Imperial College London

Imperial College London

Faculty of Natural Sciences, Department of Mathematics

Deep Neural Networks for Real-time Trajectory Planning

Author:
Amaury Francou

Supervisor:
Dr Dante Kalise

A report submitted in partial fulfilment of the requirements for the degree

MSc Applied Mathematics

September 9, 2022

The work contained in this thesis is my own work unless otherwise stated.

Abstract

This project addresses the problem of robotic locomotion under the frameworks of deep learning and optimal control theory. We consider the set of controls, expressed as feedback laws, that navigate an unmanned aerial vehicle (UAV) to a given destination. By setting, parameterizing and solving a dynamic optimization problem, we build a synthetic dataset combining current state inputs with their applicable optimal control outputs. Accordingly, we cast a supervised learning problem to approximate an optimal feedback law to be used for general locomotion. Therein, this project is concerned with taking advantage of the versatility of neural networks along with the strong data efficiency provided by the optimal control modeling of the problem.

Acknowledgements

I would like to sincerely thank the following people who have helped me in the accomplishment of this project :

- My supervisor Dr Dante Kalise for his help and guidance throughout the research process
- My family and my parents for their unwavering support throughout my studies
- My valuable friends for their cheerfulness throughout the course

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This manuscript was produced for the partial completion of the MSc Applied Mathematics course, given at Imperial College London. We introduce here the considered problem and the related objectives to pursue. We outline the challenges to be met, review previous related works and present the contributions of this thesis to its related fields. We also present the structure followed in the manuscript.

1.1 Introductory point

As many industrial and physical processes are carried out autonomously, as more and more daily life services make extensive use of algorithms and computerized intelligence, we have globally entered the era of *automation*.

In his remarkable talk at the 2017 Neural Information Processing Systems conference (NeurIPS), Pieter Abbeel presented a video in which a robot was able to perform a wide range of household tasks[1], with a level of skill and efficiency unmatched by all products currently available commercially [1]. The robotic demonstration was reminiscent of science fiction elements and was far from what one could reasonably expect in terms of advances in the related field. By the end of his preliminary talk, the speaker revealed that although the robot used was truly operational hardware-wise, it was fully remote-controlled by a human operator throughout the demonstration. Pieter Abbeel used this presentation to highlight that even though current robots have the required physical capabilities for performing a wide range of operations, they are mainly limited by a lack of *intelligent embedded controls*.

Besides, major advances in the field of *artificial intelligence* and *deep learning* have emerged with algorithms capable of outperforming human operators on an ever-increasing range of tasks [2]. However, said algorithms, that are statistical by nature, usually require large amounts of data and computational power to achieve their objectives. Additionally, they are often usable only for the specific task and framework for which they were designed.

Furthermore, controlling an agent undergoing dynamical effects can be done using a branch of mathematical optimization called *optimal control theory*. This framework proposes a collection of methods and results to compute continuous controls that dynamically optimize a given objective function. In particular, optimal control theory is adapted for describing the behavior of self-controlled robots, providing the real-time actions to be applied on a set of available actuators, in order to optimize a given criteria under a series of constraints.

Given these three observations, it is naturally possible to consider leveraging the inherent performances of deep learning models, along with data provided by the optimal control framework, with the objective of synthesizing an adequate intelligent embedded robotic controller. This approach is designed as a trade-off between having a high efficiency in the use of data and having a good level of automation provided to the robotic agent. In particular, on the one hand, while ad-hoc servoing

[1]The household tasks performed by the robot included, but not limited to, vacuuming with a non-robotic vacuum cleaner, placing dishes in a dishwasher, bringing and uncapping a beer to a user.

makes maximum use of the knowledge and data available on a given problem, it offers minimal autonomous capabilities to the agent. On the other hand, as deep learning methods[2] may provide high levels of automation, they require significant amounts of training data and computational power. Said data may be explored in an ineffective way, for instance laboriously approximating well-known explicit laws.

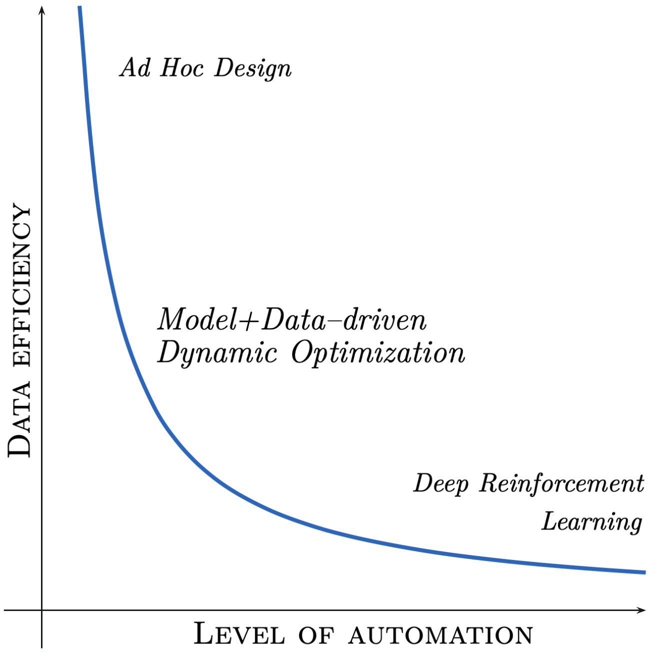


Figure 1.1: Automation level against data efficiency. Reprinted from [3] with permission.

We intend to use physics-informed optimal control laws to effortlessly generate synthetic, yet relevant training data, holding the seeds of optimality. Subsequently, we aim at using deep learning models to unveil consistent patterns inside the generated data, in order to synthesize an optimal controller.

1.2 Considered problem and motivations

We consider the specific case of robotic locomotion in which we seek to build a controller capable of navigating the agent from an initial to a final position. From an optimal control point of view, we examine a *trajectory planning* problem. Under the dynamics of a given model, we compute the controls inducing a trajectory in the related *state space*, while minimizing an objective function. Therein, the optimal control framework builds a *policy* mapping instant states to the relevant applicable controls.

In order to obtain quantitative results and leverage physics-led natural intuitions, we apply our approach to the case of unmanned aerial vehicles (UAVs), in particular focusing on the concrete case of *quadrotor drones*. Such aircrafts are nowadays widely used in a variety of applications such as infrastructure inspection, search and rescue operations or aerial photography. As they are operated through a simple set of actuators, quadrotors offer a perfect framework to study optimal control problems and are therein broadly used in the related literature.

Moreover, quadrotors offer a relevant setting for benchmarking the capabilities of an embedded controller based on deep learning methods. As such aircrafts are often remote-controlled by human operators in current practical applications, our main motivation is to obtain a UAV capable

[2] In particular, when it is possible to access simulation environments, the field of deep reinforcement learning offers state-of-the-art methods to give rise to highly autonomous agents, yet often at the cost of significant computational power.

of autonomous navigation, assessed in a simulated environment. Many new applications could benefit from automated quadrotor drones including, but not limited to, traffic management, aerial delivery, road maintenance or fire watch flights.

Figure 1.2: Walkera QR X350 Quadcopter hovering [4]

We will consider the robotic locomotion problem while optimizing the transportation time and the total energy consumption over the flight. In the optimal control framework this criteria is refereed to as *time-energy optimality*. Said criteria are straightforwardly relevant for the vast majority of autonomous navigation applications.

1.3 Objectives and challenges

We seek to generate relevant data from optimal control theory, and shape the obtained training set according to our specific needs. In particular, we build synthetic data points arising from a theoretical analysis, with the objective of obtaining a specific behavior of the quadrotor drone. Hence, fine-tuning the data is key to the success of the operation. In particular, we require that our model efficiently learns known physics, such as Newton's laws of motion, valid in our non-relativistic setting. Hence, the optimal control model must be well designed to pass the relevant behavior to the trained model.

As we aim using deep learning methods, we intend to generate the training data at large scale. Therefore, the efficiency of the algorithmic procedure for producing data points is a central consideration. Specifically, we require designing a data generation process that is more computationally effective than unsupervised learning methods and that is more time effective than collecting training data from practical points of supply.

While defining our deep learning model, choosing the correct set of predictors and adjusting hyperparameters are key points that should be treated with special care. In particular, identifying the correct predictors plays a central role in the model performance. A good choice of input variables can avoid processing unnecessary data and improve the quality of the predictions that are made. Hyperparameters influence the program's efficiency in terms of computation time and memory size. The main challenge is to produce an algorithm that can be embedded on a UAV and executed on-board in real-time.

Moreover, in order to correctly assess the designed deep learning controller, we require building an adequate simulation environment, in which we can handily investigate the behavior of the UAV. This simulation environment must reproduce the dynamics of the quadrotor and give access to all the observations necessary for an in-depth evaluation of the model.

Optimal control problems can arise with great amounts of complexity. Namely, as the number of equations composing the agent's dynamics may be high, computing the numerical solutions of the partial differential equation expressing the optimality conditions may become intractable. This downfall is known as the *curse of dimensionality.* In particular, we intend to use deep learning methods to bypass the impossibility of finding direct numerical solutions in real-time and on the edge. In the quadrotor navigation setting, the dynamics are complex enough to prevent any explicit optimal planning to be performed using the elements of optimal control theory. Managing this inherent complexity through deep learning approximations is the main challenge faced in this thesis.

1.4 Related work

Among robotic locomotion, the quadrotor setting has been extensively addressed. In particular, thorough analysis of time-optimal control problems applied to UAVs have been made [5, 6]. Energy optimality has been addressed as well in the case of quadrotor navigation [7]. The previous studies usually performed off-line computations of optimal controls. However, non deep learning related online computation methods have been studied with success [8]. Additionally, the quadrotor setting has been broadly considered as a suitable application case for reinforcement learning methods [9, 10], some of which have effectively taken advantage of physics-informed principles [11].

Hard solving of optimal controls in the general case has been addressed in numerous ways. As such controls can be seen as a solution of a specific partial differential equation, sparse grid methods can be applied [12], while still being causality-free [13]. Moreover, the previously presented approaches can make use of variational iterations [14] or be based on pseudospectral methods [15].

Deep learning models have been used to approximate the solutions of optimal control problems in the general case. Said models can involve deep Galerkin methods [16], or be used in the particular setting of stochastic control [17]. However, the use of deep learning algorithm has often been made under some specific assumptions, such as having an affine dependency between dynamics and controls or having a quadratic dependency between cost and controls [18]. Nonlinear model predictive control has also been studied while using deep learning models with specific top-ups such as correction factors [19].

Moreover, artificial intelligence methods have also been broadly studied in the field of robotics. While some approaches make solely use of lighter machine learning models [20], most of intelligent controllers make use of several intricate machine learning and deep learning algorithms [21]. Usually said models are trained using practical points of supply [22], for instance using sets of images to feed vision-based systems [23].

The use of optimal control solutions to train deep learning model for automating vehicle navigation has been attempted several times with success [24], especially for automatic spacecraft guidance [25, 26]. In general, said approaches make use of hard-solved solutions of the associated control problem, in order to generated a suitable training set. Furthermore, approximating optimal controls by using deep learning has been studied in in the general case as well. For instance, in [27], the authors leverage problem physics to favor data efficiency, and describe a method to produce trained models based on optimality conditions. Said trained models are intended to compute optimal controls in real-time.

1.5 Main contributions

We here consider time-energy optimality in the quadrotor setting, as it hasn't been addressed previously. Requiring minimizing both transportation time and energy consumption brings some relevant subtlety to the analysis, as intuitively a trade-off is to be found between two contradictory objectives : moving fast towards the final state while lowering the controls magnitude. In this sense, the model used will need to encompass a higher level of complexity. In addition, the use of deep learning methods trained upon synthetic data, to compute on-board real-time controls, is newly considered for this type of UAVs. Synthetic datasets have been mainly generated on systems with steadier dynamics then what can be encountered in the case of versatile quadrotors. The use of artificial intelligence in said vehicles is usually made through deep reinforcement learning. The algorithms related to this field mainly require solid simulation environments, computational power and a finely tuned reward parameter. We will here apply a usually more direct supervised learning approach on data that has non been arduously harvested from practical points of supply.

Among deep learning approaches, we know of no model trained with optimal control solutions that have been computed using other algorithms than hard-solvers. For instance, the analysis made in [27] relies on firstly hard-solving the optimal controls with a discretization step and the use of collocation formula. This process is highly sensible to the provided initial guesses. Bypassing this obstacle requires employing a time marching trick and a pre-training of the model. In this project, we will leverage the specific framework and physics associated with quadrotor drones to give rise to a straightforward and effortless way of generating the training data controls. In particular, we materialize a latent space of optimal trajectories and, to some extent, open a connection to the field of generative models, by randomly sampling optimal training data. We identify a theoretical asset that we leverage to generate sizeable amounts of data points and use the solvers for comparison purposes only.

1.6 Structure

The project will be divided in 3 main parts.

1. In chapter 2, we will first perform an in-depth analysis of the optimal control framework in the quadrotor context. In particular, we will begin by defining the model and the dynamics at stake. Subsequently, we will derive the mathematical optimality conditions that must be verified by the controls. We will moreover parametrize optimal trajectories using a set of constants, thus giving rise to a latent space of optimal controls.

2. In chapter 3, we will secondly devise an algorithm for efficiently generating a dataset of optimal trajectories based on the theoretical analysis made in chapter 2. We will discuss optimality of the generated data points, present all technicalities of the designed process and provide the implementation details.

3. In chapter 4, we finally compose a suitable deep learning model, adapting all hyperparameters to the specificities of our considered context. In particular we apply regularization techniques, as well as cross-validation methods to enhance the training process. We present the results obtained. The evolution of the loss function characterizing the model's performance is discussed. Thereafter, we present a set of test trajectories automatically followed by the quadrotor navigated using the deep learning feedback controller. These tryouts are carried out in a custom-made simulation environment. The suitability and optimality of the produced closed-loop controls is discussed.

Additionally, the full code and implementations can be found in the appendix.

Chapter 2

Model and optimal control problem

In the following, we define the model that will be used and the optimal control problem that will be addressed along this project. Thereon, we analytically derive the conditions that have to be verified by the optimal controls. We provide a boundary value problem which solutions are optimality candidates for the related control problem.

We present a table of the different abbreviations used in this chapter.

UAV	Unmanned aerial vehicle
OCP	Optimal control problem
HJB	Hamilton-Jacobi-Bellman equation
PMP	Pontryagin's minimum principle
BVP	Boundary value problem

Table 2.1: List of abbreviations used in Chapter 2.

2.1 The planar-quadrotor model

We study the case of the planar-quadrotor model: a 2-dimensional UAV moving in the $\mathbb{R}^2$ plane. This lower-dimensional model arise with simpler dynamics, **yet capturing the essence of the control-motion relationship**, and is consequently extensively studied in the related literature [6, 5, 10, 28, 8]. The generalization to the full 3-dimensional four-rotors aircraft model is straightforward although it comes along with sizeable extra calculations.

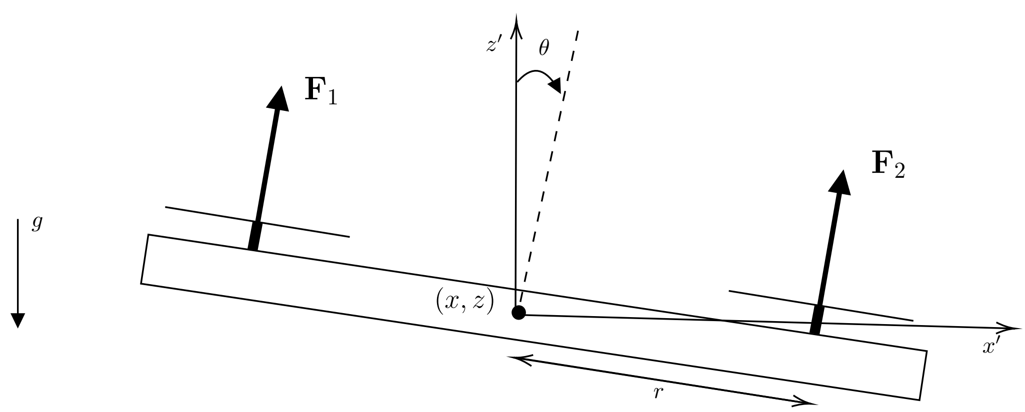


Figure 2.1: The planar-quadrotor model.

As shown in figure 2.1, the planar-quadrotor is driven by 2 vertical propellers, which produce 2 controllable upward thrust forces $\mathbf{F_1}$ and $\mathbf{F_2}$. The center of mass of the vehicle is located at (x, z) in the plane. The angle $\theta \in [0, 2\pi)$ measures the aircraft's tilt with respect to the vertical.

The quadrotor UAV is initially characterized by its state $\mathbf{q} := (x, \dot{x}, z, \dot{z}, \theta, \dot{\theta})^\top$, where the dot refers to the derivative with respect to time. We are concerned with computing time-optimal and energy-optimal point-to-point trajectories for the vehicle. Starting at $t_0 \in \mathbb{R}$ in state $\mathbf{q}(t = t_0) = \mathbf{q}_0$, we aim reaching $\mathbf{q}_f$ in minimal time t_f and minimizing the overall energy consumption of the UAV. Without loss of generality, we take $t_0 = 0$ as the departing time. Defining $F_1 := \|\mathbf{F_1}\|$ and $F_2 := \|\mathbf{F_2}\|$, the dynamics governing the UAV's motion are given by Newton's laws of motion.

$$\begin{cases} m\ddot{x} = (F_1 + F_2)\sin\theta \\ m\ddot{z} = (F_1 + F_2)\cos\theta - mg \\ I\ddot{\theta} = r(F_1 - F_2) \end{cases} \tag{2.1}$$

The parameter I is the given moment of inertia and the parameter r refers to the lever arm.

It is usual to control the current supplying the electric motors powering the propellers via a programmable board. Moreover, as the laws mapping current supply to the magnitude of the thrust forces are well-known, it is natural to consider defining controls directly deriving from the magnitudes F_1 and F_2.

Consequently, we first define $u_T := \frac{F_1+F_2}{m}$ as a control related to thrust and $\tilde{u}_R := \frac{r(F_1-F_2)}{I}$ as a control related to torque.

However, the literature advise us that **we can assume controlling the angular velocity $\dot{\theta}$ directly without dynamical effects and delay**. This simplification does not compromise with the predictability of the model. It remains a good approximation of reality as quadrotors are capable of reaching very high angular accelerations, while having angular velocities limited by sensor technology [11, 6]. Furthermore, it has been observed in a experimental in-lab setting that this assumption produces only minor and tractable discrepancies [5].

Subsequently, we will consider having a rotationnal controller u_R such that $u_R := \dot{\theta}$. Therefore, our general control is set as in the following.

$$\mathbf{u} := (u_T, u_R)^\top \tag{2.2}$$

Hence, we further characterize the quadrotor UAV by the reduced *state vector* given hereby.

$$\mathbf{q} := (x, \dot{x}, z, \dot{z}, \theta)^\top \tag{2.3}$$

Thereon, the controlled dynamics can be defined in the compact following way.

$$\dot{\mathbf{q}} = \boldsymbol{f}(\mathbf{q}, \mathbf{u}), \text{ where } \boldsymbol{f} \text{ is given by } \boldsymbol{f}(\mathbf{q}, \mathbf{u}) := \begin{pmatrix} \dot{x} \\ u_T \sin\theta \\ \dot{z} \\ u_T\cos\theta - g \\ u_R \end{pmatrix} \tag{2.4}$$

The example problem of the planar-quadrotor, although simplified, **arises with 5 equations dynamics and is consequently relevantly subject to the effects of *the curse of dimensionality***. Therein, it constitutes a suitable model to illustrate the advantages of the optimal control approximation method developed in the project.

2.2 Time-energy optimal control problem

We define our optimal control problem (OCP) with the objective of minimizing the transportation time t_f, as well as the overall energy consumption of the vehicle. By definition, this OCP is of *finite horizon*.

The first criteria is directly minimized as a final time penalty. The second criteria is taken into account through the running cost $\mathcal{L}(t, \mathbf{q}, \mathbf{u}) := \|\mathbf{u}\|^2$, as an instant expenditure conditioning our control budget[1]. As the current and the voltage at the input of the electric motors directly regulate the thrust amplitudes, the running cost varies according to the instant electric power supplied to the propellers. Hence, we consider that the magnitude of controls drives the energy flow rate [7]. Furthermore, the designed OCP takes into account the fixed starting state $\mathbf{q}_0$, as well as the expected final reference position $\mathbf{q}_f$.

Furthermore, as the maximum collective thrust is constrained by the size and technical design of the propellers and as the maximum pitch rate is limited by the sensitivity of the on-board sensors, we define a *set of admissible controls*.

$$U := \left\{ \begin{array}{cccc} \mathbf{u} & : & [0, t_f] & \rightarrow & \mathbb{R}^2 \\ & & t & \mapsto & \big(u_T(t), u_R(t)\big) \end{array} \;\middle|\; \underline{u_T} \leq u_T(t) \leq \overline{u_T} \text{ and } |u_R(t)| \leq \overline{u_R} \right\} \tag{2.5}$$

The constants $\overline{u_T}$ and $\underline{u_T}$ refer to maximum and minimum thrust respectively and $\overline{u_R}$ refers to the maximum torque.

Namely, we consider the following *Bolza problem*.

$$\begin{aligned} \min_{\mathbf{u}(\cdot)} \quad & J[\mathbf{u}(\cdot)] := \; t_f + \int_0^{t_f} \|\mathbf{u}\|^2 dt \\ \text{s.t.} \quad & \mathbf{q}(t = t_f) = \mathbf{q}_f, \\ & \mathbf{q}(t = 0) = \mathbf{q}_0, \\ & \dot{\mathbf{q}} = \boldsymbol{f}(\mathbf{q}, \mathbf{u}), \\ & \mathbf{u} \in U, \\ & t \in [0, t_f] \end{aligned} \tag{OCP}$$

Defining $X := \mathbb{R}^4 \times [0, 2\pi)$, the state is here given by a function $\mathbf{q} : [0, t_f] \rightarrow X$, the control by a function $\mathbf{u} \in U \subset \mathcal{F}(\mathbb{R}, \mathbb{R}^2)$, and the dynamics are encoded in a vector field $\boldsymbol{f} : X \times \mathbb{R}^2 \rightarrow X$ which is Lipschitz continuous[2]. Moreover, the cost functional J is here convex[3]. **Note that the method developed in the following of this project generalizes to generic-form Bolza problems under the same regularity assumptions**.

The following table provides the values used in the project for the constants of the model.

m	0.792 kg
r	0.145 m
I	0.04 kg.m^2
g	9.81 m.s^{-2}
$\overline{u_T}$	14 m.s^{-2}
$\underline{u_T}$	0.8 m.s^{-2}
$\overline{u_R}$	4.6 rad.s^{-1}

Table 2.2: Constants of the model used in numerical computations. Quadrotor constants are based on the Walkera QR X350 Quadcopter technical sheet.

[1]The euclidean norm is used here: $\|\mathbf{u}\|^2 = \mathbf{u}^\top \mathbf{u}$.
[2]The total derivative is bounded.
[3]The squared euclidean norm is convex and the integral is linear.

2.3 Optimality conditions

We consider the optimality conditions that the control $\mathbf{u} \in U$ must verify to minimize the cost functionnal J. The process of control can be either *open loop* or *closed loop*.

Open loop control solves the (OCP) equation solely based on the initial state $\mathbf{q}_0$. An optimal open loop control is in the form $\mathbf{u}^\star = \mathbf{u}^\star(t, \mathbf{q}_0)$, the instant output state having no effect on the upcoming control operation.

In practical real-time applications, feedback-based controls are more consistent and commonly used. Such closed loop controls allow the system to maintain a given level of accuracy, stabilize dynamic processes and automate disturbance management. An optimal closed loop control is in the form $\mathbf{u}^\star = \mathbf{u}^\star(t, \mathbf{q})$, the instant output state being used to condition the upcoming control operation. In this project we are concerned with closed loop controls which can be computed instantly on-board and at real-time $t \in [0, t_f]$, based on a measurement of the state $\mathbf{q}(t) \in X$.

