

POLITECNICO DI MILANO
Master of Science in Space Engineering
School of Industrial and Information Engineering



**A DAGGER APPROACH FOR SUPERVISED
IMITATION LEARNING APPLIED TO
AUTONOMOUS LUNAR LANDING**

Advisor: Prof. Pierluigi Di Lizia
Co-advisors: Prof. Roberto Furfaro
Prof. Francesco Topputo

Candidate:
Emanuele Bolognesi
Matr. 883373

Academic Year 2018-2019



POLITECNICO
MILANO 1863



THE UNIVERSITY
OF ARIZONA

Emanuele Bolognesi: *A DAgger approach to supervised imitation learning applied to autonomous lunar landing*

|| Master of Science in Space Engineering, Politecnico di Milano

|| Research in collaboration with University of Arizona, Tucson (AZ)

© Copyright July 2018

*A mia nonna Neda,
che ha aspettato
a lungo questo momento.*

Acknowledgments

First I want to thank Prof. Pierluigi Di Lizia for being my advisor and for his distinguished kindness and availability; he put me in contact with Prof. Furfaro, who allowed me to have such a great experience in doing the thesis abroad. I thank Prof. Roberto Furfaro for giving me the opportunity to carry out this research at the University of Arizona, and for his presence and relationship also out of the office. I thank Prof. Francesco Topputo for giving some corrections to my thesis and for introducing me to Git, a tool that I found very useful for this work.

I thank my companions in Arizona, in particular Luca, Andrea, Enrico, Mario and Shahin, for their friendship and for the beautiful trips we had around Arizona and California. I thank Raffa and Genna, who welcomed me in Los Angeles and with whom I spent a couple of nice days.

I thank all my Polimi friends and classmates, with whom I shared these last years pursuing the Master's degree: Alessio, Andre Gioia, Andre Pasquale, Azzi, Corimba, Cri, Erkin, Faggio, Fra Romanó, Fra Scala, Gare, Giobbe, Jack Negri, Laurona, Luca Mariani, Marghe, Marzia, Monta, Nando, Pez, Pit, Polventu, Quiro, Seba, Simo, Teo Caruso, Tico, Tognotommi, Tom Bellosta.

Last but most important, I thank my family for supporting me and for giving me the opportunity and the financial means to study.

Sommario

Grazie all'esponenziale sviluppo dei recenti anni della tecnologia informatica e della potenza computazionale, sempre piú interesse viene dedicato al Machine Learning. Questa branca dell'Intelligenza Artificiale ha dimostrato di essere uno strumento estremamente potente, e per questa ragione ha trovato larghe applicazioni in innumerevoli campi come la visione artificiale, robotica, motori di ricerca, finanza, medicina, solo per citarne alcuni. É naturale che anche il settore spaziale sia interessato dal Machine Learning, che é molto promettente nell'ottica di permettere l'esecuzione autonoma e in tempo reale di compiti complessi. Il seguente lavoro applica il Machine Learning, in particolare tecniche di supervised imitation learning, per implementare un sistema di controllo in feedback per un atterraggio autonomo ottimo lunare. Tecniche di supervised imitation learning per atterraggio ottimi sono già state argomento di alcuni studi, ma nessuno di questi ha considerato il fatto che problemi di predizione sequenziale, dove le osservazioni future dipendono dalle predizioni precedenti, violano l'assunzione di variabili indipendenti e identicamente distribuite che viene spesso presa nel Machine Learning, e questo spesso inficia le prestazioni sia in teoria che in pratica. La soluzione proposta in questo lavoro a tale problema é l'applicazione sistematica della tecnica del DAgger, il cui obiettivo é incrementare le prestazioni del modello di Machine Learning al momento della simulazione. Il problema di atterraggio é affrontato per il caso di energy optimal e fuel optimal, entrambi in casi 1D e 3D. Due differenti modelli di Machine Learning sono usati e confrontati, ovvero la Deep Neural Network e l'Extreme Learning Machine.

Abstract

Thanks to the exponential development of the computer technology and computational power during the recent years, more and more interest is dedicated to Machine Learning. This branch of Artificial Intelligence has proved to be extremely powerful and for this reason has found wide applications in countless fields like computer vision, robotics, search engines, finance, medicine, just to name a few. It is natural that also the space field is interested by Machine Learning, which is very promising in the perspective of performing difficult tasks autonomously and in real-time. The following work applies Machine Learning, in particular supervised imitation learning techniques, to implement a control feedback loop for autonomous optimal lunar landing. Supervised imitation learning for optimal landing has already been subject of a few studies, but none of them considered the fact that sequential prediction problems, where future observations depend on previous predictions, violate the independent-and-identically-distributed assumption commonly used in statistical learning, and this often leads to poor performances both in theory and practice. The solution proposed in this work to such problem is the systematic application of the DAgger approach, which is supposed to increase the performances of the Machine Learning model at prediction time. The landing problem is studied for energy optimal and fuel optimal landing, both in 1D and 3D cases. Two different Machine Learning models, which are Deep Neural Network and Extreme Learning Machine, are used and compared.

Contents

Acknowledgments	I
Sommario	II
Abstract	III
Nomenclature	X
1 Introduction	1
1.1 State of the art	3
1.2 Work justification and purposes	4
1.3 Proposed approach	4
1.4 Thesis structure	5
2 Machine Learning Theory	6
2.1 Supervised Machine Learning	6
2.2 Imitation Learning	7
2.3 The Markov Property	8
2.4 DAgger	8
2.5 Artificial Neural Networks	10
2.6 Extreme Learning Machine	14
2.7 Performance Indexes	16
3 Energy Optimal Landing Problem	20
3.1 ZEM/ZEV	21
3.2 Choice of the input features	23

3.3	1D case	24
3.3.1	Problem formulation	24
3.3.2	Train and test dataset generation	25
3.3.3	Dynamics simulator	26
3.3.4	Dagger procedure	27
3.3.5	Visualize Dagger benefits	29
3.3.6	Deep Neural Network	31
3.3.7	Extreme Learning Machine	35
3.4	3D case	38
3.4.1	Problem formulation	38
3.4.2	Train and test dataset generation	39
3.4.3	Dynamics simulator	40
3.4.4	Dagger procedure	41
3.4.5	Visualize Dagger benefits	42
3.4.6	Deep Neural Network	43
3.4.7	Extreme Learning Machine	46
4	Fuel Optimal Landing Problem	49
4.1	GPOPS	50
4.2	Choice of the input features	51
4.3	1D case	52
4.3.1	Problem formulation	52
4.3.2	Train and test dataset generation	53
4.3.3	Dynamics simulator	56
4.3.4	Dagger procedure	56
4.3.5	Visualize Dagger benefits	58
4.3.6	Deep Neural Network	59
4.3.7	Extreme Learning Machine	62
4.4	3D case	63
4.4.1	Problem formulation	63
4.4.2	Train and test dataset generation	64
4.4.3	Dynamics simulator	65

4.4.4	Dagger procedure	66
4.4.5	Deep Neural Network	66
5	Conclusions and future work	70
5.1	Conclusions	70
5.2	Future work	71
	Bibliography	72

List of Figures

2.1	The issue of sequential prediction problems in imitation learning. Image taken from [17].	9
2.2	Scheme of recurrent and feedforward neural networks. Image taken from [18].	11
2.3	Scheme of a SLFN. Image taken from [19].	11
2.4	Scheme of a SLFN with connections. Image taken from [6].	15
2.5	Example of regression curve	17
2.6	Confusion matrix. Image taken from [20].	18
3.1	Example of ZEM/ZEV solution for 1D case	23
3.2	EO problem 1D train set	26
3.3	Collection criteria of the new states for DAgger	28
3.4	Predicted trajectories over train set, before and after DAgger	30
3.5	1D State distribution, before and after DAgger	31
3.6	EO 1D DNN architecture	32
3.7	EO 1D DNN results DAgger vs Supervised. The subplots represent the validation loss, the average final position, the average final velocity and the Cramer-Von Mises distance.	35
3.8	EO 1D ELM architecture	36
3.9	EO 1D ELM results DAgger vs Supervised. The subplots represent the validation loss, the average final position, the average final velocity and the Cramer-Von Mises distance.	37
3.10	EO 3D train set	40
3.11	3D State distribution, before and after DAgger	42

3.12	EO 3D DNN architecture	43
3.13	Choosing the best architecture with TensorBoard	44
3.14	EO 3D DNN example of regression curve	45
3.15	EO 3D DNN results DAgger vs Supervised. The subplots represent the validation loss, the average final position, the average final velocity and the Cramer-Von Mises distance.	46
3.16	EO 3D ELM architecture	47
3.17	EO 3D ELM example of regression curve	47
3.18	EO 3D ELM results DAgger vs Supervised. The subplots represent the validation loss, the average final position, the average final velocity and the Cramer-Von Mises distance.	48
4.1	Examples of GPOPS solution for 1D case	51
4.2	FO 1D train set	54
4.3	Partial dependence plot 1	55
4.4	Partial dependence plot 2	55
4.5	Predicted trajectories over train set, before and after DAgger, 2D view	58
4.6	Predicted trajectories over train set, before and after DAgger, 3D view	58
4.7	FO 1D DNN architecture	59
4.8	FO 1D DNN Confusion matrix	60
4.9	FO 1D DNN results DAgger vs Supervised. The subplots represent the validation loss, the average final position, the average final velocity and the Cramer-Von Mises distance.	61
4.10	FO 1D ELM Confusion Matrix	62
4.11	FO 1D ELM State Distribution	62
4.12	FO 3D train set	65
4.13	FO 3D DNN architecture	67
4.14	FO 3D DNN results DAgger vs Supervised. The subplots represent the validation loss, the average final position, the average final velocity and the Cramer-Von Mises distance.	69

List of Tables

2.1	Common activation functions	12
3.1	EO 1D train set initial condition	25
3.2	EO 1D DNN results DAgger	34
3.3	EO 1D ELM results DAgger	37
3.4	EO 3D train set initial condition	39
3.5	EO 3D DNN results DAgger	45
3.6	EO 3D ELM results DAgger	48
4.1	FO 1D train set initial condition	53
4.2	FO 1D DNN results DAgger	61
4.3	EO 3D train set initial condition	64
4.4	FO 3D DNN results DAgger	68

Nomenclature

Acronym	Definition
AI	Artificial Intelligence
ANN	Artificial Neural Network
BC	Boundary Conditions
CNN	Convolutional Neural Network
Dagger	Dataset Aggregation
DL	Deep Learning
DNN	Deep Neural Network
ELM	Extreme Learning Machine
EO	Energy Optimal
FO	Fuel Optimal
GPOPS	General Pseudospectral Optimal Control Software
HJB	Hamilton-Jacobi-Bellman
i.i.d	Indipendent and Identically Distributed
LSTM	Long Short Term Memory
LTU	Linear Threshold Unit
ML	Machine Learning
PDP	Partial Dependence Plot
RNN	Recurrent Neural Network
SLFN	Single Layer Feedforward Network
TPBVP	Two Point Boundary Value Problem
ZEM/ZEV	Zero-Effort-Miss, Zero-Effort-Velocity

Symbol	Definition	[UoM]
D_{CM}	Cramer-Von Mises distance	
g	Gravitational field of the Moon at surface	[N/kg]
g_0	Gravitational field of the Earth at sea level	[N/kg]
I_{sp}	Specific Impulse	[s]
J	Cost Function	
m	Mass of the lander	[kg]
\mathbf{r}	Position vector	[m]
T	Thrust	[N]
\mathbf{v}	Velocity vector	[m/s]
\mathbf{X}	Input features vector	
\mathbf{y}	Target vector	

Chapter 1

Introduction

Being the nearest celestial body to the Earth, it is natural that the Moon has always been of interest for space science and exploration. The exploration of the Moon started during the Cold War between the Soviet Union and the United States of America [11]. After World War II, it began what is called the Space Race, that is a kind of technological competition between the two main world powers on who would achieve first in spaceflight capabilities. The first lunar mission was in 1959, just two years after the launch of the Sputnik, with the Soviet probe Luna 1, which was followed in the later years by a large number of other Soviet and American missions. In 1966 the first lander Luna 9, by Soviet Union, reached the lunar surface. The Soviet Union achieved a few others primacies, but it was the United States that, thanks to the rocket technology brought by the legendary German aerospace engineer Wernher Von Braun, achieved the primacy of bringing the man to the Moon in 1969 with the Apollo 11 mission [12], from the Apollo Program. Apollo 11 was followed by other successful Apollo missions (except Apollo 13), some of which installed on the lunar surface scientific instruments and brought back to the Earth samples of lunar rocks. From the mid-1960s to the mid-1970s there were 65 Moon landings, but after Luna 24 in 1976 they suddenly stopped: the Soviet Union started focusing on Venus and space stations and the U.S. on Mars and beyond, and on the Skylab and Space Shuttle programs.

