last 50 episodes. Notice that the agent experiences a quick improvement during the first 400 training episodes, and reaches a stable plateau around episode 500.

2.4.1 Experiment I - Angular velocity stabilization

The goal of this first experiment is to train a controller capable of stabilizing the angular velocity of a satellite by reaching $\boldsymbol{\omega}_{desired} = [0, 0, 0]$ rad/s. In this experiment we will simplify the states of the environment discussed in Section 2.3.2 and the states will be 3-dimensional vectors that represent the angular velocity in rad/s around the three body axes

$$[w_x, w_y, w_z]$$

The satellite is considered to be the microsatellite of cubic shape with the inertia tensor given by Equation (2.5). The actions available to the agent are the ones specified in Table 2.1, and the reward function is given by Equation (2.7).

The agent has been trained for a total of 2500 episodes, each one with a length of 120 seconds and a time-step of 0.1 seconds, resulting in 1200 steps for each training episode. The hyperparameters of the actor/critic networks and the training process are the ones specified in table 2.2.

Figure 2.2 shows the shift in total episode reward through the full training process. After the training process is completed, the agent is run through 25 different simulations in order to evaluate its performance. Figure 2.3 shows the results obtained during one of these simulations. Notice that the controller is able to quickly control the angular velocity and keep it stable, with the exception of some instantaneous spikes due to having discrete control torque actions.

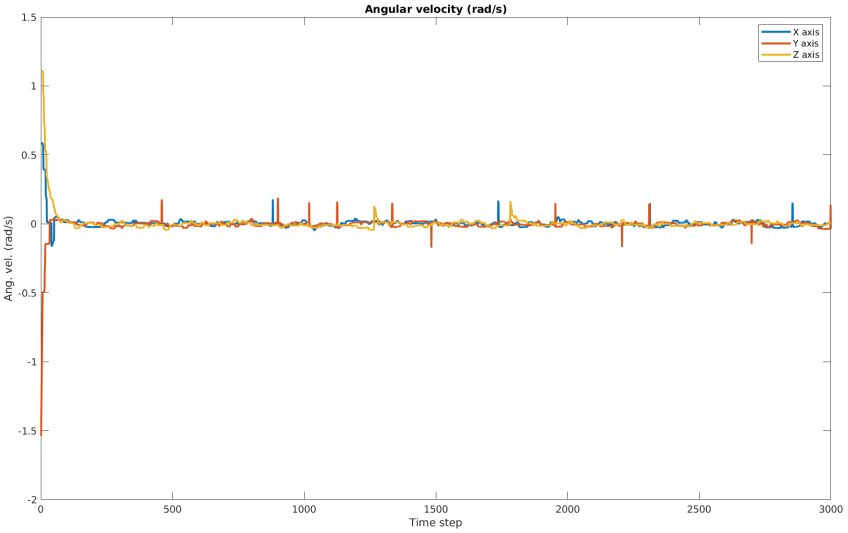


Fig. 2.3: Simulation results for the angular velocity controller agent used in Experiment I.

2.4.2 Experiment II - Full attitude control

Having been able to train a controller capable of stabilizing the angular velocity of a satellite, the next step is to control the full attitude (both orientation and angular velocity). This experiment has been divided into two different sub-experiments, each one with a unique inertia matrix configuration and slightly different available actions and training conditions.

II.a - Microsatellite

In this sub-experiment, the satellite is also considered to be a microsatellite with the same shape and mass defined in Experiment I.

The spacecraft begins each training episode at an arbitrary orientation, as explained in Section 2.3.3. The goal of the agent will be to drive the satellite to the attitude $\mathfrak{q}_{desired} = [\pm 1, 0, 0, 0]^T$ and $\omega_{desired} = [0, 0, 0]^T$. Recall that two unit quaternions with opposite sign represent exactly the same orientation. Therefore, the quaternions $[\pm 1, 0, 0, 0]$ represent the same rotation, and the controller will be successful if it manages to reach any of them.

The actions available to the agent are the same 31 actions that appear in Table 2.1, and the reward function is given by Equation (2.8). Note that, in contrast to Experiment I, this reward function takes into account both the orientation and the angular velocity of the satellite.

The agent has been trained for 6000 episodes, each one with a length of 300 seconds and a time-step of 0.1 seconds, resulting in 3000 steps for each episode. The hyperparameters for the PPO training process are specified in Table 2.2.

The results of the training process for the control agent are shown in Figure 2.4. We observe that the agent performs poorly during the first 500 training iterations and with a great dispersion in total rewards, which is to be expected since the actor network has not yet learn a suitable control policy. The agent experiences a steep increase in performance after training iteration 500, and reaches a plateau around iteration 1000. Afterwards, the improvements are slower and seem to converge towards 500.

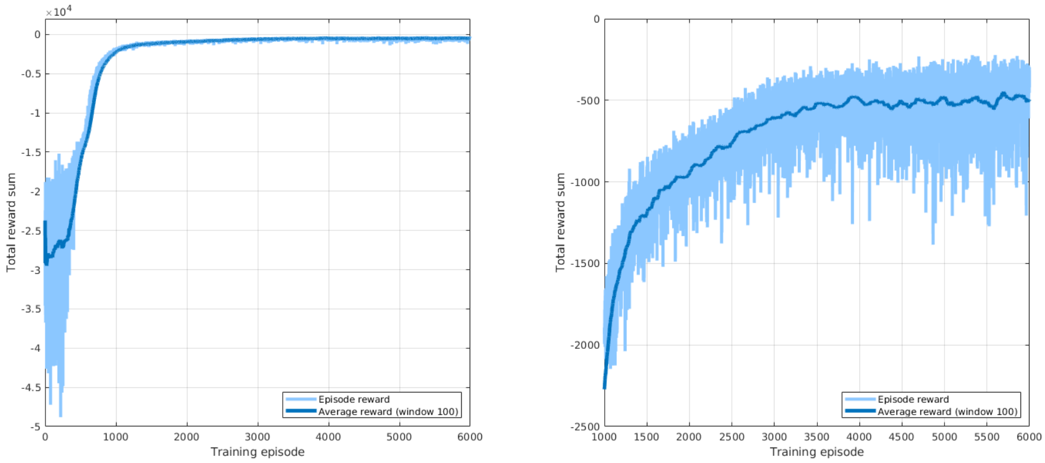


Fig. 2.4: Training process of the microsat. Light blue lines represent the reward for each individual training episode. The hard blue line is the average reward of the previous 100 episodes. Left image is shows the full training process, while image on the right shows the training results after episode 1000.

After the training process is done, the agent is tested in 25 simulation episodes, without training. Figure 2.5 show the behaviour of the satellite during the full length of one of these simulations. We can observe that the controller is able to quickly stabilize the attitude and maintain it stable afterwards, despite some small spikes in angular velocity due to having a discrete action torque. Figure 2.6 is a closeup of the attitude error after time step 500, showing that the precision of the controller is less than ± 2.5 degrees, with an average around ± 1 degrees.

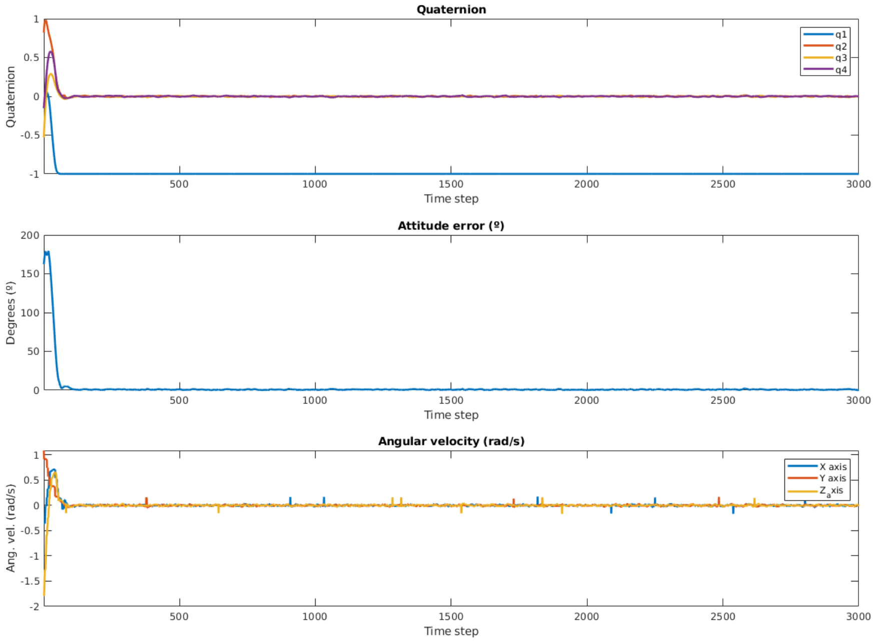


Fig. 2.5: Simulation results for the microsatellite control agent.