2.3.1 Hamilton-Jacobi-Bellman equation

We seek to compute the optimal closed loop feedback control $\mathbf{u}^\star = \mathbf{u}^\star(t, \mathbf{q})$. We undertake the standard procedure [29, 27], defining the *value function* $V : [0, t_f] \times X \to \mathbb{R}$, which represents the *optimal cost-to-go* of the (OCP), starting in state $\mathbf{q}$ at time t on-wards.

$$V(t, \mathbf{q}) := \begin{cases} \inf\limits_{\mathbf{u}(\cdot) \in U} & \displaystyle\int_t^{t_f} 1 + \|\mathbf{u}\|^2 \, d\tau \\ \text{s.t.} & \tilde{\mathbf{q}}(\tau = t) = \mathbf{q}, \ \tilde{\mathbf{q}}(\tau = t_f) = \mathbf{q}_f \\ & \dot{\tilde{\mathbf{q}}} = \boldsymbol{f}(\tilde{\mathbf{q}}, \mathbf{u}), \ \tau \in [t, t_f] \end{cases} \tag{2.6}$$

The value function (2.6) is shown to be the unique *viscosity solution*[4] of the Hamilton-Jacobi-Bellman partial differential equation [30].

$$\begin{cases} -\dfrac{\partial V}{\partial t}(t, \mathbf{q}) - \min\limits_{\mathbf{u}(\cdot) \in U} \left(1 + \|\mathbf{u}\|^2 + \left[\dfrac{\partial V}{\partial \mathbf{q}}(t, \mathbf{q}) \right]^\top \boldsymbol{f}(\mathbf{q}, \mathbf{u}) \right) = 0 \\ V(t_f, \mathbf{q}) = 0 \end{cases} \tag{HJB}$$

We introduce a *costate vector* $\boldsymbol{\lambda} : [0, t_f] \to \mathbb{R}^5$, and build the corresponding *Hamiltonian*.

$$H(t, \mathbf{q}, \boldsymbol{\lambda}, \mathbf{u}) := 1 + \|\mathbf{u}\|^2 + \boldsymbol{\lambda}^\top \boldsymbol{f}(\mathbf{q}, \mathbf{u}) \tag{2.7}$$

We have that **the optimal control minimizes the Hamiltonian**.

$$\mathbf{u}^\star = \mathbf{u}^\star(t, \mathbf{q}, \boldsymbol{\lambda}) = \arg \min_{\mathbf{u}(\cdot) \in U} H(t, \mathbf{q}, \boldsymbol{\lambda}, \mathbf{u}) \tag{2.8}$$

By defining the minimal Hamiltonian $H^\star(t, \mathbf{q}, \boldsymbol{\lambda}) := H(t, \mathbf{q}, \boldsymbol{\lambda}, \mathbf{u}^\star)$, (HJB) rewrites as in the following.

$$\begin{cases} -\dfrac{\partial V}{\partial t}(t, \mathbf{q}) - H^\star \left(t, \mathbf{q}, \dfrac{\partial V}{\partial \mathbf{q}} \right) = 0 \\ V(t_f, \mathbf{q}) = 0 \end{cases} \tag{HJB$\star$}$$

[4]Moreover, if V is C^2, it is the unique classical solution of HJB.

Obtaining the optimal control can be made by computing the viscosity solution of the (HJB) equation, as it yields necessary and sufficient optimality conditions. With a computed solution V of (HJB), by setting $\boldsymbol{\lambda} = \partial V/\partial \mathbf{q}$, the optimal control is then obtained by minimizing the Hamiltonian: $\mathbf{u}^\star = \arg \min_{\mathbf{u}(\cdot)\in U} H(t, \mathbf{q}, \partial V/\partial \mathbf{q}, \mathbf{u})$.

Although methods to solve (HJB) directly have been developed [31, 13, 14, 12], we focus on a strongly related and commonly used procedure that provides necessary conditions for optimality. In particular, the *characteristics* of the solutions of the (HJB) equation verify a specific boundary value problem (BVP) : *Pontryagin's minimum principle.*

2.3.2 Pontryagin's minimum principle

An optimal feedback control $\mathbf{u}^\star = \mathbf{u}^\star(t, \mathbf{q}, \boldsymbol{\lambda})$ verifies the following Pontryagin's minimum principle equations.

$$\begin{cases} \mathbf{u}^\star = \arg \min_{\mathbf{u}(\cdot)\in U} H(t, \mathbf{q}, \boldsymbol{\lambda}, \mathbf{u}) \\ \dot{\mathbf{q}} = \frac{\partial H}{\partial \boldsymbol{\lambda}} = \boldsymbol{f}(\mathbf{q}, \mathbf{u}^\star), \quad \mathbf{q}(0) = \mathbf{q}_0, \quad \mathbf{q}(t_f) = \mathbf{q}_f \\ -\dot{\boldsymbol{\lambda}}(t) = \frac{\partial H}{\partial \mathbf{q}}\Big(t, \mathbf{q}, \boldsymbol{\lambda}, \mathbf{u}^\star(t, \mathbf{q}, \boldsymbol{\lambda})\Big) \qquad \text{(adjoint equation)} \end{cases} \tag{PMP}$$

The (PMP) **boundary value problem provides a necessary condition for optimality**. Even though the characteristics of the value function (2.6) verify the (PMP) equations, the solution may not be unique. Hence, computed solutions of (PMP) may be sub-optimal. For instance, sufficiency of (PMP) for optimality may be obtained globally under convexity assumptions [32], or locally near an equilibrium point [33]. Notwithstanding the absence of sufficiency in our current setting, **solutions of** (PMP) **remain strong optimality candidates and will be considered optimal in the following of the project**.

Noting $\boldsymbol{\lambda} = (\lambda_1, \lambda_2, \lambda_3, \lambda_4, \lambda_5)^\top$, we specify the Hamiltonian (2.7) in our considered setting.

$$\begin{aligned} H(t, \mathbf{q}, \boldsymbol{\lambda}, \mathbf{u}) &= 1 + u_T^2 + u_R^2 + \lambda_1 \dot{x} + \lambda_2 u_T \sin\theta + \lambda_3 \dot{z} + \lambda_4 (u_T \cos\theta - g) + \lambda_5 u_R \\ &= 1 + \lambda_1 \dot{x} + \lambda_3 \dot{z} - \lambda_4 g + \Big(\lambda_5 u_R + u_R^2\Big) + \Big((\lambda_2 \sin\theta + \lambda_4 \cos\theta) u_T + u_T^2\Big) \end{aligned} \tag{2.9}$$

The adjoint equation of (PMP) gives us the following conditions on the costate variables.

$$\begin{cases} \dot{\lambda}_1 = 0 \Rightarrow \lambda_1 = c_1 \in \mathbb{R} \\ \dot{\lambda}_2 = -\lambda_1 \Rightarrow \lambda_2 = c_2 - c_1 t,\ c_2 \in \mathbb{R} \\ \dot{\lambda}_3 = 0 \Rightarrow \lambda_3 = c_3 \in \mathbb{R} \\ \dot{\lambda}_4 = -\lambda_3 \Rightarrow \lambda_4 = c_4 - c_3 t,\ c_4 \in \mathbb{R} \\ \dot{\lambda}_5 = -\lambda_2 u_T \cos\theta + \lambda_4 u_T \sin\theta \end{cases} \tag{2.10}$$

As (PMP) requires that the optimal set of controls minimizes the Hamiltonian, and as the two controls appear in separate terms in (2.9), the minimization can be held separately.

We first consider the thrust control.

$$\text{For all } t \in [0, t_f],\ u_T^\star(t) = \arg \min_{u_T(t)\in[\underline{u_T}, \overline{u_T}]} \Big((\lambda_2 \sin\theta + \lambda_4 \cos\theta) u_T + u_T^2\Big) \tag{2.11}$$

The second order polynomial in u_T directly gives us the desired minimizing value, regulated by a function $a : [0, t_f] \to \mathbb{R}$.

$$\text{For all } t \in [0, t_f],\ a(t) := -\frac{\lambda_2 \sin\theta + \lambda_4 \cos\theta}{2} = -\frac{(c_2 - c_1 t)\sin\theta + (c_4 - c_3 t)\cos\theta}{2} \tag{2.12}$$

$$\text{For all } t \in [0, t_f],\ u_T^\star(t) = \left\{\begin{array}{ll} \overline{u_T} & \text{if } a(t) \geq \overline{u_T} \\ \underline{u_T} & \text{if } a(t) \leq \underline{u_T} \\ a(t) & \text{otherwise} \end{array}\right\} = \min\Big(\max\big(a(t),\, \underline{u_T}\big),\, \overline{u_T}\Big) \tag{2.13}$$

The obtained thrust control is not *bang-bang*, nor *bang-singular* as what can be encountered usually in the related literature [6, 34]. Instead, we here have a **regulating function undergoing saturation effects**, arising from the (PMP) conditions.

We secondly consider the rotational control.

$$\text{For all } t \in [0, t_f],\ u_R^\star(t) = \arg \min_{u_R(t) \in [-\overline{u_R}, \overline{u_R}]} \Big(\lambda_5 u_R + u_R^2\Big) \tag{2.14}$$

As previously, the second order polynomial in u_R directly gives us the desired minimizing value, regulated by another function $b : [0, t_f] \to \mathbb{R}$.

$$\text{For all } t \in [0, t_f],\ b(t) := -\frac{\lambda_5(t)}{2} \tag{2.15}$$

$$\text{For all } t \in [0, t_f],\ u_R^\star(t) = \left\{\begin{array}{ll} \overline{u_R} & \text{if } b(t) \geq \overline{u_R} \\ -\overline{u_R} & \text{if } b(t) \leq -\overline{u_R} \\ b(t) & \text{otherwise} \end{array}\right\} = \min\Big(\max\big(b(t),\, -\overline{u_R}\big),\, \overline{u_R}\Big) \tag{2.16}$$

The computed rotational control is also based on a regulating function that is subject to saturation. As we see in (2.15), the definition of b is not explicit as it depends on the costate variable λ_5, which is itself governed by a differential equation found in the adjoint equation (2.10). In particular, to fully determine λ_5 as an initial value problem, we need to determine $\lambda_5(0)$. At first, we recall that this value cannot be chosen freely as **the Hamiltonian** (2.7) **must be zero along all optimal trajectories since the terminal time is free** [35]. However, as a matter of simplification, we will require $u_R(0) = 0$ as an additional constraint defining the admissible controls.

Subsequently, given the characterisation of $u_R^\star$ in (2.16) and assuming $|\overline{u_R}| > 0$, we require $b(0) = 0$. By differentiating b with respect to time and using the last component of the adjoint equation, we set the requested initial value problem.

$$\forall t \in [0, t_f]\ \ \dot{b}(t) = \left(\frac{\big(c_2 - c_1 t\big)\,\cos\theta - \big(c_4 - c_3 t\big)\,\sin\theta}{2}\right) u_T(t)\,, \quad b(0) = 0 \tag{2.17}$$

As in [6], we build an *augmented system* using a vector $\mathbf{y} := (x, \dot{x}, z, \dot{z}, \theta, b)^\top$. We solve the related augmented initial value problem to generate an appropriate trajectory. The state components are computed according to the dynamics given by (2.4). The controls u_T and u_R are computed using (2.13) and (2.16) respectively.

$$\dot{\mathbf{y}} = \boldsymbol{g}\left(\mathbf{y}, \mathbf{u}\right) := \begin{pmatrix} \dot{x} \\ u_T \sin\theta \\ \dot{z} \\ u_T \cos\theta - g \\ u_R \\ \left(\frac{(c_2 - c_1 t)\ \cos\theta - (c_4 - c_3 t)\ \sin\theta}{2}\right) u_T \end{pmatrix}, \quad \mathbf{y}(0) = (\mathbf{q}_0,\, 0)^\top \tag{2.18}$$

We define the vector $\mathbf{c} := (c_1, c_2, c_3, c_4)^\top$. Notice that any choice of said constants fully determines a trajectory. The augmented system (2.18) has been derived using the adjoint equation of (PMP). Nonetheless, it is still to find $\mathbf{c}$ such that $\mathbf{q}(t_f) = \mathbf{q}_f$ and such that the computed control $\mathbf{u}$ minimizes the Hamiltonian. We now have all the theoretical building blocks to proceed with numerical solving of optimal trajectories and controls.

Chapter 3

Synthetic dataset generation

We now define a method for generating a dataset of near-optimal trajectories and controls based on the analysis made in Chapter 2. We devise an algorithm for efficiently computing requested data points at large scale. Moreover, we specify the implementation details and examine a set of example trajectories. Additionally, we compare our custom generated trajectories with solver-based ones.

We present a table of the different abbreviations used in this chapter.

IVP	Initial value problem
NLP	Nonlinear programming
ICLOCS2	Imperial College London optimal control software 2
IPOPT	Interior point optimizer

Table 3.1: List of abbreviations used in Chapter 3.

3.1 A brief introduction to supervised learning

We aim at synthesizing an optimal closed loop control $\mathbf{u}^\star = \mathbf{u}^\star(t, \mathbf{q})$, such that the considered quadrotor navigates to the desired final state $\mathbf{q}_f$, minimizing the functional J defined in the (OCP). Our objective is to approximate said closed loop control via a supervised learning method.

3.1.1 Definitions

In a supervised learning scheme, we are given a dataset of *predictor* input variables, along with their corresponding *outcome* output variables, in the form of M pairs $\{(\check{x}_{(i)},\, \check{y}_{(i)})\}_{i=1}^{M}$. The goal is to approximate the relationship between predictors and outcomes via a model function $\boldsymbol{\pi} : \check{X} \to \check{Y}$, that minimizes a given loss $L(\boldsymbol{\pi}(\check{x}),\, \check{y})$, where $L : \check{X} \times \check{Y} \to \mathbb{R}$. Therein, said *loss function* computes the error made between the model's *prediction* and the considered *ground truth*. The loss function is minimized by evaluating the data points in a subset of the available dataset which is called the *training set*. We denote N the number of training points ($N < M$). Specifically, the *mean sample loss*, which is the mean of all losses across the training set, is minimized.

$$E(L) = \frac{1}{N} \sum_{i=1}^{N} L(\boldsymbol{\pi}(\check{x}_{(i)}),\, \check{y}_{(i)}) \tag{3.1}$$

This optimization process is made by varying the parameters of the model function $\boldsymbol{\pi}$. The final objective is to predict outcomes from previously unseen inputs. Hence, preventing the model function from *overfitting* the training data is key. Consequently, the supervised learning process is performed monitoring an *expected out-of-sample loss*, which ideally remains small. This mean sample loss is computed on a on a different subset of the dataset called the *validation set*. This procedure is intended to ensure *generalisability* : obtaining a trained model function that can process unseen data effectively.

3.1.2 The quadrotor setting : using synthetic data points

Our goal is to build a model function $\boldsymbol{\pi} : X \to U$, that matches a measured state to the applicable instant controls for bringing the quadrotor to the desired final state $\mathbf{q}_f$. Hence, the predictor input variable needs to involve both said current measured state $\mathbf{q}(t)$, as well as the desired final state to meet. We design our predictor as the difference between the current and final states.

$$\hat{\mathbf{q}}(t) := \mathbf{q}_f - \mathbf{q}(t), \text{ with } \hat{\theta}(t) \equiv \theta_f - \theta(t) \mod 2\pi \tag{3.2}$$

Furthermore, the outcome is consistently the instant control vector $\mathbf{u}(t) = (u_T(t), u_R(t))^\top$. As said outcome is continuous[1], we are concerned with a *regression* task. Thus, the intended dataset is designed as the difference of states and control pairs.

$$\mathcal{D} := \left\{ \left(\hat{\mathbf{q}}_{(i)}, \mathbf{u}_{(i)} \right) \right\}_{i=1}^{M} \subset X \times U \tag{3.3}$$

The training set is denoted $\mathcal{T} \subset \mathcal{D}$, with $|\mathcal{T}| = N$. The validation set is denoted $\mathcal{V} \subset \mathcal{D}$, with $|\mathcal{V}| = K$. Our minimization is made using the *mean squared error*.

$$MSE_{\boldsymbol{\pi}} := \frac{1}{N} \sum_{i=1}^{N} \left(\boldsymbol{\pi}\left(\hat{\mathbf{q}}_{(i)}\right) - \mathbf{u}_{(i)} \right)^2 \tag{3.4}$$

Whilst supervised training is often conducted using data collected from experimental or practical points of supply, in the considered quadrotor setting we aim to use trajectories and controls arising from the theoretical analysis of optimality conditions. In this sense, we intend building a *synthetic dataset* of states and control pairs, as given by the analysis made in Chapter 2.

Among the advantages of this approach, as the training set is synthetic, it is possible to access a virtually unlimited supply of data being solely limited by the time and computational power available. Moreover, it is possible to shape the data according to the user's needs, for example varying its statistical distribution in order to obtain a specific behavior of the model function.

3.2 Generating trajectories from constants

To generate the set $\mathcal{D}$, we need a range of optimal trajectories and controls. As seen in Chapter 2, optimal controls are given by (2.13) and (2.16). Moreover, optimal trajectories are computed solving the initial value problem defined in (2.18). Thereon, for any given initial state $\mathbf{q}_0$ and any chosen time horizon T, we have that any choice of four constants $c_1, c_2, c_3, c_4 \in \mathbb{R}^4$ fully determines a set of controls and its corresponding trajectory on $[0, T]$. **Thus, there exists a specific subset of $\mathbb{R}^4$ that acts as a *latent space* of optimal trajectories**. In particular, each optimal trajectory has a compressed representation in this lower dimensional space and is located by a four constants address. From this, it remains necessary to investigate said latent space and characterize the associated trajectories.

[1] As opposed to categorical outcomes.

3.2.1 Discussing eligibility and optimality

As the structure of the latent space remains largely unknown, as a first approach we consider the characteristics of trajectories generated by any quadruplets of $\mathbb{R}^4$. We examine the relationship that maps the constants vector $\mathbf{c}$ to the final state $\mathbf{q}_f$, for a fixed time horizon T.

For instance, consider the quadrotor in the initial state $\mathbf{q}_0 = \mathbf{0}$, i.e. initially positioned at $(0,0)$ and at rest. We require having the quadrotor in the final state $\mathbf{q}_f = (0,0,1,0,0)^\top$, i.e. positioned at $(0,1)$ and at rest. A possible trajectory is the straight vertical line joining $z=0$ to $z=1$. In this case, as the center of mass of the quadrotor follows said straight line, the tilt angle remains zero during all the manoeuvre. Subsequently, the angular acceleration vanishes and we have $u_R = 0$ at all times. Consequently, as $\overline{u_R} > 0$, $\dot{b}(t) = (c_2 - c_1 t)/2 = 0$ for all $t > 0$, which give us that $c_1 = c_2 = 0$ for this specific trajectory. In particular, given this strong condition and in the context of uniformly sampled constants over $\mathbb{R}^4$, this precise situation is atomless and occurs with probability zero.

Subsequently, we examine the characteristics and optimality of randomly generated trajectories, obtained by sampling a random vector $\mathbf{c}$ and solving the IVP (2.18). In this context, to perform the analysis we consider comparing said generated trajectory with a corresponding solver-based one.

3.2.2 Interior point optimizer method

To solve optimal controls and trajectories, we use the Imperial College London optimal control software 2 (ICLOCS2) [36]. This MATLAB package uses transcription and discretization methods to transpose the considered optimal control problem into series of nonlinear programming subproblems (NLP). It offers a modelling environment to handle our defined (OCP).

In particular, we use ICLOCS2 along with an interior point optimizer method (IPOPT) [37]. This NLP solver carries out the primal-dual interior point method proposed in [38]. The method minimizes a nonlinear objective function under inequality constraints. Specifically, it converges to the minimum of an associated unconstrained objective function, which is the original objective function to which is subtracted the scaled sum of the logarithms of the several constraint functions. This associated unconstrained objective function is referred to as the *logarithmic barrier function*. By computing the gradient of said logarithmic barrier function, adding a dual variable to the constraint functions and applying Newton's method, an iterative update equation is found. The update is performed until convergence.

Due to the time and computational power required, an IPOPT-based solver does not constitute a feasible solution to compute optimal controls in real-time and on board the quadrotor. Hence, the model function $\boldsymbol{\pi}$, approximating the optimal closed loop control, will be pre-trained and designed so that it can perform predictions time-efficiently and requiring a reasonable amount of computational power. In this work, the use of the ICLOCS2 solver will be mainly limited to data exploration and model validation.

3.2.3 Example of a generated quadrotor trajectory

We generate a sizeable amount of trajectories and compare the obtained controls with the corresponding solver-based ones. We present a specific example trajectory computed via a random draw of a vector $\mathbf{c} \in \mathbb{R}^4$ and an initial state $\mathbf{q}_0 \in X$. We set a time horizon $T = 2$ seconds.

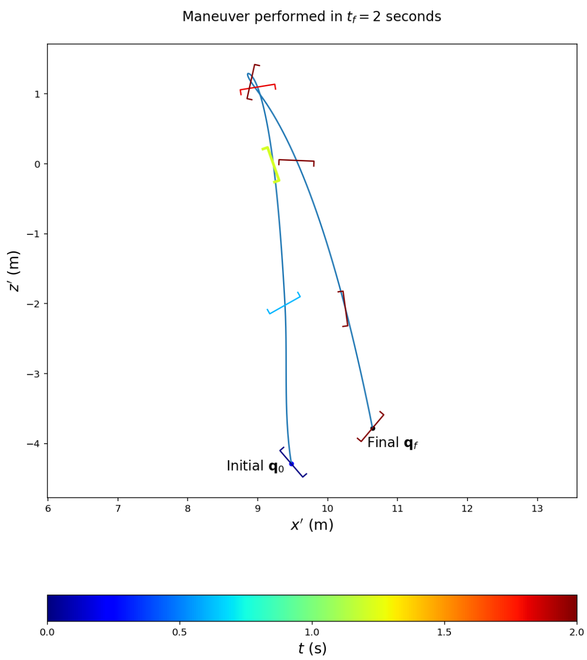
$\mathbf{q}_0$	$(9.48,\ -1.12,\ -4.29,\ 7.14,\ 0.86)^\top$
$\mathbf{c}$	$(107.90,\ -199.84,\ 273.42,\ -75.36)^\top$
T	2 s

Table 3.2: Parameters used for the example trajectory generation.

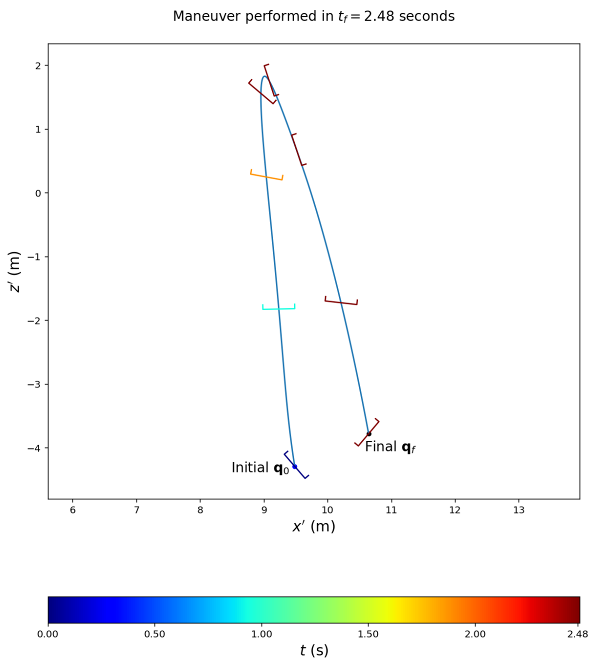
We denote $\tilde{\mathbf{q}} := (x, \dot{x}, z, \dot{z})^\top$ the position and velocity state coordinates. We have $\mathbf{q} = (\tilde{\mathbf{q}}, \theta)^\top$. The parameters have been drawn following normal and uniform distributions : $\tilde{\mathbf{q}}_0 \sim \mathcal{N}(\tilde{\mu},\, \tilde{\sigma}^2\,\mathbf{I})$, $\theta(0) \sim \mathcal{U}([0, 2\pi))$ and $\mathbf{c} \sim \mathcal{N}(\mu_c,\, \sigma_c^2\,\mathbf{I})$, with $\tilde{\mu} = 0,\ \tilde{\sigma}^2 = 16,\ \mu_c = 0,\ \sigma_c^2 = 40\,000$.

We generate the trajectory solving the IVP (2.18) by using a Runge-Kutta method of order 4. The final state obtained is $\mathbf{q}(t_f) = \mathbf{q}(T) = \big(10.64,\, 1.87,\, -3.78,\, -10.12,\, -0.87\big)^\top$. We also use the ICLOCS2 package to compute the solver-based trajectory going from $\mathbf{q}_0$ to $\mathbf{q}_f$, while minimizing the functional J, defined in the (OCP).

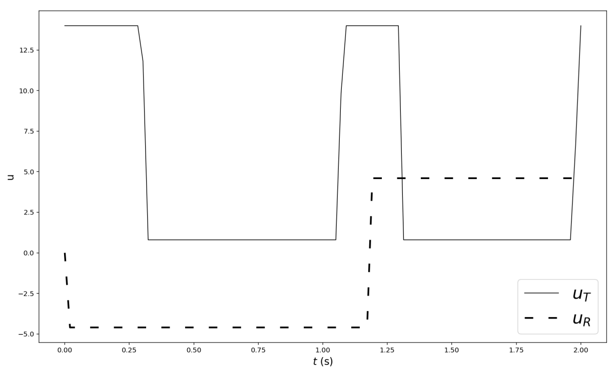
The MATLAB code used for computing the trajectory via the ICLOCS2 package can be found in the appendix A.



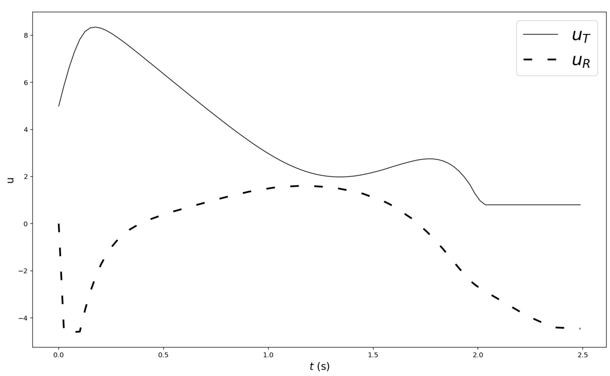
(a) Theoretically generated trajectory



(b) Solver-based trajectory



(c) Theoretically generated controls



(d) Solver-based controls

Figure 3.1: Quadrotor trajectory starting at $\mathbf{q}_0 = \big(9.48,\, -1.12,\, -4.29,\, 7.14,\, 0.86\big)^\top$, targeting final state $\mathbf{q}_f = \big(10.64,\, 1.87,\, -3.78,\, -10.12,\, -0.87\big)^\top$. Both theoretically generated and solver-based trajectories and controls are presented.