In more recent years, new countries decided to send their own probes to carry out scientific exploration of the Moon. In particular Japan, with the Hiten spacecraft in 1990 and with the SELENE spacecraft in 2007, with the goal "to obtain scientific data of the lunar origin and evolution and to develop the technology for the future lunar exploration", according to the JAXA official website [13]. Also ISRO, India's space agency, sent a spacecraft on the Moon, discovering the presence of water molecules. China recently sent a rover on the dark side of the Moon, to explore a zone that we know very little about. The Moon is of scientific interest in order to better understand the origins of the solar system. There is still a lot to explore on the Moon, because the Apollo Missions only covered a bit part of the surface, in particular on the equatorial zone.

While SpaceX is working to colonize Mars in order to make Humanity a multi-planetary specie [14], other space agencies like the European Space Agency (ESA) have targeted the Moon as a possible outpost to build a base [15]. In order to achieve this, it is required a robust technology to perform planetary landing. The number of landing failures also in recent years proves that the technology is still not mature enough. During the landing of Apollo 11, the lander risked of finishing the fuel, and it was mainly thank to the exceptional piloting skills of Neil Armstrong that it managed to perform a soft landing in a safe zone. Mars was denominated the Great Galactic Ghoull because of the great number of probes lost trying to reach its surface [16]. The problem of Mars is that it is so far that also the telecommunications suffer a delay big enough to jeopardize the mission. The failures on Mars were not only restricted to human errors, but also software failures (the most recent was the Schiaparelli lander). These are all reasons why it is desirable to be able to rely on a robust software than can perform autonomous and optimal landing. The optimal landing problem requires to solve a two-points-boundary-value-problem (TPBVP), which is computationally intense and thus cannot be performed on-board in real-time. The goal of this thesis is contributing in the study of Machine Learning techniques to perform an optimal landing in real-time.

1.1 State of the art

Supervised imitation learning has been applied to optimal lunar landing in a few research studies. In one of these works, Sanchez and Izzo [1] implemented a deep neural network (DNN), assuming perfect knowledge of position and velocity, for 2D a fuel optimal (FO) landing problem. They demonstrated how DNNs can be trained to learn the optimal state-feedback in a number of continuous time, deterministic, non-linear systems of interest in the aerospace domain. They suggested that the networks trained did learn the solution to Hamilton-Jacobi-Bellman (HJB) equations underlying the optimal control problem, because they were able to generalize on points well outside the train set. They also analyzed and tried different network architectures, varying the number of hidden layers and neurons per layer, and different activation functions, demonstrating that shallow networks are not sufficient to represent satisfactorily the complexity of the problem.

In another study, Bloise and Orlandelli [2] implemented a recurrent neural network (RNN) for the fuel optimal landing problem. They generated the optimal trajectories for the train set with the General Pseudospectral Optimal Control Software (GPOPS), both in 1D and 2D cases. They trained the network on two main problems: first, assuming perfect knowledge of position and velocity and mapping them to the thrust; then, generating images of the Moon surface and mapping them to the thrust, thus avoiding the need of knowing position and velocity. A RNN combined to a CNN is used for the second problem, because having a series of image is needed in order to have information about the lander velocity. The CNN processes the images, while the RNN extracts information from the images sequence.

None of these studies considered the fact that sequential prediction problems such as imitation learning, where future observations depend on previous predictions (actions), violate the *independent-and-identically-distributed* (i.i.d.) assumption commonly made in statistical learning. In the following section it is explained what this means and what consequences it implies.

1.2 Work justification and purposes

As Ross et al. stated in [3]: *”Sequential prediction problems such as imitation learning, where future observations depend on previous predictions (actions), violate the common i.i.d. assumptions made in statistical learning. This leads to poor performance in theory and often in practice. In particular, a classifier that makes a mistake with probability ϵ under the distribution of states/observations encountered by the expert, can make as many as $T^2\epsilon$ mistakes in expectation over T -steps under the distribution of states the classifier itself induces.”* Put in other words, it means that the agent can be trained with optimal trajectories, but when it comes to the simulation, every prediction error will cause the agent to find itself in a state which is likely not present in the training set (unless the train set covers the entire space of possible states). This increases the probability of committing another prediction error, thus making the trajectory diverge from the target. The solution to this problem proposed in [3] is the DAgger algorithm (which will be explained in detail in section 2.4), an iterative procedure to train a deterministic policy that achieves good performances guarantees under its induces distribution of states. In that paper, DAgger is applied to simple classification problems in video games. The aim of this work is to prove that the DAgger algorithm can be applied to the optimal landing problem successfully and effectively improving the performances of the ML model during the Monte Carlo simulation. It will be shown that the DAgger algorithm allows achieving good performances with relatively small datasets, and to outperform classical supervised learning with parity of training set size.

1.3 Proposed approach

The approach proposed to carry out the purposes explained in the previous section is demonstrating the effectiveness of the DAgger algorithm to problems with gradually increasing difficulty. The energy optimal (EO) problem is first tackled:

the nature of the EO problem is much simpler than FO problem, and the thrust provided by the engine is not constrained in order to simplify even further. Then, once the DAgger procedure has been tested and proved to work, the FO problem is studied. For both problems, it is first studied the 1D case and then the 3D case. Considerations on the train set size, type of network, model architecture and input feature are made for each case, as well as the criteria to apply DAgger.

The software used to generate optimal trajectories are respectively ZEM/ZEV for the EO problem and GPOPS for the FO problem. They are both implemented in Matlab. The Machine Learning framework used is Keras/Tensorflow, and the code used to train the models, to perform the Monte Carlo simulations and to carry out all the data analysis is in Python.

1.4 Thesis structure

Chapter 2 covers the basics of Machine Learning theory, starting from the definition of supervised and imitation learning, to the explanation of DAgger and finally the theory behind the two ML models used in this work, that is DNN and ELM. The performance indexes used to quantify the quality of the training for each kind of problem are explained. Chapter 3 is on the EO landing problem. It begins formalizing the EO problem and giving some useful information about the software used to generate the train and test trajectory for this problem. Then, it shows how the train set is generated, the procedure to apply DAgger to the EO problem, the tools to visualize the improvements brought by the application of DAgger, and the performances of DNN and ELM, both in 1D and 3D cases. Chapter 4 follows the same structure of the previous chapter, but this time on FO problem. Chapter 5 is about conclusions and future work.

Chapter 2

Machine Learning Theory

In this chapter are presented the theoretical aspects of Machine Learning that are required to better understand what is done in the next chapters.

2.1 Supervised Machine Learning

Machine Learning (ML) is a particular branch of Artificial Intelligence (AI), and it aims at making computer systems able to perform high-level complex tasks effectively and without receiving explicit instructions, relying instead on patterns learned from data. Machine Learning systems can be classified according to the amount and type of supervision they get during training. There are four major categories: *supervised learning*, *unsupervised learning*, *semisupervised learning* and *reinforcement learning*. In supervised learning, the training data fed to the model includes both the input features \mathbf{X}_i and the desired solutions \mathbf{y}_i , called *labels*. The train set is thus constituted by the pairs $\{\mathbf{X}_i, \mathbf{y}_i\}$, $i = 1, \dots, N$, where N is the number of samples in the train set. The goal of the training is making the model able to learn the relation between the input features and the target labels, so that it is able to generalize on new, never seen, inputs. A model is said to *overfit*

if it does perform well on the train data, but poorly on never seen data [7], [8]. Instead, it is said to *underfit* if does not perform well on the train data either. In this work supervised learning algorithms are used, and the pairs state-actions will be provided to the learner during training.

It is possible to make a further distinction of the problem: if the label \mathbf{y} is categorical, it is a *classification* problem, while if it is numerical, it is a *regression* problem. It will be seen every combination of these problems: only regression (EO problem 1D and 3D), only classification (FO problem 1D) and both classification and regression (FO problem 3D).

2.2 Imitation Learning

Imitation learning is a control design paradigm that seeks to learn a control policy reproducing demonstrations from experts. For the landing problem studied in this work, the expert is a software which generates optimal trajectories. The pairs of states and control actions are fed to the ML model during training, so that it learns to map a given state to an optimal control action. This makes the type of Machine Learning supervised.

Consider the following scenario: an agent, which is the ML model, receives in input some *observations* \mathbf{o}_t , and takes an *action* \mathbf{u}_t basing its decision on a *policy* $\pi_\theta(\mathbf{u}_t|\mathbf{o}_t)$, which can be deterministic or stochastic (in this work a deterministic policy is trained) and depends on some parameters θ . The actions have an effect on future observation. The goal of imitation learning is to train the best policy possible, in order to perform the optimal actions for each observation, learning from what is called the *expert policy*. The expert can be a human, or in this case a software which solves the optimal landing problem and generates optimal trajectories.

2.3 The Markov Property

A process has the *Markov property* if the conditional probability distribution of future states of the process (conditional on both past and present states) depends only upon the present state, not on the sequence of events that preceded it. This property can be formalized as following:

$$P(s_t \mid s_{t-1}, s_{t-2}, \dots, s_1) = P(s_t \mid s_{t-1}) \quad (2.1)$$

where P is the probability of happening the state s at time t , given the states at previous times. Thus, the full knowledge of the dynamics of the system, and so of the state \mathbf{x}_t other than the equations of motion, is required to assume the Markov property. The observation indicates the quantity of information that is available to the agent, and is in general different from the state. In this work it is assumed complete knowledge of the state, so that there is no distinction between the state \mathbf{x}_t and the observation \mathbf{o}_t , and in the following sections they will be used interchangeably. The Markov assumption is important because it allows to use models like DNNs and ELMs, which take only one observation at a time, and are not able to reconstruct information from previous observations like recurrent neural networks (RNNs).

2.4 DAgger

Dataset Aggregation (DAgger) techniques are already used in Machine Learning, and they usually consist in modifying slightly the train data (e.g. adding noise), and augmenting with that new artificial data the train set. For the sequential prediction problem, instead, DAgger works in a different way, and it aims at solving a very specific issue. As already observed in section 1.2, sequential prediction problems violate the i.i.d. assumption because future observations depend on previous predictions. Taking as a reference fig. 2.1, the model is trained on the

distribution of data coming from the expert policy $p_{data}(\mathbf{o}_t)$, but at simulation time it encounters a distribution of data coming from a different policy $p_{\pi_\theta}(\mathbf{o}_t)$ of the model. In fact, the distribution of observations induced by $p_{\pi_\theta}(\mathbf{o}_t)$ is not the same as the distribution of observations from which the train data came from, because of the prediction errors.