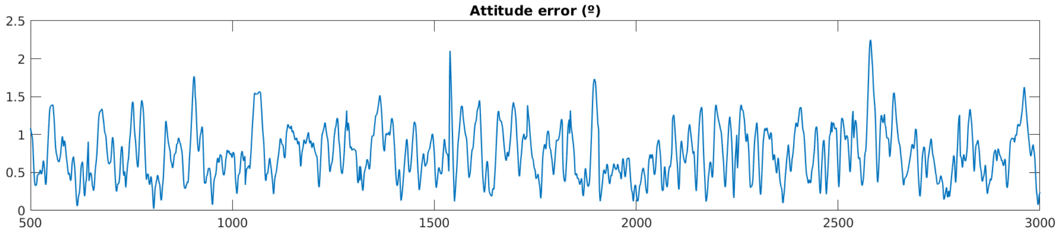


Fig. 2.6: Closeup on the attitude error of the microsatellite control agent after time step 500.

In conclusion, it would seem that our AI-based controller is perfectly capable to stabilize a microsatellite in a short time, even with a relatively large initial error in attitude.

II.b - Cubesat

A *Cubesat* is a type of miniaturized satellite that tipically perform simple earth observation tasks or serve as a proof of concept or prototype for new spacecraft technologies. These types of satellites usually weight no more than 2Kg and are shaped as cubes (hence the name), tipically with a side length of 10 centimeters or less and a cost of around $100.000 (excluding launch costs). Cubesats tend to be put into a Low Earth Orbit (LEO) towed by other bigger spacecrafts.

In this experiment we simulated the ^{3}Cat4 Cubesat [1], currently being manufactured and tested by the *Universitat Politècnica de Catalunya* (UPC), and tried to control its attitude using our reinforcement learning controller, with results similar to those obtained in the previous experiments.

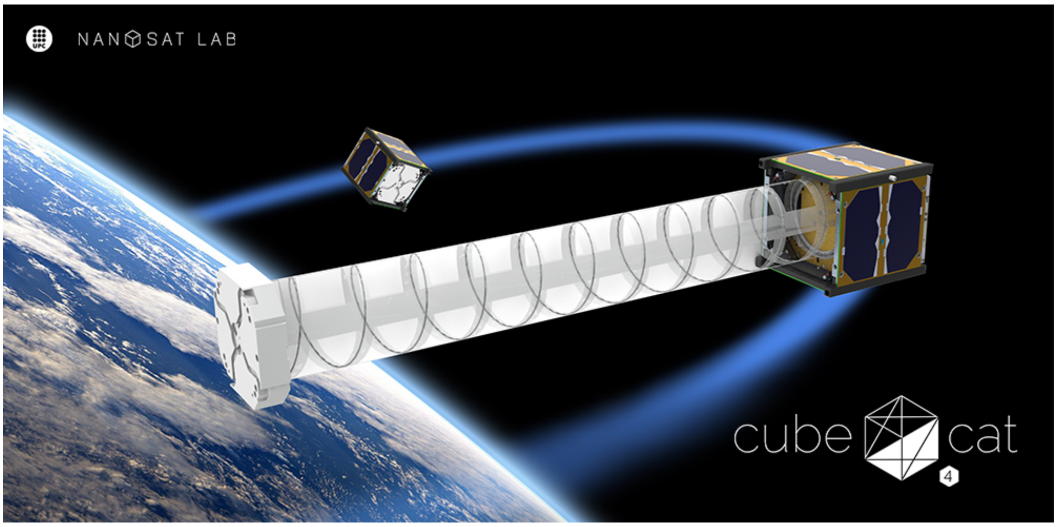


Fig. 2.7: Artistic render of the ^{3}Cat4 cubesat both in its stowed and deployed shapes. We only consider its stowed form when computing the inertia matrix. Credits: [1].

The inertia matrix of this satellite is represented, according to the specifications in the official website (Mass = 1.18Kg, side = 100mm), as

$$I = 1.18 \cdot \frac{0.1^2}{6} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The discrete actions available have been modified with respect to the ones in Table 2.1, given that a torque produced by the actuators of a cubesat is usually very limited:

Action number	**Torque** $(N \cdot m)$
1	$[0, 0, 0]$
2 and 3	$[\pm 10^{-1}, 0, 0]$
4 and 5	$[0, \pm 10^{-1}, 0]$
6 and 7	$[0, 0, \pm 10^{-1}]$
8 and 9	$[\pm 10^{-2}, 0, 0]$
10 and 11	$[0, \pm 10^{-2}, 0]$
12 and 13	$[0, 0, \pm 10^{-2}]$
⋮	⋮
26 and 27	$[\pm 10^{-5}, 0, 0]$
28 and 29	$[0, \pm 10^{-5}, 0]$
30 and 31	$[0, 0, \pm 10^{-5}]$

Notice that the maximum torque output available to the cubesat is smaller than the one available to the microsat in the previous experiment. For this reason, the duration of each training episode has been increased from 300 to 500 seconds in order to give the spacecraft more time to reach its goal. The reward function used during the experiment is the same defined Equation (2.8) . The other hyperparameters of the neural networks and training process remain unchanged with respect to the other experiments.

Figure 2.8 shows the training process for the Cubesat controller. Notice that the agent experiences a quick and steep increase in performance that reaches a plateau around training episode 300. The average sum of rewards for each episode is lower than the previous episode, mainly beacause the cubesat can apply less torque than the microsatellite and also because training episodes are longer (and, therefore, the agent accumulates more negative reward).

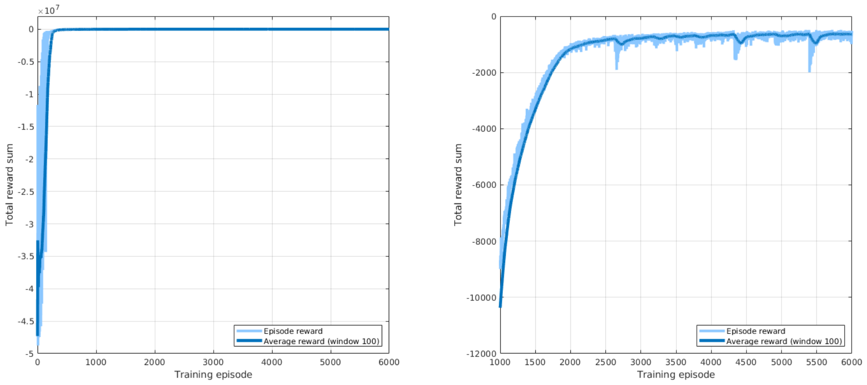


Fig. 2.8: Training process for the cubesat controller. Light blue lines represent the reward for each individual training episode. The hard blue line is the average reward of the previous 100 episodes. Left image is shows the full training process, while image on the right shows the training results after episode 1000.

After the training process has been completed, the agent has been run through various simulations. Figure 2.9 shows the results for one of these simulations. Figure 2.10 show a closeup on the attitude error after simulation time step 500.

In conclusion, it would seem that our AI-based controller is also able to stabilize a small cubesat in a very short time, even though the output torque available to the agent is very limited.

2.4.3 Experiment III - Perturbations

In this third experiment, we will try to obtain a controller capable of controlling and stabilizing the attitude of the microsatellite employed in Experiments I and II.a, but being subject to random enviromental perturbations as described in Section 2.3.3. The characteristics of the environment and actions are the same used in Experiment II.a (except for the fact that now we have perturbations). The hyperparameters of the training process and the topology of the critic and actor networks are also unchanged.

Figure 2.11 shows the training process for the control agent through all 6000 training episodes. Notice that the agent converges to a stable solution around iteration $500 - 1000$, and then it improves slowly to an average episodic reward of $550 - 600$. This shows that this agent improves more slowly than the one used in Experiment II.a, which is to be expected since it is facing a much harder problem.