The simulations presented in figure 3.1a and figure 3.1b describe the quadrotor's trajectory in the (x', z') plane. The custom simulator has been freely inspired from figures presented in [5]. The vehicle is represented by a bracket which right angle edges are directed in the same direction as the thrust forces. The bracket color refers to the time at which the quadrotor is printed on the trajectory. The color-time equivalence is given by the attached colormap.

In this trajectory, the given initial and final states are rather close position-wise. However, although the UAV starts with a significant ascending instant vertical velocity ($\dot{z}(0) = 7.14$), it is required to arrive at its final position with a large descending instant vertical speed ($\dot{z}(t_f) = -10.12$). To perform this task, both the theoretical and solver-based trajectories draw a bell-shape, leveraging gravitational effects to absorb their initial ascending speed and gain the desired final descending velocity.

The trajectories differ from their tilt angle variation. The theory-based trajectory is sharper, with a higher applied tilt rate stopping the vehicle at its apex. The solver-based trajectory is smoother, letting the quadrotor glide down towards the final position. The controls differ accordingly : the theoretical controls are mainly bang-bang while the solver controls remain primarily in singular arcs.

We continue the analysis by defining the energy contribution to the cost functional.

$$E := \int_0^{t_f} \|\mathbf{u}\|^2 dt \tag{3.5}$$

Thereon, we can compare the optimality of the two trajectories. The energy contribution is numerically computed using a Simpson method.

	Theoretically generated	Solver-based
t_f	**2.00**	2.48
E	65.42	**57.36**
J	67.42	**59.84**

Table 3.3: Cost functionals and contributions associated with the example trajectory.

Although the theoretically generated trajectory and controls are slightly more time-optimal, we observe that the energy consumption is lower for the solver-based controls. On end, the solver-based trajectory is more optimal regarding the cost functional J, as the theoretically generated trajectory is roughly 12.5% higher.

Consequently, we have that the latent space of optimal trajectories is a strict subset of $\mathbb{R}^4$. Notwithstanding this observation, most of randomly sampled theoretical trajectories arise being near-optimal, in comparison with the solver-based corresponding ones. Therefore, **trajectories generated from any $\mathbf{c} \in \mathbb{R}^4$ will be considered as optimal for the following of this project**.

3.3 Ensuring data representivity

From the exploration of randomly generated trajectories, we observe that sampling the constants $c_1, c_2, c_3, c_4 \in \mathbb{R}$ from a normal distribution results in obtaining comparatively similar routes between $\mathbf{q}_0$ and $\mathbf{q}_f$. To ensure the generalisability of the model $\boldsymbol{\pi}$, we require a training set that maintains *data representivity.* In particular, we require having a variance within the training data that is reasonably close to what can be encountered in the unseen data to be subsequently processed by the model function.

3.3.1 Sampling initial states and final times

In the same manner as the example trajectory presented in section 3.2.3, we draw the initial states by processing $\tilde{\mathbf{q}}_0$ and $\theta(0)$ separately.

Since the predictor variables are built subtracting the final and current states, as we can control the admissible final state $\mathbf{q}_f$, the absolute value of the initial position and velocity is not significant. Since the dataset used is synthetic, we can shape the data arbitrarily. In particular, we are mainly concerned by the relative position between $\mathbf{q}_0$ and $\mathbf{q}_f$ and can discard any final states that are not verifying some set of conditions. Hence, the initial state will be sampled using a multivariate normal distribution with zero covariance between variables.

The initial tilt angle will be sampled with a higher probability for having the quadrotor upward facing at the initial time[2]. Precisely, we define $\mathbb{P}_\theta : \mathcal{P}\big([0, 2\pi)\big) \to [0, 1]$, such that $\mathbb{P}_\theta\big([0, \frac{\pi}{2}) \cup [\frac{3\pi}{2}, 2\pi)\big) = 0.65$ and $\mathbb{P}_\theta\big([\frac{\pi}{2}, \frac{3\pi}{2})\big) = 0.35$.

$$\mathcal{D} \text{ is generated sampling } \mathbf{q}_0 \text{ with } \tilde{\mathbf{q}}_0 \sim \mathcal{N}\big(\tilde{\mu},\, \tilde{\sigma}^2\, \mathbf{I}\big) \text{ and } \theta(0) \sim \mathbb{P}_\theta \tag{3.6}$$

While building the dataset, we fix the final time t_f beforehand. Said final time will be sampled using a log-normal distribution[3].

$$\mathcal{D} \text{ is generated with } t_f \sim \ln\mathcal{N}\big(\mu_t,\, \sigma_t^2\big) \tag{3.7}$$

In particular, we aim at having a mean maneuver duration close to 2 seconds. This necessary choice may constrain the range of use of the trained model and must be made by the user according to the specificity of his problem context.

3.3.2 Evenly distributed relative final positions

We sample the generating constants $c_1, c_2, c_3, c_4 \in \mathbb{R}$ from a multivariate normal distribution with zero covariance and perform a relevant post-processing on the obtained trajectories.

$$\mathcal{D} \text{ is generated with } \mathbf{c} \sim \mathcal{N}\big(\mu_c,\, \sigma_c^2\, \mathbf{I}\big) \tag{3.8}$$

In the case of a constant vector $\mathbf{c}$ drawn from a normal distribution, among the final states $\mathbf{q}_f$ collected from the related generated trajectory, a significant number of final positions (x_f, z_f) are located at a lower altitude relatively to their corresponding initial position, i.e. $z_f < z(0)$. This situation arises from the characteristics of the latent space structure.

To ensure the representivity of the dataset and hedge ourselves against gathering exclusively any type of particular composition of trajectories, we require evenly distributed relative final positions. We require 25% of final positions to be located at the upper-right relatively to their corresponding initial position, i.e. having $z_f > z(0)$ and $x_f > x(0)$. Accordingly, we require 25% of final positions to be located at the lower-right, upper-left and lower-left relatively to their corresponding initial position. To perform this task, we implement a straightforward testing procedure and progressively discard any over-represented final states, until we reached the desired number of data points. Thus, we obtain the adapted equidistribution of relative final positions.

[2]This is an arbitrary requirement.

[3]Ensuring $t_f > 0$.

3.4 Dataset generation method

We describe the tools, algorithmic method and implementation details used to generate the synthetic dataset of near-optimal trajectories and controls.

3.4.1 Main algorithm

The main algorithm uses a data multiplication procedure as well as an encoding method of the tilt angle variable. We detail these operations before describing the entire algorithmic process.

3.4.1.1 Data multiplication procedure

As we generate a trajectory navigating from a state $\mathbf{q}_0$ to a state $\mathbf{q}_f$ and obtain controls at all time for said trajectory, we can leverage the *determinism* of the dynamics to multiply the number of data points.

This procedure is based on the fact that if we know all states $\mathbf{q}(t)$ and controls $\mathbf{u}(t)$, for $t \in [0, t_f]$, then any trajectory starting at $\mathbf{q}(t_1)$, with $t_1 \in [0, t_f]$, and finishing at $\mathbf{q}(t_1 + \Delta t)$, with $(t_1 + \Delta t) \in [t_1, t_f]$, along with all corresponding controls, are also admissible in the dataset $\mathcal{D}$. Hence, we are able to collect a set of sub-trajectories, extracted from any principal trajectory that has been computed solving the IVP (2.18).

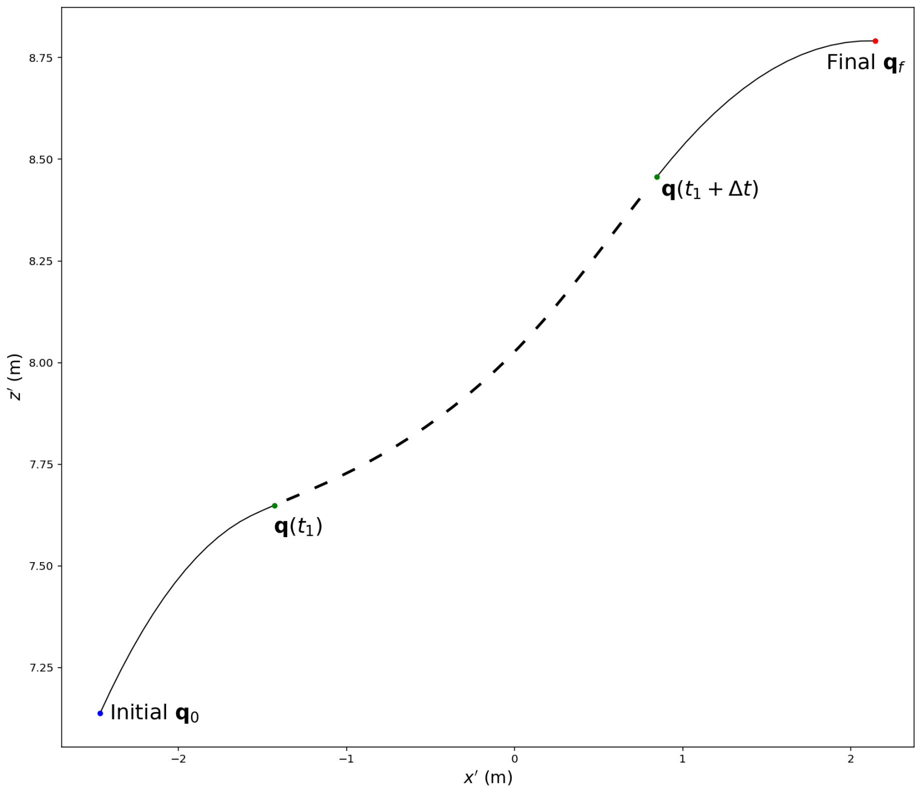


Figure 3.2: Example of a sub-trajectory extraction.

We start extracting the data points related to the principal trajectory fixing the final state $\mathbf{q}_f = \mathbf{q}(t_f)$ and going back through time. We compute the predictors $\hat{\mathbf{q}}(t) = \mathbf{q}_f - \mathbf{q}(t)$, selecting times $t = t_f - \Delta t$ progressively increasing $\Delta t > 0$, until reaching $t = 0$. We repeat the operation by fixing an earlier final state $\mathbf{q}_f^{(new)} = \mathbf{q}(t_f - dt)$. This operation defines the new sub-trajectory to be extracted. The process is continued until we reach $\mathbf{q}_f^{(new)} = \mathbf{q}_0$.

We consider a trajectory characterized by $n \in \mathbb{N}$ points $\big(\mathbf{q}_i, \mathbf{u}_i\big)_{i \in [\![1,n]\!]}$, with $\mathbf{q}_n = \mathbf{q}_f$. We describe the data multiplication procedure from a numerical approach.

Algorithm 1 Data multiplication procedure

⟨ *Generate a trajectory* $\big(\mathbf{q}_i, \mathbf{u}_i\big)_{i \in [\![1,n]\!]}$ *solving* (2.18) ⟩
for $i = n$ to 1 **do**
  $\mathbf{q}_f \leftarrow \mathbf{q}_i$
  for $j = i - 1$ to 0 **do**
    $\hat{\mathbf{q}}_{i,j} \leftarrow \mathbf{q}_i - \mathbf{q}_j$
    ⟨ *Add* $\big(\hat{\mathbf{q}}_{i,j}, \mathbf{u}_j\big)$ *to* $\mathcal{D}$ ⟩

For a generated trajectory made of n points, the multiplication procedure adds $n(n-1)/2$ data points to $\mathcal{D}$. Intuitively, predictors built subtracting states that are concurrent time-wise, i.e. such that $\hat{\mathbf{q}} = \mathbf{q}(t) - \mathbf{q}(t - dt)$, express the dynamics governing the UAV, while predictors built subtracting states that are separate time-wise, i.e. such that $\hat{\mathbf{q}} = \mathbf{q}(t) - \mathbf{q}(t - \Delta t)$, express the long-term transportation planning scheme followed by the quadrotor.

3.4.1.2 Angle encoding

Passing an angle variable as input to the model function $\boldsymbol{\pi}$ requires special care. Since having a tilt angle θ close to 0 or close to 2π leads to having a similarly oriented quadrotor, a modification of the predictors is necessary. In particular, we select the sine and cosine pairs as modified predictors.

$$\text{The predictors } \hat{\mathbf{q}} \text{ are modified according to } \hat{\theta} \rightarrow (\cos\hat{\theta},\, \sin\hat{\theta}) \tag{3.9}$$

Recall that $\hat{\theta} = \theta_f - \theta$. Intuitively, on the one hand, $\cos\hat{\theta}$ is a relevant predictor for the magnitude of the angle variation u_R to be applied in order to join $\theta(t_f) = \theta_f$. On the other hand, the sign of $\sin\hat{\theta}$ may relevantly account for the direction of the rotation to be applied in order to join said final state.

3.4.1.3 Complete procedure

We describe the algorithm used to generate the dataset $\mathcal{D}$, built for our supervised learning task.

Algorithm 2 Dataset generation

$K \leftarrow 0$
while $K < M$ **do**
  ⟨ *Sample parameters* $\tilde{\mathbf{q}}_0 \sim \mathcal{N}\big(\tilde{\mu},\, \tilde{\sigma}^2\,\mathbf{I}\big)$, $\theta(0) \sim \mathbb{P}_\theta$, $t_f \sim \ln\mathcal{N}\big(\mu_t,\, \sigma_t^2\big)$, $\mathbf{c} \sim \mathcal{N}\big(\mu_c,\, \sigma_c^2\,\mathbf{I}\big)$ ⟩
  ⟨ *Generate a trajectory* $\big(\mathbf{q}_i, \mathbf{u}_i\big)_{i \in [\![1,n]\!]}$ *solving* (2.18) ⟩
  ⟨ *Collect* $\mathbf{q}_f = \mathbf{q}_n$ ⟩
  if $\mathbf{q}_f$ *is admissible according to criteria defined in 3.3.2* **then**
    ⟨ *Apply data multiplication procedure described in* **Algorithm 1** ⟩
    ⟨ *Encode predictor* $\hat{\theta}$ *according to* (3.9) ⟩
    ⟨ *Add* $\big(\hat{\mathbf{q}}, \mathbf{u}\big)$ *to* $\mathcal{D}$ ⟩
    $K \leftarrow K + \frac{n(n-1)}{2}$

3.4.2 Implementation details

We implement the procedure described in **Algorithm 2** in `Python`.

We use vectorized function in all relevant situations via the `numpy` library. Said library offers partly pre-compiled code in `C`, hence producing faster computations. Moreover, we use the `numpy.random` sub-library to perform the parameter draw.

As previously, we solve (2.18) using a Runge-Kutta method of order 4 with the `scipy.integrate` sub-library.

We implement a custom function for testing the conditions defined 3.3.2, using dictionaries to characterize each sub-category of admissible trajectories. In particular, we add an extra condition, requiring $\sqrt{\left(x_f - x(0)\right)^2 + \left(z_f - z(0)\right)^2} < d_{\max}$ to narrow down the variance of the computed dataset.

We generate the data points in batches of trajectories, marking each batch and each trajectory by a unique identification number.

$d_{\max}$	6.0
$\tilde{\mu}$	0
$\tilde{\sigma}$	4
μ_t	0.75
σ_t	0.20
μ_c	0
σ_c	200

Table 3.4: Parameters used for the dataset generation.

The `Python` code used for the dataset generation can be found in the appendix B.

3.4.3 Generated dataset overview

We built a dataset $\mathcal{D}$ composed of $M = 123\,750\,000$ points $(\hat{\mathbf{q}}_{(i)}, \mathbf{u}_{(i)})$. The dataset is made of 25 000 principal trajectories, themselves composed of $n = 100$ state-control pairs each. Using the data multiplication procedure described in 3.4.1.1, each principal trajectory produces 4 950 data points.

We use a 85%/15% split to divide the dataset into a training set and a validation set. Hence, we have $N = 105\,187\,500$ and $K = 18\,562\,500$.

We provide an overview of the dataset by printing a subset of the trajectories in the (x', z') plane. We add an arrow to highlight the direction in which the motion is performed.

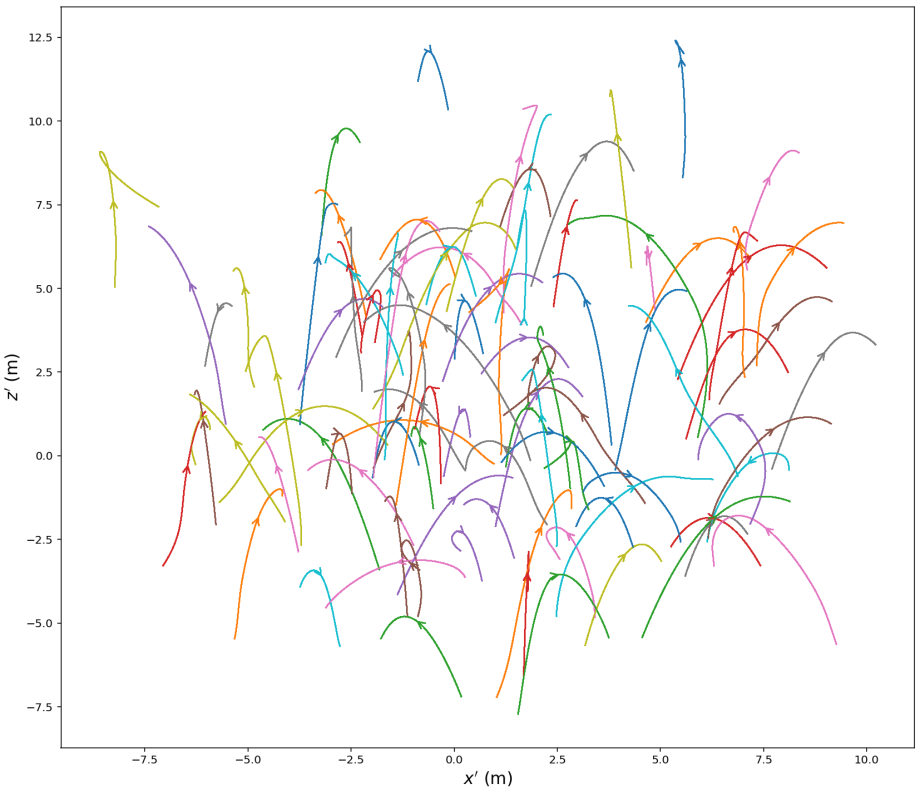


Figure 3.3: Subset of $\mathcal{D}$ in the (x', z') plane.

We can now use the presented dataset $\mathcal{D}$ to fit a relevantly chosen model function $\boldsymbol{\pi}$. This model function will be implemented by a **deep neural network**.

Chapter 4

Deep neural network for near-optimal closed loop controls

We present the model used to approximate the optimal control $\mathbf{u}^\star = \mathbf{u}^\star(t, \mathbf{q})$. In particular, the model function $\boldsymbol{\pi}$ is implemented using a deep neural network and fitting the data provided in $\mathcal{D}$. We describe the specific structure used and examine the behavior of the quadrotor while navigated by the synthesized controls in a closed loop manner.

We present a table of the different abbreviations used in this chapter.

DNN	Deep neural network
ReLU	Rectified linear unit
SGD	Stochastic gradient descent

Table 4.1: List of abbreviations used in Chapter 4.

4.1 A brief introduction to deep neural networks

The model $\boldsymbol{\pi} : \hat{\mathbf{q}} \to \mathbf{u}$ will be implemented as a *deep neural network* (DNN). We define a such function and discuss its use to approximate optimal controls.

4.1.1 Overview and definitions

The main building block of deep neural networks is the artificial neuron [39]. For a given nonlinear function $\sigma : \mathbb{R} \to \mathbb{R}$ called *activation*, a set of parameters $\omega_1, \dots, \omega_n \in \mathbb{R}$ called *weights*, and an extra parameter $b \in \mathbb{R}$ called *bias*, an artificial neuron is a function $h : \mathbb{R}^n \to \mathbb{R}$ such that $h(a_1, \dots, a_n) := \sigma\Big(\sum_{j=1}^{n} \omega_j a_j + b\Big)$.

The artificial neural network is a function $\check{\boldsymbol{\pi}} : \mathbb{R}^p \to \mathbb{R}^q$ that assembles successive layers of neurons [40]. In particular, it is made of an *input layer* processing a predictor variable $\mathbf{x} \in \mathbb{R}^p$, followed by several *hidden layers* performing core manipulation on the provided data, and terminates by an *output layer* providing the outcome variable $\mathbf{y} \in \mathbb{R}^q$.

Each hidden layer is composed of neurons handling information incoming from previous layers and passing updated information to the next layers. We will consider a *fully-connected* DNN, in which each neuron of a hidden layer passes a copy of its output $h(a_1, \dots, a_n) \in \mathbb{R}$ to all the neurons of the next layer.

Specifically, we consider a fully-connected DNN, taking $n_0 = p$ inputs, composed by $L \in \mathbb{N}$ hidden layers. Each hidden layer is made of $(n_k)_{k \in [\![1,L]\!]}$ neurons, having their weights and bias encoded in $\mathbf{W}^{(k-1)} \in \mathbb{R}^{n_k \times n_{k-1}}$ and $\mathbf{b}^{(k-1)} \in \mathbb{R}^{n_k}$, for $k \in [\![1, L+1]\!]$. The activations are given by $\left(\boldsymbol{\sigma}^{(k)}\right)_{k \in [\![1,L]\!]}$, and apply any chosen nonlinear function on the neuron outputs element-wise. The artificial neural network $\check{\boldsymbol{\pi}}$ is evaluated as in the following [41].

$$\check{\boldsymbol{\pi}}(\mathbf{x}) := \boldsymbol{\sigma}\big(\mathbf{W}^{(L)}\mathbf{h}^{(L)} + \mathbf{b}^{(L)}\big) \tag{4.1}$$

$$\mathbf{h}^{(k)} = \boldsymbol{\sigma}\big(\mathbf{W}^{(k-1)}\mathbf{h}^{(k-1)} + \mathbf{b}^{(k-1)}\big), \text{ for } k \in [\![1, L]\!] \tag{4.2}$$

$$\mathbf{h}^{(0)} = \mathbf{x} \tag{4.3}$$

Deep neural networks are *compositional* in essence. They generate complexity by successively composing a large number of elementary functions, namely the functions related to artificial neurons. Therefore, they are adapted for approximating complex functions, fitting data, and recognizing patterns [42].

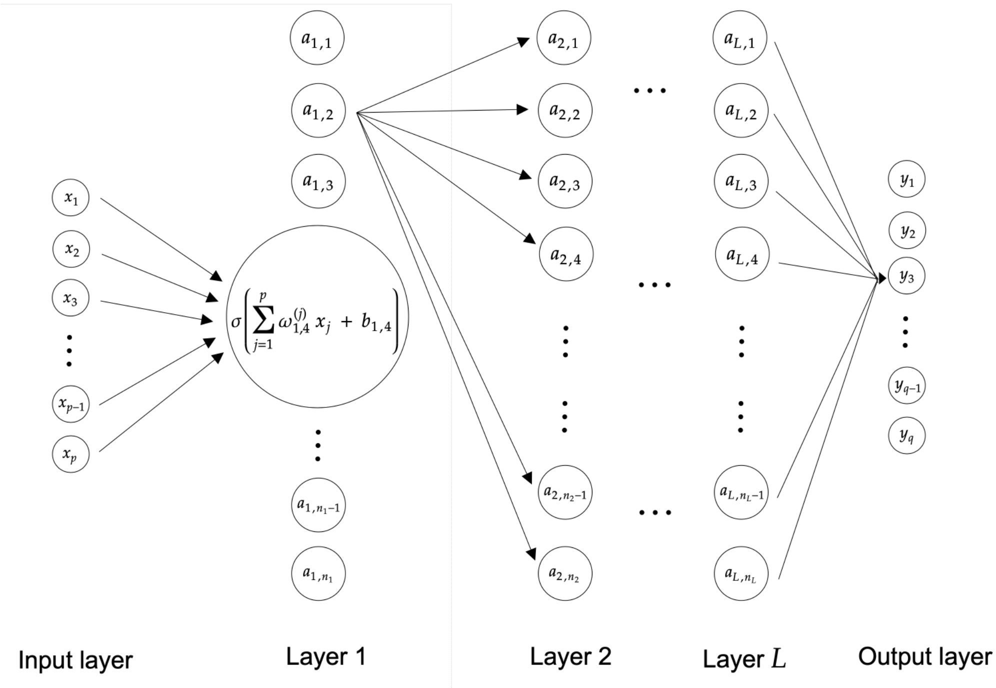


Figure 4.1: Structure of a deep neural network composed of fully-connected layers.

A deep neural network can be used in a supervised scheme, to fit a set of predictors and outcomes pairs. The training process is done by successively updating the parameters of the DNN, which are the set of weights $\omega_{i,j}^{(k)}$ and biases $b_{i,j}$, in order to minimize a mean sample loss $E(L)$.