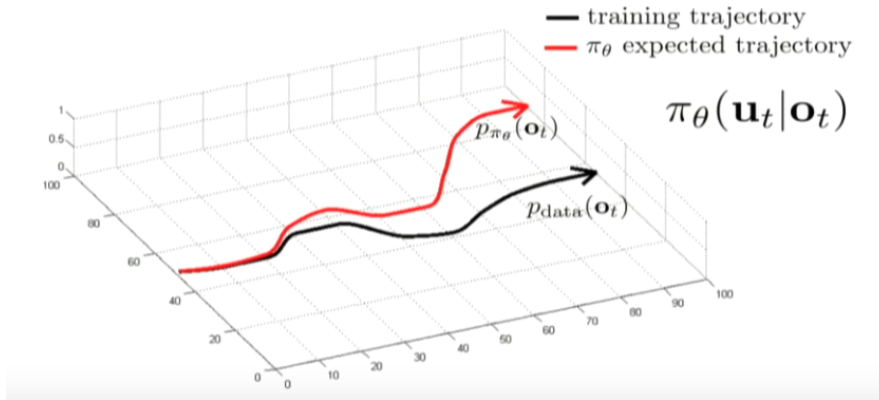


Figure 2.1: The issue of sequential prediction problems in imitation learning. Image taken from [17].

To this problem, there are in particular two possible solution: one is training a very good policy, which is the same of the expert policy, but this is very difficult to achieve; a simpler solution is to use DAgger, whose purpose is to make the distribution of data encountered at prediction time, the same on which the model has been trained ($p_{data}(\mathbf{o}_t) = p_{\pi_\theta}(\mathbf{o}_t)$). This is done just running the policy $\pi_\theta(\mathbf{u}_t | \mathbf{o}_t)$ and augmenting the train data with the observations encountered during the simulation. The DAgger algorithm can be summarized as follow:

1. Train $\pi_\theta(\mathbf{u}_t | \mathbf{o}_t)$ from human data $\mathbb{D} = \{\mathbf{o}_1, \mathbf{u}_1, \dots, \mathbf{o}_N, \mathbf{u}_N\}$
2. Run $\pi_\theta(\mathbf{u}_t | \mathbf{o}_t)$ to get dataset $\mathbb{D}_\pi = \{\mathbf{o}_1, \dots, \mathbf{o}_N\}$
3. Ask human to label \mathbb{D}_π with actions \mathbf{u}_t
4. Aggregate: $\mathbb{D} \leftarrow \mathbb{D} \cup \mathbb{D}_\pi$ and return to point 1.

The DAgger algorithm has already been applied to situations where the model is supposed to learn from human behavior, for example in problem such as driving car [10] or video games [27], [28]. In the landing problem, however, a human cannot label the observations with optimal actions; instead, a software is used to generate optimal trajectories: ZEM/ZEV for the EO problem, and GPOPS for the FO problem. The following algorithm is then applied:

1. Train $\pi_\theta(\mathbf{u}_t|\mathbf{o}_t)$ from software data $\mathbb{D} = \{\mathbf{o}_1, \mathbf{u}_1, \dots, \mathbf{o}_N, \mathbf{u}_N\}$
2. Run $\pi_\theta(\mathbf{u}_t|\mathbf{o}_t)$ to get dataset $\mathbb{D}_\pi = \{\mathbf{o}_1, \dots, \mathbf{o}_N\}$
3. Use the software to label \mathbb{D}_π with actions \mathbf{u}_t
4. Aggregate: $\mathbb{D} \leftarrow \mathbb{D} \cup \mathbb{D}_\pi$ and return to point 1.

2.5 Artificial Neural Networks

Artificial Neural Networks (ANNs) are a Machine Learning model that take inspiration from the biological brain's architecture. They are versatile, powerful and scalable, characteristics that make them ideal for highly complex AI tasks.

Artificial neural networks can be grouped in *feedforward neural networks* and *recurrent neural networks*, depending on the structure of the connections between the neurons. Recurrent neural networks are structured in a way that some neurons receive in input their own output, plus the input of other neurons, and this makes these kind of networks ideal to process sequences of input and extract information about their history. If there are no such feedback connections, the network is classified as a feedforward neural network. The simplest feedforward architecture is the *Single Layer Feedforward Network* (SLFN), which is composed by an *input layer*, a *hidden layer* and an *output layer*.

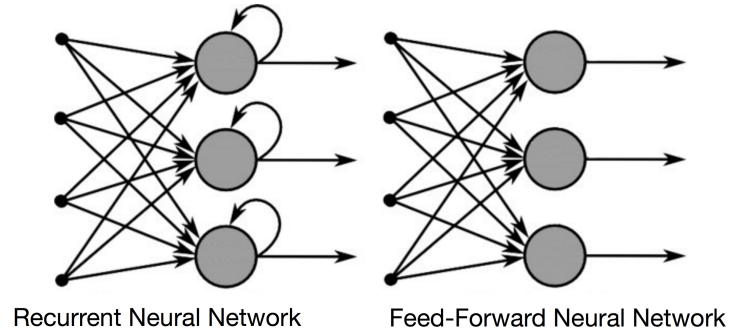


Figure 2.2: Scheme of recurrent and feedforward neural networks. Image taken from [18].

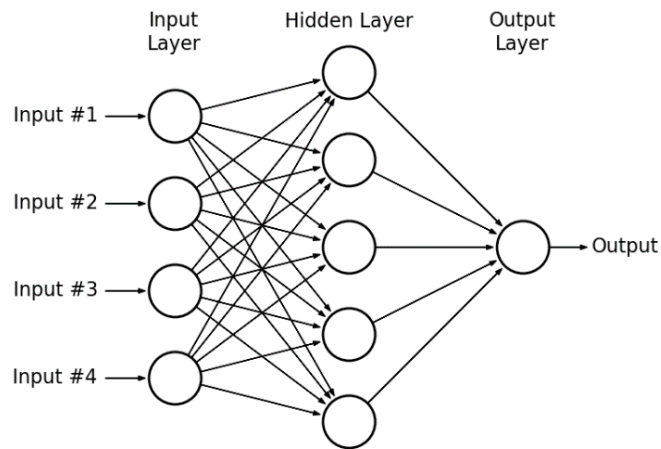


Figure 2.3: Scheme of a SLFN. Image taken from [19].

In general, the output of function of a SLFN can be written as:

$$\mathbf{f}_L(\mathbf{x}) = \sum_{i=1}^L \beta_i h_i(\mathbf{x}, \mathbf{w}_i, b_i) \quad (2.2)$$

where L is the number of neurons in the hidden layer, β_i with $i = 1, \dots, L$ is the output weights vector of the i -th node, \mathbf{w}_i and b_i are respectively the input weights vector and the bias, \mathbf{x} is the input vector. Functions h_i are the activation

functions of the neurons and are in general non linear piece-wise continuous. Their purpose is to introduce in the network a non linearity, which is necessary for the model to represent complex non linear dynamics. Different activation functions have different behavior: for example, they can be discontinuous in the zero (step), continuous but not differentiable (ReLU) or continuous and differentiable (sigmoid and tanh); furthermore, they can be saturating (step, sigmoid, tanh, ReLU for negative inputs) if the output is bounded, or non saturating (ReLU for positive inputs) if the output is not bounded. These properties ultimately affects the model behavior and the activation functions have to be chosen for each problem. A list of activation functions commonly used in ML is reported in tab. 2.1.

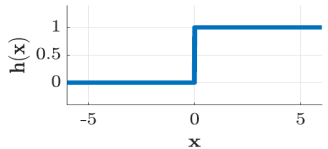
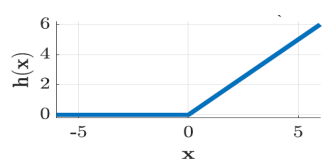
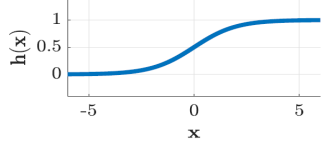
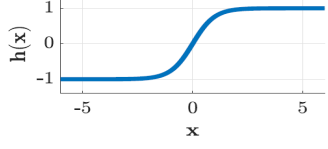
Activation	Formula	Plot	Output range
Step	$h(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$		$h = 0 \vee h = 1$
ReLU (rec-tified linear unit)	$h(x) = \max(0, x)$		$h \in [0, \infty)$
Sigmoid	$h(x) = \frac{1}{1 + e^{-x}}$		$h \in (0, 1)$
Tanh (hyper-bolic tangent)	$h(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$		$h \in (-1, 1)$

Table 2.1: Common activation functions

Deep neural networks are feedforward neural networks with two ore more hidden

layers, and every layer except the output layer is fully connected to the next layer. Training a neural networks it means finding the values of the weights and biases such that, given an input, the desired output is obtained. The training is done with an algorithm called *backpropagation*, which can be described as a gradient descent algorithm with reverse-mode autodiff. It works as follows: for each training instance, the backpropagation algorithm first makes a prediction computing the output of every neuron in each consecutive layer (forward pass), than measures the error (it knows the true label, since it is supervised learning), and then goes back through each layer to measure the error contribution from each connection (reverse pass), and finally tweaks the connection weights to reduce the error (gradient descent step).

There are some parameters, called *hyperparameters*, which can be chosen and are fundamental in determining the quality and success of the training. The most common and important hyperparameters are:

- Number of hidden layers: having more hidden layers allows the model to represent more complex structures. Too many hidden layers may lead to overfitting and slow all the computations (training and prediction), but not enough layers may not be adequate to represent the complexity of the problem (underfit).
- Number of neurons per layer: the number of neurons in the input and output layer are determined by the input and output size. The number of neurons in the hidden layers are hyperparameters, and like the number of layers, too many of them cause overfitting, but not enough underfitting.
- Activations: depending on the problem, it may be desirable to choose a saturating activation (like step, sigmoid or tanh) or a non saturating one (like ReLU, for positive inputs).
- Learning rate: influences the speed of the weight update. If it is too high, the training is very fast but may not reach convergence. If it is too low, the training is too slow and may be stuck in a local minimum.

- Mini-batch size: indicates the number of samples fed to the network at each iteration to update the weights.
- Normalization: all the inputs are rescaled within a specified range. In this way, it is avoided having input features with different orders of magnitude, which in general helps the training.

Unfortunately, there are only a few rule of thumbs, but other than, that there is no strict rule to choose the hyperparameters. They absolutely depend on the nature and kind of problem that has to be solved, so a trial and error procedure is required.

2.6 Extreme Learning Machine

Extreme learning machines are a type of single layer feedforward networks, but their training process is completely different from classical ANNs. In fact, it has been proved by Huang et al. [4] that if a SLFN with tunable hidden nodes parameters can learn a regression of a target function $\mathbf{f}(\mathbf{x})$, then, if the hidden nodes activation functions $h_i(\mathbf{x}, \mathbf{w}_i, b_i)$, $i = 1 \dots L$ are non-linear piecewise continuous, training of the network does not require tuning of those parameters. This means that input weights \mathbf{w}_i and biases b_i of hidden nodes can be assigned randomly, and a SLFN will still maintain the property of universal approximator, as long as the output weights β_i are computed properly. Referring to eq. 2.2 and generalizing to N samples, the output of a SLFN can be expressed as:

$$\mathbf{y}_j = \sum_{i=1}^L \beta_i h_i(\mathbf{x}_j, \mathbf{w}_i, b_i) \quad j = 1 \dots N \quad (2.3)$$

with $\beta_i \in \mathbb{R}^{m_{out} \times 1}$ and $\mathbf{w}_i \in \mathbb{R}^{m_{in} \times 1}$, being m_{in} the input size and m_{out} the output size. Fig. 2.4 represents a SLFN with the connections between input, hidden and output layer. The network is fed a batch of N samples \mathbf{x}_i and gives in output N

predictions \mathbf{y}_i .