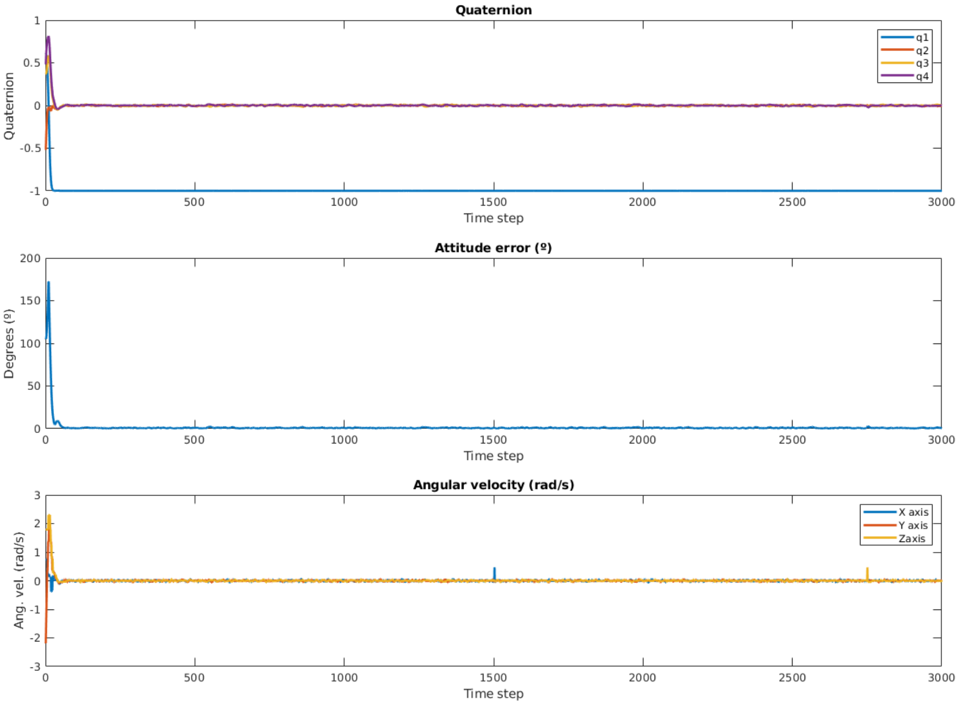


Fig. 2.9: Simulation results for the Cubesat control agent.

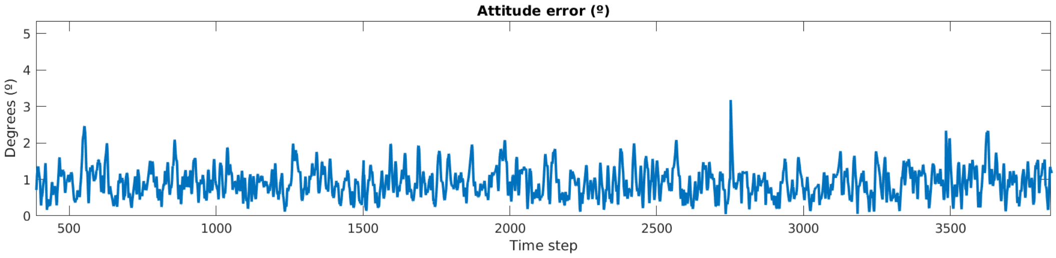


Fig. 2.10: Closeup on the attitude error after simulation step 500. Notice that the absolute error is never superior to ±4 degrees.

Figures 2.12 and 2.13 show the results of a simulation performed on the trained agent, with the randomized inertia matrix:

$$I = \begin{pmatrix} 0.5777 & 0.0422 & 0.0352 \\ 0.0422 & 0.6042 & 0.0255 \\ 0.0352 & 0.0255 & 0.6277 \end{pmatrix}$$

In conclusion, this experiment shows that our AI-based controller is able to successfully and quickly correct the attitude even in the presence of environmental perturbations and a varying inertia matrix.

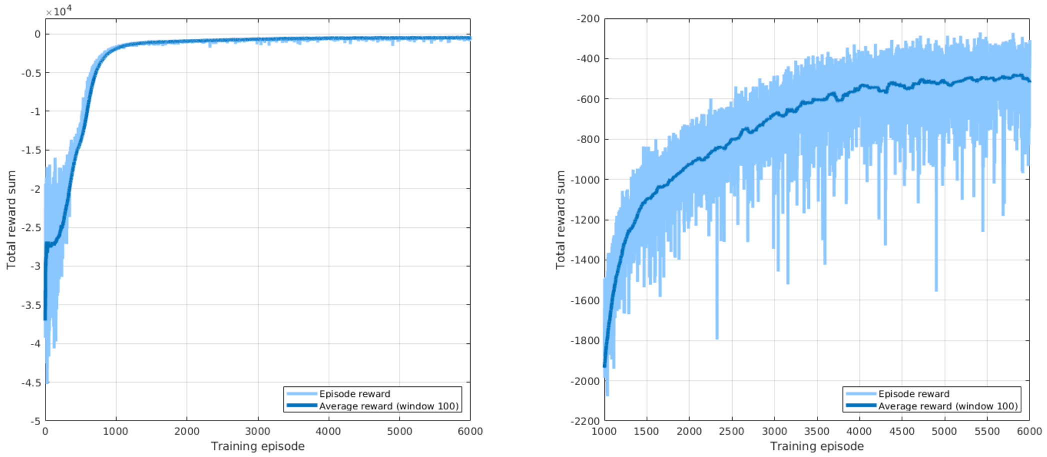


Fig. 2.11: Training process for the agent trained in an enviroment with perturbations. Left image is shows the full training process, while image on the right shows the training results after episode 1000.

2.5 Future work

Although the obtained results are acceptable and in the lines of other recent, similar works (see [16], [17]), further research should be made in order to obtain a controller that could, in theory, be uploaded to the computer of a real mission. In particular, the author thinks that research efforts should be focused on the following aspects:

- **Precision:** the controller should be improved to have greater precision and achieving a smaller attitude error, since some real-world space missions (such as radiotelescopes or communication buses) require precise alignment. In order to achieve this, the author suggests exploring alternative reward functions that exponentially grow when the attitude error decreases, and also using continuous actions instead of discrete.
- **Multiple inertia tensors:** the author thinks that a single controller capable of controlling multiple inertia matrix configurations would be of great use since the shape, mass and size of spacecraft usually changes during missions (due to the loss of propellant, collisions with other bodies with the consequent loss of mass, etc). Some experiments have been performed during this thesis to obtain a controller able to work with multiple satellite shapes by fully randomizing the inertia tensor at the beginning of each epsiode. Results have been very poor, even when an approximation of the inertia tensor is encoded into the input vector of the neural network. In future work, the author would suggest trying a different class of neural networks, such as Spiking Neural Networks or LSTMs. These network architectures have the peculiarity that their current output depends on previous outputs, thus giving the agent some notion of *memory* and allowing it to infer the inertia matrix from past experiences in the current training/simulation episode.

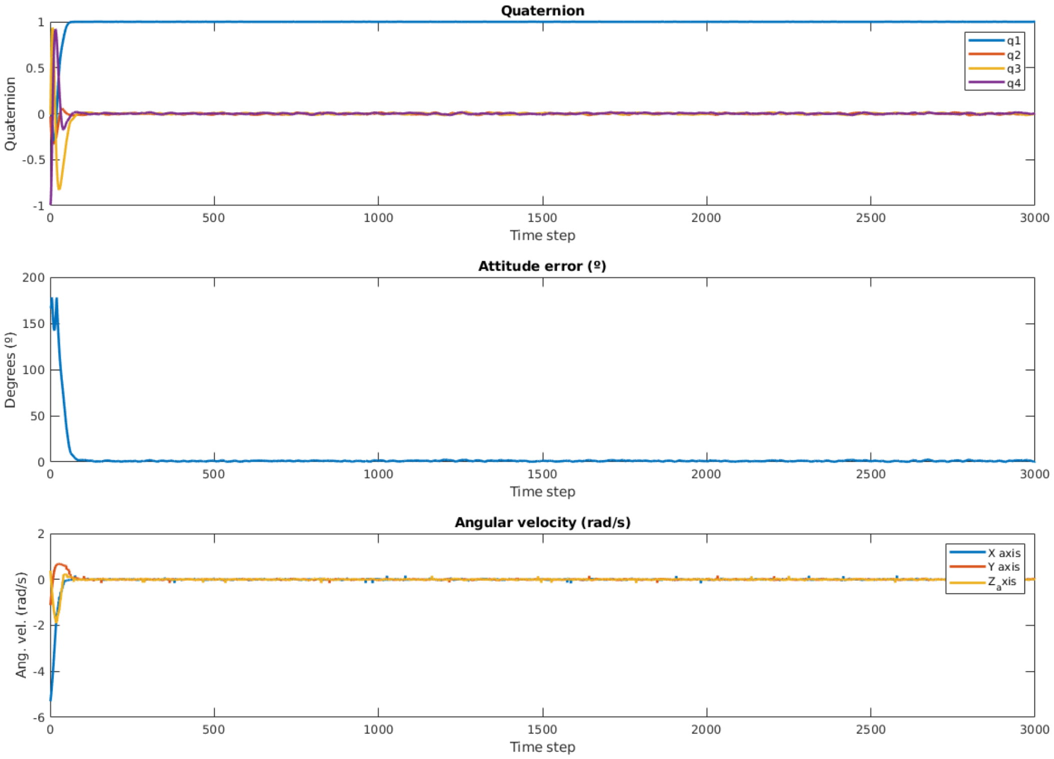


Fig. 2.12: Simulation results for the control agent. The graphs show that the agent is able to quickly orient the satellite and mantain its correct attitude afterwards, despite constant environmental perturbations.