The update of parameters is made using *gradient descent*. We denote Θ_s the set of parameters of the DNN, at iteration $s \in \mathbb{N}$ of the optimization process. The update is made by modifying the parameters, performing a small step towards the direction of steepest descent, with respect to the mean sample loss function, in said parameters space.

$$\Theta_{s+1} = \Theta_s - \eta \nabla_\Theta E(L_{\Theta s}) \tag{4.4}$$

The parameter η is called the learning rate. In the update (4.4), the gradient is usually not evaluated on every points of the training set. As this can be computationally expensive, an estimate of the gradient is instead computed on a minibatch of randomly drawn training points. This technique is referred as *stochastic gradient descent*. It accelerates the training process and does not compromise with the minimization of the loss function if η has been chosen sufficiently small.

The computation of the gradient is made by a specfic method called *backpropagation* [43, 44]. The gradient of the loss function is computed with respect to the weights and biases using the chain rule. Its value is first computed regarding parameters of the output layer and is backward-iteratively computed for hidden layers, via an explicit update. This method efficiently avoids redundancy in the gradient computations.

4.1.2 Discussing DNN as optimal control approximators

Optimal closed loop controllers can be obtained by solving the Hamilton-Jacobi-Bellman partial differential equation in a discretized manner, via a space and time grid. The size of the grid and the computational power required increases substantially with the number of equations that are given by the dynamics of the considered system. In the context of the planar-quadrotor, the curse of dimensionality forbids any on-board computing of optimal controls by solving the (HJB) equation.

The use of a DNN to approximate an optimal controller is relevant in this case, as the network can be trained offline, and used on-board to predict the instant controls. A prediction is obtained by passing the current measured state $\hat{\mathbf{q}}$ as input into the network, and performing a *forward pass* to compute the outcome control $\mathbf{u}$. The forward pass is efficient time and complexity wise for reasonable activation functions $\boldsymbol{\sigma}$. As the memory required to store the model increases with the number of parameters Θ, for any acceptable yet effective DNN structure, the memory required remains transferable to a portable drive. Hence, the predictions can be performed on the edge, loading the model on an on-board memory unit and computing forward passes via a specialized processing unit at small time steps.

Moreover, deep neural networks verify a set of *universal approximation theorems*, which characterize their ability to approximate any given continuous function, to a certain arbitrary extent and under specific conditions [45, 46, 47]. Even though no assumptions are made concerning the regularity of the optimal closed loop control function $(t, \mathbf{q}) \to \mathbf{u}^\star$, a DNN could perform a continuous approximation of the controls. The precision of said approximation could be pushed as far as needed by the user.

4.2 Defining our deep neural network model

We present the custom DNN $\boldsymbol{\pi}$ and its structure that will be implemented for fitting the data points in $\mathcal{D}$, in order to synthesize our controller.

4.2.1 Activations

Two different types of activation functions are used in the DNN.

4.2.1.1 Hidden layer activations

All hidden layers in $\boldsymbol{\pi}$ will have rectified linear units (ReLU) as activations.

$$\begin{array}{rcccc} \text{ReLU} & : & \mathbb{R} & \to & \mathbb{R} \\ & & x & \mapsto & \max(0, x) \end{array} \tag{4.5}$$

Deep neural networks composed by neurons activated using ReLU represent a piecewise affine function. Such networks approximate general functions by fitting the affine regions accordingly. In particular, they can be used in model predictive control under certain conditions [48]. In our specific context, ReLU was found to be heuristically more efficient in synthesizing the controller.

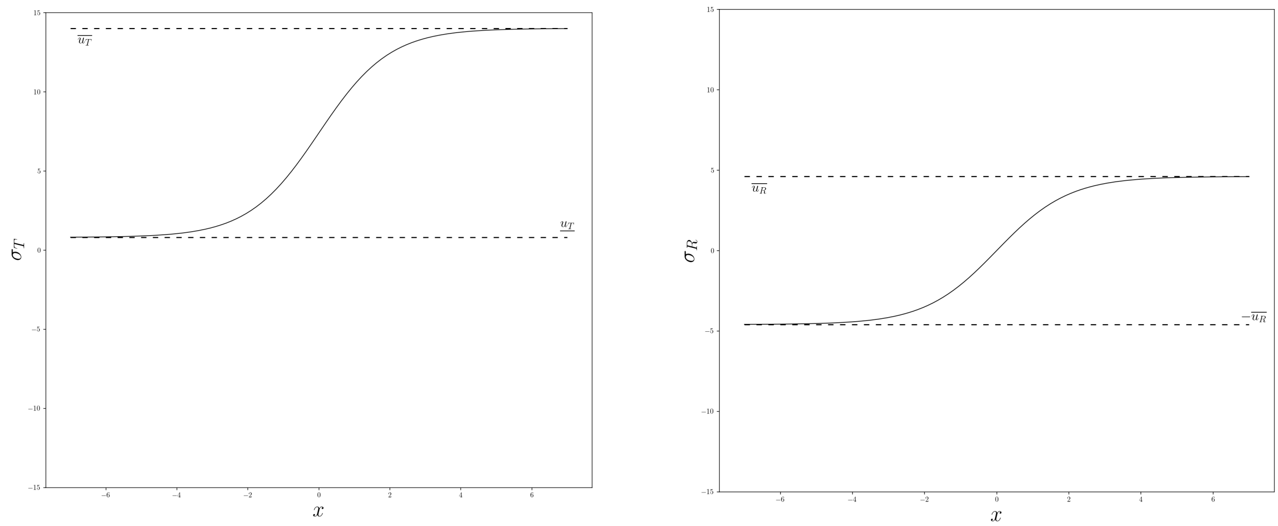
4.2.1.2 Output layer activations

The output layer will be composed by two neurons for computing both u_T and u_R. Two specific activation functions will be used to ensure that $\mathbf{u} \in U$. The *sigmoid* function is a common activation applied in DNNs. We build two modified sigmoids $\sigma_T : \mathbb{R} \to \mathbb{R}$ and $\sigma_R : \mathbb{R} \to \mathbb{R}$ to ensure that $\underline{u_T} \leq u_T \leq \overline{u_T}$ and $-\overline{u_R} \leq u_R \leq \overline{u_r}$.

$$\sigma_T(x) := \ (\overline{u_T} - \underline{u_T}) \ \frac{1}{1+e^{-x}} \ + \ \underline{u_T} \tag{4.6}$$

$$\sigma_R(x) := \ \overline{u_R} \left(\frac{2}{1+e^{-x}} - 1 \right) \tag{4.7}$$

Said functions will ensure that the controls are admissible as $\sigma_T(x) \xrightarrow[x\to+\infty]{} \overline{u_T}$, $\sigma_T(x) \xrightarrow[x\to-\infty]{} \underline{u_T}$, $\sigma_R(x) \xrightarrow[x\to+\infty]{} \overline{u_R}$, and $\sigma_R(x) \xrightarrow[x\to-\infty]{} -\overline{u_R}$.



(a) Modified sigmoid for thrust control σ_T

(b) Modified sigmoid for rotational control σ_R

Figure 4.2: Graphs of modified sigmoid activations used on the output layer.

Intuitively, for controls saturated at their admissible minimum (respectively admissible maximum), the value passed to the last neurons must be arbitrarily small (respectively arbitrarily large). In this sense, the custom activations allow the DNN to adapt to the required output space U. Furthermore, the smoothness of the sigmoids will facilitate computations of the gradient and will subsequently ease the training process.

4.2.2 The model

We describe the complete deep neural network used to approximate the controls.

4.2.2.1 Regularization techniques

To avoid overfitting, we first apply the *dropout* technique to our DNN. This technique consists in randomly selecting weights and setting their values to zero during the training process [49]. This procedure is equivalent to 'cutting' some connections between neurons. It prevents subsequent neuron connections to adjust to the training points and lose generalisability.

Secondly, we apply *early stopping* during the training process. As presented in section 3.1.1, to avoid overfitting we compute the loss function on a validation set. The training process is performed going through the complete training set several times. Each complete pass is referred to as an *epoch*. As the loss computed on the training set continuously decreases for increasing numbers of epochs, the loss computed on the validation set first decreases then increases. The early stopping technique consists in stopping the training process at the minimum of the validation loss, i.e. stopping the process before the model starts overfitting the training data.

4.2.2.2 Cross-validation

The number of hidden layers and the number of neurons per layer are central hyperparameters to be tuned, in order to obtain good performances of the model. As no exact rule exists for finding such hyperparameters, we apply heuristics to find optimal values.

In particular, we use a specific 'rule of thumb' to bound the total number of neurons composing the DNN. Recalling that $N \in \mathbb{N}$ is the number of training points, that we have 6 input values[1], 2 output values and additionally denoting $\alpha \in [\![2, 10]\!]$ a scaling factor, we can estimate the maximum number of neurons $N_e^{(\max)}$ that can be used, while reasonably avoiding overfitting.

$$N_e^{(\max)} = \frac{N}{\alpha\,(2+6)} \tag{4.8}$$

Overall, **we generated and trained more than 40 different DNN architectures** and assessed their performance using cross-validation techniques. We split the dataset to avoid bias and monitored the metrics with different hyperparameters.

4.2.2.3 Model summary

The final structure has been chosen with respect to the validation loss, as well as monitoring the ability of the simulated quadrotor to perform navigations between arbitrary and random initial and final states.

The chosen deep neural network is composed of :

- An input layer made of 6 neurons
- 5 hidden layers made of 900 neurons each and activated using ReLU functions
- All connections between hidden layers are subject to dropout with a probability of 25%
- An output layer made of 2 neurons, respectively activated by σ_T and σ_R

The network is composed of 3 251 702 trainable parameters. The DNN is optimized using SGD and the training is made performing early stopping.

[1] Even though the state difference vector $\hat{\mathbf{q}}$ holds 5 numbers, it is modified according to the angle encoding procedure 3.4.1.2, bringing the number of inputs to 6.

4.2.2.4 Implementation details

We implement the DNN and perform its training in `Python`. We use the `tensorflow` library as computing engine and the `keras` library as high-level design environment.

Minibatch size	128
Maximum number of epochs	50
Learning rate η	10^{-3}

Table 4.2: Parameters used in the DNN training.

The code related to said implementation can be found in appendix C.1.

4.3 Results

We first present the metrics monitored during the training process. Subsequently, we describe how to simulate a quadrotor trajectory using the trained DNN as a controller. Finally, we present a range of closed loop trajectories and analyze their optimality, as well as the general behavior of the feedback-controlled quadrotor.

4.3.1 Training and validation

As explained in section 3.1.1 and section 4.2.2, during the training process the mean sample loss is computed on the training set $\mathcal{T}$ and the validation set $\mathcal{V}$. As the training goes on, the gradient descent minimizes the loss function. Said loss continuously decreases until the early stopping is triggered.

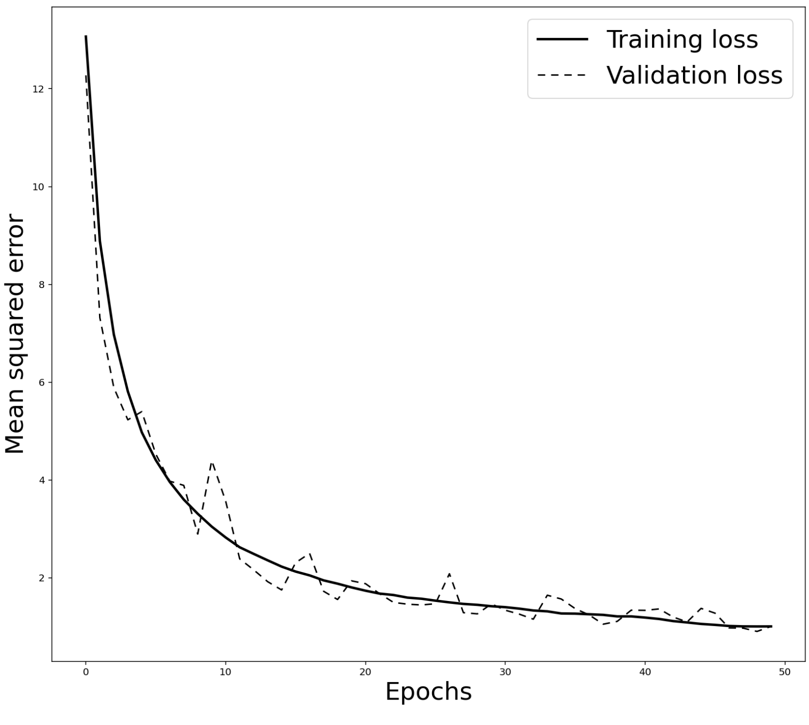


Figure 4.3: Mean sample loss on training and validation sets across SGD optimization.

We observe that the mean sample loss computed on the training set strictly decreases for an increasing number of epochs. Moreover, we have that the mean sample loss computed on the validation set decreases sinusoidally, globally following the same trend as the training loss : the DNN fits the data to some reasonable extent.

4.3.2 Simulator

We obtained a DNN $\boldsymbol{\pi}$ able of predicting applicable controls, given a measured instant state $\mathbf{q}$. We develop a specific algorithm for simulating closed loop trajectories based on the evaluation of $\boldsymbol{\pi}$. We use a process based on the Euler method with a stepsize $h = 2.5 \cdot 10^{-3}$ and a tolerance parameter $\epsilon > 0$.

Algorithm 3 Closed loop simulation process

$\mathbf{q} \leftarrow \mathbf{q}_0$
$\hat{\mathbf{q}} \leftarrow \mathbf{q}_f - \mathbf{q}$
$\langle$ *Encode predictor* $\hat{\theta}$ *according to* (3.9) $\rangle$
$\mathbf{u} \leftarrow \boldsymbol{\pi}(\hat{\mathbf{q}})$
while $\|\hat{\mathbf{q}}\| > \epsilon$ **do**
    $\mathbf{q} \leftarrow \mathbf{q} + h \cdot \boldsymbol{f}(\mathbf{q}, \mathbf{u})$
    $\hat{\mathbf{q}} \leftarrow \mathbf{q}_f - \mathbf{q}$
    $\langle$ *Encode predictor* $\hat{\theta}$ *according to* (3.9) $\rangle$
    $\mathbf{u} \leftarrow \boldsymbol{\pi}(\hat{\mathbf{q}})$

We implement the custom simulator in `Python`. The corresponding code can be found in appendix C.2 and C.3.

4.3.3 Examples of closed loop navigation

We perform tryouts of closed loop trajectories at a large scale. We examine the behavior of the quadrotor on autonomous navigation. In the majority of the studied flights, the vehicle proves capable of joining the final state in a satisfactory amount of time. However, for some given states the quadrotor performs irregular manoeuvres. Overall, the synthesized controller satisfies the constraints and objective defined in the OCP to a reasonable extent.

4.3.3.1 From $z = 0$ to $z = 1$

The first trajectory we study is the straight take off line, from an initial position $x(0) = 0$ and altitude $z(0) = 0$, to a similar position $x_f = 0$ and a higher altitude $z_f = 1$.

The quadrotor has no initial horizontal and vertical velocities, i.e. $\dot{x}(0) = 0$ and $\dot{z}(0) = 0$. We require no final horizontal and vertical velocities equally, i.e. $\dot{x}_f = 0$ and $\dot{z}_f = 0$.

The quadrotor starts with a tilt angle set to zero, i.e. $\theta(0) = 0$ and is required to terminate its flight having the same orientation, i.e. $\theta_f = 0$.

The initial state vector is $\mathbf{q}_0 = (0, 0, 0, 0, 0)^\top$. The final targeted state is $\mathbf{q}_f = (0, 0, 1, 0, 0)^\top$.

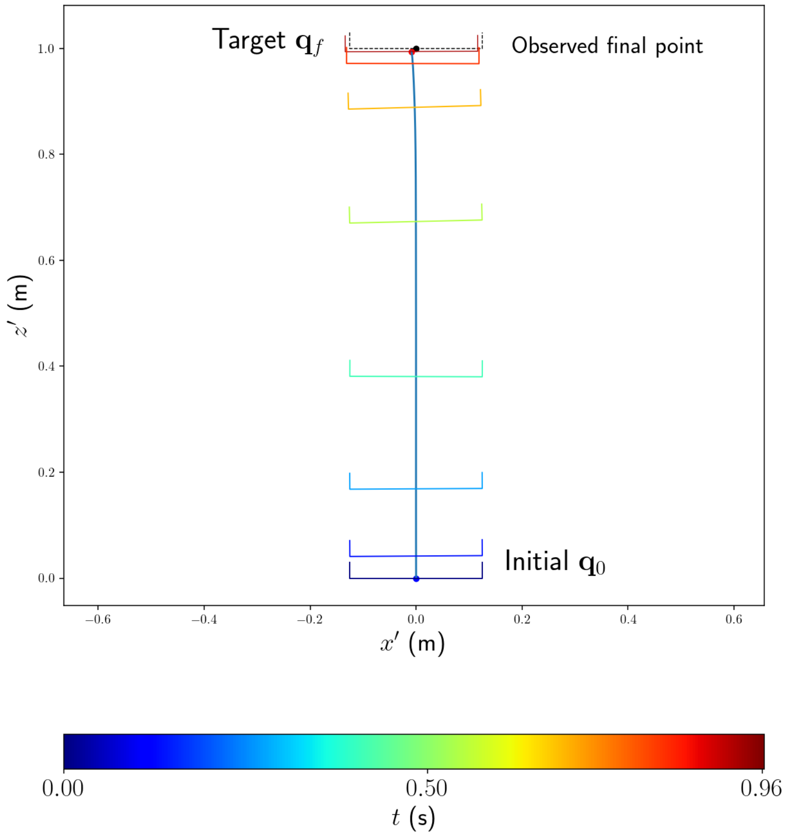


Figure 4.4: Closed loop quadrotor trajectory starting at $\mathbf{q}_0 = (0, 0, 0, 0, 0)^\top$, targeting final state $\mathbf{q}_f = (0, 0, 1, 0, 0)^\top$.

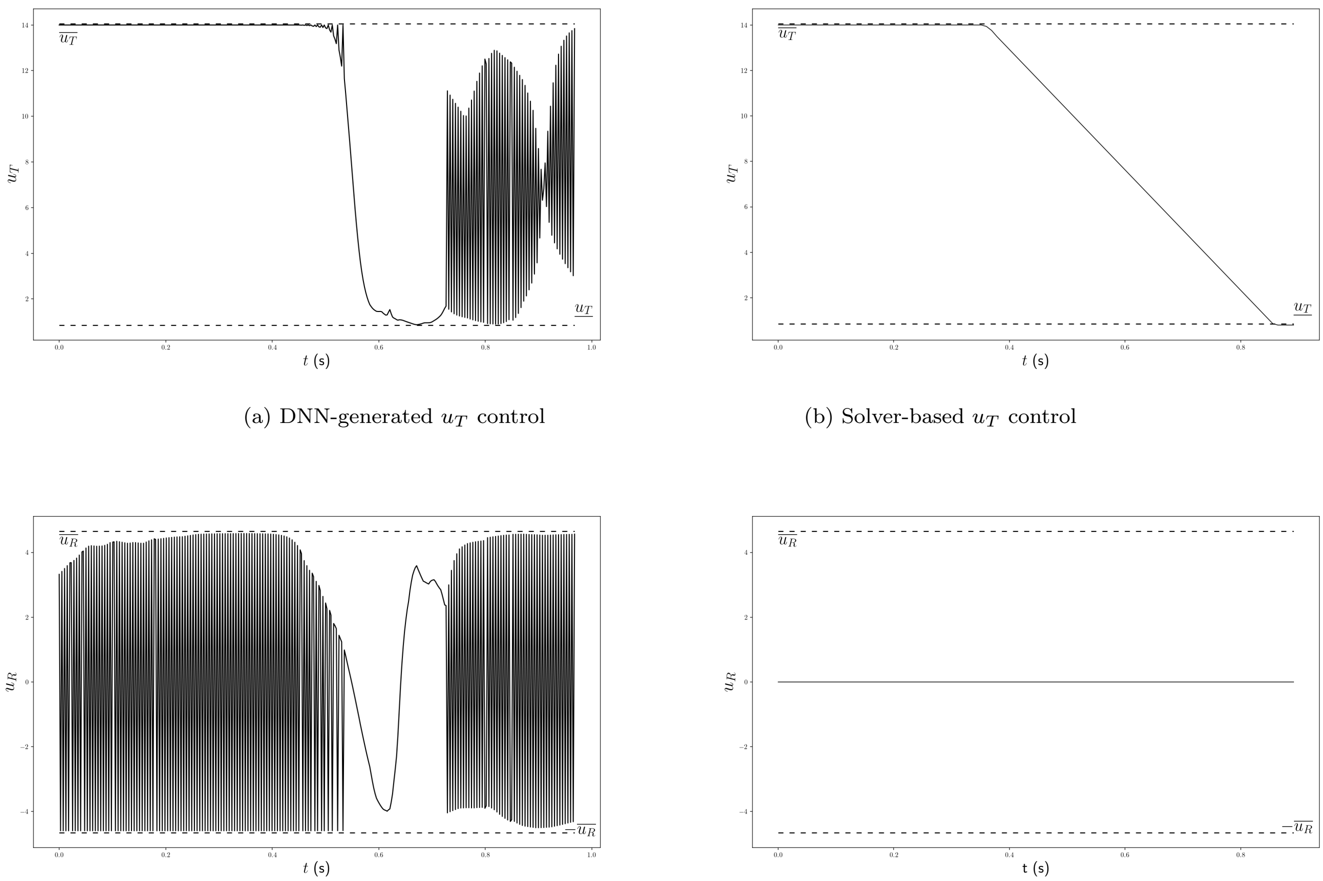
We observe that the quadrotor effectively joins the required final state within the tolerance, in a time $t_f = 0.96\, s$. It follows a straight vertical line, the tilt angle remaining small along all the navigation. The vehicle starts by vertically accelerating, before decelerating soon enough to arrive at $z_f = 1$ with no vertical velocity.

We recall that the analysis made in section 3.2.1 prevents such vertical trajectories from being represented in the dataset $\mathcal{D}$, given the conditions under which the constant vector $\mathbf{c}$ has been drawn. Yet, the autonomous quadrotor is able to perform the upward straight line navigation. This ability may arise from the data multiplication procedure described in 3.4.1, as small sections of more complicated trajectories may also constitute similar straight lines. Hence, in this situation, the DNN controller encounters predictors that are comparable to others that were present in $\mathcal{D}$.

Note that this straight line trajectory is optimal regarding the OCP. In particular, we computed the solver-based solution which is also the vertical line joining the initial and target altitude. We compare the energy contributions and cost functionals of both solver-based and closed loop controls.

	DNN-generated	Solver-based
t_f	0.96	**0.89**
E	138.75	**104.98**
J	139.71	**105.8**

Table 4.3: Cost functionals and contributions associated with the $z(0) = 0$ to $z_f = 1$ trajectory.



(a) DNN-generated u_T control

(b) Solver-based u_T control

(c) DNN-generated u_R control

(d) Solver-based u_R control

Figure 4.5: Quadrotor controls starting at $\mathbf{q}_0 = (0, 0, 0, 0, 0)^\top$, targeting final state $\mathbf{q}_f = (0, 0, 1, 0, 0)^\top$. Both DNN-generated and solver-based controls are presented.

We observe that although both set of controls lead to the same final state, the DNN-generated closed loop controls have a tendency to strongly oscillate around a mean value.

We first focus on the u_R values 4.5c and 4.5d. Notice that the optimal trajectory requires no tilt variation, i.e. $u_R = 0$ at all times. While the solver-based controller achieves this goal steadily, the closed loop u_R alternates between opposite saturated values $-\overline{u_R}$ and $\overline{u_R}$, averaging to zero over time. **This shows a propensity of the model $\boldsymbol{\pi}$ to output mainly bang-bang controls**.

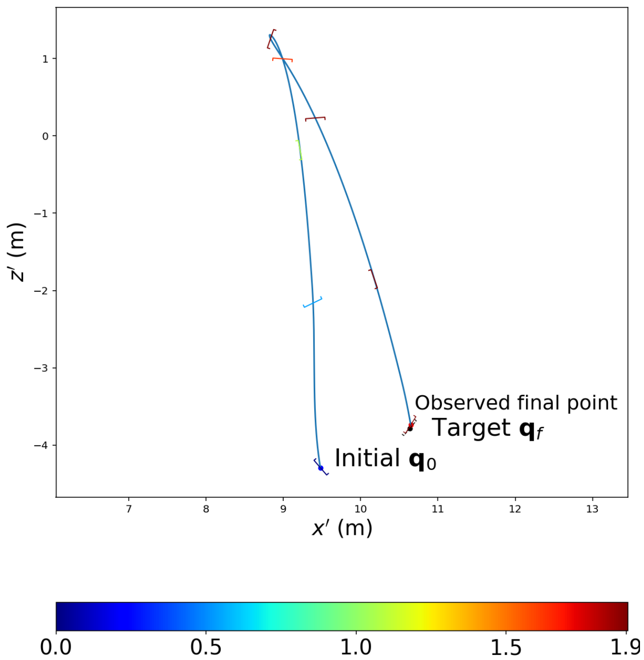
We secondly focus on the u_T values 4.5a and 4.5b. We observe that both closed loop and solver-based controllers start accelerating the quadrotor by an initial saturated thrust value $\overline{u_T}$. After a short time interval, while the solver control continuously decelerates to arrive at the final state with the required velocity, the DNN controller jumps directly to the saturated lower bound $\underline{u_T}$. This results in an excessive deceleration, which is ultimately compensated by an oscillating re-acceleration, averaging to an adequate value for obtaining $\dot{z}_f = 0$.