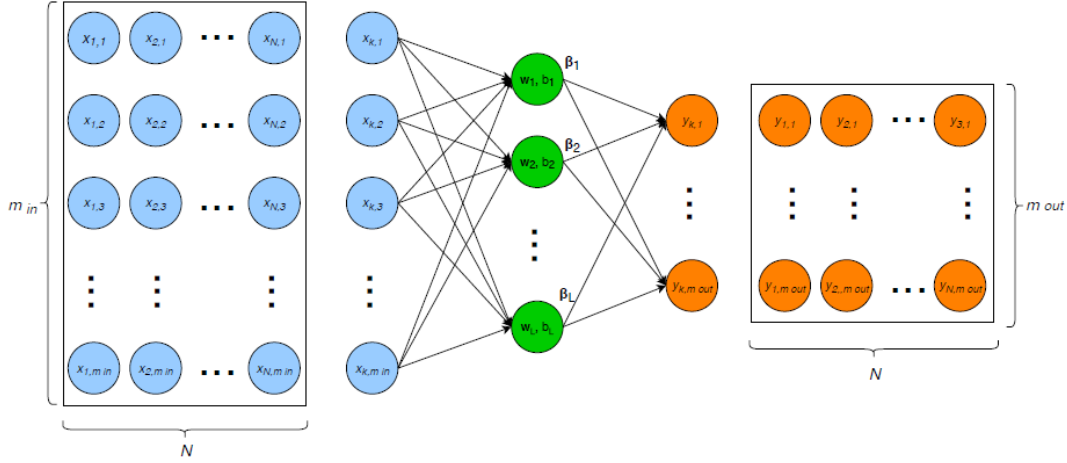


Figure 2.4: Scheme of a SLFN with connections. Image taken from [6].

The hidden layer matrix of the network is called \mathbf{H} , whose i -th column is the output of the i -th hidden node with respect to the set of inputs $\mathbf{X} = [\mathbf{x}_1, \dots, \mathbf{x}_N] \in \mathbb{R}^{m_{in} \times N}$:

$$\mathbf{H} = \begin{bmatrix} h(\mathbf{x}_1, \mathbf{w}_1, b_1) & \dots & h(\mathbf{x}_1, \mathbf{w}_L, b_L) \\ \vdots & \ddots & \vdots \\ h(\mathbf{x}_N, \mathbf{w}_1, b_1) & \dots & h(\mathbf{x}_N, \mathbf{w}_L, b_L) \end{bmatrix}, \quad \mathbf{H} \in \mathbb{R}^{N \times L} \quad (2.4)$$

Now it is possible to write in matrix formulation eq. 2.3:

$$\mathbf{f}_L(\mathbf{X}) = \mathbf{Y} = \mathbf{H}\boldsymbol{\beta} \quad \text{with} \quad \boldsymbol{\beta} = \begin{bmatrix} \beta_1^T \\ \vdots \\ \beta_L^T \end{bmatrix} \quad \boldsymbol{\beta} \in \mathbb{R}^{L \times m_{out}} \quad (2.5)$$

The training algorithm of ELM is aimed to minimize the cost functional E which represents the training error of the SLFN:

$$E = \|\mathbf{H}\boldsymbol{\beta} - \mathbf{T}\|^2, \quad \text{with} \quad \mathbf{T} = \begin{bmatrix} \mathbf{t}_1^T \\ \vdots \\ \mathbf{t}_L^T \end{bmatrix}, \quad \mathbf{T} \in \mathbb{R}^{N \times m_{out}} \quad (2.6)$$

Where \mathbf{T} is the matrix collecting the true labels. The trained network is a universal approximator if $\boldsymbol{\beta}$ are assigned according to the least square error of the overdetermined system:

$$\mathbf{H}\boldsymbol{\beta} = \mathbf{T} \quad (2.7)$$

In order to have the solution $\bar{\boldsymbol{\beta}}$ with minimum L_2 norm among all least squares solutions, it is necessary and sufficient condition to evaluate $\boldsymbol{\beta}$ using the Moore-Penrose generalized inverse of the hidden layer matrix \mathbf{H} . Thus, the training algorithm of ELM can be written as:

$$\bar{\boldsymbol{\beta}} = \mathbf{H}^\dagger \mathbf{T}, \quad \mathbf{H}^\dagger = (\mathbf{H}^T \mathbf{H})^{-1} \mathbf{H}^T \quad (2.8)$$

Note that the training algorithm of ELM does not require iterative tuning as backpropagation does, but it just consists of the evaluation of the pseudo-inverse of \mathbf{H} , usually obtained via single value decomposition with a computational complexity $\mathcal{O}(NL^2)$. For this reason, training an ELM requires in general much less time and computational resources than training a deep network.

2.7 Performance Indexes

Performance indexes are used to verify the quality of the training. The choice of the indexes depends on the problem and on the information that is needed. In this work for the regression problem, it is used the root mean square error:

$$RMSE = \sqrt{\frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2} \quad (2.9)$$

The RMSE represents the average error made for every prediction, compared to the true label, and it is used also as loss function to train the DNN. A visual tool used to verify the quality of a regressor is the *regression curve*. An example of regression curve is represented in fig. 2.5: on the x-axis are the target labels, on the y-axis the predictions. The more the predictions are similar to the targets, the more the blue points lie on the orange 45 degrees line.

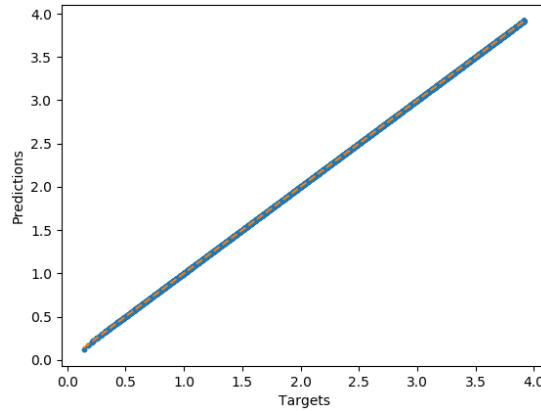


Figure 2.5: Example of regression curve

For the classification problem instead, as loss function for the training of the DNN is used the *binary cross-entropy*, or log loss:

$$H_p(q) = -\frac{1}{N} \sum_{i=1}^N y_i \cdot \log(p(y_i)) + (1 - y_i) \cdot \log(1 - p(y_i)) \quad (2.10)$$

It is a particular case of the more general cross-entropy loss, but for binary classification. The idea of the cross-entropy is to compute the confidence of the model in predicting a certain class, and penalize the weights associated if it is wrong, or reinforce the weights if it is right, in a way proportional to the confidence of the

prediction. As a performance index, it is used the accuracy, which is the ratio between the number of correct predictions over the total number of predictions:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Number of predictions}} = \frac{1}{N} \sum_{i=1}^N (1 - \|y_i - \hat{y}_i\|) \quad (2.11)$$

Another tool to evaluate the performance of a classifier is the *confusion matrix*. The idea is to count the number of times that instances of class A are classified as class B.

		Actual Values	
		Positive (1)	Negative (0)
Predicted Values	Positive (1)	TP	FP
	Negative (0)	FN	TN

Figure 2.6: Confusion matrix. Image taken from [20].

Taking as a reference the confusion matrix for binary classification reported in fig. 2.6: there are two classes, Positive (1) and Negative (0). The confusion matrix counts:

- How many Positive values are predicted as Positive (true positive, TP)
- How many Negative values are predicted as Negative (true negative, TN)
- How many Positive values are predicted as Negative (false negative, FN)

- How many Negative values are predicted as Positive (false positive, FP)

With the confusion matrix, it is possible to have a much deeper insight of the classifier's behavior than just looking at the accuracy.

Chapter 3

Energy Optimal Landing Problem

Consider a lander in a 3D cartesian reference system with axes $\{\hat{x}, \hat{y}, \hat{z}\}$, subject to the gravity field \mathbf{g} and control thrust \mathbf{T} . It starts at time t_0 from initial position \mathbf{r}_0 with initial velocity \mathbf{v}_0 , and it has to reach the target state $\mathbf{r}_f, \mathbf{v}_f$ at time t_f . The equations of motion are:

$$\begin{cases} \dot{\mathbf{r}} = \mathbf{v} \\ \dot{\mathbf{v}} = \frac{\mathbf{T}}{m} + \mathbf{g} \\ \dot{m} = -\frac{\|\mathbf{T}\|}{I_{sp} \cdot g_0} \end{cases} \quad BC : \begin{cases} \mathbf{r}(t_0) = \mathbf{r}_0 \\ \mathbf{v}(t_0) = \mathbf{v}_0 \\ m(t_0) = m_0 \end{cases} \quad \begin{cases} \mathbf{r}(t_f) = \mathbf{r}_f \\ \mathbf{v}(t_f) = \mathbf{v}_f \end{cases} \quad (3.1)$$

where $\mathbf{g} = -g\hat{z}$ is the gravitational field vector of the Moon, and g is assumed constant and equal to 1.62 N/kg . $I_{sp} = 200 \text{ s}$ is the specific impulse of the lander engine, and $g_0 = 9.81 \text{ N/kg}$ is the value of the gravitational field of the Earth at sea level.

ZEM/ZEV provides in output the control acceleration \mathbf{a} instead of the thrust \mathbf{T} , furthermore, as explained in Section 1.3, the control acceleration is not constrained, so it is assumed that the engine is always able to provide enough acceleration. This means that the dynamics is not dependent on the mass, which

can be computed a posteriori. The equations of motion for the EO problem are re-written as:

$$\begin{cases} \dot{\mathbf{r}} = \mathbf{v} \\ \dot{\mathbf{v}} = \mathbf{a} + \mathbf{g} \end{cases} \quad BC : \begin{cases} \mathbf{r}(t_0) = \mathbf{r}_0 \\ \mathbf{v}(t_0) = \mathbf{v}_0 \end{cases} \quad \begin{cases} \mathbf{r}(t_f) = \mathbf{r}_f \\ \mathbf{v}(t_f) = \mathbf{v}_f \end{cases} \quad (3.2)$$

Solving the energy optimal (EO) problem means finding the control acceleration which minimizes the cost function:

$$J_{EO} = \min \int_{t_0}^{t_f} \mathbf{a} \cdot \mathbf{a} \, dt \quad (3.3)$$

3.1 ZEM/ZEV

ZEM/ZEV is the software used to generate the trajectories for the energy optimal landing problem. At the core of the theory of ZEM/ZEV there are the two variables zero-effort-miss (ZEM) and zero-effort-velocity (ZEV) which are defined as the respectively difference of position and velocity between the target and the lander once the equation of motions 3.2 are integrated without control until final time t_f :

$$\begin{cases} \mathbf{ZEM}(t) = \mathbf{r}_T(t_f) - \mathbf{r}(t_f) \\ \mathbf{ZEV}(t) = \mathbf{v}_T(t_f) - \mathbf{v}(t_f) \end{cases} \quad \text{for } t < \xi < t_f \quad (3.4)$$

For the minimum energy problem it is demonstrated in [5] that the control acceleration for every time t can be computed as a function of ZEM, ZEV and t_{go} :

$$\mathbf{a}(t) = \frac{6}{t_{go}^2} \mathbf{ZEM}(t) - \frac{2}{t_{go}} \mathbf{ZEV}(t) \quad (3.5)$$

where time-to-go $t_{go} = t_f - t$ is defined as the time to arrive at the terminal state. If the gravitational field \mathbf{g} is assumed constant, it is demonstrated in [5] that for the EO problem the t_{go} can be obtained solving numerically the following quartic equation:

$$\begin{aligned}
 t_{go}^4 A + t_{go}^2 B + t_{go} C + D &= 0 \\
 A &= \mathbf{g} \cdot \mathbf{g} \\
 B &= -2(\mathbf{v} \cdot \mathbf{v} + \mathbf{v}_f \cdot \mathbf{v} + \mathbf{v}_f \cdot \mathbf{v}_f) \\
 C &= 12(\mathbf{r}_f - \mathbf{r}) \cdot (\mathbf{v} + \mathbf{v}_f) \\
 D &= -18(\mathbf{r}_f - \mathbf{r}) \cdot (\mathbf{r}_f - \mathbf{r})
 \end{aligned} \tag{3.6}$$