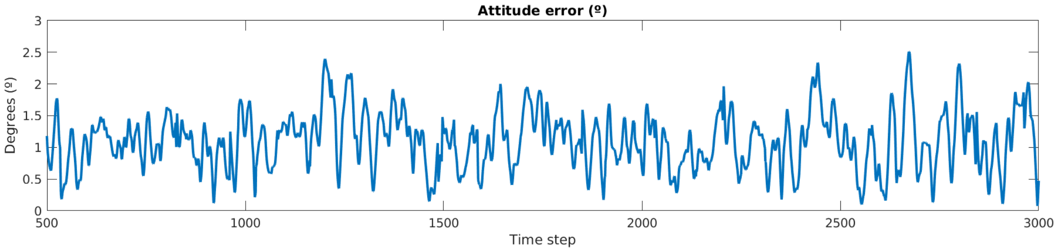


Fig. 2.13: Closeup on the attitude error after time step 500 (50 seconds). The attitude error is always less than ±3 degrees.

Poincaré map approximation 3

Suppose we have an orbital system consisting of two large objects with great mass (e.g. the system composed by the Sun and Earth), and suppose we have a small object of negligible mass (such as a satellite, or a small spaceship). The Restricted Three Body Problem (RTBP) is the problem of determining the movement of the small object with respect to the two large bodies when subject only to gravitational forces. The RTBP is of great interest, for example, because the knowledge of its dynamics is the basis to construct real missions such the SOHO or the James Webb telescope that move through quasi-periodic orbits around equilibrium points.

Another interesting case of dynamical system are models of classical atomic physics. The Coulomb law is essentially the same as the gravitational (Newtonian) one, so that the type of systems of differential equations that models the problem are quite similar. An example that we will present here is the motion of an electron of a Hydrogen molecule under a circularized polarized microwave field. One challenging problem is the study on which condition the ionization of the molecule occurs (when the electron is able to escape). For this, the comprehension of the dynamics is crucial.

A very useful tool to study the global dynamics of these type of models around stable equilibrium points or periodic orbits is the *Poincaré map*. Informally, the Poincaré map of a dynamical system is a way to represent the dynamics in a lower-dimensional space, usually a section Σ given by some hyperplane. Roughly speaking, given a point $x_0 \in \Sigma$ and the solution (trajectory) that goes through this point, the Poincaré map is a function $P : \Sigma \to \Sigma$ which assings to the point x_0 the next intersection of the trajectory with the section Σ.

In this third chapter we want to approximate the Poincaré map of a dynamical system using an artificial neural network in two different ways. More concretely, we want to build a neural network that is able to reproduce a Poincaré map, first forwards in time, second backwards in time.

Unfortunately, results obtained on this second problem have not been as good as expected, and the Poincaré map approximated with the neural network does not yield an accurate representation of the phase dynamics. Nevertheless, we will review the results obtained, try to justify why it might be failing and propose a future line of research in which we believe this program could be improved upon.

3.1 Dynamical systems and Poincaré maps

(Contents on this section will be based on Sections 1 and 2 of Ch. I of [10]).

In this section we will introduce a broad definition for dynamical systems and Poincaré maps.

In the context of physics, a **system** is a collection of objects that can be detected or measured (the position of a collection of objects, the temperature of a gas, etc).

The dynamics (i.e. changes over time) of a system can be usually modelled as a **dynamical system**.

Definition 3.1.1 (Dynamical system)**.** Let X be a metric space. A **dynamical system** over X is a triple $(X, \mathbb{R}, \phi)$ where $\phi : \mathbb{R} \times X \to X$ is a map satisfying the following three axioms:

- $\phi(0, x) = x$, $\forall x \in X$.
- $\phi(t_2, \phi(t_1, x)) = \phi(t_1 + t_2, x)$, $\forall t_1, t_2 \in \mathbb{R}$ and $x \in X$.
- ϕ is continuous.

The map ϕ is called the **phase map**, and the space X is called the **phase space** (of the dynamical system).

Consider now a differential equation in the following form (for simplicity, we just consider the autonomous case):

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

where $f : \mathbb{R}^n \to \mathbb{R}^n$ is a continuous function. Assume that for each $x_0 \in \mathbb{R}^n$ a unique solution $\varphi(t, x_0)$ of Equation (3.1) exists which is defined over $\mathbb{R}^n$ and $\varphi(0, x_0) = x_0$. It can be proven that the map $\phi : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ with $\phi(t, x) = \varphi(t, x)$ defines a dynamical system over $\mathbb{R}$.

We will usually call the function $\varphi(t, x_0)$ the flux of x_0, or (in the context of astrodynamics and celestial mechanics) orbit or trajectory.

Let Σ be a section over X (usually we will work with sections with the form $x =$ constant, $y =$ constant, etc).

Definition 3.1.2 (Poincaré map)**.** Let $\Sigma \subset X$ be a section. A Poincaré map is a function

$$\begin{aligned} P : \Sigma &\longrightarrow \Sigma \\ x &\longrightarrow P(x) = \phi(t_f, x) \end{aligned}$$

where t_f is the time that ϕ needs to reach Σ when starting from x at $t = 0$.

Finally, a system has a first integral if there exists a function $F(x)$ such that it remains constant along the solutions.

3.2 From Celestial Mechanics to molecular dynamics

(The contents on this Section are based on [4]).

For the second part of this thesis the original intention was to approximate a Poincaré map of the RTBP. However, in order perform the first experiments with our neural-network regressor, we decided to begin with an alternative, more well-behaved problem: the CP problem, which studies the movement of an hydrogen atom when it is subject to a polarized microwave field. Although this is not an astrodynamics problem, its equations and general behaviour are analogous to the ones found in restricted three body problem.

Due to the poor results obtained with the neural network trying to approximate the Poincaré Map of the CP problem, we did not have time to actually work with the RTBP during this thesis.

The CP problem consists on studying the relative motion of an hydrogen atom when subject to a circularly polarized (CP) microwave field, and it is similar to the restricted three body problem in celestial mechanics.

The problem can be described in a rotating frame by the system of equations

$$\begin{cases} \dot{x} = \dot{x}, \\ \dot{y} = \dot{y}, \\ \ddot{x} = 2\dot{y} + x - \frac{x}{r^3} - K, \\ \ddot{y} = -2\dot{x} + y - \frac{y}{r^3}. \end{cases} \tag{3.2}$$

where (x, y) are the coordinates position of the particle, $r^2 = x^2 + y^2$, and $K = F/\omega^{4/3}$, where $F > 0$ is the strength of the microwave field (in Volts/meter) and ω is the angular frequency of the microwave field. The system has a first integral (is conservative) given by the function

$$H = \frac{1}{2}\left(p_x^2 + p_y^2\right) - xp_y + yp_x - \frac{1}{r} + Kx. \tag{3.3}$$

That is, $H = h$ (called energy) is constant along the solutions.

The problem has several features described in [4]. For example, exists a family of stable periodic orbits. We fix a value of the energy h and the periodic orbit for that energy level. To study the dynamics around it, the Poincaré map is a very good tool. Figure 3.1 shows an example of a Poincare map for system (3.2).

3.3 Approximating Poincaré maps as a supervised learning problem

Supervised learning is a paradigm of machine learning which uses labeled data examples to teach a model (such as an ANN, see Section 1.2) to produce a desired output. Regression is a supervised learning technique in which models try to predict a continuous value.