As a matter of consequence, as given in table 4.3, the solver-based controls are more optimal time and energy wise. To lower the energy contribution of the controls given by $\boldsymbol{\pi}$, we may for instance average the outcomes on a number of subsequent predictions. This would result in smoothing the bang-bang effects on u_R and consequently lowering E. We also recall that contrarily to the solver-based controls, the DNN controller is usable on the edge and the predictions may be made in real-time.

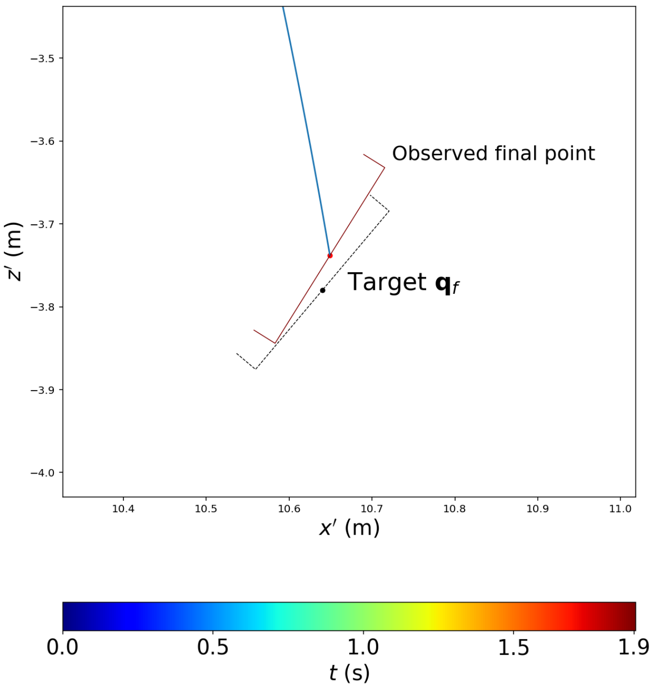
Moreover, said bang-bang behavior may be moderated adding a parameter μ to the core sigmoids of the custom output activations. Tuning said hyperparameter in a sigmoid $x \to 1/(1 + e^{-\mu\, x})$ may help modulate the slope at the inflection point and favor a wider range of unsaturated controls.

4.3.3.2 Example trajectory analyzed in 3.2.3

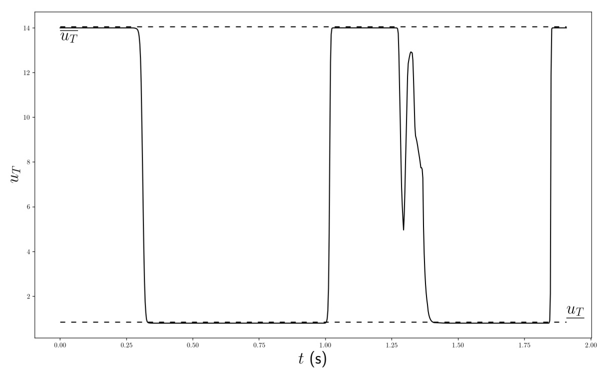
We examine the same trajectory analyzed in 3.2.3 in the case of a closed loop navigation.



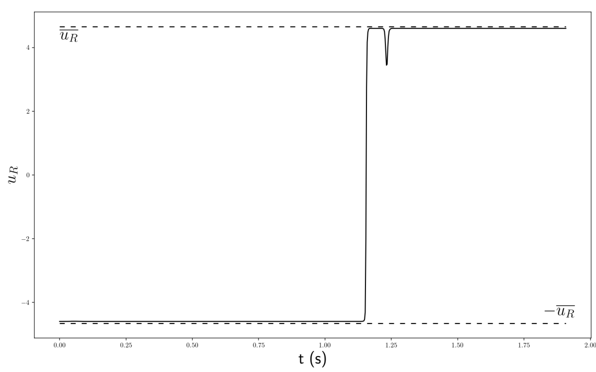
(a) Closed loop trajectory



(b) Zoom box around the targeted final state



(c) DNN-generated u_T control



(d) DNN-generated u_R control

Figure 4.6: Closed loop quadrotor trajectory starting at $\mathbf{q}_0 = \left(9.48, -1.12, -4.29, 7.14, 0.86\right)^\top$, targeting final state $\mathbf{q}_f = \left(10.64, 1.87, -3.78, -10.12, -0.87\right)^\top$.

The quadrotor joins the final state in $t_f = 1.90$ seconds. Unsurprisingly the trajectory followed by the vehicle is very similar to the theoretically-generated one. Although the example analyzed in 3.2.3 has been generated solving the system 2.18, it has not been added to the dataset $\mathcal{D}$. The model function $\boldsymbol{\pi}$ predicts instant controls based on patterns learnt in the training data. As similarities may exist between some trajectories composing $\mathcal{D}$ and the example trajectory considered in 3.2.3, it is probable to obtain similar behaviors.

The controls are also often saturated and have a tendency to oscillate for this navigation, which is in accordance with the results obtained in section 4.3.3.1. In this context, we notice that the switching times are similar to the ones in the theoretically generated controls. We examine the optimality of the considered closed loop navigation.

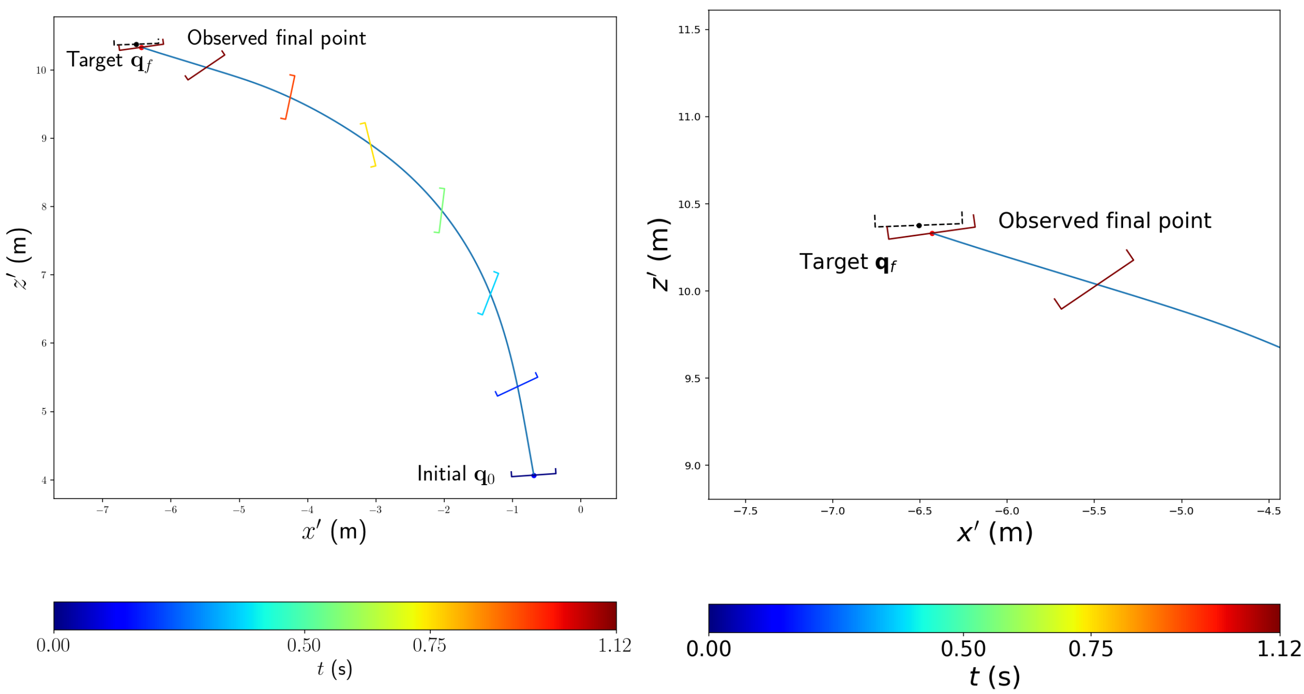
	Theoretically generated	Solver-based	Closed loop
t_f	**2.00**	2.48	**1.90**
E	65.42	**57.36**	71.11
J	67.42	**59.84**	73.01

Table 4.4: Cost functionals and contributions associated with the example trajectory. The closed loop navigation has been added.

Although the closed loop navigation is more time-optimal than the theoretically generated and solver-based ones, it produces a higher energy contribution. The cost functional value remains reasonable as it is 22% greater than the minimum acheived.

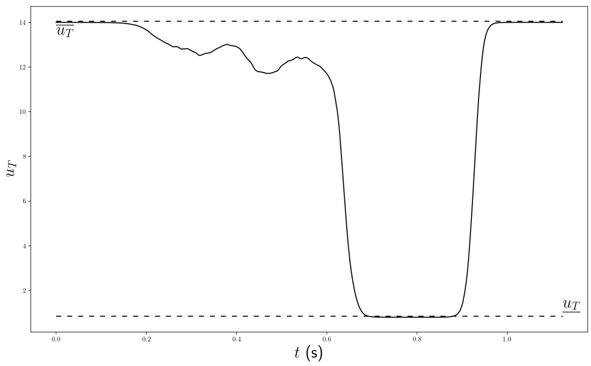
4.3.3.3 Randomly drawn initial and final state

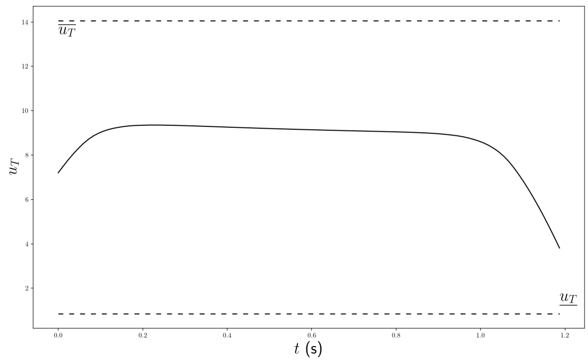
We consider a closed loop navigation made between two randomly drawn initial and final states. Said states are drawn according to (3.6).



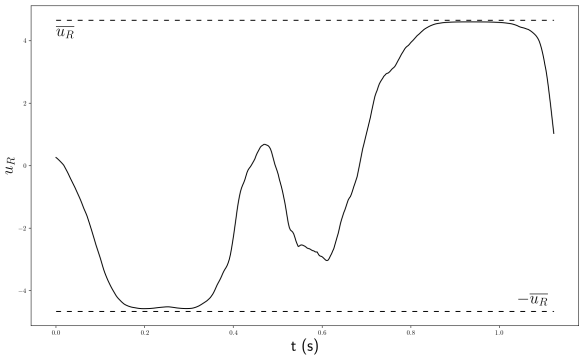
(a) Closed loop trajectory (b) Zoom box around the targeted final state

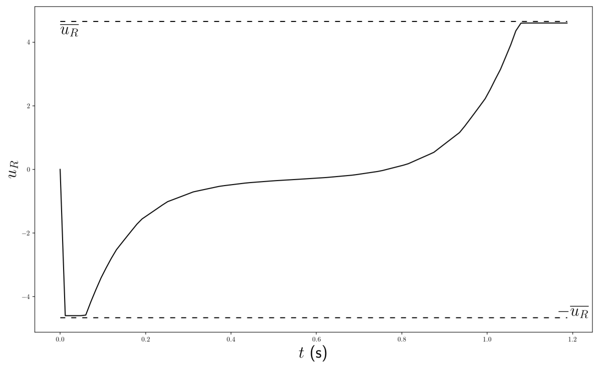
Figure 4.7: Closed loop quadrotor trajectory starting at $\mathbf{q}_0 = \big(-0.69,\ -1.29,\ 4.07,\ 7.40,\ 6.21\big)^\top$, targeting final state $\mathbf{q}_f = \big(-6.51,\ -8.47,\ 10.38,\ 2.78,\ 6.25\big)^\top$.





(a) DNN-generated u_T control (b) Solver-based u_T control





(c) DNN-generated u_R control (d) Solver-based u_R control

Figure 4.8: Quadrotor controls for same initial and final states as in figure 4.7. Both DNN-generated and solver-based controls are presented.

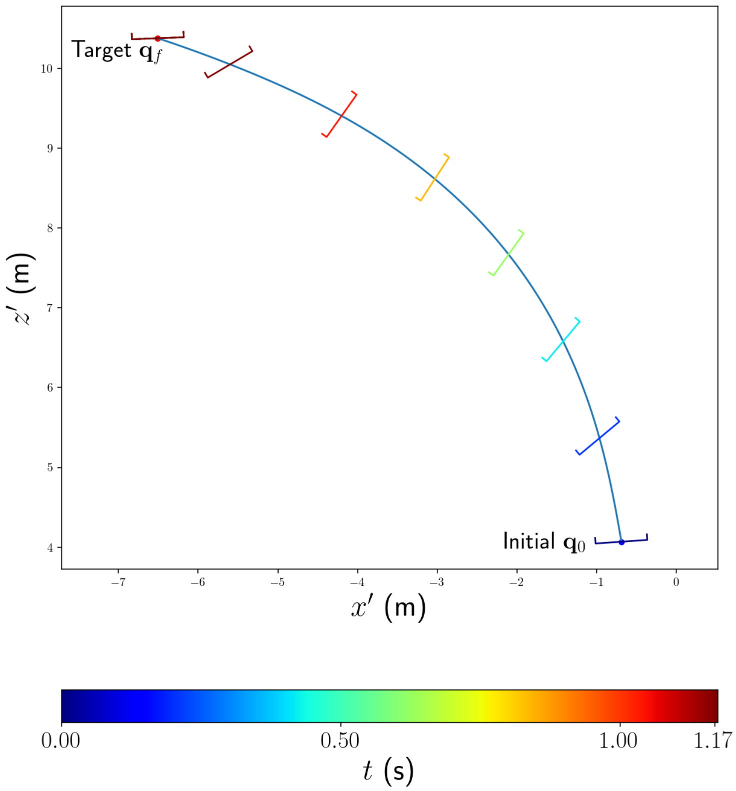


Figure 4.9: Solver-based quadrotor trajectory for same initial and final states as in figure 4.7.

	DNN-generated	Solver-based
t_f	**1.12**	1.17
E	116.25	**97.12**
J	117.37	**98.29**

Table 4.5: Cost functionals and contributions associated with the trajectory having initial and final states as in figure 4.7.

The closed loop controller performs the navigation in $t_f = 1.12$ seconds. We compare the obtained trajectory and controls with the solver-based ones. As in previous tested trajectories, even though the DNN is more time-optimal, it generates larger controls norm-wise, which results in a higher energy contribution and a reasonably higher cost functional. In particular, J is 19 % higher in this case.

As observed in figure 4.8, the closed loop controls have a tendency to vary too strongly at first, before correcting in a second time. This behavior produces wiggling u_R and u_T functions. The dynamical consequences of said behavior are visible on the trajectory in figure 4.7 : the tilt angle becomes too steep at first, before being modified to match the correct final orientation.

Note that, as visible in figure 4.7b, the final state is never perfectly reached with closed loop controls. The quadrotor joins $\mathbf{q}_f$ within a reasonable tolerance for a wide range of drawn initial and final states.

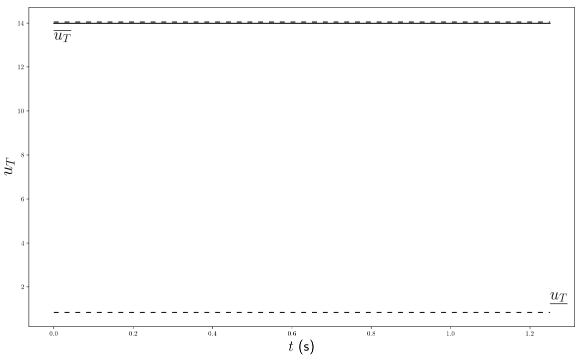
4.3.3.4 Atypical trajectory

We examine the particular trajectory in which the vehicle is initially at rest and is required to come back to its starting position while being upside down.

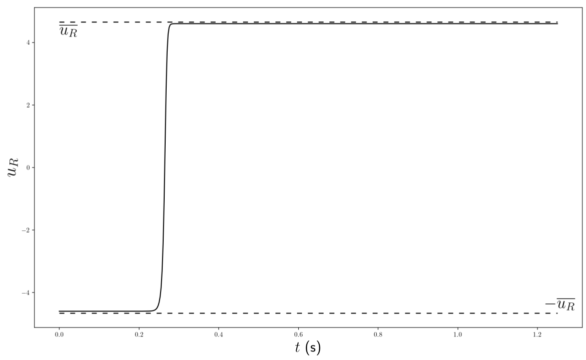
The vehicle is located at $x(0) = 0$ and $z(0) = 0$ and is required to come back, i.e. $x_f = 0$ and $z_f = 0$. It has no initial horizontal and vertical velocities, i.e. $\dot{x}(0) = 0$ and $\dot{z}(0) = 0$. We require no final horizontal and vertical velocities equally, i.e. $\dot{x}_f = 0$ and $\dot{z}_f = 0$.

The quadrotor starts with a tilt angle set to zero, i.e. $\theta(0) = 0$ and is required to terminate its flight being upside down, i.e. $\theta_f = \pi$.

The initial state vector is $\mathbf{q}_0 = (0, 0, 0, 0, 0)^\top$. The final targeted state is $\mathbf{q}_f = (0, 0, 0, 0, \pi)^\top$.



(a) DNN-generated u_T control



(b) DNN-generated u_R control

Figure 4.10: Closed loop quadrotor controls for trajectory starting at $\mathbf{q}_0 = (0, 0, 0, 0, 0)^\top$, targeting final state $\mathbf{q}_f = (0, 0, 0, 0, \pi)^\top$.

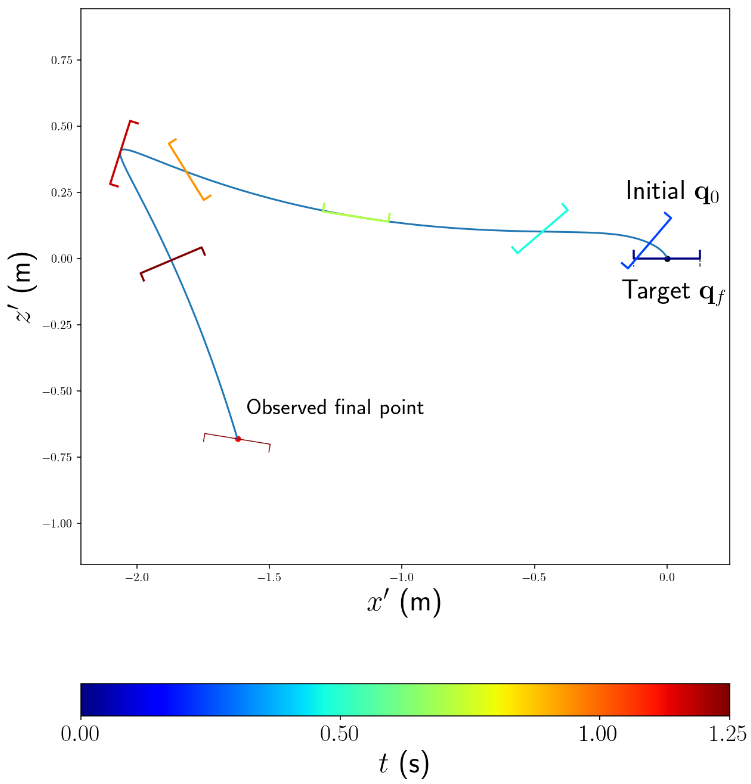


Figure 4.11: Quadrotor trajectory starting at $\mathbf{q}_0 = (0, 0, 0, 0, 0)^\top$, targeting final state $\mathbf{q}_f = (0, 0, 0, 0, \pi)^\top$.

We observe that the synthesized controller $\boldsymbol{\pi}$ fails to join the final state $\mathbf{q}_f$. Even though the quadrotor initiates a half turn, the vehicle unsuccessfully manages the downward pulling gravitational effect.

In a such particular situation, the failure of the navigation is probable. As the states have been continuously drawn from a normal distribution in the training set, the initial inputs passed to the DNN in this trajectory are likely to be missing or to constitute outliers in $\mathcal{D}$. Thus, the DNN may output inappropriate controls in the first few iterations of the trajectory, compromising the hole navigation.

In figure 4.11, the trajectory has been stopped after 500 predictions. For larger iterations, we observe that the vehicle continues steadily falling to increasingly lower altitudes. Once the quadrotor starts moving away from the final state, the outcome controls start becoming erratic. This behavior of the model function arises from the fact that the dataset $\mathcal{D}$ has been built taking into account only trajectories in which the initial and final states were arbitrarily close location-wise. Large $\hat{\mathbf{q}}$ inputs constitute unseen data for the DNN, which ultimately outputs rather random controls. Hence, the vehicle follows a random downward path, the only constant effect applied being gravity.

4.3.4 Discussing the obtained closed loop trajectories and controls

The trained DNN proves usable in a range of situations. Simple trajectories such as the straight take off line are well performed. More complicated navigations are successful as well, in particular when initial and final states have been drawn in the same way as for the training set.

Since the closed loop controls have a tendency to be bang-bang, the associated trajectories are usually less energy optimal. Regardless of this observation, in a vast range of tested navigations, the quadrotor successfully joins the required final state in a reasonable amount of time. However, for very specific and arbitrary $\mathbf{q}_f$, the vehicle doesn't manage to navigate towards the final location.

As a matter of consequence, we observe that the synthesized controller has specialized in a subset of possible trajectories and may need further improvements to manage more arbitrary initial and final states.

Chapter 5

Conclusion and future research

We summarize and discuss the research outcomes obtained previously. Subsequently, we present the remaining relevant questions to be answered and the future research axis to be followed.

5.1 Main results

In chapter 2, we introduced the planar-quadrotor model and derived optimality equations from the (PMP) conditions. In particular, we parameterized all optimal trajectories by 4 constants $c_1, c_2, c_3, c_4 \in \mathbb{R}$. By this mean, we materialized a latent space for optimal controls, each constants vector $\mathbf{c}$ fully determining a trajectory. This result connects our optimal control problem to the field of generative models, each optimal trajectory being mapped to a compressed representation lying in a subset of $\mathbb{R}^4$.

In order to cast a supervised learning problem, we devised a synthetic dataset generation process in chapter 3. We started exploring the latent space structure by looking to an example trajectory and comparing to the solver-based related one. We then formulated a series of procedures to enhance the predictors and to ensure data representivity. We proposed and implemented an algorithm efficient enough to produce a dataset $\mathcal{D}$ made of more than 100 million training points. A such enterprise would have been highly demanding if the training data had been generated by hard-solving the (OCP). Leveraging the theoretical framework allowed us to create synthetic data while minimizing wall time and computational power.

Finally, in chapter 4 we designed a deep neural network $\boldsymbol{\pi}$ to approximate optimal controls, using the previously produced dataset. We fine-tuned the different hyperparameters to obtain the best model possible, over more than 40 different tries. We minimized the mean squared error upon the validation set, in particular using specific regularization techniques. We assessed the trained model in a custom simulation environment on a set of autonomous navigations. Specifically, the synthesized deep learning feedback controller was able to perform autonomous flights between a wide range of initial and final states, while being close to optimality. However, some specific outlying initial and final states have proven more difficult to approach. Hence, we propose several future research axis to boost the performances of the controller and broaden our understanding of the autonomous quadrotor setting.

5.2 Future research

An intuitive way to improve the performances of the deep learning model would be to separate the controller in two different networks, each of them having only one output neuron specifically dedicated to u_T or u_R. Decorrelating the networks, doubling the parameters and restricting the set of weights and biases to predict only one control at a time may ease the training process and enhance the approximation made by the model.

Moreover, as seen in the example trajectory presented in 4.3.3.4, the state may not hold enough information to predict the correct instant controls. The time elapsed from the beginning of the navigation, i.e. the sub-phase in which the trajectory is currently in, may play a central role in forging the action to apply to the vehicle. Depending on the current time elapsed since the beginning of the manoeuvre, same states may produce different controls. Hence, future models may consider augmented predictors including the time variable, in the form of $\hat{\mathbf{q}} = (x_f - x,\, \dot{x}_f - \dot{x},\, z_f - z,\, \dot{z}_f - \dot{z},\, \cos(\theta_f - \theta),\, \sin(\theta_f - \theta),\, t)^\top$, practically having discretized $t_i = hi$ with h being the time step.

To take into account progressiveness and temporal dependence of the state control relationship, one can also make use of memory-full deep learning models. Algorithms such as long short-term memory (LSTM) arise with built-in feedback loops that are precisely appropriate for approximating closed loop controls. Such algorithms can process data in sequences and extract relevant information for an in-stream chain of data points.