In the landing problem the target state is fixed and is placed at $\mathbf{r}_f = [0, 0, 0]$ m and $\mathbf{v}_f = [0, 0, 0]$ m/s, so eq. 3.6 can be simplified:

$$\mathbf{g} \cdot \mathbf{g} t_{go}^4 - 2 \mathbf{v} \cdot \mathbf{v} t_{go}^2 - 12 \mathbf{r} \cdot \mathbf{v} t_{go} - 18 \mathbf{r} \cdot \mathbf{r} = 0 \tag{3.7}$$

In Fig. 3.1 is reported an example of solution of ZEM/ZEV for a 1D landing problem:

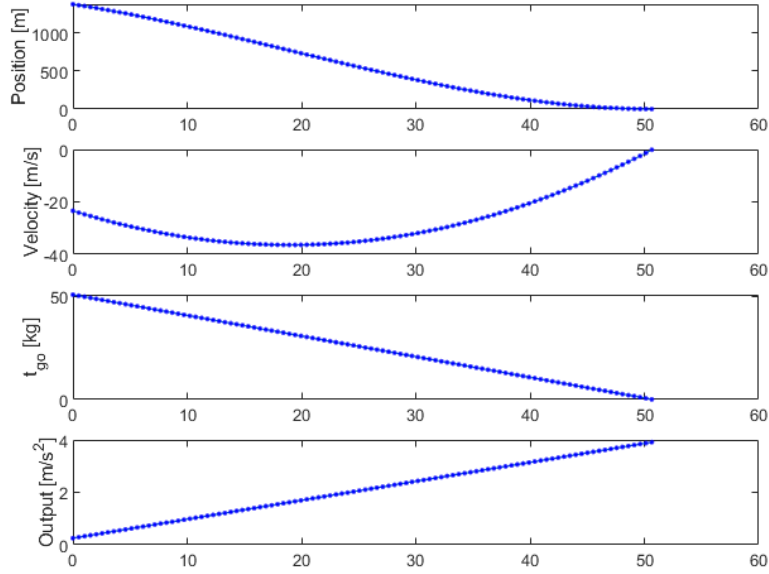


Figure 3.1: Example of ZEM/ZEV solution for 1D case

3.2 Choice of the input features

The choice of the input features is critical in every ML problem. The input features in fact represent the information given in input to the model in order to make a prediction. It is then desirable that the input features reflect as much as possible the relevant information about the data. Consider as an example a regression problem where the price of a house has to be predicted: in this case the features that are more likely to affect the price of a house may be the location, the number of rooms/bathrooms, the walkable area, just to name a few. Instead, some information like the color of the walls may not be so relevant in determining the price of the house.

The input features chosen for the EO problem are position, velocity and time-to-go. While the first two are trivial, one may wonder why it is used the t_{go} . The reason is that, other than proving to increase the performances with respect to not

using it, looking at eq. 3.6 one can see that it does add information, that is the target state \mathbf{r}_f and \mathbf{v}_f , which would be otherwise unknown to the model. Thus, during the simulation it is assumed perfect knowledge of position and velocity, which are computed during the integration of the dynamics, and of time-to-go, which is computed with eq. 3.7.

3.3 1D case

The unidimensional problem is of interest for two main reasons: first, it is a simplified problem which allows to test the network training and the DAgger techniques fast and easily, since it requires less training data than a more complex 3D case; second, the reduced number of dimensions allows to visualize results that offer interesting insights of the ML model behavior, and that would be otherwise more difficult to visualize with more dimensions.

3.3.1 Problem formulation

Consider a lander subject to the gravity field and control acceleration. It starts at time $t_0 = 0$ s from altitude h_0 with initial velocity v_0 , and it has to reach the target state $h_f = 0$ m and $v_f = 0$ m/s at time t_f . The equations of motion 3.2 become:

$$\begin{cases} \dot{h} = v \\ \dot{v} = a - g \end{cases} \quad BC : \begin{cases} h(0) = h_0 \\ v(0) = v_0 \end{cases} \quad \begin{cases} h(t_f) = 0 \\ v(t_f) = 0 \end{cases} \quad (3.8)$$

where h is the altitude, v the vertical velocity, a the control acceleration.

The goal is to train a network able to give for each state of the trajectory an optimal control acceleration a , such that the lander reaches the target state

minimizing the cost function:

$$J_{EO} = \min \int_0^{t_f} \|a\|^2 dt \quad (3.9)$$

In order to train the model, supervised learning techniques are used. The altitude h , velocity v and time-to-go t_{go} are used as input features. The target is the control acceleration a :

- Input features:

$$X = [h, v, t_{go}] \quad X \in \mathbb{R}^{3 \times 1}$$

- Target:

$$y = a \quad y \in \mathbb{R}^{1 \times 1}$$

3.3.2 Train and test dataset generation

For the EO 1D landing problem, 500 optimal trajectory are generated with ZEM/ZEV for the train set, for a total of approximately 450.000 states (as it will explained in the next sections, it is not necessary to use all these states for the train set). The initial conditions for these trajectories are sampled uniformly randomly within the ranges in Table 3.1:

Variable	Min. value	Max. value
h_0	1000 m	1500 m
v_0	-40 m/s	-20 m/s

Table 3.1: EO 1D train set initial condition

The train set is visualized in Fig. 3.2, where the red rectangle represents the area of the initial conditions.

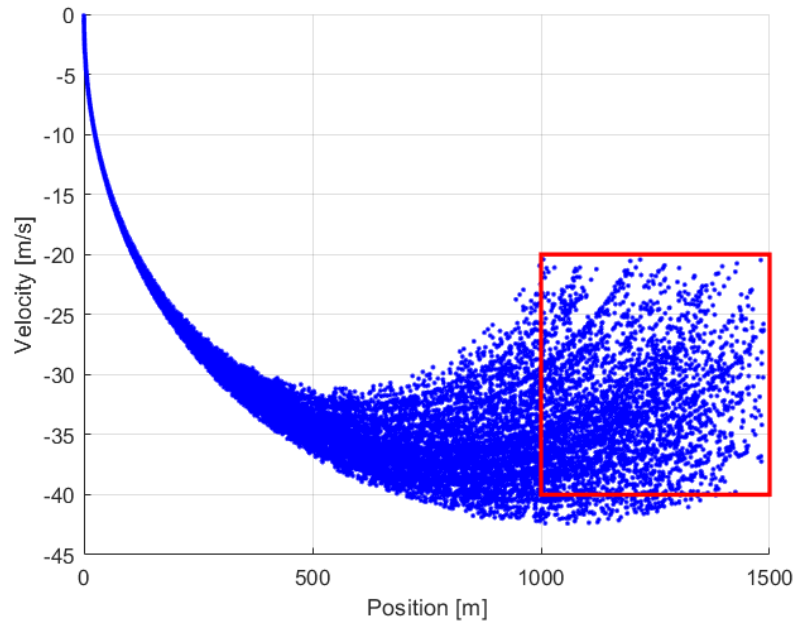


Figure 3.2: EO problem 1D train set

For each DAgger iteration, 50 new optimal trajectories are generated as test set. In this way, at each iteration the Monte Carlo simulation is run on different trajectories.

3.3.3 Dynamics simulator

In order to perform the Monte Carlo simulation integrating the predicted trajectories, a dynamics simulator is implemented in Python. The pseudo-code is:

For each initial condition of the test trajectories:

1. Initialize position $h = h_0$ and velocity $v = v_0$
2. Compute t_{go} solving eq. 3.7

3. Apply MinMaxScaler to input features $X = [h, v, t_{go}]$, and predict control acceleration $y_{pred} = a$
4. Integrate equation of motion 3.8 with a time step of 0.1 s (10 Hz)
5. If the altitude $h > 0$ m and the velocity $v < 0$ m/s go back to point 2, otherwise interrupt integration

The integration is performed with a frequency of 10 Hz, which is a common frequency for landing engines, and it is stopped either when the lander reaches the ground or when its vertical velocity is no more negative (i.e. the lander starts rising before touching ground).

3.3.4 DAgger procedure

Here it is explained how the DAgger algorithm already described in section 2.4 is implemented in practice for the EO unidimensional problem:

1. The train set is generated with ZEM/ZEV. The next step is to choose the best architecture for the ML model. In order to do this, different architectures are trained and the best in terms of validation loss is chosen. This step is critical since, as it will be shown later, the same model that performs well on the original train set, may not be so good on the augmented train set. This means that for each DAgger it is required an analysis to search the best architecture possible. As will be shown in sections 3.3.6 and 3.3.7, different techniques will be used for DNN and ELM to carry out such analysis systematically.
2. A Monte Carlo simulation is performed: the current policy is run on the test trajectories to obtain the predicted trajectories.
3. The predicted trajectories are labeled with ZEM/ZEV: each state of all the predicted trajectories is used as initial condition to generate an optimal trajectory with ZEM/ZEV, and *only the first labeled state is retained*. In this

way it is possible to compare for each predicted state the acceleration predicted y_{pred} and the optimal acceleration y_{opt} . This is useful to visualize the network performance at the simulation time, and also to have a criterion to collect new states for DAgger. The predicted states where the prediction error $y_{err} = ||y_{pred} - y_{opt}||$ is larger than a threshold ϵ_{min} and below a threshold ϵ_{max} are collected. Taking as a reference Fig. 3.3, the left plot shows a single trajectory: for each time it compares the predicted acceleration with respect to the optimal one (corrected with ZEM/ZEV at previous point). The states collected are those where the prediction error is in a certain range (error between 5 % and 50 %). On the right plot, it is shown the predicted trajectory inside the train set, and it is confirmed that the prediction errors are bigger where the input states are outside the train set, and this causes the trajectory to diverge from the target state.

4. The train set is augmented with the states collected in the previous point. The procedure is repeated going back to point 1 and training a new model on the augmented train set.

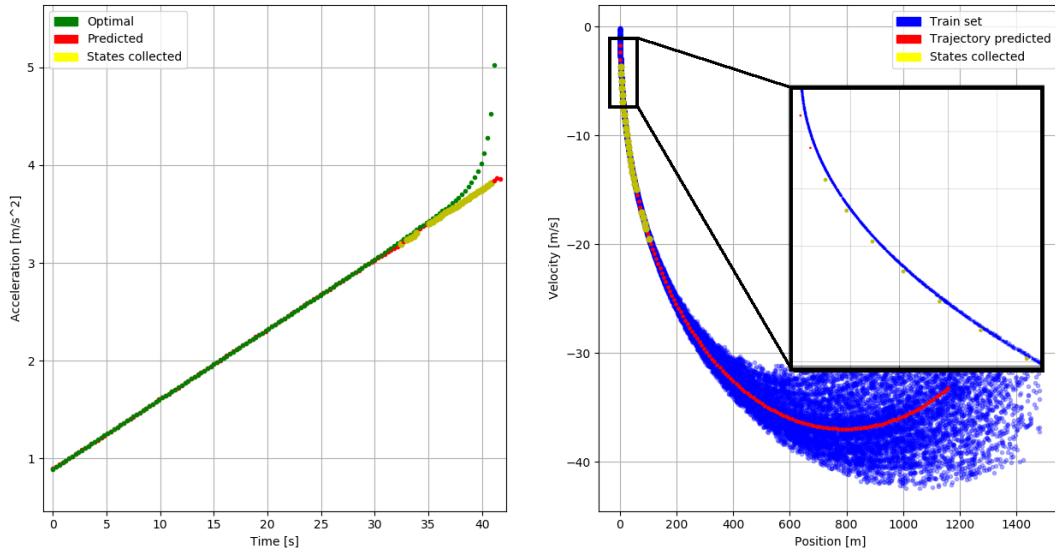


Figure 3.3: Collection criteria of the new states for DAgger

Point 3 of the previous list is the one that required the most number of experiments to come up with. First, it was to be decided if, when augmenting the dataset, retaining only the first state of the corrected trajectory or the entire trajectory. Various tests proved that it is better to augment with only the first state for a few reasons: adding the entire trajectory does not increase the performances in a significant way with respect to adding only the first state, but it increases a lot the train set; moreover, adding the entire trajectory basically means to add data from the expert policy $p_{data}(\mathbf{o}_t)$ (with reference to Fig. 2.1), whereas only the first state is taken from the trained policy $p_{\pi_{\theta}}(\mathbf{o}_t)$. Remember that the goal of DAgger is collecting data from the trained policy instead of the expert policy. That said, it would be perfectly possible to add the entire trajectory, but it would be less efficient.