The steps to train a neural network using a supervised learning approach are shown in Algorithm 4.

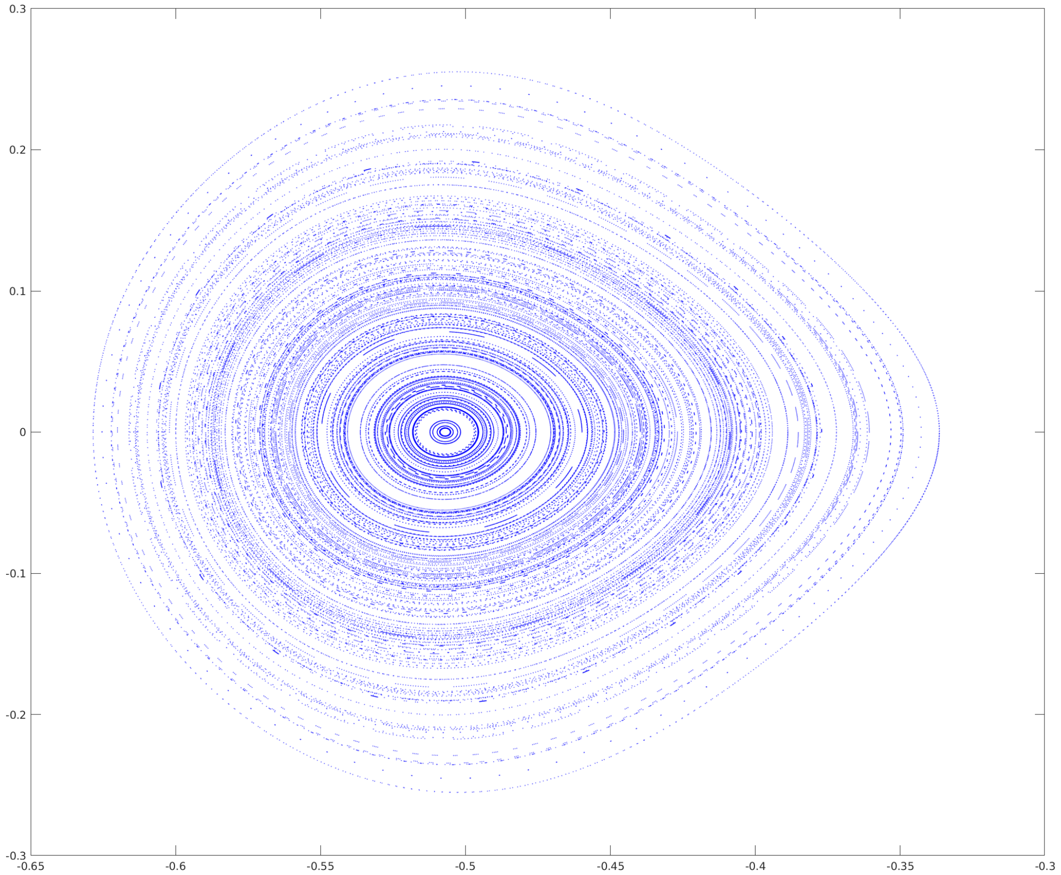


Fig. 3.1: Example of the Poincaré map of the CP problem with section $\dot{x} = 0$, $\dot{y} < 0$, with $H = h = -1.7$ and $K = 0.0015749$. It can be proven that a periodic orbit (a point p such that $P(p) = p$) exists at $x = -0.5070085504151148$, $y = \dot{x} = 0$ and $\dot{y} = -0.8963$. This map has been generated by taking 270 initial conditions, numerically integrating them and saving the first 5000 intersections of each one with the Poincaré section. Note that the figure displays some very defined patterns of invariant curves (which show the existence of quasi-periodic orbits), islands that surround other periodic orbits, etc.

Algorithm 4 | Supervised learning (with ANNs)

Inputs: a dataset of n labeled examples (x_i, y_i), $i = 1, \dots, n$, a multilayer ANN $M_\theta(x_i)$ with weights θ, a measure $Loss(y, \hat{y})$ of the error between some predictions $\hat{y} = \{\hat{y}_0, ..., \hat{y}_m\}$ of the ANN and the actual labels $y = \{y_0, ..., y_m\}$, a method for optimizing the weights θ (such as Backpropagation + ADAM).

Repeat the following steps, until some stopping criterion is reached (such as having a $Loss$ less than some threshold):

Step 1: Collect a batch of m examples from the dataset, with $m \leq n$.

Step 2: for each collected example $(x_0, y_0), ..., (x_m, y_m)$, predict $\hat{y}_i \leftarrow M_\theta(x_i)$.

Step 3: calculate $Loss(y, \hat{y})$, where $y = \{y_0, ..., y_m\}$ and $\hat{y} = \{\hat{y}_0, ..., \hat{y}_m\}$.

Step 4: optimize the ANN weights θ according the the calculated $Loss$ value.

In our particular case, we want to teach two neural networks to approximate the Poincaré map of the CP problem. We will generate a dataset that will contain a large quantity of points of a poincaré map for the CP problem (see Section 3.3.1), and use it to train two different regressors. The first one will recieve some initial conditions $(x, y, \dot{x}, \dot{y})$ as input and will try to predict the first intersection with the poincaré section $\dot{x} = 0, \dot{y} < 0$ of the orbit that runs through this point. The second regressor will do the oposite: given the coordinates of an intersection point, it will try to predict the asociated initial conditions. From now on, in order to avoid confusion (and even though both models are actually regressors) we will call the first model the *progressor*, and the second one the *regressor*.

In this thesis, the chosen $Loss$ function has been the *Mean Absolute Error*, which is often employed in machine learning regression tasks:

$$MAE(y, \hat{y}) := \frac{\sum_{i=0}^{m} |y_i - \hat{y}_i|}{m},$$

where m is the length of both y and $\hat{y}$. The chosen weight optimization method is Backpropagation (see Algorithm 3) with the ADAM optimizer.

3.3.1 Dataset generation

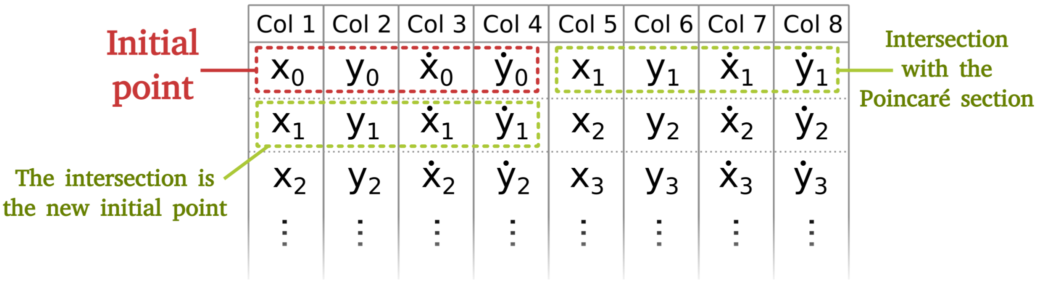
The first step towards building both the *regressor* and the *progressor* is having a dataset to train the neural networks. In our case, we generated three different datasets training datasets plus two testing datasets, each consisting of thousands of points calculated from the system given by Equation (3.2), with energy level $h = 1.7, K = 0.0015749$ and the Poincaré section given by $\dot{x} = 0, \dot{y} < 0$. By having different training datasets, we can perform multiple experiments and compare the performance of the various models.

The first dataset has been generated from sequence of equidistant points $(x_i, 0, 0, \dot{y}_i)$, with $x_0 = -0.63, x_1 = -0.62, x_2 = -0.61, ..., x_n = -0.36$ and $\dot{y}_i$ obtained by solving Equation (3.3) for $\dot{y}$:

$$\dot{y}^2 = 2h + r^2 + \frac{2}{r} - 2Kx. \tag{3.4}$$

Note that, since $x, y, \dot{x}_0$ are defined and $\dot{y}$ must be less than zero (otherwise it would not belong to our chosen Poincaré section), the previous equation has a unique solution for $\dot{y}$.

For each of these initial conditions $(x_i, 0, 0, \dot{y}_i)$, Equation (3.2) has been numerically integrated using RKF78 to find 5000 intersections with the Poincaré section. These intersections have been saved in a `.csv` file with the convention shown in Figure 3.2 (this convention is consistent among all other datasets). We will call this first dataset the **equidistant dataset**.