Furthermore, the latent space described in chapter 2 remains largely unexplored. Revealing the structure of this strict subset of $\mathbb{R}^4$ can greatly benefit the dataset generation and ultimately lead to a better understanding of this particular compression phenomenon. Nonetheless, the relationship mapping initial and final states $(\mathbf{q}_0, \mathbf{q}_f)$ to the corresponding vector $\mathbf{c}$ is yet unknown. One approach could consist in approximating said relationship using a supplementary helper neural network, collecting pairs of initial and final states with the corresponding 4 constants. Yet, this collection cannot be made by sampling the initial states, solving (2.18) and collecting the final states. Indeed, such a process would de facto exclude all atypical trajectories such as 4.3.3.4 from the training set and would end up having the same biases as the ones detected in our model $\boldsymbol{\pi}$. However, going back to the theoretical analysis, **we observe that having the Hamiltonian** (2.9) **equal to zero along all optimal trajectory arises with having a linear relationship in the constants c_1, c_2, c_3, c_4 to be satisfied at all times**, i.e. having a matrix $\mathbf{A} = \mathbf{A}(t)$, a vector $\mathbf{b} = \mathbf{b}(t)$, such that $\mathbf{Ac} = \mathbf{b}$ is an overdetermined system valid for all $t \in [0, t_f]$. Thus, said helper network could be trained by sampling pairs of initial and final states $(\mathbf{q}_0, \mathbf{q}_f)$, hard-solving the optimal trajectories, and extracting the constants using, for instance, the linear least square method $\mathbf{c} \approx (\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top\mathbf{b}$. We attempted this procedure several times but it proved ineffective in practice, as the extraction of the constants is highly dependent on the quality of the solved trajectory. Although it works well for simple navigations, it fails to retrieve an acceptable vector $\mathbf{c}$ in more complicated cases. Future research may focus on fine-tuning the solver's settings to obtain better results for said procedure. Note that obtaining a direct relationship between $(\mathbf{q}_0, \mathbf{q}_f)$ and $\mathbf{c}$ is equivalent to obtaining the closed loop controls, by simply solving (2.18) afterwards. The feedback controller may be approximated using the principal neural network. The training data could be obtained in a chain operation : first sampling $(\mathbf{q}_0, \mathbf{q}_f)$ according to the users needs, including for example any atypical trajectory, secondly finding the corresponding latent constants with the helper neural network and thirdly generating the associated trajectory by solving the augmented system (2.18).

The quadrotor automation has been extensively studied under the framework of deep reinforcement learning. Such methods approximate controls using one or more neural networks that are progressively tailored to maximize a reward variable. Said reward is manually designed with great care, to guide the state to control policy towards some desired behavior. The models reinforce in a trial and error scheme, at first applying random controls before progressively shifting to outcomes that maximize the reward expectation. The models may have to do some extensive exploration before seeing some improvement patterns in the reward stream. This gives rise to a central dilemma in deep reinforcement learning which is the balancing between exploration and exploitation of beneficial patterns. This extensive trial and error process may come along with sizeable computation times and high computational power requirements. In particular, an interesting future research axis may be the assessment of an hybrid approach, in which the policy network is not initialized randomly at start, but is pre-trained using a supervised learning approach on an optimal control generated synthetic dataset. Hence, the model would pre-hold the seeds of optimality in its parameters and would benefit from a physics-informed warm-start at the beginning of the reinforcement learning phase. In this case, optimal control theory would help balance the exploration versus exploitation dilemma by reducing the exploration needs. This would mitigate the aforementioned downturns of deep reinforcement learning by kickstarting the model using elements of optimal control theory. Usually, the methods available in deep reinforcement learning apply to discrete action spaces, which is not relevant for our continuous controls. Nevertheless, said **optimal control boosted deep reinforcement learning** can be applied in our setting, using methods specifically designed for continuous action spaces [50]. In particular, as a starting point for future work, we implemented a deep deterministic policy gradient method (DDPG), using our kickstarted model $\boldsymbol{\pi}$. Our code can be found in appendix C.4, and is inspired from [51].

Lastly, although the analysis made in chapter 2 has shown the existence of singular arcs in the optimal controls, a final research axis may make the assumption of exclusively bang-bang controls. The regression task considered all along this project would become a classification task. Two networks for u_T and u_R would be used, each having two output neurons. By the means of a softmax activation function, said output neurons would hold a probability of having one of the two saturated bangs as the optimal response to the input state predictor. Applying deep reinforcement learning would become more straightforward as the action space would be discrete in such a context. In particular, in this situation, a policy gradient method would randomly sample an action according to the probability vector formed by the output neurons. This also opens the way to the optimal control boosted deep reinforcement learning strategy, and constitutes a relevant future approach to be undertaken for improving the autonomous navigation of quadrotors.

Appendix A

ICLOCS2 solver code

```matlab
function [dx] = QuadrotorTrajectory_Dynamics_Internal(x,u,p,t,data)
% Imperial College London
% MSc Applied Mathematics
% This code has been written as part of the MSc project 'Deep Neural Networks
% for Real-time Trajectory Planning'
% Author : Amaury FRANCOU - CID: 01258326
% Supervisor : Dr Dante KALISE
%
% This code uses the ICLOCS2 optimization based control software in Matlab/Simulink
% (http://www.ee.ic.ac.uk/ICLOCS/default.htm).
%
% It has been inspired by the Two-link robot arm example problem
% found on the ICLOCS2 website
% (http://www.ee.ic.ac.uk/ICLOCS/ExampleRobotArm.html) and written by
% Yuanbo Nie, Omar Faqir, and Eric Kerrigan.
%
% Syntax:
%          [dx] = QuadrotorTrajectory_Dynamics_Internal(x,u,p,t,vdat) (Dynamics Only)
%
% Inputs:
%    x  - state vector
%    u  - input
%    p  - parameter
%    t  - time
%    vdat - structured variable containing the values of additional data used inside
%          the function
% Output:
%    dx - time derivative of x
%

% State variables
x1 = x(:,1); % x
x2 = x(:,2); % xDot
x3 = x(:,3); % z
x4 = x(:,4); % zDot
x5 = x(:,5); % theta

% Controls
uT = u(:,1);
uR = u(:,2);

% Dynamics

dx(:,1) = x2;

dx(:,2) = uT .* sin(x5);

dx(:,3) = x4;

dx(:,4) = uT .* cos(x5) - 9.81;

dx(:,5) = uR;
```

```matlab
function [dx] = QuadrotorTrajectory_Dynamics_Sim(x,u,p,t,data)
% Imperial College London
% MSc Applied Mathematics
% This code has been written as part of the MSc project 'Deep Neural Networks
% for Real-time Trajectory Planning'
% Author : Amaury FRANCOU - CID: 01258326
% Supervisor : Dr Dante KALISE
%
% This code uses the ICLOCS2 optimization based control software in Matlab/Simulink
% (http://www.ee.ic.ac.uk/ICLOCS/default.htm).
%
% It has been inspired by the Two-link robot arm example problem
% found on the ICLOCS2 website
% (http://www.ee.ic.ac.uk/ICLOCS/ExampleRobotArm.html) and written by
% Yuanbo Nie, Omar Faqir, and Eric Kerrigan.
%
% Syntax:
%          [dx] = QuadrotorTrajectory_Dynamics_Sim(x,u,p,t,vdat)  (Dynamics Only)
%
% Inputs:
%    x  - state vector
%    u  - input
%    p  - parameter
%    t  - time
%    vdat - structured variable containing the values of additional data used
   inside
%          the function
% Output:
%    dx - time derivative of x
%

% State variables
x1 = x(:,1); % x
x2 = x(:,2); % xDot
x3 = x(:,3); % z
x4 = x(:,4); % zDot
x5 = x(:,5); % theta

% Controls
uT = u(:,1);
uR = u(:,2);

% Dynamics

dx(:,1) = x2;

dx(:,2) = uT .* sin(x5);

dx(:,3) = x4;

dx(:,4) = uT .* cos(x5) - 9.81;

dx(:,5) = uR;
```

```matlab
function options = settings_QuadrotorTrajectory(varargin)
% Imperial College London
% MSc Applied Mathematics
% This code has been written as part of the MSc project 'Deep Neural Networks
% for Real-time Trajectory Planning'
% Author : Amaury FRANCOU - CID: 01258326
% Supervisor : Dr Dante KALISE
%
% This code uses the ICLOCS2 optimization based control software in Matlab/Simulink
% (http://www.ee.ic.ac.uk/ICLOCS/default.htm).
%
% It has been inspired by the Two-link robot arm example problem
% found on the ICLOCS2 website
% (http://www.ee.ic.ac.uk/ICLOCS/ExampleRobotArm.html) and written by
% Yuanbo Nie, Omar Faqir, and Eric Kerrigan.
%
% This function provides various settings to the solver.
```

```
%
% Syntax:  options = settings_QuadrotorTrajectory(varargin)
%          When giving one input with varargin, e.g. with settings(20), will use h-method of your choice with N=20 nodes
%          When giving two inputs with varargin, hp-LGR method will be used with two possibilities
%           - Scalar numbers can be used for equally spaced intervals with the same polynoimial orders. For example, settings_hp(5,4) means using 5 LGR intervals each of polynomial degree of 4.
%           - Alternatively, can supply two arrays in the argument with customized meshing. For example, settings_hp([-1 0.3 0.4 1],[4 5 3]) will have 3 segments on the normalized interval [-1 0.3], [0.3 0.4] and [0.4 1], with polynomial order of 4, 5, and 3 respectively.
%
% Output:
%    options - Structure containing the settings
%


%% Transcription Method

% Direct_collocation
options.transcription = 'direct_collocation';

% Integrated residual minimization : alternating method
options.min_res_mode = 'alternating';

% Priorities for the solution property : lower integrated residual error
options.min_res_priority = 'low_res_error';

% Error criteria : local absolute error
options.errortype='local_abs';


%% Discretization Method

% Discretization method : Hermite-Simpson method
options.discretization = 'hermite';

% Result Representation: direct interpolation in correspondence with the transcription method
options.resultRep = 'default';

%% Derivative generation

% Derivative computation method : finite differences  ('numeric')
options.derivatives = 'numeric';
options.adigatorPath = '../../adigator';

% Perturbation sizes for numerical differentiation
options.perturbation.H = [];
options.perturbation.J = [];

%% NLP solver

% NLP solver : IPOPT
options.NLPsolver = 'ipopt';

% IPOPT settings
options.ipopt.tol = 1e-9;                        % Convergence tolerance (relative).
options.ipopt.print_level = 5;                   % Print level.
options.ipopt.max_iter = 5000;                   % Maximum number of iterations.

options.ipopt.mu_strategy = 'adaptive';         % Adaptive update strategy.

options.ipopt.hessian_approximation = 'exact';   %  Use second derivatives provided by ICLOCS.

options.ipopt.limited_memory_max_history = 6;   % Maximum size of the history for the limited quasi-Newton Hessian approximation.

options.ipopt.limited_memory_max_skipping = 1;  % Threshold for successive
```

```
    iterations where update is skipped for the quasi-Newton approximation.


%% Meshing Strategy

% Type of meshing : with local refinement
options.meshstrategy = 'mesh_refinement';

% Mesh Refinement Method : Automatic refinement
options.MeshRefinement = 'Auto';

% Mesh Refinement Preferences : efficient
options.MRstrategy = 'efficient';

% Maximum number of mesh refinement iterations
options.maxMRiter = 50;

% Discountious Input
options.disContInputs = 0;

% Minimum and maximum time interval
options.mintimeinterval = 0.001;
options.maxtimeinterval = inf;

% Distribution of integration steps : equispaced steps.
options.tau = 0;


%% Other Settings

% Cold/Warm/Hot Start
options.start = 'Cold';

% Automatic scaling
options.scaling = 0;

% Reorder of LGR Method
options.reorderLGR = 0;

% Early termination of residual minimization if tolerance is met
options.resminEarlyStop = 0;

% External Constraint Handling
options.ECH.enabled = 0;

% A conservative buffer zone
options.ECH.buffer_pct = 0.1;

% Regularization Strategy
options.regstrategy = 'off';

% LEAVE THIS PART UNCHANGED AND USE FUNCTION SYNTAX (AS DESCRIBED ON THE TOP) TO
    DEFINE THE ITEGRATION NODES
if nargin==2
    if strcmp(varargin{2},'h')
        options.nodes=varargin{1};
        options.discretization='hermite';
    else
        if length(varargin{1})==1
            options.nsegment=varargin{1};
            options.pdegree=varargin{2};
        else
            options.tau_segment=varargin{1};
            options.npsegment=varargin{2};
        end
        options.discretization='hpLGR';
    end
else
    options.nodes=varargin{1};
end


%% Output settings
```

```

% Display computation time
options.print.time = 1;

% Display relative local discretization error
options.print.relative_local_error = 1;

% Display cost (objective) values
options.print.cost = 1;


% Plot figures : plot all figures
options.plot = 1;
```

```
function [problem,guess] = QuadrotorTrajectory
% Imperial College London
% MSc Applied Mathematics
% This code has been written as part of the MSc project 'Deep Neural Networks
% for Real-time Trajectory Planning'
% Author : Amaury FRANCOU - CID: 01258326
% Supervisor : Dr Dante KALISE
%
% This code uses the ICLOCS2 optimization based control software in Matlab/Simulink
% (http://www.ee.ic.ac.uk/ICLOCS/default.htm).
%
% It has been inspired by the Two-link robot arm example problem
% found on the ICLOCS2 website
% (http://www.ee.ic.ac.uk/ICLOCS/ExampleRobotArm.html) and written by
% Yuanbo Nie, Omar Faqir, and Eric Kerrigan.
%
% Syntax:  [problem,guess] = QuadrotorTrajectory
%
% Outputs:
%    problem - Structure with information on the optimal control problem
%    guess   - Guess for state, control and multipliers.
%

% Defining q0 and qf
q0 = [ 9.481  -1.117  -4.29    7.135   0.86 ];
qf = [ 10.641    1.871   -3.779  -10.123   -0.869];

% Plant model name, used for Adigator
InternalDynamics = @QuadrotorTrajectory_Dynamics_Internal;
SimDynamics = @QuadrotorTrajectory_Dynamics_Sim;

% Settings file
problem.settings = @settings_QuadrotorTrajectory;

% Initial time t0.
problem.time.t0_min = 0;
problem.time.t0_max = 0;
guess.t0 = 0;

% Final time.
problem.time.tf_min=0.01;
problem.time.tf_max=inf;
guess.tf = 2; % Guessing final time

% Initial conditions for system q0.
problem.states.x0 = q0;

% Initial conditions for system.
problem.states.x0l = q0;
problem.states.x0u = q0;

% State bounds.
problem.states.xl = [-inf -inf -inf -inf -inf];
problem.states.xu = [inf inf inf inf inf];

% State error bounds
problem.states.xErrorTol_local = [0.0001 0.0001 0.0001 0.0001 0.0001];
```

```
problem.states.xErrorTol_integral = [0.0001 0.0001 0.0001 0.0001 0.0001];
% State constraint error bounds
problem.states.xConstraintTol = [0.0001 0.0001 0.0001 0.0001 0.0001];

% Terminal state bounds qf.
problem.states.xfl = qf;
problem.states.xfu = qf;

% Guess the state trajectories with [x0 xf]
guess.states(:,1)=[q0(1) qf(1)];
guess.states(:,2)=[q0(2)  qf(2)];
guess.states(:,3)=[q0(3)  qf(3)];
guess.states(:,4)=[q0(4) qf(4)];
guess.states(:,5)=[q0(5) qf(5)];

% Number of control actions N
problem.inputs.N = 0;

% Input bounds
uTmax = 14;
uTmin = 0.8;
uRmax = 4.6;

problem.inputs.ul = [uTmin -uRmax];
problem.inputs.uu = [uTmax uRmax];

problem.inputs.u0l = [uTmin 0]; % uR(0) = 0
problem.inputs.u0u = [uTmax 0];

% Input constraint error bounds
problem.inputs.uConstraintTol = [0.0001 0.0001];

% Guess the input sequences with [u0 uf]
guess.inputs(:,1) = [uTmax uTmax];
guess.inputs(:,2) = [0 0];

% Not required
problem.parameters.pl=[];
problem.parameters.pu=[];
guess.parameters=[];
problem.setpoints.states=[];
problem.setpoints.inputs=[];
problem.constraints.ng_eq=0;
problem.constraints.gTol_eq=[];
problem.constraints.gl=[];
problem.constraints.gu=[];
problem.constraints.gTol_neq=[];
problem.constraints.bl=[];
problem.constraints.bu=[];
problem.constraints.bTol=[];

% Problem parameters used in the functions
% Get function handles and return to Main.m
problem.data.InternalDynamics = InternalDynamics;
problem.data.functionfg  =@fg;
problem.data.plantmodel = func2str(InternalDynamics);
problem.functions = {@L,@E,@f,@g,@avrc,@b};
problem.sim.functions = SimDynamics;
problem.sim.inputX = [];
problem.sim.inputU = 1:length(problem.inputs.ul);
problem.functions_unscaled = {@L_unscaled,@E_unscaled,@f_unscaled,@g_unscaled,@avrc
    ,@b_unscaled};
problem.data.functions_unscaled = problem.functions_unscaled;
problem.data.ng_eq = problem.constraints.ng_eq;
problem.constraintErrorTol = ...
    [problem.constraints.gTol_eq,problem.constraints.gTol_neq,...
    problem.constraints.gTol_eq,problem.constraints.gTol_neq,problem.states.
    xConstraintTol,...
    problem.states.xConstraintTol,problem.inputs.uConstraintTol,problem.inputs.
    uConstraintTol];


function stageCost = L_unscaled(x,xr,u,ur,p,t,vdat)
```

```
% Returns the running cost.
%
% Syntax:  stageCost = L(x,xr,u,ur,p,t,data)
%
% Inputs:
%    x  - state vector
%    xr - state reference
%    u  - input
%    ur - input reference
%    p  - parameter
%    t  - time
%    data- structured variable containing the values of additional data used inside
%          the function
%
% Output:
%    stageCost - Scalar or vectorized stage cost
%

stageCost =(u(:,1).*u(:,1)+u(:,2).*u(:,2));


function boundaryCost=E_unscaled(x0,xf,u0,uf,p,t0,tf,vdat)

% Returns the boundary value cost.
%
% Syntax:  boundaryCost = E(x0,xf,u0,uf,p,tf,data)
%
% Inputs:
%    x0  - state at t=0
%    xf  - state at t=tf
%    u0  - input at t=0
%    uf  - input at t=tf
%    p   - parameter
%    tf  - final time
%    data- structured variable containing the values of additional data used inside
%          the function
%
% Output:
%    boundaryCost - Scalar boundary cost
%

boundaryCost = tf;


function bc = b_unscaled(x0,xf,u0,uf,p,t0,tf,vdat,varargin)

% Not useful here. - Returns a column vector containing the evaluation of the
% boundary constraints.
%
% Syntax:  bc=b(x0,xf,u0,uf,p,tf,data)
%
% Inputs:
%    x0  - state at t=0
%    xf  - state at t=tf
%    u0  - input at t=0
%    uf  - input at t=tf
%    p   - parameter
%    tf  - final time
%    data- structured variable containing the values of additional data used inside
%          the function
%
%
% Output:
%    bc - column vector containing the evaluation of the boundary function
%

bc = [];
```

```
% Imperial College London
% MSc Applied Mathematics
```

```
% This code has been written as part of the MSc project 'Deep Neural Networks
% for Real-time Trajectory Planning'
% Author : Amaury FRANCOU - CID: 01258326
% Supervisor : Dr Dante KALISE
%
% This code uses the ICLOCS2 optimization based control software in Matlab/Simulink
% (http://www.ee.ic.ac.uk/ICLOCS/default.htm).
%
% It has been inspired by the Two-link robot arm example problem
% found on the ICLOCS2 website
% (http://www.ee.ic.ac.uk/ICLOCS/ExampleRobotArm.html) and written by
% Yuanbo Nie, Omar Faqir, and Eric Kerrigan.
%
% This is the main solver script.



clear all;
close all;
format compact;

[problem,guess] = QuadrotorTrajectory;          % Problem definition
options = problem.settings(20);                 % Get options and solver settings
[solution,MRHistory] = solveMyProblem(problem,guess,options);
[ tv, xv, uv ] = simulateSolution( problem, solution, 'ode113', 0.001 );

%% figure

xx = linspace(solution.T(1,1),solution.T(end,1),100);

figure
plot(speval(solution,'X',1,xx), speval(solution,'X',3,xx), 'r-' )
xlabel('x [m]')
ylabel('z [m]')
grid on

figure
plot(xx,speval(solution,'U',1,xx),'b-' )
hold on
plot(xx,speval(solution,'U',2,xx),'r-' )
plot(tv,uv(:,1),'k-.')
plot(tv,uv(:,2),'k-.')
plot([solution.T(1,1); solution.tf],[problem.inputs.ul(1), problem.inputs.ul(1)],'r
    -' )
plot([solution.T(1,1); solution.tf],[problem.inputs.uu(1), problem.inputs.uu(1)],'r
    -' )
xlim([0 solution.tf])
xlabel('Time [s]')
grid on
ylabel('Control Input')
legend('uT','uR')
```

Appendix B

Dataset generation code

B.1 Solving (2.18)

```
"""
Imperial College London
MSc Applied Mathematics
This code has been written as part of the MSc project 'Deep Neural Networks
for Real-time Trajectory Planning'
Author : Amaury FRANCOU - CID: 01258326
Supervisor : Dr Dante KALISE
"""

# Imports
import numpy as np
from scipy.integrate import solve_ivp


# Setting constants
g = 9.81
uTmax = 14
uTmin = 0.8
uRmax = 4.6


def aFunc(t, y, c) :
    """
    This function computes the value of the modulating function a at time t,
    according to (2.12).

    Parameters
    ----------
    t : float - the time value
    y : 6-dimensional numpy array - the augmented state vector
    c : 4-dimensional numpy array - the costate constants
    Returns
    -------
    a : float - the modulating function a at time t
    """
    return - 0.5 * ((c[1] - c[0] * t) * np.sin(y[4]) + (c[3] - c[2] * t) * np.cos(y
    [4]))

def aFuncVec(tf, N, y, c) :
    """
    This function computes the value of the modulating function a,
    according to (2.12), in a vectorized fashion.

    Parameters
    ----------
    tf : float - the final time value
    N : integer - the number of evaluation points
    y : 6xN-dimensional numpy array - the augmented state vectors on [0,tf]
    c : 4-dimensional numpy array - the costate constants
    Returns
    -------
    aVec : N-dimensional numpy array - the modulating function a on [0,tf]
```

```
    """

    time = np.linspace(0,tf,N)
    return - 0.5 * ((c[1] - c[0] * time) * np.sin(y[4,:]) \
                   + (c[3] - c[2] * time) * np.cos(y[4,:]))


def uTcontrol(a, uTmin = uTmin, uTmax = uTmax) :
    """
    This function computes the thrust control as defined in (2.13).

    Parameters
    ----------
    a : float - the modulating function value a(t)
    uTmin : float - the minimal admissible thrust control
    uTmax : float - the maximal admissible thrust control

    Returns
    -------
    uTcontrol : float - the computed thrust control
    """
    return min(max(a, uTmin), uTmax)


def uTcontrolVec(aVec, uTmin = uTmin, uTmax = uTmax) :
    """
    This function computes the thrust controls as defined in (2.13),
    in a vectorized fashion.

    Parameters
    ----------
    aVec  : N-dimensional numpy array - the modulating function a on [0,tf]
    uTmin : float - the minimal admissible thrust control
    uTmax : float - the maximal admissible thrust control

    Returns
    -------
    uTcontrolVec : N-dimensional numpy array - the computed thrust controls on [0,tf]
    """
    return np.clip(aVec, uTmin, uTmax)


def uRcontrol(b, uRmax = uRmax) :
    """
    This function computes the torque control as defined in (2.16).

    Parameters
    ----------
    b : float - the modulating function value b(t)
    uRmax : float - the maximal admissible torque control

    Returns
    -------
    uRcontrol : float - the computed torque control
    """
    return min(max(b, -uRmax), uRmax)


def uRcontrolVec(bVec, uRmax = uRmax) :
    """
    This function computes the torque control as defined in (2.16),
    in a vectorized fashion.

    Parameters
    ----------
    bVec : N-dimensional numpy array - the modulating function value b on [0,tf]
    uRmax : float - the maximal admissible torque control

    Returns
    -------
    uRcontrolvec : N-dimensional numpy array - the computed torque controls on [0,tf]
```

```
    """
    return np.clip(bVec, -uRmax, uRmax)


def solve(q0, c, tf, N, extractControls = False) :
    """
    This function solves the initial value problem defined in (2.18) for a target
    final time tf.

    Parameters
    ----------
    q0 : 5-dimensional numpy array - the initial state vector
    c : 4-dimensional numpy array - the costate constants
    tf : float - the final time value
    N : integer - the number of evaluation points
    extractControls - boolean - if set to True the function returns the controls

    Returns
    -------
    sol : scipy OdeSolution instance - sol.y contains the values of the solution
    u : 2xN-dimensional numpy array - the controls on [0,tf]
    """

    def fDynamics(t, y, c = c) :
        """
        This function computes the derivative of the augmented state vector according
        to (2.18).

        Parameters
        ----------
        t : float - the time value
        y : 6-dimensional numpy array - the augmented state vector
        c : 4-dimensional numpy array - the costate constants

        Returns
        -------
        f : 6-dimensional numpy array - the augmented state vector derivative
        """

        # Computing controls
        uT = uTcontrol(a = aFunc(t = t, y = y, c = c))
        uR = uRcontrol(b = y[5])

        return np.array([y[1], uT * np.sin(y[4]), y[3], uT * np.cos(y[4]) - g, uR, \
                         0.5 * uT * ((c[1] - c[0] * t) * np.cos(y[4])  \
                                     - (c[3] - c[2] * t) * np.sin(y[4]))])

    # Initial value of augmented state vector
    y0 = np.append(q0,0)
    # Scipy IVP solver using explicit Runge-Kutta method of order 5(4)
    sol = solve_ivp(fDynamics, [0, tf], y0, method = 'RK45', max_step = 1e-2, \
                    t_eval = np.linspace(0,tf,N))

    # Controls extraction
    if extractControls :
        # uT
        aVec = aFuncVec(tf = tf, N = N, y = sol.y, c = c)
        print(aVec)
        uTcontrols = uTcontrolVec(aVec)
        # uR
        uRcontrols = uRcontrolVec(bVec = sol.y[5,:])
        # u = [uT, uR]
        u = np.row_stack((uTcontrols,uRcontrols))

        return sol, u

    return sol
```