Another decision taken in point 3 is the criteria to collect only states where the prediction error is in a certain range. The reason why there is a minimum threshold, is to avoid adding states that the model already predicts well. The reason for the maximum threshold, is to avoid adding states that are so outside the train set that is likely that the model in a future iteration will not encounter. Again, it would be possible to simply augment the train set with all the predicted states, but this would be less efficient.

3.3.5 Visualize DAgger benefits

In this section are presented a few metrics and plots used to visualize and quantify the benefits provided by the application of DAgger. Fig. 3.4 shows how augmenting the train dataset helps the predicted trajectories to converge to the target state $[0 \text{ m}, 0 \text{ m/s}]$. In fact, before DAgger (a), the policy is trained on the original train set, and the predicted trajectory tend to diverge toward the last states because the model finds itself in states that it was not trained on, and so the error predictions are bigger as shown in Fig. 3.3. After DAgger (b), the train set has been augmented with the states collected as described in the previous section, and now the predicted trajectory tend to converge much better to the target state.

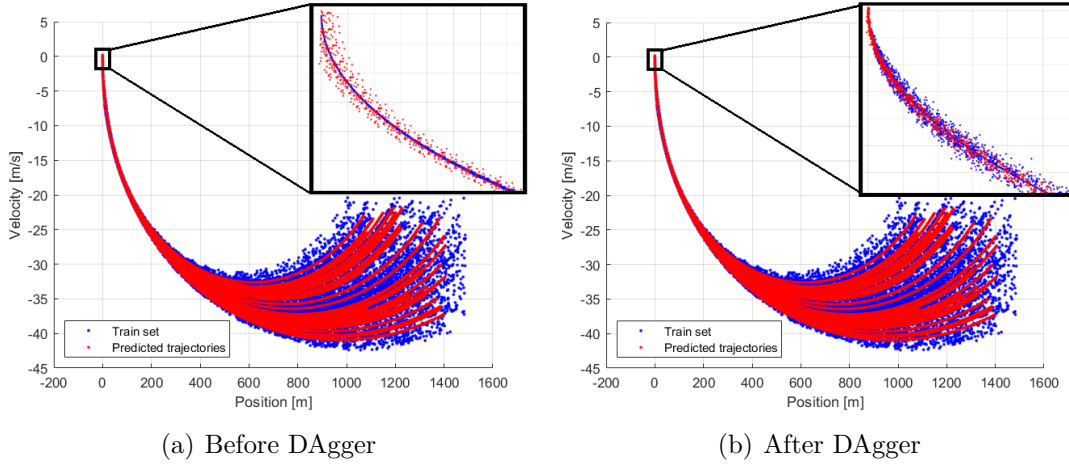


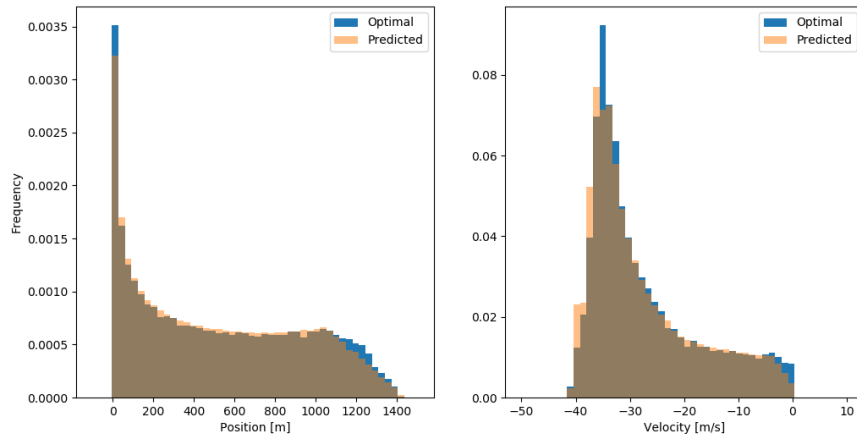
Figure 3.4: Predicted trajectories over train set, before and after DAGger

Another tool to visualize the improvements on the policy is the plot in Fig. 3.5 which compares the states distribution of the predicted trajectories and the optimal ones. On the x-axis, there is a specific variable (i.e. position or velocity), on the y-axis the normalized frequency of states that fall in a specific range. The more the two distributions - optimal and predicted - match, the more the predicted trajectories are similar to the optimal ones, which is indeed desirable. It can be observed how the application of DAGger make the distribution match better, in particular in the last states where altitude and velocity are near zero.

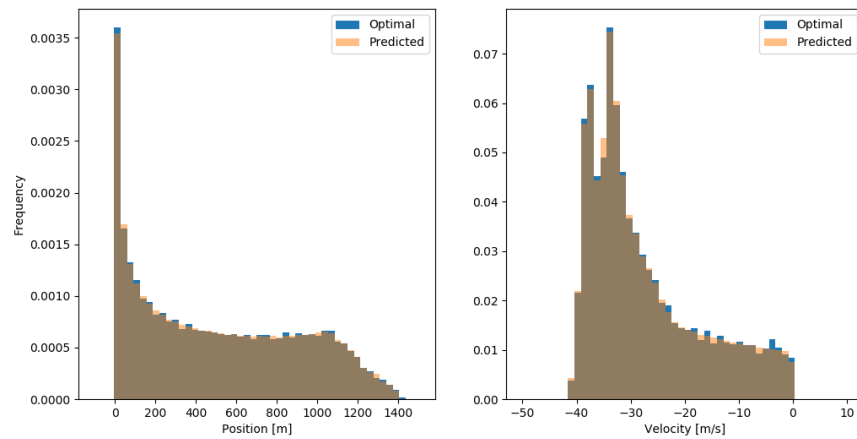
Finally, it is here defined a quantity that will be used to measure how two distributions are similar. It is the Cramer-Von Mises distance:

$$D_{CM} = \sum_i (h_{pred}(i) - h_{opt}(i))^2 \quad (3.10)$$

where $h(i)$ is the normalized frequency of a certain state. For example, referencing to Fig. 3.5, before DAGger the Cramer-Von Mises distance is $D_{CM} = 7.5 \cdot 10^{-4}$, while after DAGger is $D_{CM} = 3.3 \cdot 10^{-4}$.



(a) Before DAGger



(b) After DAGger

Figure 3.5: 1D State distribution, before and after DAGger

3.3.6 Deep Neural Network

The deep neural network is the first ML model used. The original size of the 1D train set with 500 trajectories is approximately 450,000 states, but there is no need to use them all. In fact, applying DAGger means being smart about the train set, and a little sized original train set is sufficient to train a fair enough initial policy to collect new states to augment. This in fact is the philosophy of DAGger:

collect data in a clever way from the trained policy instead of the expert policy. Thus, for the initial train set of DNN, only 20.000 states are sampled randomly from the original train set.

As already explained in section 3.3.4, the main problem of training the ML model, is to find the most performing architecture in terms of validation loss, at each DAgger iteration. In order to carry out such analysis systematically for the DNN, it is used TensorBoard [22], a visualization tool offered by TensorFlow. It allows to visualize the training graphs of different architectures, in order to choose the best.

An interesting result obtained during the application of DAgger, is that augmenting the train set usually requires to change the model architecture. In fact, an architecture that performs well on the original train set, may perform much worse on the augmented train set. In general, going forward with the iterations, it is needed to increase the complexity of the network, for example adding hidden layers and neuron per layer. For example, Table 3.6 shows that on the original train set (iteration 0), the architecture chosen has 2 hidden layers and 64 neurons per layer, while after a few DAgger iterations (iteration 4), the architecture chosen has 4 hidden layers and 128 neurons per layer.

Layer (type)	Output Shape	Param #
main_input (InputLayer)	(None, 3)	0
dense_1 (Dense)	(None, 64)	256
dense_2 (Dense)	(None, 64)	4160
regr_output (Dense)	(None, 1)	65
Total params: 4,481 Trainable params: 4,481 Non-trainable params: 0		

(a) Iteration 0

Layer (type)	Output Shape	Param #
main_input (InputLayer)	(None, 3)	0
dense_1 (Dense)	(None, 128)	512
dense_2 (Dense)	(None, 128)	16512
dense_3 (Dense)	(None, 128)	16512
dense_4 (Dense)	(None, 128)	16512
regr_output (Dense)	(None, 1)	129
Total params: 50,177 Trainable params: 50,177 Non-trainable params: 0		

(b) Iteration 4

Figure 3.6: EO 1D DNN architecture

The following hyperparameters are common to all the architectures:

- Learning rate: 10^{-3} , decreases by a factor of 0.9 each time the validation loss does not decrease for 5 consecutive epochs.
- Mini-batch size: 8, same used by Izzo in [1].
- Activation function: ReLU, same used by Izzo in [1].
- Normalization: input features are rescaled in a range $[0, 100]$ using the `MinMaxScaler` of Keras.

The following hyperparameters, instead, are searched again for each iteration:

- Number of hidden layers.
- Number of neurons per layer.

During the training phase, not the whole train set is actually used for the training, because a fraction of it (15%) is used for the validation set. The validation set consists in data which the model has never seen during the training, and it is useful to avoid overfitting. The same train-validation split ratio will be used for all the other cases. The train is stopped if the validation loss does not decrease for 20 consecutive epochs.

In Table 3.2 are reported the results of the DNN over DAgger iterations. The performances shown are the validation loss, the average final position and velocity reached during the Monte Carlo simulation and the Cramer-Von Mises distance D_{CM} . Finally, there is the number of training data collected during DAgger at the previous iteration, and the total training set size. It is interesting how just a little amount of training data collected with DAgger (in this case about 13% of the initial train set size) is enough to increase dramatically the performances of the network.

Iter	Val loss	Pos [m]	Vel [m/s]	D_{CM}	New data	Data
0	$2.0 \cdot 10^{-2}$	0.56	1.94	$7.5 \cdot 10^{-4}$	-	20,000
1	$4.3 \cdot 10^{-3}$	0.06	0.44	$8.3 \cdot 10^{-5}$	724	20,724
2	$2.0 \cdot 10^{-2}$	0.03	0.16	$4.9 \cdot 10^{-5}$	951	21,675
3	$6.5 \cdot 10^{-3}$	0.01	0.12	$1.1 \cdot 10^{-4}$	589	22,264
4	$6.7 \cdot 10^{-3}$	0.01	0.10	$3.3 \cdot 10^{-4}$	360	22,624

Table 3.2: EO 1D DNN results DAgger

In Fig. 3.7 the performance of DAgger against classical supervised learning is plotted, with parity of train set size. The difference in the train set is that for DAgger, new states are collected running the policy trained in the previous iteration, while for classical supervised, they are collected running the expert policy. It is clear how DAgger outperforms classical supervised, in particular in final position and velocity, which reach a precision respectively of centimeters and centimeters per second.