Initial point
Intersection with the Poincaré section
The intersection is the new initial point

Col 1	Col 2	Col 3	Col 4	Col 5	Col 6	Col 7	Col 8
x_0	y_0	$\dot{x}_0$	$\dot{y}_0$	x_1	y_1	$\dot{x}_1$	$\dot{y}_1$
x_1	y_1	$\dot{x}_1$	$\dot{y}_1$	x_2	y_2	$\dot{x}_2$	$\dot{y}_2$
x_2	y_2	$\dot{x}_2$	$\dot{y}_2$	x_3	y_3	$\dot{x}_3$	$\dot{y}_3$
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Fig. 3.2: Format of the `.csv` dataset files.

The second dataset has been generated in a similar way than the *equidistant dataset*, but the x values of the initial points have been randomly generated by taking 500 values from a Uniform$(-0.63, -0.36)$ distribution, without repetition. As with the *equidistant dataset*, Equation (3.2) has been numerically integrated from these initial points to find 5000 intersections with the Poincaré section. We will name this second dataset the **random dataset**.

The third dataset has been generated by taking a grid of points with $x \in (-0.54, -0.46)$ with a step of 0.001, $y \in (-0.1, 0.1)$ also with a step of 0.001, $\dot{x} = 0$ and $\dot{y}$ given by Equation (3.4). Each initial point has been integrated only until its first intersection. We will call this third dataset the **limited dataset**. This dataset was generated after the first tests with the neural networks, due to the poor results obtained. With this dataset, we wanted to check if by limiting the area which the neural network has to learn, the precision improved. As seen in Section 3.5.3, this unfortunately has not been the case.

Two additional dataset has been generated to test the performance of the ANNs. The first one is called the **testing dataset**. This dataset has been generated in a similar way than the *equidistant dataset*, but using a step of 0.001 between the x values, and calculating only 300 intersections instead of 5000. This dataset has been used to test the precision of the networks trained with the *random* and *limited* datasets.

The second testing dataset is called the **limited testing dataset**, and has been generated by taking initial x_0 conditions between $(-0.54, -0.46)$ with a step of 0.001, $y_0 = \dot{x}_0 = 0$, $\dot{y}_0$ given by the energy level, and calculating 1000 intersections for each one of these initial conditions with the Poincaré section.

3.4 Architecture and hyperparameters of the ANNs

Having generated the training and testing datasets, the next step is building both the *progressor* and *regressor* models. Both the *progressor* and *regressor* consist of three fully-connected neural networks. Each of these networks will recieve a 3-dimensional vector as input, consisting of the coordinates $(x, y, \dot{y})$ of an initial point ($\dot{x}$ is ignored, since in our Poincaré section it is always zero), and each network outputs the value of one of the coordinates x, y or $\dot{y}$ (see Figure 3.3).

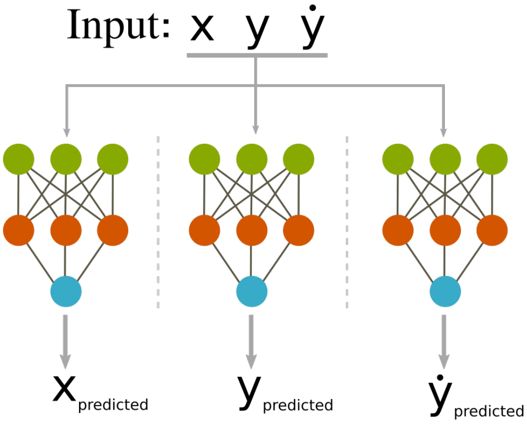


Fig. 3.3: Each model consists of three ANNs. Each ANN recieves the coordinates of a point (excluding $\dot{x}$) and outputs the prediction of a coordinate.

In both models, the network that predicts the x coordinate is a fully-connected ANN with 3, 32, 64, 128, 64, 32 and 1 nodes, ReLU activation function, and a regression layer as output. The networks that predict y and $\dot{y}$ are bigger, consisting of 8 layers of 3, 256, 512, 512, 1024, 1024, 512 and 1 nodes, ReLU activation function, and a regression layer as output. The hyperparameters used to train all networks are shown in Table 3.1 . In addition to this, the rows on the dataset have been shuffled at the beginning of each training epoch.

Hyperparameter	Value
Initial learning rate	10^{-4}
Learning rate drop factor	0.1
Learning rate drop period	every 5 epochs
Max. training epochs	15
Mini-batch size	4096/2048/64

Tab. 3.1: Hyperparameters used during the training process. Note that the batch size used is different for the networks trained with the limited dataset. Also note that we have multiple values for the batch size: the value 4000 was used during the first tests with the equidistant dataset. The value 2048 was used in the subsequent attempts to improve the performance, and during the tests performed with the random dataset. The value 64 is the one used with the limited dataset.

Recall that, as shown in Figure 3.2, columns 1-4 of the datasets contain the values $(x, y, \dot{x}, \dot{y})$ of the initial points, and columns 5-8 contain the coordinates of the first intersection with the Poincaré section. The ANNs of the *progressor* are trained using columns 1-2-4 as features and columns 5-6-8 as targets/labels, while the *regressor*'s ANNs are trained using cols. 5-6-8 as features and cols. 1-2-4 as targets/labels.

3.5 Results

In this section we will review the results obtained with the models when trying to predict the contents of the test datasets. As stated at the beginning of this chapter, results obtained have not been as good as expected, and the approximated Poincaré map learned by the neural networks does not provide an accurate representation of the phase dynamics of the CP problem. Although in all cases the ANNs have been trained with a final MAE error less than 10^{-5}, Figures included in this section show that none of the two models is able to correctly learn the Poincaré map of the CP problem, regardless of the dataset used for training.

3.5.1 Equidistant dataset

Figure 3.6 shows the predictions of the *progressor* over the points of the testing dataset when trained with the equidistant dataset and using the hyperparameters specified in 3.4. Blue markers represent the real (x, y) values of the intersection points, and red markers represent the predictions done by the neural network. Ideally, both sets of markers should match perfectly; however, it can be easily noticed that although the networks seems to preserve some of the global dynamics of the system, it does not provide an accurate representation of the poincaré map for the CP problem.

Figures 3.4 and 3.5 show the first results obtained in one of the experiments when training the networks with the equidistant dataset, and with a batch size of 4096. Note that both figures show innacurate results, and an important offset betwen the real and predicted points. However, neural networks are able to learn some resemblance of the dynamics of the system.

In order to try to palliate this displacement, we have introduced two modifications to the program. First, we have reduced the batch size to 2048 (as shown in Table 3.1). However the offset (the displacement with respect the location of the central periodic orbit) was still very large. In order to palliate this offset, we calculated the distance between the fixed point (that corresponds to a periodic orbit of the system) $x = -0.507, y = \dot{x} = 0, \dot{y} = -0.8963$ and its prediction. Then, we applied a translation to each point according to this distance, thus removing this offset. Figure 3.6 shows the predictions of the *progressor* with the equidistant dataset, while Figure 3.7 shows the predictions of the *regressor*. We can observe that the predictions are still very imprecise, although the offset has decreased.

3.5.2 Random dataset

Figure 3.8 shows the predictions of the *progressor* over the points of the testing dataset when trained with the random dataset, while Figure 3.9 shows the predictions of the *regressor*. Results obtained when training the models using the random

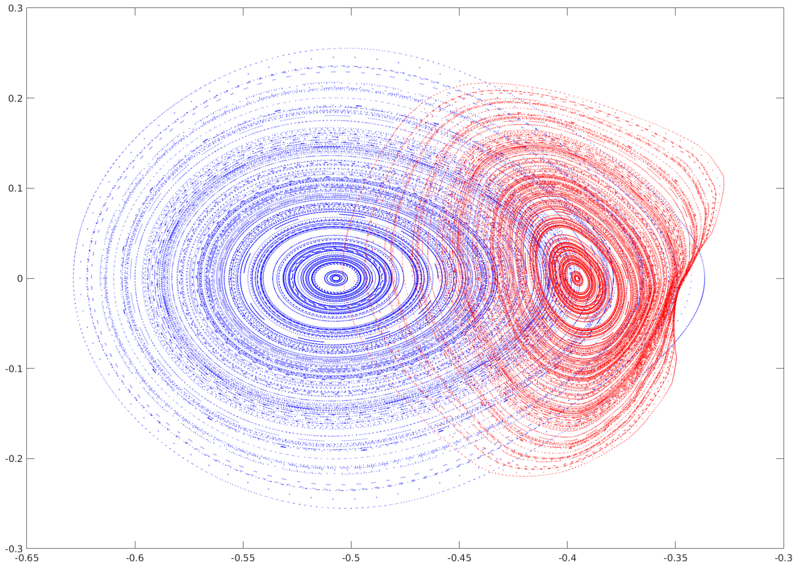


Fig. 3.4: One of the first tests with the *Progressor* over the testing dataset when trained with the equidistant dataset. Markers in blue represent the real (x, y) values of the points, and markers in red represent the predictions calculated by the neural networks.