B.2 Generating dataset $\mathcal{D}$

```
"""
Imperial College London
MSc Applied Mathematics
This code has been written as part of the MSc project 'Deep Neural Networks
for Real-time Trajectory Planning'
Author : Amaury FRANCOU - CID: 01258326
Supervisor : Dr Dante KALISE
"""

# Imports
import numpy as np
from IVPsolver import *
from numpy import random
from math import pi
import uuid
import json
from progress.bar import FillingCirclesBar
import time

# Computing only once
twoPi = 2 * pi
piOn2 = pi / 2
threePiOn2 = (3 * pi) / 2

# Initializing random number generator
random.seed(1008*1996)

def parameterDraw() :
    """
    This function performs the parameters draw for the trajectory generation.

    Parameters
    ----------
    None

    Returns
    -------
    c : 4-dimensional numpy array - the costate constants
    q0 : 5-dimensional numpy array - the initial state vector
    tf : float - the final time value
    """

    # Generating c
    c = random.normal(loc = 0, scale = 200, size = 4)

    # Generating q0
    x0 = random.normal(loc = 0, scale = 4.0)
    xDot0 = random.normal(loc = 0, scale = 4.0)
    z0 = random.normal(loc = 0, scale = 4.0)
    zDot0 = random.normal(loc = 0, scale = 4.0)
    # Better chances having the UAV not upside down at start
    possibleTheta0 = [random.uniform(low = 0, high = piOn2), \
                      random.uniform(low = threePiOn2, high = twoPi), \
                          random.uniform(low = piOn2, high = threePiOn2)]
    theta0 = random.choice(possibleTheta0, p = [0.325, 0.325, 0.35])
    q0 = np.array([x0, xDot0, z0, zDot0, theta0])

    # Generating tf
    tf = random.lognormal(mean = 0.75, sigma = 0.20)

    return c, q0, tf


def qfLocation(q0, qf) :
    """
    This function returns the location of the terminal state position (xf,zf)
    with respect to the initial position (x0,z0).

    Parameters
    ----------
    q0 : 5-dimensional numpy array - the initial state vector
```

```python
    qf : 5-dimensional numpy array - the final state vector

    Returns
    -------
    location : string - the terminal state position
    """
    if qf[0] >= q0[0] :
        if qf[2] >= q0[2] :
            return 'upper right'
        else :
            return 'lower right'
    else :
        if qf[2] >= q0[2] :
            return 'upper left'
        else :
            return 'lower left'

def distribInit() :
    """
    This function initializes the distribution dictionnary of final positions
    with respect to corresponding initial positions.

    Parameters
    ----------
    None

    Returns
    -------
    distrib : dictionnary - the initialized distribution of final positions
        with respect to corresponding initial positions.
    """

    return {'upper left' : {'num' : 0, 'full' : False},
            'lower left' : {'num' : 0, 'full' : False},
            'upper right' : {'num' : 0, 'full' : False},
            'lower right' : {'num' : 0, 'full' : False}}


def qfAdmissible(q0, qf, maxDist, distrib, maxPerLoc) :
    """
    This function returns True if the final state is admissible regarding
    our arbitrary conditions.

    Parameters
    ----------
    q0 : 5-dimensional numpy array - the initial state vector
    qf : 5-dimensional numpy array - the final state vector
    maxDist : float - the maximum distance authorized between the initial
        and final positions
    distrib : dictionnary - the current distribution of final positions
        with respect to corresponding initial positions
    maxPerLoc : integer - the maximum final positions per location

    Returns
    -------
    admissibility : boolean - set to True if the given qf is admissible in the dataset
    """

    # Testing distance
    diff = qf - q0
    xzDiff = np.array([diff[0], diff[2]])
    if np.linalg.norm(xzDiff) > maxDist :
        return False

    # Testing location
    loc = qfLocation(q0, qf)
    if distrib[loc]['full'] :
        return False
    else :
        distrib[loc]['num'] += 1
        if distrib[loc]['num'] == maxPerLoc :
            distrib[loc]['full'] = True
```

```python
        return True


def generateBatch(targetBatchSize, N, maxDist) :
    """
    This function generates a batch of trajectories and controls, computed using
    (2.18).

    Parameters
    ----------
    targetBatchSize : integer - the number of trajectories and controls samples
        in the batch, must be divisible by 4
    N : integer - the number of evaluation points
    maxDist : float - the maximum distance authorized between the initial
        and final positions

    Returns
    -------
    None
    """

    # Divisibility by 4 required
    targetBatchSize = targetBatchSize - targetBatchSize % 4

    # Initialization
    currentBatchSize = 0
    distrib = distribInit()
    batch = {}

    # Evenly distributed final positions with respect to initial positions
    maxPerLoc = int(targetBatchSize / 4)

    # Progress bar
    bar = FillingCirclesBar('Processing current batch', max = targetBatchSize)

    # Generating batch
    while currentBatchSize < targetBatchSize :

        # Drawing parameters
        c, q0, tf = parameterDraw()

        # Solving IVP
        sol, u = solve(q0, c, tf, N, extractControls = True)
        qf = sol.y[:5,N-1]

        # Verifying if qf is admissible
        if qfAdmissible(q0, qf, maxDist, distrib, maxPerLoc) :

            currentBatchSize += 1
            bar.next()
            trajId = str(uuid.uuid4().int)[:12] # Unique id for each trajectory

            # Storing values
            batch[trajId] = {}
            batch[trajId]['q'] = sol.y.tolist()
            batch[trajId]['u'] = u.tolist()
            batch[trajId]['c'] = c.tolist()


    # Save as you go
    batchId = str(uuid.uuid4().int)[:6]
    fileDir = '/.../.../.../.../' \
    '.../trajectoryBatches.nosync/'
    with open(fileDir + batchId + '.json', 'w') as file:
        json.dump(batch, file)

    bar.finish()
    return

# Settings
nbBatches = 250
targetBatchSize = 100
```

```python
N = 100
maxDist = 6.0


def generateDataset(nbBatches = nbBatches, targetBatchSize = targetBatchSize, \
                    N = N, maxDist = maxDist) :
    """
    This function generates the synthetic trajectories and controls dataset.

    Parameters
    ----------
    nbBatches : integer - the number of batches required
    targetBatchSize : integer - the number of trajectories and controls samples
        in the batch, must be divisible by 4
    N : integer - the number of evaluation points
    maxDist : float - the maximum distance authorized between the initial
        and final positions

    Returns
    -------
    None
    """

    print('')
    for batchNb in range(nbBatches):
        print('----------------------------------------')
        print('Batch number ' + str(batchNb + 1) + '/' + str(nbBatches))
        t0 = time.time()
        # Generate batch
        generateBatch(targetBatchSize, N, maxDist)
        t1 = time.time()
        print('Completed in ' + str(t1-t0)[:5] +'s')

    print('........................................')
    print('........................................')
    print('Dataset generated')
    print('')


if __name__ == '__main__' :
    generateDataset()
```

B.3 Preparing and post-processing dataset $\mathcal{D}$

```python
"""
Imperial College London
MSc Applied Mathematics
This code has been written as part of the MSc project 'Deep Neural Networks
for Real-time Trajectory Planning'
Author : Amaury FRANCOU - CID: 01258326
Supervisor : Dr Dante KALISE
"""

# Imports
import numpy as np
from os import listdir
from os.path import isfile, join
import json
from math import pi
from progress.bar import ChargingBar
from sklearn.model_selection import train_test_split
import copy


def encodeTheta(theta, pi = pi) :
    """
    This function encodes the angle theta for use as input of the neural network.

    Parameters
    ----------
```

```
    theta : float - the angle

    Returns
    -------
    sHat : float - the first encoded number
    sCheck : float - the second encoded number

    """

    return np.cos(theta), np.sin(theta)

def prepareDataset(datasetPath) :
    """
    This function prepares the dataset in order to be ready for use in tensorflow
    model.

    Parameters
    ----------
    datasetPath : string - the absolute path to the dataset files

    Returns
    -------
    X_train : Nx5-dimensional numpy array - the states used for training
    X_test : Nx5-dimensional numpy array - the states used for testing
    Y_train : Nx2-dimensional numpy array - the controls used for training
    Y_test : Nx2-dimensional numpy array - the controls used for testing

    """

    # Getting batch file names
    batchList = [fileName for fileName in listdir(datasetPath) if isfile(join(datasetPath, fileName))]
    batchList.remove('.DS_Store')
    batchFiles = [datasetPath + '/' + batchName for batchName in batchList]

    # Computing only once
    twoPi = 2 * pi

    # Retrieval of the number of points in each trajectories
    batchOne = open(batchFiles[0])
    batchOneData = json.load(batchOne)
    firstTrajId = next(iter(batchOneData))
    N = np.array(batchOneData[firstTrajId]['q']).shape[1]

    # Total size of dataset
    lenBatchList = len(batchList)
    pointsPerTraj = int((N-1)*N/2)
    trajPerFile = len(batchOneData)
    totalPoints = pointsPerTraj * trajPerFile * lenBatchList
    print('Total number of points in dataset : ', totalPoints)

    # Preparing storage
    qStorage = np.zeros(shape = (totalPoints,6))
    uStorage = np.zeros(shape = (totalPoints,2))

    # Progress bar
    print('...........................................')
    print('...........................................')
    print('Loading data')
    bar = ChargingBar('Loading data', max = lenBatchList)

    for fileNum, batchFile in enumerate(batchFiles) :

        # Batch loading
        batch = open(batchFiles[0])
        batchData = json.load(batch)
        bar.next()

        for batchNum, trajId in enumerate(batchData) :

            # For stacking
            startAt = 0 + batchNum * pointsPerTraj + fileNum * trajPerFile * pointsPerTraj
```

```
            stopAt = N-1 + batchNum * pointsPerTraj + fileNum * trajPerFile *
    pointsPerTraj

            # Extracting arrays of trajectory
            q = np.array(batchData[trajId]['q'])[:5]
            u = np.array(batchData[trajId]['u'])

            # Augmentation procedure
            for k in range(N-1) :
                k+=1

                # Subtrajectory
                uSubtraj = copy.deepcopy(u[:,:N-k])
                qSubtraj = copy.deepcopy(q[:,:N-k+1])

                # Last state in subtrajectory
                qf = copy.deepcopy(q[:,N - k])
                qf = qf.reshape((5,1))
                qSubtraj = qf - qSubtraj # Considering the difference with qf
                qSubtraj = qSubtraj[:,:N-k]
                qSubtraj[4,:] = np.mod(qSubtraj[4,:], twoPi) # Theta in [0,2pi)

                # Encoding
                cosTheta, sinTheta = encodeTheta(qSubtraj[4,:])
                qSubtrajAugm = np.row_stack((qSubtraj[:4,:],cosTheta))
                qSubtrajAugm = np.row_stack((qSubtrajAugm,sinTheta))

                # Shaping the data for tensorflow
                qSubtrajAugm = qSubtrajAugm.T
                uSubtraj = uSubtraj.T

                # Adding to numpy storage
                qStorage[startAt:stopAt, :] = qSubtrajAugm
                uStorage[startAt:stopAt, :]  = uSubtraj

                # Updating
                startAt = stopAt
                stopAt += (N-k-1)

                # Flushing variables
                del uSubtraj, qSubtraj, qf, cosTheta, sinTheta, qSubtrajAugm

    bar.finish()

    # Splitting train and test sets with shuffling
    X_train, X_test, Y_train, Y_test = \
        train_test_split(qStorage, uStorage, test_size = 0.15, shuffle = True)


    print('Data loaded')
    print('.........................................')
    print('.........................................')

    assert X_train.shape[0] + X_test.shape[0] == totalPoints

    print('Total number of training points : ', X_train.shape[0])
    print('Total number of testing points : ', X_test.shape[0])

    return X_train, X_test, Y_train, Y_test
```

Appendix C

Neural network training and evaluation

C.1 Training the deep neural network

```
"""
Imperial College London
MSc Applied Mathematics
This code has been written as part of the MSc project 'Deep Neural Networks
for Real-time Trajectory Planning'
Author : Amaury FRANCOU - CID: 01258326
Supervisor : Dr Dante KALISE
"""

# Imports
import numpy as np
import tensorflow as tf
import keras.backend as K
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input, Dropout
import os
from datasetPreparation import prepareDataset
from tensorflow.keras.layers import Activation
from tensorflow.keras.models import model_from_json

input_shape = (6,) # Inputs are state vectors

# Constants
uTmax = 14
uTmin = 0.8
uRmax = 4.6

##############################################################################
# Choose dataset old/new
whichDataset = 'old'
# Settings
training_number = 47
batch_size = 128
numberEpochs = 50
numberNeurons = 900
patience = 3
learning_rate = 1e-3
# Load previous model to enhance
loadModel = False
modelNumber = 34
##############################################################################


def controlsActivation(x, uTmin = uTmin, uTmax = uTmax, uRmax = uRmax) :
    """
    This function performs the activation for the output layer of the neural
    network based on the controls limits using a custom sigmoid function.

    Parameters
```

```
    ----------
    x  : Mx2-dimensional numpy array - the pre-activations for the output layer over all the batch
    uTmin : float - the minimal admissible thrust control
    uTmax : float - the maximal admissible thrust control
    uRmax : float - the maximal admissible torque control

    Returns
    -------
    activations : Mx2-dimensional numpy array - the activations for the output layer over all the batch
    """

    # Neurons
    n0 = x[:,0:1] # Shape is (batch_size, 1)
    n1 = x[:,1:2]

    # Neuron 0 provides uT
    x0 = ((uTmax-uTmin) * K.sigmoid(n0) + uTmin)
    # Neuron 1 provides uR
    x1 = uRmax * (2 * K.sigmoid(n1) - 1)

    return K.concatenate([x0,x1], axis = -1)


# Defining our tensorflow neural network model
model = Sequential([ # Tensorflow model
        Input(shape = input_shape), # Fully-connected layer
        Dense(numberNeurons, activation = 'relu'),
        Dropout(0.25),
        Dense(numberNeurons, activation = 'relu'),
        Dropout(0.25),
        Dense(numberNeurons, activation = 'relu'),
        Dropout(0.25),
        Dense(numberNeurons, activation = 'relu'),
        Dropout(0.25),
        Dense(numberNeurons, activation = 'relu'),
        Dense(2, activation = controlsActivation) # Output layer
    ])


# Using stochastic gradient descent
learning_rate = learning_rate
opt = tf.keras.optimizers.SGD(learning_rate = learning_rate)
# Load previous weights
if loadModel :
    modelNumber = str(modelNumber)
    pathToModel = '//../../../../...'\
        '.../../.../training_' + modelNumber
    json_file = open(pathToModel + '/model' + modelNumber + '.json', 'r')
    loaded_model_json = json_file.read()
    json_file.close()
    model = model_from_json(loaded_model_json,\
                            custom_objects={'controlsActivation': Activation(controlsActivation)})
    model.load_weights(pathToModel + '/model' + modelNumber + '.h5')
# Using mean squared error loss
model.compile(loss = 'mse', optimizer = opt, metrics = ['mse', 'mae', 'mape'])


# Saving model weights along the way
training_number = str(training_number)
filePath = "Trained Models/training_" + training_number +'/'
checkpoint_path = filePath + "cp.ckpt"
checkpoint_dir = os.path.dirname(checkpoint_path)
cp_callback = tf.keras.callbacks.ModelCheckpoint(filepath = checkpoint_path,
                                                 save_weights_only = True,
                                                 verbose = 1)

# Perform early stopping
es_callback = tf.keras.callbacks.EarlyStopping(monitor = 'val_loss', patience = patience)

```

```
datasetPath = '/.../.../.../' \
'.../.../.../trajectoryBatches_' + whichDataset + '.nosync'

if __name__ == '__main__' :

    print('')
    # Loading data
    X_train, X_test, Y_train, Y_test = prepareDataset(datasetPath)
    print('')

    # Training the neural network
    model.summary()
    print('')
    print('Starting training')
    print('......................................')
    print('......................................')
    with tf.device('/device:GPU:0'):
      metrics = model.fit(x = X_train, y = Y_train, \
                          batch_size = batch_size, epochs = numberEpochs, verbose = 1,
                          validation_data = (X_test, Y_test), shuffle = True, \
                              callbacks = [es_callback,cp_callback])
    print('......................................')
    print('......................................')
    print('End of training')
    print('')

    # Saving model and metrics
    model_json = model.to_json()
    with open(filePath + "model" + training_number + ".json", "w") as json_file :
        json_file.write(model_json)
    # Save weights to HDF5
    model.save_weights(filePath + "model" + training_number + ".h5")
    print("Saved model to HDD")
    np.save(filePath + 'metricsTheoretical' + training_number + '.npy', metrics.history)
    print("Saved metrics to HDD")
    print('')
```

C.2 Simulator

```
"""
Imperial College London
MSc Applied Mathematics
This code has been written as part of the MSc project 'Deep Neural Networks
for Real-time Trajectory Planning'
Author : Amaury FRANCOU - CID: 01258326
Supervisor : Dr Dante KALISE
"""

# Imports
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from tensorflow.keras.models import model_from_json
from tensorflow.keras.layers import Activation
from dynamicalSolver import eulerSolveIVP, RK4solveIVP
from matplotlib.lines import Line2D
from trainNN import controlsActivation


def simulate(modelNumber, method, qf, q0 = np.zeros(5), Nmax = 500, h = 2.5 * 1e-3, eps = 1e-3,
             Tmax = 5.0, maxStep = 1e-3, targetXZ = False, targetQ = False, printCircle = False, printState = False, \
                 printControls = False) :
    """
    This function simulates a quadrotor flight, performed using the controller
    synthesized with the trained neural network.

```

```
    Parameters
    ----------
    modelNumber : int - the model number which to use
    method : string - preferred method
    qf : 5-dimensional numpy array - the final state vector
    q0 : 5-dimensional numpy array - the initial state vector
    Nmax : integer - the maximum number of time iterations before stopping
    h : float - the time step used
    eps : float - the tolerance of closeness to required final state
    Tmax : float - the maximum time on which to compute the trajectory
    targetXZ : boolean - Set to True if the end point is chosen to minimize
        the distance with the position input (xf,zf)
    targetQ : boolean - Set to True if the end point is chosen to minimize
        the distance with the state input qf
    printCircle : boolean - If set to True, prints an error circle with a radius
        corresponding to the XZ distance to the target endpoint
    printState : boolean - if set to true, prints the difference of states qhat
    printControls : boolean - if set to true, prints the controls u

    Returns
    -------
    converged : boolean - the variable is set to True if the final state has been reached
                                within the tolerance
    qStorage : 5xM-dimensional numpy array - the state vectors across time
    uStorage : 2xM-dimensional numpy array - the control vector across time

    """


    ##########################################################################


    #### Loading model ####
    modelNumber = str(modelNumber)
    pathToModel = '//.../.../.../.../'\
        '.../.../.../.../training_' + modelNumber
    json_file = open(pathToModel + '/model' + modelNumber + '.json', 'r')
    loaded_model_json = json_file.read()
    json_file.close()
    model = model_from_json(loaded_model_json,\
                            custom_objects={'controlsActivation': Activation(controlsActivation),\
                                            'Activation': Activation(controlsActivation)})
    model.load_weights(pathToModel + '/model' + modelNumber + '.h5')


    ##########################################################################


    #### Simulating trajectory ####

    if method == 'Euler' :
        qStorage, uStorage, converged, errArray, errXZarray  = \
            eulerSolveIVP(model = model, q0 = q0, qf = qf, Nmax = Nmax, h = h, eps = eps, \
                          printState = printState, printControls = printControls)
    if method == 'RK4' :
        qStorage = RK4solveIVP(model = model, q0 = q0, qf = qf, Tmax = Tmax, Nmax = Nmax, \
                               maxStep = maxStep, extractControls = True, printState = printState, \
                                   printControls = printControls)
        converged = True
        uStorage = None


    ##########################################################################


    #### Chosing end point ####
    if not converged :
```

```
        if targetXZ :
            tBestXZ = np.argmin(errXZarray)
            qStorage = qStorage[:,:tBestXZ]
        if targetQ :
            tBestq = np.argmin(errArray)
            qStorage = qStorage[:,:tBestq]


    ##########################################################################


    #### Plot ####
    fig, ax = plt.subplots(figsize=(9.75,12.25))


    # Initial and final points
    plt.plot(q0[0],q0[2],'bo', label ='$\mathbf{q}_0$', markersize = 4)
    plt.text(q0[0]-0.070, q0[2]-.035, 'Initial $\mathbf{q}_0$', fontsize = 12)
    plt.plot(qf[0],qf[2],'o', label = 'Target $\mathbf{q}_f$', markersize = 4, color = 'black')
    plt.text(qf[0]-.085, qf[2]+.02, 'Target $\mathbf{q}_f$', fontsize = 12)
    plt.plot(qStorage[0,-1], qStorage[2,-1], 'ro', label = 'Observed final point', markersize = 4)
    plt.text(qStorage[0,-1]+.03, qStorage[2,-1]+.03, 'Observed final point', fontsize = 12)

    # XZ Trajectory
    X = qStorage[0,:]
    Z = qStorage[2,:]
    plt.plot(X,Z)
    plt.xlabel("$x'$ (m)", fontsize = 15)
    plt.ylabel("$z'$ (m)", fontsize = 15)

    # Error circle
    xzf = np.array([qf[0],qf[2]])
    xzaf = np.array([qStorage[0,-1],qStorage[2,-1]])
    err = np.linalg.norm(xzf-xzaf)
    if printCircle :
        circle = plt.Circle((qf[0],qf[2]), err, fill = False, ls = '--')
        plt.text(qf[0]-err-0.005,qf[2]-err-0.005, 'Error on $(x_f,z_f)$ : ' + str(err)[:4] + 'm')
        plt.gca().add_patch(circle)

    # Parameters
    N = qStorage.shape[1]
    rate = int(0.15 * N)
    length = 0.5
    lengthOn2 = length / 2
    height = 0.07

    # Colormap
    cmap = matplotlib.cm.get_cmap('jet')

    # Target Final quadrotor drawing
    thetaf = np.rad2deg(qf[4])
    quadrotorX = np.array([qf[0] - lengthOn2, qf[0] - lengthOn2, \
                           qf[0] + lengthOn2, qf[0] + lengthOn2])
    quadrotorZ = np.array([qf[2] + height, qf[2], qf[2], qf[2] + height])
    rotate = matplotlib.transforms.Affine2D().rotate_deg_around(qf[0], qf[2], -thetaf)
    quadrotor = Line2D(quadrotorX, quadrotorZ, linewidth = 1.37,  \
                       drawstyle = 'steps-mid', color = 'black', linestyle = '--')
    quadrotor.set_transform(rotate + ax.transData)
    plt.gca().add_line(quadrotor)


    # Actual final quadrotor drawing
    thetafActual = np.rad2deg(qStorage[4,-1])
    quadrotorX = np.array([qStorage[0,-1] - lengthOn2, qStorage[0,-1] - lengthOn2, \
                           qStorage[0,-1] + lengthOn2, qStorage[0,-1] + lengthOn2])
    quadrotorZ = np.array([qStorage[2,-1] + height, qStorage[2,-1], qStorage[2,-1], \
```

```
                                qStorage[2,-1] + height])
    rotate = matplotlib.transforms.Affine2D().rotate_deg_around(qStorage[0,-1], qStorage[2,-1],\
                                                                            -thetafActual)
    quadrotor = Line2D(quadrotorX, quadrotorZ, linewidth = 1.37, drawstyle = 'steps-mid',\
                            color = cmap((N-1)*h))
    quadrotor.set_transform(rotate + ax.transData)
    plt.gca().add_line(quadrotor)


    # Drawing quadrotor orientation
    for i in range(N) :

        if i % rate == 0 :

            xNow = qStorage[0,i]
            zNow = qStorage[2,i]
            thetaNow = np.rad2deg(qStorage[4,i])
            quadrotorX = np.array([xNow - lengthOn2, xNow - lengthOn2, xNow + lengthOn2, xNow + lengthOn2])
            quadrotorZ = np.array([zNow + height, zNow, zNow, zNow + height])
            rotate = matplotlib.transforms.Affine2D().rotate_deg_around(xNow, zNow, -thetaNow)
            quadrotor = Line2D(quadrotorX, quadrotorZ, linewidth = 1.37, drawstyle = 'steps-mid', \
                                    color = cmap(i * h))
            quadrotor.set_transform(rotate + ax.transData)
            plt.gca().add_line(quadrotor)

    # Colorbar
    norm = matplotlib.colors.Normalize(vmin = 0,vmax = (N-1) * h)
    sm = plt.cm.ScalarMappable(cmap = cmap, norm = norm)
    cbar = plt.colorbar(sm, orientation = "horizontal")
    cbar.set_label('$t$ (s)', fontsize = 15)

    # Title
    plt.title('Quadrotor trajectory starting at $\mathbf{q}_0$' \
              + ' = ' + np.array2string(q0) + ', \n targeting endpoint $\mathbf{q}_f$ = ' \
                   + np.array2string(qf)  +r'$^\top$' + '.' + '\n')

    plt.gca().set_aspect('equal', adjustable='box')
    plt.axis('equal')
    plt.show()

    return converged, qStorage, uStorage
```