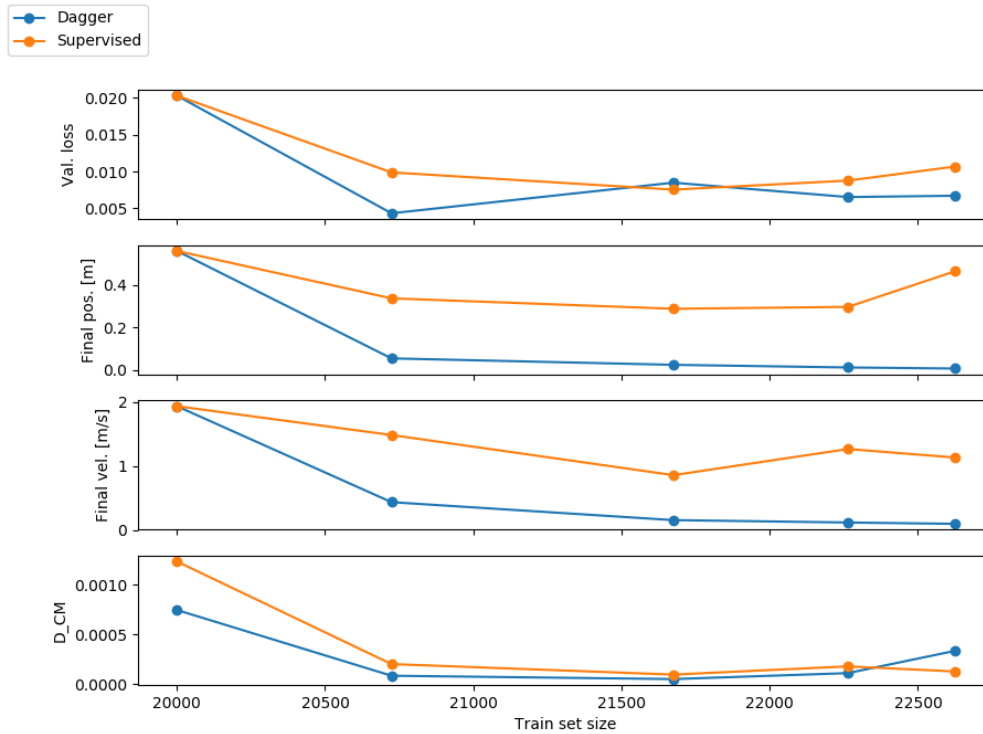


Figure 3.7: EO 1D DNN results DAGger vs Supervised. The subplots represent the validation loss, the average final position, the average final velocity and the Cramer-Von Mises distance.

3.3.7 Extreme Learning Machine

One drawback of ELMs is that they are counter intuitive. Sometimes, they give better results in terms of generalization with less train data, which is the opposite to what is usually expected instead in Deep Learning, and in general in Machine Learning [7], [8]. After a few experiments, it is chosen to use an initial train set with 10.000 states for the ELM.

On the bright size, one of the advantages of ELM is that it is easier to tune than a DNN, because there is only one hyperparameter, which is the number of neurons. The procedure adopted to choose the best architecture for each iteration is in principle the same of the one already used for DNN. Different models

with different number of neurons are trained, and the model which achieves the minimum validation loss is chosen. The code with the implementation of ELM in Python has been downloaded from [24]. The input features are rescaled in a range $[0, 1]$ with the MinMaxScaler of Keras.

Fig. 3.8 shows that the optimal number of neurons (where the validation loss reaches a minimum) increases along the DAgger iterations. This is the same phenomena encountered with the DNN: augmenting the train size with DAgger, a more complex network is required.

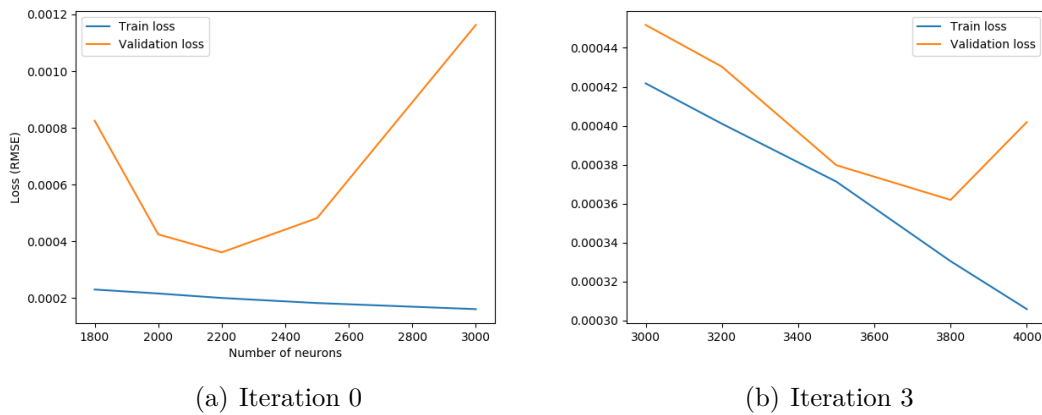


Figure 3.8: EO 1D ELM architecture

ELM proved to be extremely powerful on the regression problem, obtaining already on the original training set performances as good as DNN after DAgger. This is even more impressive, considering that they needed a training set half of the size and a training time in the order of few seconds, instead of several minutes needed by the neural networks. The performances of ELM over the DAgger iterations are reported in Table 3.3, and compared to classical supervised learning in Fig. 3.9.

Iter	Val loss	Pos [m]	Vel [m/s]	D_{CM}	New data	Data
0	$5.2 \cdot 10^{-4}$	0.01	0.11	$3.0 \cdot 10^{-4}$	-	10,000
1	$2.9 \cdot 10^{-4}$	0.01	0.09	$3.9 \cdot 10^{-4}$	120	10,120
2	$3.8 \cdot 10^{-3}$	0.0	0.07	$3.2 \cdot 10^{-4}$	385	10,505
3	$3.7 \cdot 10^{-3}$	0.0	0.08	$2.6 \cdot 10^{-4}$	100	10,605

Table 3.3: EO 1D ELM results DAGger

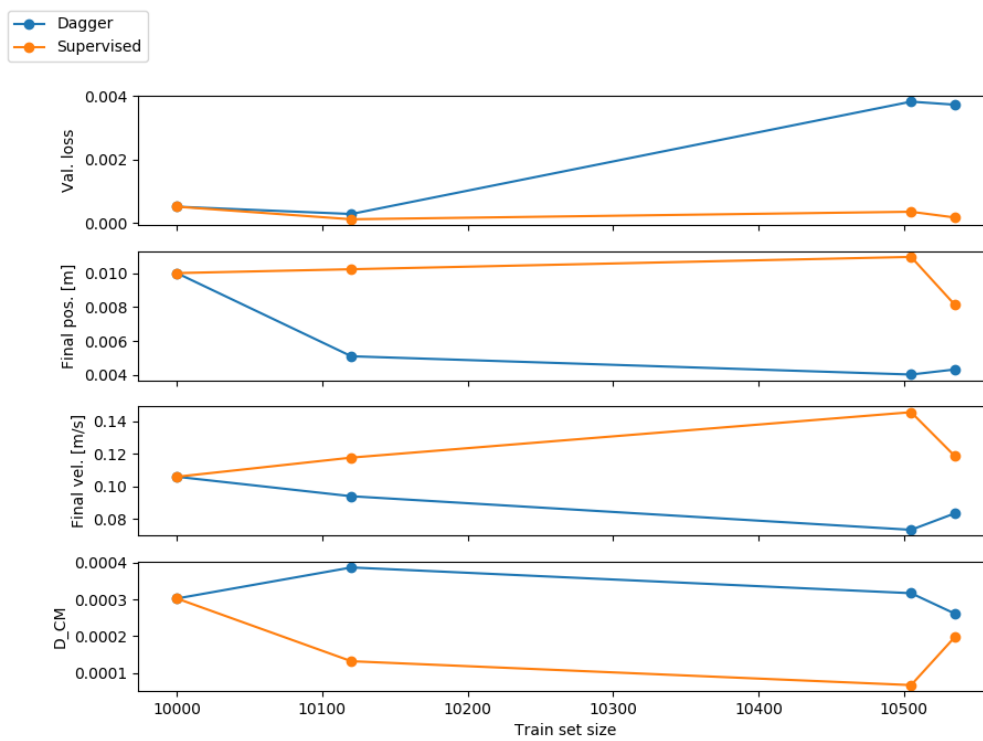


Figure 3.9: EO 1D ELM results DAGger vs Supervised. The subplots represent the validation loss, the average final position, the average final velocity and the Cramer-Von Mises distance.

3.4 3D case

Once DAgger has been successfully applied to the 1D case, it is now time to tackle the more complete 3D case.

3.4.1 Problem formulation

Consider a lander subject to the downward gravity force and a 3D vector control force. It starts at time $t_0 = 0$ at initial position $[x_0, y_0, z_0]$ with initial velocity $[v_{x0}, v_{y0}, v_{z0}]$, and it has to reach the final state $x_f = y_f = z_f = 0$ m and $v_{xf} = v_{yf} = v_{zf} = 0$ m/s at time t_f . The equations of motion are:

$$\begin{cases} \dot{x} = v_x \\ \dot{y} = v_y \\ \dot{z} = v_z \\ \dot{v}_x = a_x \\ \dot{v}_y = a_y \\ \dot{v}_z = a_z - g \end{cases} \quad BC : \quad \begin{cases} x(0) = x_0 \\ y(0) = y_0 \\ z(0) = z_0 \\ v_x(0) = v_{x0} \\ v_y(0) = v_{y0} \\ v_z(0) = v_{z0} \end{cases} \quad \begin{cases} x(t_f) = 0 \\ y(t_f) = 0 \\ z(t_f) = 0 \\ v_x(t_f) = 0 \\ v_y(t_f) = 0 \\ v_z(t_f) = 0 \end{cases} \quad (3.11)$$

The goal is to train a network able to give for each state of the trajectory a control acceleration $\mathbf{a} = a_x \hat{\mathbf{x}} + a_y \hat{\mathbf{y}} + a_z \hat{\mathbf{z}}$, such that the lander reaches the final state minimizing the cost function:

$$J_{EO} = \min \int_0^{t_f} \mathbf{a} \cdot \mathbf{a} \, dt \quad (3.12)$$

where $\mathbf{v} = v_x \hat{\mathbf{x}} + v_y \hat{\mathbf{y}} + v_z \hat{\mathbf{z}}$.