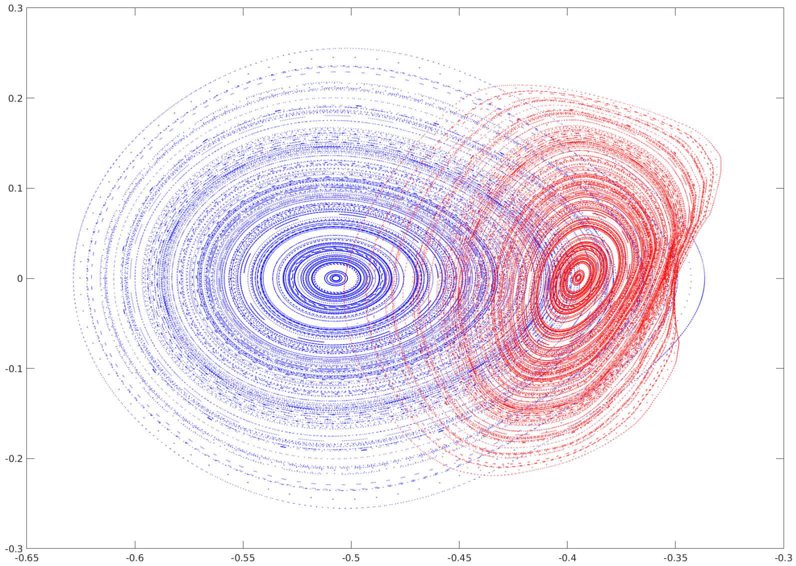


Fig. 3.5: One of the first tests with the *Regressor* over the testing dataset when trained with the equidistant dataset. Markers in blue represent the real (x, y) values of the initial conditions, and markers in red represent the predictions calculated by the neural networks.

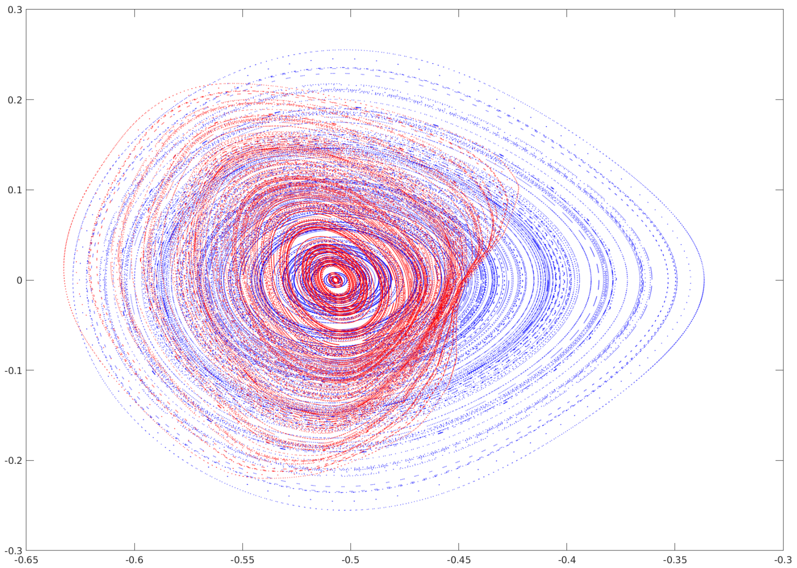


Fig. 3.6: Predictions of the *Progressor* over the testing dataset when trained with the equidistant dataset, after reducing the batch size and correcting the offset. Markers in blue represent the real (x, y) values of the points, and markers in red represent the predictions calculated by the neural networks.

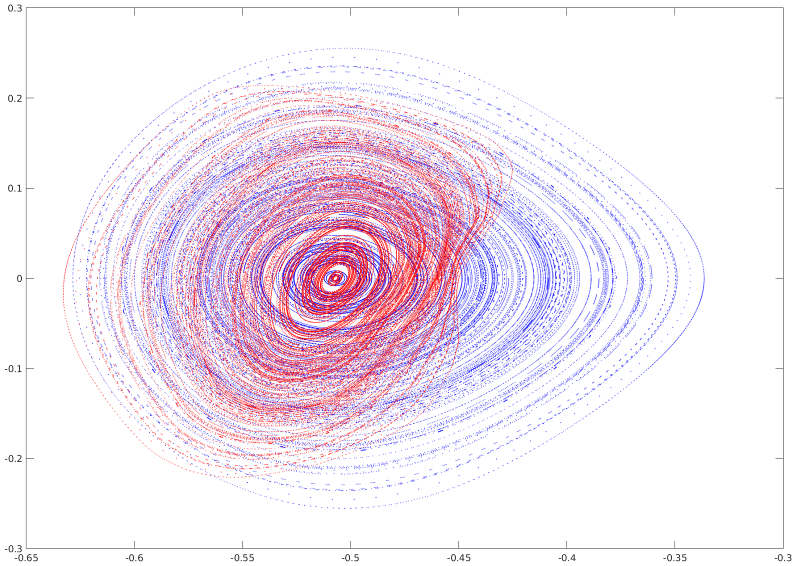


Fig. 3.7: Predictions of the *Regressor* over the testing dataset when trained with the equidistant dataset, after reducing the batch size and correcting the offset. Markers in blue represent the real (x, y) values of the initial conditions, and markers in red represent the predictions calculated by the neural networks.

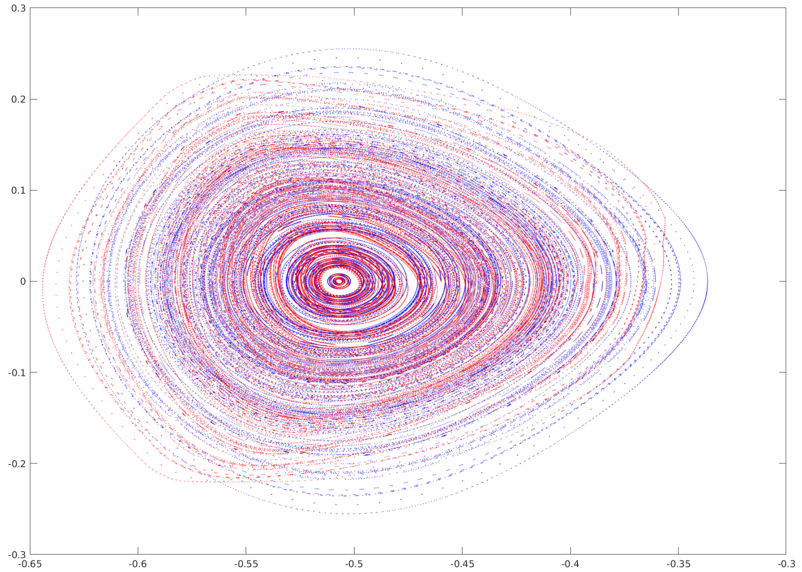


Fig. 3.8: Predictions of the *progressor* over the testing dataset when trained with the random dataset. Markers in blue represent the real (x, y) coordinates of the points, and markers in red represent the predictions calculated by the neural networks.

dataset seem to perform slightly better than the ones shown in the previous section. Still, the predictions remain very inaccurate.

3.5.3 Limited dataset

Figure 3.10 shows the predictions made by the *progressor* over the points of the limited testing dataset when trained with the limited dataset, while Figure 3.11 shows the predictions made by the *regressor* using the same training and test datasets. Results show that the models predict with more accuracy the points closer to the center $(0.5, 0)$, but they quickly lose accuracy as the distance from this center grows.