C.3 Simulation script

```
"""
Imperial College London
MSc Applied Mathematics
This code has been written as part of the MSc project 'Deep Neural Networks
for Real-time Trajectory Planning'
Author : Amaury FRANCOU - CID: 01258326
Supervisor : Dr Dante KALISE
"""

# Imports
import numpy as np
from simulator import simulate
from datasetGenerator import parameterDraw
from IVPsolver import solve

#############################################################################
# Settings
method = 'Euler' # RK4 or Euler
modelNumber = 37 # Model to use

```

```
q0 = np.zeros(5)
qf = np.array([0,0,1,0,0])

printState = True
printControls = True

drawParameters = True # Draw q0 and qf in the same manner as dataset generator
maxDist = 6.0 # Max XZ distance between q0 and qf
onlyUp = False # Only accept upwards XZ trajectories



                        #############################

### Euler ####
Nmax = 150 # Number of points along trajectory
h = 2.5 * 1e-3 # Time step

eps = 1e-4 # Tolerance on state

targetXZ = False
targetQ = False
printCircle = False


### RK4 ###
Tmax = 2.0
#Nmax = 100 # Number of points along trajectory
maxStep = 1e-4


##############################################################################



def getInitialFinalStates(maxDist = maxDist, onlyUp = onlyUp) :
    """
    This function samples initial and final states in the same manner as in
    the dataset generator.

    Parameters
    ----------
    None

    Returns
    -------
    q0 : 5-dimensional numpy array - the initial state vector
    qf : 5-dimensional numpy array - the final state vector

    """

    # Sampling until condition is verified
    condition = True
    while condition :
        # Drawing parameters
        c, q0, tf = parameterDraw()

        # Solving IVP
        N = 100
        sol = solve(q0, c, tf, N = N, extractControls = False)
        qf = sol.y[:5,N-1]

        # Computing XZ distance
        diff = qf - q0
        xzDiff = np.array([diff[0], diff[2]])
        if onlyUp :
            if np.linalg.norm(xzDiff) < maxDist and diff[2] > 0 :
                condition = False
        else :
            if np.linalg.norm(xzDiff) < maxDist :
                condition = False

    return q0, qf
```

```


if __name__ == '__main__' :

    np.set_printoptions(precision=3)

    if drawParameters :
        q0, qf = getInitialFinalStates()

    converged, qStorage, uStorage = simulate(modelNumber = modelNumber, method = method,\
                                            qf = qf, q0 = q0,\
                                                Nmax = Nmax, h = h, \
                                    eps = eps, Tmax = Tmax, maxStep = maxStep,\
                                        targetXZ = targetXZ, targetQ = targetQ, \
                                        printCircle = printCircle,
                                    printState = printState, printControls = printControls)
    print('')
    print('Converged : ', converged)
    print('')
```

C.4 Deep reinforcement learning attempts

```
"""
Imperial College London
MSc Applied Mathematics
This code has been written as part of the MSc project 'Deep Neural Networks
for Real-time Trajectory Planning'
Author : Amaury FRANCOU - CID: 01258326
Supervisor : Dr Dante KALISE
"""

# Imports
import numpy as np
from math import pi
import tensorflow as tf
from dynamicalSolver import fDynamics
from buffer import BasicBuffer_b
from trainNN import controlsActivation
from tensorflow.keras.layers import Activation
from tensorflow.keras.models import model_from_json
from datasetPreparation import encodeTheta
import copy
import os
# Constants
uTmax = 14
uTmin = 0.8
uRmax = 4.6
twoPi = 2 * pi
g = 9.81

def qfDistanceMeasure(q, qf) :
    """
    This function samples initial and final states in the same manner as in
    the dataset generator.

    Parameters
    ----------
    q : 5-dimensional numpy array - the current state vector
    qf : 5-dimensional numpy array - the final state vector

    Returns
    -------
    g : float - a measure of the distance to the final state

    """

    qhat = qf - q
    qhat[4] = qhat[4] % twoPi
```

```python
    qhat[4] = np.square(np.arctan2(np.sin(qhat[4]), np.cos(qhat[4])))

    return np.linalg.norm(qhat)**2


def getReward(qNext, q, u ,qf, h, alpha, beta, gamma) :
    """
    This function computes the reward of the DDPG method.

    Parameters
    ----------
    qNext : 5-dimensional numpy array - the next state vector
    q : 5-dimensional numpy array - the current state vector
    u : 2-dimensional numpy array - the control vector
    qf : 5-dimensional numpy array - the final state vector
    h : float - the time step used
    alpha : float - reward parameter
    beta : float - reward parameter
    gamma : float - reward parameter


    Returns
    -------
    r : float - the instant reward

    """
    dist = qfDistanceMeasure(q, qf)
    newDist = qfDistanceMeasure(qNext, qf)
    sign = np.sign(dist - newDist)

    if sign >= 0 :
        r = alpha * np.abs(1 / newDist) - beta * np.linalg.norm(u)**2
    else :
        r = 0 - gamma * np.linalg.norm(u)**2
    return r

def nextState(q, u, h) :
    """
    This function computes the next state of the quadrotor based on the Euler
    method.

    Parameters
    ----------
    q : 5-dimensional numpy array - the current state vector
    u : 2-dimensional numpy array - the control vector
    h : float - the time step used

    Returns
    -------
    qNew : 5-dimensional numpy array - the next state vector

    """
    qNew = copy.deepcopy(q)
    qNew += h * fDynamics(t = -1, q = q, u = u)

    return qNew

def isArrived(q, qf, eps) :
    """
    This function returns True if the vehicle has arrived close to its final
    state within the given tolerance.

    Parameters
    ----------
    q : 5-dimensional numpy array - the current state vector
    qf : 5-dimensional numpy array - the final state vector
    eps : float - the tolerance of closeness to required final state

    Returns
    -------
    isArrived : boolean - True if the quadrotor has arrived to qf

    """
```

```python

    return (qfDistanceMeasure(q, qf) < eps)


def predictControl(q, qf, piModel, noiseScale) :
    """
    This function computes controls using the policy network and adds random
    noise for DRL exploration.

    Parameters
    ----------
    q : 5-dimensional numpy array - the current state vector
    qf : 5-dimensional numpy array - the final state vector
    piModel : keras object - the trained neural network
    noiseScale : 2-dimensional numpy array - scale parameters for random noise

    Returns
    -------
    u : 2-dimensional numpy array - the control vector

    """

    qhat = qf - q
    qhat[4] = qhat[4] % twoPi
    cosTheta, sintheta = encodeTheta(qhat[4])
    qhatAugm = np.concatenate((qhat[:4],np.array([cosTheta, sintheta])))

    u = piModel.predict(np.array([qhatAugm]), verbose = 0)[0]
    u += np.multiply(noiseScale, np.random.randn(2))

    u[0] = np.clip(u[0], uTmin, uTmax)
    u[1] = np.clip(u[1], -uRmax, uRmax)

    return u

q0 = np.zeros(5)
qf = np.array([0,0,0,0,3.1415])

def ddpg(trainingNumber, modelNumber = 2, numberEpisodes = 100, \
         q0 = q0, qf = qf, h = 1 * 1e-3, eps = 1e-2, maxEpisodeLength = 10000, \
         startSteps = 500, noiseScale = np.array([1.25,1.00]), batchSize = 32,
    discount = 0.99, \
             decayFactor = 0.99, alpha = 100, beta = 50, gamma = 50) :
    """
    This function performs the deep deterministic policy gradient method, loading a
    previously trained quadrotor controller.

    Parameters
    ----------
    trainingNumber : int - the current training number
    modelNumber : int - the pre-trained model number to use
    numberEpisodes : int - the number of episodes to perform
    q0 : 5-dimensional numpy array - the initial state vector
    qf : 5-dimensional numpy array - the final state vector
    h : float - the time step used
    eps : float - the tolerance of closeness to required final state
    maxEpisodeLength : int - maximum number of iteration per episode
    startSteps : int - the number of start steps to reach before adding noise
        to controls for exploration
    noiseScale : 2-dimensional numpy array - scale parameters for random noise
    batchSize : int - size of batch
    discount : float - discount factor for future Q values
    decayFactor : float - decay factor for networks update
    alpha : float - reward parameter
    beta : float - reward parameter
    gamma : float - reward parameter


    Returns
    -------
    rewards : M-dimensional numpy array - the rewards across training
    qLosses : N-dimensional numpy array - loss of Q network across training
    piLosses : N-dimensional numpy array - loss of Pi network across training
```

```
    """

    ############################################################################
    # Loading model Pi
    modelNumber = str(modelNumber)
    pathToModel = '//.../.../Documents/.../' \
        '.../.../.../Trained Models/training_' + modelNumber
    json_file = open(pathToModel + '/model' + modelNumber + '.json', 'r')
    loaded_model_json = json_file.read()
    piModel = model_from_json(loaded_model_json, \
                          custom_objects={'controlsActivation': Activation(
    controlsActivation)})
    piModel.load_weights(pathToModel + '/model' + modelNumber + '.h5')
    # Loading model piTarget
    piTarget = model_from_json(loaded_model_json, \
                           custom_objects={'controlsActivation': Activation(
    controlsActivation)})
    piTarget.load_weights(pathToModel + '/model' + modelNumber + '.h5')
    json_file.close()


    # Creating model Q
    Q = tf.keras.Sequential()
    Q.add(tf.keras.layers.Input(shape  = 5 + 2))
    for layer in range(5) :
        Q.add(tf.keras.layers.Dense(units = 32, activation='relu'))
    Q.add(tf.keras.layers.Dense(units = 1, activation = None))
    # Creating model Qtarget
    Qtarget = tf.keras.Sequential()
    Qtarget.add(tf.keras.layers.Input(shape  = 5 + 2))
    for layer in range(5) :
        Qtarget.add(tf.keras.layers.Dense(units = 32, activation='relu'))
    Qtarget.add(tf.keras.layers.Dense(units = 1, activation = None))

    ############################################################################

    # Replay buffer
    replay_buffer = BasicBuffer_b(size = int(1e6), obs_dim = 5, act_dim = 2)

    # For network training
    piModel_optimizer = tf.keras.optimizers.Adam(learning_rate = 1e-3)
    Q_optimizer = tf.keras.optimizers.Adam(learning_rate = 1e-3)


    ############################################################################

    # Episodes and training
    rewards = []
    qLosses = []
    piLosses = []
    numSteps = 0

    for episodeNb in range(numberEpisodes) :

        # Initialization
        q = copy.deepcopy(q0)
        episodeRewards = 0
        episodeLength = 0
        arrived = False

        while not (arrived or (episodeLength == maxEpisodeLength)) :

            if numSteps == startSteps + 1 :
              print('.......... Starting using physics-informed model
    ............')


            if numSteps > startSteps :
                u = predictControl(q, qf, piModel, noiseScale)
            else :
                uT = np.random.normal(loc = 7.4, scale = 3.3)
                uT =  np.clip(uT, uTmin, uTmax)
```

```python
                uR = np.random.normal(loc = 0, scale = 2.3)
                uR = np.clip(uR, -uRmax, uTmax)
                u = np.array([uT, uR])

            numSteps += 1

            # Next step in navigation
            qNext = nextState(q, u, h)
            qNext[4] = qNext[4] % twoPi
            reward = getReward(qNext, q, u ,qf, h, alpha, beta, gamma)
            arrived = isArrived(q, qf, eps)


            episodeRewards += reward
            episodeLength += 1

            # Ignore arrived if time horizon reached
            if episodeLength == maxEpisodeLength :
                arrivedStorage = False
            else :
                arrivedStorage = arrived

            # Store navigation to replay buffer
            replay_buffer.push(q, u, reward, qNext, arrivedStorage)

            # Moving on
            q = copy.deepcopy(qNext)

        # Perform the gradient descent/ascent updates
        for _ in range(episodeLength) :

            # Sampling from buffer
            States, Controls, Rewards, NextStates, ArrivalStatus = \
                replay_buffer.sample(batchSize)
            States = np.asarray(States, dtype=np.float32)
            Controls = np.asarray(Controls, dtype=np.float32)
            Rewards = np.asarray(Rewards, dtype=np.float32)
            NextStates = np.asarray(NextStates, dtype=np.float32)
            ArrivalStatus = np.asarray(ArrivalStatus, dtype=np.float32)
            StatesTensor = tf.convert_to_tensor(States)

            # Optimizating pi
            with tf.GradientTape() as tape2 :
                StatesHat = qf - States
                cosTheta, sinTheta = encodeTheta(StatesHat[:,4])
                StatesHat = StatesHat[:,:4]
                StatesHat = np.column_stack((StatesHat,cosTheta))
                StatesHat =  np.column_stack((StatesHat,sinTheta))
                EvaluatedControls = piModel(StatesHat)
                args = tf.keras.layers.concatenate([StatesTensor, EvaluatedControls], axis=1)
                Qval = Q(args)
                piLoss =  -tf.reduce_mean(Qval)
                piGrad = tape2.gradient(piLoss, piModel.trainable_variables)
            array = np.random.normal(size=6)
            piModel_optimizer.apply_gradients(zip(piGrad, piModel.trainable_variables))
            piLosses.append(piLoss)

            # Optimizating Q
            with tf.GradientTape() as tape :
                NextStatesHat = qf - NextStates
                cosTheta, sinTheta = encodeTheta(NextStatesHat[:,4])
                NextStatesHat = NextStatesHat[:,:4]
                NextStatesHat = np.column_stack((NextStatesHat,cosTheta))
                NextStatesHat =  np.column_stack((NextStatesHat,sinTheta))
                nextControls = piTarget(NextStatesHat)
                args = np.concatenate((NextStates, nextControls), axis=1)
                QtargetVals = Rewards + discount * (1 - ArrivalStatus) * Qtarget(args)
                args2 = np.concatenate((States, Controls), axis=1)
                Qvals = Q(args2)
                Qloss = tf.reduce_mean((Qvals - QtargetVals)**2)
```

```python
                Qgrad = tape.gradient(Qloss, Q.trainable_variables)
            Q_optimizer.apply_gradients(zip(Qgrad, Q.trainable_variables))
            qLosses.append(Qloss)

            # Updating Q
            QtWeights = np.array(Qtarget.get_weights(), dtype = object)
            QWeights = np.array(Q.get_weights(), dtype = object)
            QFinalWeights = decayFactor * QtWeights + (1 - decayFactor) * QWeights
            Qtarget.set_weights(QFinalWeights)

            # Updating pi
            piTargetWeights = np.array(piTarget.get_weights(), dtype = object)
            piWeights = np.array(piModel.get_weights(), dtype = object)
            piFinalWeights = decayFactor * piTargetWeights + (1 - decayFactor) * piWeights
            piTarget.set_weights(piFinalWeights)

        if episodeNb == 0 :
            print('')
        print("Episode : ", episodeNb + 1, "Reward : ", '{:,}'.format(episodeRewards),\
              'Episode length : ', episodeLength, 'Last state : ', q)

        rewards.append(episodeRewards)

    # ##################################################################################

    # Saving model and metrics
    trainingNumber = str(trainingNumber)
    path = '/.../.../.../.../'\
        '.../.../.../DRL Models/' + 'Training_' + trainingNumber
    os.mkdir(path)
    filePath = '/.../.../.../.../' \
        '.../.../.../DRL Models/' + 'Training_' + trainingNumber + '/'

    model_json = piTarget.to_json()
    with open(filePath + 'piModel' + trainingNumber + '.json', 'w') as json_file :
        json_file.write(model_json)
    # Save weights to HDF5
    piTarget.save_weights(filePath + 'piModel' + trainingNumber + '.h5')

    model_json = Qtarget.to_json()
    with open(filePath + 'Qmodel' + trainingNumber + '.json', 'w') as json_file :
        json_file.write(model_json)
    # Save weights to HDF5
    Qtarget.save_weights(filePath + 'Qmodel' + trainingNumber + '.h5')
    print("Saved models to HDD")
    print('')

    return rewards, qLosses, piLosses

if __name__ == "__main__" :

    trainingNumber = 9
    modelNumber = 2
    numberEpisodes = 1500

    maxEpisodeLength = 200 # 3.56 seconds
    startSteps = maxEpisodeLength * 150
    noiseScale = np.array([0.15 * (uTmax-uTmin), 0.15 * 2 * uRmax])

    alpha = 10000
    beta = 1
    gamma = 10


    discount = 0.95 # 0.99
    decayFactor = 0.95 # 0.99
    batchSize = 32

    h = 3.5 * 1e-2
    eps = 1e-2
```

```
    q0 = np.zeros(5)
    qf = np.array([0,0,0,0,3.1415])
    print('')
    rewards, qLosses, piLosses  = ddpg(trainingNumber, modelNumber, numberEpisodes,
     \
        q0, qf, h, eps, maxEpisodeLength, \
        startSteps, noiseScale, batchSize, discount, \
            decayFactor, alpha, beta, gamma)
```

Bibliography

[1] Pieter Abbeel. Deep learning for robotics. Neural Information Processing Systems - NeurIPS, conference, 2017.

[2] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[3] Dante Kalise. Week 11: Dynamic optimization, lecture notes. Optimization, MATH70005. Imperial College London, delivered 21 March 2022.

[4] Wikimedia Commons. Walkera QR X350 Quadcopter Hovering. `https://commons.wikimedia.org/wiki/File:Walkera_QR_X350_Quadcopter_Hovering.jpg`, 2013. Accessed: 03-07-22.

[5] Robin Ritz, Markus Hehn, Sergei Lupashin, and Raffaello D'Andrea. Quadrocopter performance benchmarking using optimal control. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5179–5186. IEEE, 2011.

[6] Markus Hehn, Robin Ritz, and Raffaello D'Andrea. Performance benchmarking of quadrotor systems using time-optimal control. *Autonomous Robots*, 33(1):69–88, 2012.

[7] Fabio Morbidi and Dominik Pisarski. Practical and accurate generation of energy-optimal trajectories for a planar quadrotor. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 355–361. IEEE, 2021.

[8] Teodor Tomić, Moritz Maier, and Sami Haddadin. Learning quadrotor maneuvers from optimal control and generalizing in real-time. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1747–1754. IEEE, 2014.

[9] Yu-Hsin Hsu and Rung-Hung Gau. Reinforcement learning-based collision avoidance and optimal trajectory planning in uav communication networks. *IEEE Transactions on Mobile Computing*, 21(1):306–320, 2020.

[10] Zhikun Wang, Roderich Groß, and Shiyu Zhao. Aerobatic tic-toc control of planar quadcopters via reinforcement learning. *IEEE Robotics and Automation Letters*, 7(2):2140–2147, 2022.

[11] Sergei Lupashin, Angela Schöllig, Michael Sherback, and Raffaello D'Andrea. A simple learning strategy for high-speed quadrocopter multi-flips. In *2010 IEEE international conference on robotics and automation*, pages 1642–1648. IEEE, 2010.

[12] Wei Kang and Lucas C Wilcox. Mitigating the curse of dimensionality: sparse grid characteristics method for optimal feedback control and hjb equations. *Computational Optimization and Applications*, 68(2):289–315, 2017.

[13] Wei Kang and Lucas Wilcox. A causality free computational method for hjb equations with application to rigid body satellites. In *AIAA Guidance, Navigation, and Control Conference*, page 2009, 2015.

[14] B Kafash, A Delavarkhalafi, and SM Karbassi. Application of variational iteration method for hamilton–jacobi–bellman equations. *Applied Mathematical Modelling*, 37(6):3917–3928, 2013.

[15] Christopher L Darby, William W Hager, and Anil V Rao. An hp-adaptive pseudospectral method for solving optimal control problems. *Optimal Control Applications and Methods*, 32(4):476–502, 2011.

[16] Juan Parras, Patricia A Apellániz, and Santiago Zazo. Deep learning for efficient and optimal motion planning for auvs with disturbances. *Sensors*, 21(15):5011, 2021.

[17] Jiequn Han et al. Deep learning approximation for stochastic control problems. *arXiv preprint arXiv:1611.07422*, 2016.

[18] Philipp Grohs and Lukas Herrmann. Deep neural network approximation for high-dimensional parabolic hamilton-jacobi-bellman equations. *arXiv preprint arXiv:2103.05744*, 2021.

[19] Kimberly J Chan, Joel A Paulson, and Ali Mesbah. Deep learning-based approximate nonlinear model predictive control with offset-free tracking for embedded applications. In *2021 American Control Conference (ACC)*, pages 3475–3481. IEEE, 2021.

[20] Roberto Lampariello, Duy Nguyen-Tuong, Claudio Castellini, Gerd Hirzinger, and Jan Peters. Trajectory planning for optimal robot catching in real-time. In *2011 IEEE International Conference on Robotics and Automation*, pages 3719–3726. IEEE, 2011.

[21] Lin Zhang, Yingjie Zhang, and Yangfan Li. Path planning for indoor mobile robot based on deep learning. *Optik*, 219:165096, 2020.

[22] Shubin Yin, Wei Ji, and Lihui Wang. A machine learning based energy efficient trajectory planning approach for industrial robots. *Procedia CIRP*, 81:429–434, 2019.

[23] Ping Wu, Yang Cao, Yuqing He, and Decai Li. Vision-based robot path planning with deep learning. In *International Conference on Computer Vision Systems*, pages 101–111. Springer, 2017.

[24] Zhigang Ren, Jialun Lai, Zongze Wu, and Shengli Xie. Deep neural networks-based real-time optimal navigation for an automatic guided vehicle with static and dynamic obstacles. *Neurocomputing*, 443:329–344, 2021.

[25] Runqi Chai, Antonios Tsourdos, Al Savvaris, Senchun Chai, Yuanqing Xia, and CL Philip Chen. Six-dof spacecraft optimal trajectory planning and real-time attitude control: a deep neural network-based approach. *IEEE transactions on neural networks and learning systems*, 31(11):5005–5013, 2019.

[26] Lin Cheng, Zhenbo Wang, Fanghua Jiang, and Chengyang Zhou. Real-time optimal control for spacecraft orbit transfer via multiscale deep neural networks. *IEEE Transactions on Aerospace and Electronic Systems*, 55(5):2436–2450, 2018.

[27] Tenavi Nakamura-Zimmerer, Qi Gong, and Wei Kang. Adaptive deep learning for high-dimensional hamilton–jacobi–bellman equations. *SIAM Journal on Scientific Computing*, 43(2):A1221–A1247, 2021.

[28] Praveen Venkatesh, Sanket Vadhvana, and Varun Jain. Analysis and control of a planar quadrotor. *arXiv preprint arXiv:2106.15134*, 2021.

[29] Daniel Liberzon. *Calculus of Variations and Optimal Control Theory: A Concise Introduction*. Princeton University Press, 2012.

[30] Michael G Crandall and Pierre-Louis Lions. Viscosity solutions of hamilton-jacobi equations. *Transactions of the American mathematical society*, 277(1):1–42, 1983.

[31] Jinghao Zhu. A feedback optimal control by hamilton–jacobi–bellman equation. *European Journal of Control*, 37:70–74, 2017.

[32] Olvi L Mangasarian. Sufficient conditions for the optimal control of nonlinear systems. *SIAM Journal on control*, 4(1):139–152, 1966.

[33] Dahlard L Lukes. Optimal regulation of nonlinear dynamical systems. *SIAM Journal on Control*, 7(1):75–100, 1969.

[34] Gottfried Vossen. Switching time optimization for bang-bang and singular controls. *Journal of optimization theory and applications*, 144(2):409–429, 2010.

[35] Dimitri P Bertsekas. Dynamic programming and optimal control, volume 1 of optimization and computation series. *Athena Scientific, Belmont, MA, USA, 3rd edition. Cited on*, page 2, 2005.

[36] Yuanbo Nie, Omar Faqir, and Eric C Kerrigan. Iclocs2: Solve your optimal control problems with less pain. 2018.

[37] Andreas Wächter and Lorenz T Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical programming*, 106(1):25–57, 2006.

[38] II Dikin. Iterative solution of problems of linear and quadratic programming. In *Doklady Akademii Nauk*, volume 174, pages 747–748. Russian Academy of Sciences, 1967.

[39] McCulloch JL. A logical calculus of ideas immanent in nervous activity. *Bull. of Math. Biophysics*, 5:115–133, 1943.

[40] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[41] Kevin Webster. Chapter 8 : Neural networks, lecture notes. Methods for Data Science, MATH70026. Imperial College London, delivered 14 February 2022.

[42] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[43] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[44] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[45] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[46] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[47] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: A view from the width. *Advances in neural information processing systems*, 30, 2017.

[48] Benjamin Karg and Sergio Lucia. Efficient representation and approximation of model predictive control laws via deep learning. *IEEE Transactions on Cybernetics*, 50(9):3866–3878, 2020.

[49] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[50] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[51] Sunny Guha. Deep deterministic policy gradient (DDPG): Theory and implementation. `https://towardsdatascience.com/deep-deterministic-policy-gradient-ddpg-theory-and-implementation-747a3010e82f`, 2020.