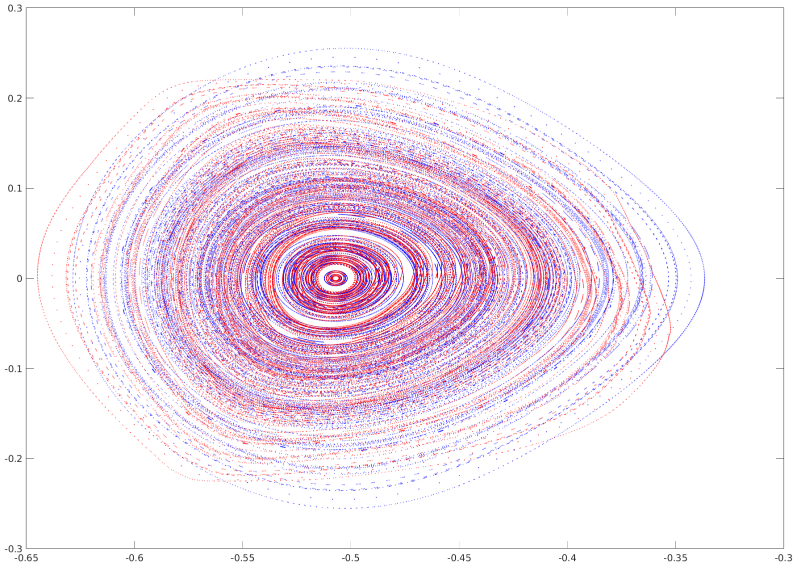


Fig. 3.9: Predictions of the *regressor* over the testing dataset when trained with the random dataset. Markers in blue represent the real (x, y) coordinates of the initial conditions, and markers in red represent the predictions calculated by the neural networks.

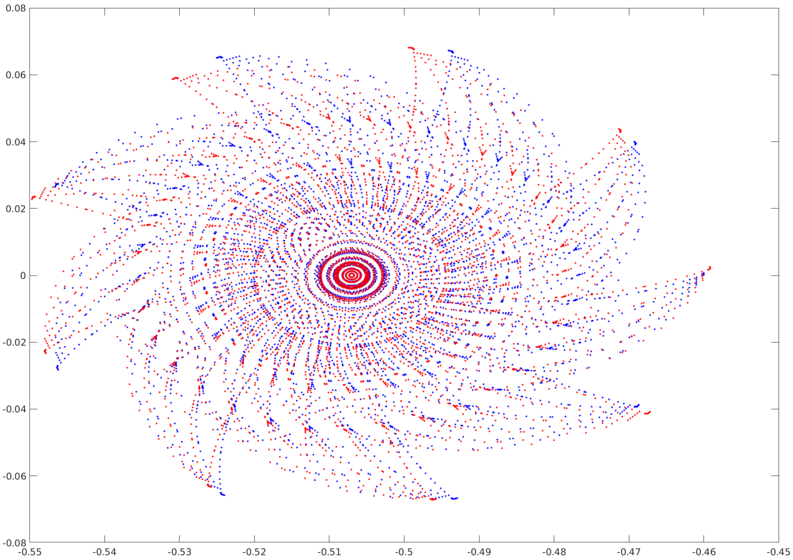


Fig. 3.10: Predictions of the *progressor* over the limited training dataset when trained with the limited dataset. Markers in blue represent the real (x, y) coordinates of the points, and markers in red represent the predictions calculated by the neural networks.

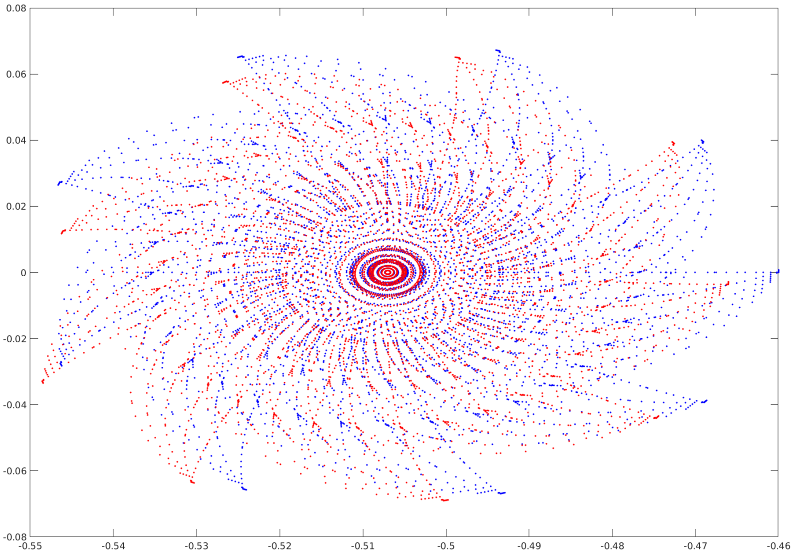


Fig. 3.11: Predictions of the *regressor* over the limited training dataset when trained with the limited dataset. Markers in blue represent the real (x, y) coordinates of the initial conditions, and markers in red represent the predictions calculated by the neural networks.

3.5.4 Conclusions and future work

The predictions of the Poincaré map for the CP problem produced by the models have been very poor and inaccurate, and further work and research is needed in order to try to improve these results. It would seem that ordinary, fully-connected neural networks are unable to accurately represent complex mathematical functions such as the ones underlying in the behaviour of a dynamical system. Therefore, the author thinks the architecture of the ANNs of our models should be rebuilt from scratch using Henon Networks, which have been used successfully to perform calculations and approximations over the Poincaré maps of some particular problems in physics (see [3]).

The author also considers that it would be interesting to try to investigate the reason why the current fully connected neural network produces these far-fetched results and why, despite this fact, it is able to learn and preserve some of the apparent dynamics of the system (such as isles, quasi-periodic orbits, torus, etc).

Bibliography

[1]$^3Cat - 4$ - *NanoSat Lab - UPC*. URL: `https://nanosatlab.upc.edu/en/missions-and-projects/3cat-4`. (accessed: 12.05.2022).

[2]Alberto Abad. *Astrodinámica*. Bubok Publishing, 2012. ISBN: 978-84-686-2857-8.

[3]J. W. Burby, Q. Tang, and R. Maulik. „Fast neural Poincaré maps for toroidal magnetic fields“. In: *arXiv* (Nov. 2020).

[4]Barrabés E., Ollé M., Borondo F., Farrelly D., and Mondelo J. M. „Phase space structure of the hydrogen atom in a circularly polarized microwave field“. In: *Elsevier* (2011).

[5]Canuto E., Novara C., Massotti L., Carlucci D., and C. Perez Montenegro. *Spacecraft Dynamics and Control - The Embedded Model Control Approach*. Butterworth-Heinemann publications, 2018. ISBN: 978-0-08-100700-6.

[6]Rumelhart D. E., Hinton G. E., and Williams R. J. „Learning representations by back-propagating errors“. In: *Nature* 323 (1986), pp. 533–536.

[7]Schulman J., Wolski F., Dhariwal P., Radford A., and Klimov O. „Proximal Policy Optimization Algorithms“. In: *arXiv* (Aug. 2017).

[8]Vinyals O. and Babuschkin I. et al. „Grandmaster level in StarCraft II using multi-agent reinforcement learning“. In: *Nature* 575 (2019), pp. 350–354.

[9]*ode45*. URL: `https://mathworks.com/help/matlab/ref/ode45.html`. (accessed: 24.05.2022).

[10]Bhatia N. P. and Szeg. *Stability Theory of Dynamical Systems*. Cambridge University Press, 2009. ISBN: 978-0-521-48181-0.

[11]Kingma D. P. and Ba J. „ADAM: A method for stochastic optimization“. In: *Nature* 323 (1986), pp. 533–536.

[12]*Proximal Policy Optimization (PPO) Agents*. URL: `https://es.mathworks.com/help/reinforcement-learning/ug/ppo-agents.html`. (accessed: 12.05.2022).

[13]Russell S. and Norvig P. *Inteligencia Artificial, un enfoque moderno (2nd edition)*. Pearson Prentice Hall, 2004. ISBN: 84-205-4003-X.

[14]Sutton R. S. and Barto A. G. *Reinforcement Learning: An Introduction (second edition)*. The MIT Press, 2018. ISBN: 978-0-262-03924-6.

[15]*Uniformly distributed random rotations*. URL: `https://mathworks.com/help/nav/ref/randrot.html`. (accessed: 25.05.2022).

[16]Vedant and Alliston J. et al. „Reinforcement Learning for Spacecraft Attitude control“. In: *70th International Astronautical Congress* (2019).

[17] Ma Z. and Wang Y. et al. „Reinforcement Learning-Based Satellite Attitude Stabilization Method for Non-Cooperative Target Capturing“. In: *MDPI / Sensors* (2018).

List of Figures