The input features and target are the same as in the 1D case, but in three dimensions:

- Input features:

$$X = [x, y, z, v_x, v_y, v_z, t_{go}] \quad X \in \mathbb{R}^{7 \times 1}$$

- Target:

$$y = [a_x, a_y, a_z] \quad y \in \mathbb{R}^{3 \times 1}$$

3.4.2 Train and test dataset generation

For the EO 3D landing problem, 1000 optimal trajectory are generated with ZEM/ZEV for the train set, for a total of approximately 1.000.000 states (even in this case, it is not necessary to use all these states for the train set). The initial conditions for these trajectories are sampled uniformly randomly within the following ranges:

Variable	Min. value	Max. value
x_0	1500 m	2000 m
y_0	-100 m	100 m
z_0	1000 m	1500 m
v_{x0}	-60 m/s	-50 m/s
v_{y0}	-10 m/s	10 m/s
v_{z0}	-30 m/s	-20 m/s

Table 3.4: EO 3D train set initial condition

The train set is visualized in Fig. 3.10, where the red volume indicates the zone where the initial conditions are sampled:

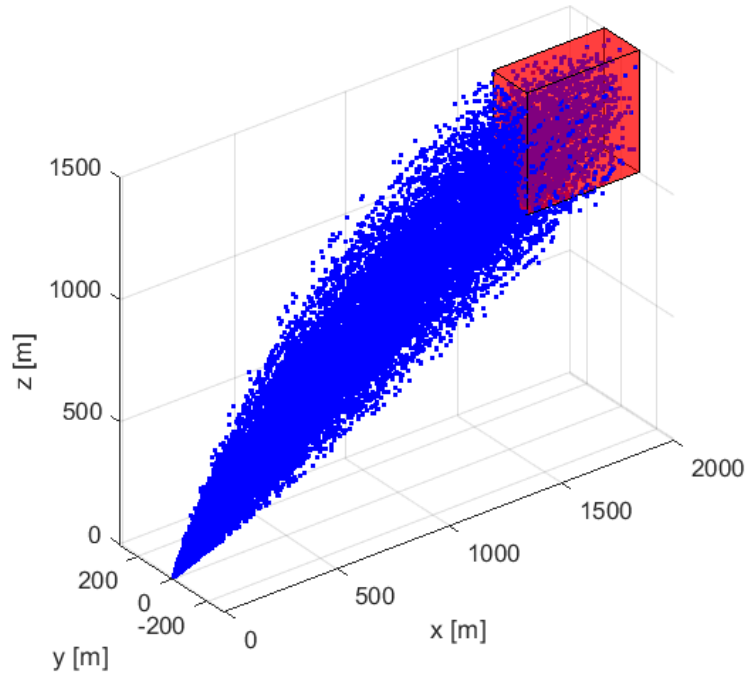


Figure 3.10: EO 3D train set

For each DAgger iteration, 100 new optimal trajectories are generated as test set. In this way, at each iteration the Monte Carlo simulation is run on different trajectories.

3.4.3 Dynamics simulator

The dynamics simulator for the 3D case is the same as the one for 1D case, described in section 3.3.3, with the appropriate modifications for the 3D case. The pseudo-code is:

For each initial condition of the test trajectories:

1. Initialize position $x = x_0$, $y = y_0$, $z = z_0$ and velocity $v_x = v_{x0}$, $v_y = v_{y0}$, $v_z = v_{z0}$
2. Compute t_{go} solving eq. 3.7
3. Apply MinMaxScaler to input features $X = [x_0, y_0, z_0, v_{x0}, v_{y0}, v_{z0}, t_{go}]$, and predict control acceleration $y_{pred} = [a_x, a_y, a_z]$
4. Integrate equation of motion 3.11 with a time step of 0.1 s
5. If the altitude $z > 0$ m and the vertical velocity $v_z < 0$ m/s go back to point 2, otherwise interrupt integration

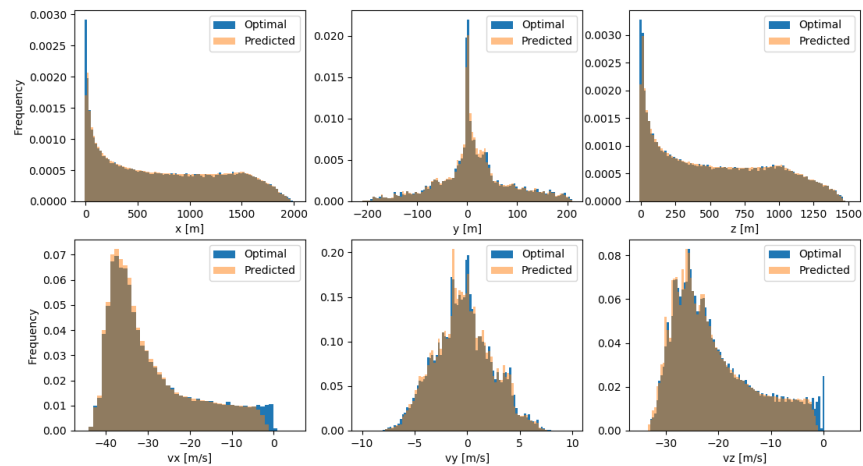
3.4.4 DAgger procedure

The application of DAgger is very similar to the procedure described in section 3.3.4 for the 1D case:

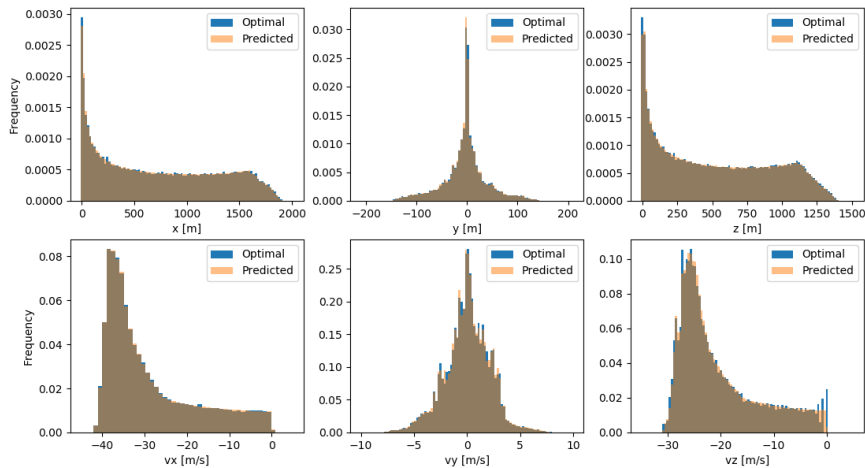
1. The train set is generated with ZEM/ZEV and the ML model is trained.
2. A Monte Carlo simulation is performed: the current policy is run with the test trajectories to obtain the predicted trajectories.
3. The predicted trajectories are labeled with ZEM/ZEV. The predicted states where the prediction error $y_{err} = ||y_{pred} - y_{opt}||$ is bigger than a threshold ϵ_{min} and below a threshold ϵ_{max} are collected.
4. The train set is augmented with the states collected in the previous point. The procedure is repeated going back to point 1 and training a new model on the augmented train set.

3.4.5 Visualize DAGger benefits

It is still possible to plot the predicted state distribution over the optimal one for each dimension of position and velocity. An example is shown in Fig. 3.11, in which it can be observed again how the predicted distribution improves after DAGger, in particular in the final states.



(a) Before DAGger



(b) After DAGger

Figure 3.11: 3D State distribution, before and after DAGger

3.4.6 Deep Neural Network

The original size of the 3D train set with 1000 trajectories is approximately 1.000.000 states. With the same considerations taken for the 1D case, for the initial train set of DNN, only 30.000 states are sampled randomly from the original train set. Also like the 1D case, the network complexity increases over the DAgger iterations: for example, Fig. 3.12 shows that on the original train set (iteration 0), the architecture chosen has 2 hidden layers and 64 neurons per layer, while after a few DAgger iterations (iteration 3), the architecture chosen has 5 hidden layers and 256 neurons per layer.

Layer (type)	Output Shape	Param #	Connected to
main_input (InputLayer)	(None, 7)	0	
dense_1 (Dense)	(None, 64)	512	main_input[0][0]
dense_2 (Dense)	(None, 64)	4160	dense_1[0][0]
x (Dense)	(None, 1)	65	dense_2[0][0]
y (Dense)	(None, 1)	65	dense_2[0][0]
z (Dense)	(None, 1)	65	dense_2[0][0]
Total params: 4,867 Trainable params: 4,867 Non-trainable params: 0			

(a) Iteration 0

Layer (type)	Output Shape	Param #	Connected to
main_input (InputLayer)	(None, 7)	0	
dense_1 (Dense)	(None, 256)	2848	main_input[0][0]
dense_2 (Dense)	(None, 256)	65792	dense_1[0][0]
dense_3 (Dense)	(None, 256)	65792	dense_2[0][0]
dense_4 (Dense)	(None, 256)	65792	dense_3[0][0]
dense_5 (Dense)	(None, 256)	65792	dense_4[0][0]
x (Dense)	(None, 1)	257	dense_5[0][0]
y (Dense)	(None, 1)	257	dense_5[0][0]
z (Dense)	(None, 1)	257	dense_5[0][0]
Total params: 265,987 Trainable params: 265,987 Non-trainable params: 0			

(b) Iteration 3

Figure 3.12: EO 3D DNN architecture

The following hyperparameters are common to all the architectures:

- Learning rate: 10^{-3} , decreases by a factor of 0.9 each time the validation loss does not decrease for 5 consecutive epochs.
- Mini-batch size: 8, same used by Izzo in [1].
- Activation function: ReLU for all layers and outputs, except tanh for y output.

- Normalization: input features are rescaled in a range $[0, 100]$ using the Min-MaxScaler of Keras.

The following hyperparameters, instead, are searched again for each iteration:

- Number of hidden layers.
- Number of neurons per layer.

Fig. 3.13 shows an example of TensorBoard: in this case the architecture with 2 layers and 64 neurons per layer and tanh activation is chosen (actually tanh activation only for y output), because it reaches a validation loss of $5.52 \cdot 10^{-4}$, by far the smallest with respect to all the other architectures tested. The intuition behind the fact that a tanh activation performs better for the y output, is that, given this problem and symmetric initial conditions, the values of y assume both positive and negative values. This makes tanh, which is non saturating on the origin, preferable to ReLU, which outputs zero for all the negative inputs.

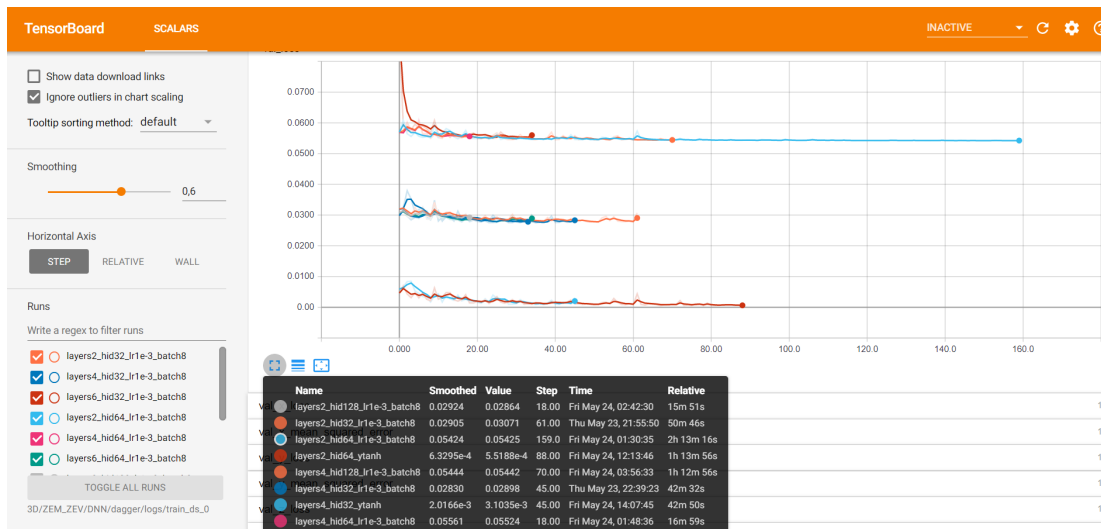


Figure 3.13: Choosing the best architecture with TensorBoard

Fig. 3.14 represents an example of regression curve for each component of the target acceleration. It is quite unclear the presence of the outlier points in each

curve. At first sight it may seem that the quality of the training is very bad, but the Monte Carlo simulation proved that these outliers do not prevent the model to obtain good performance at simulation time. Still, these regression curves do suggest that it should be possible to find an even better architecture to train.

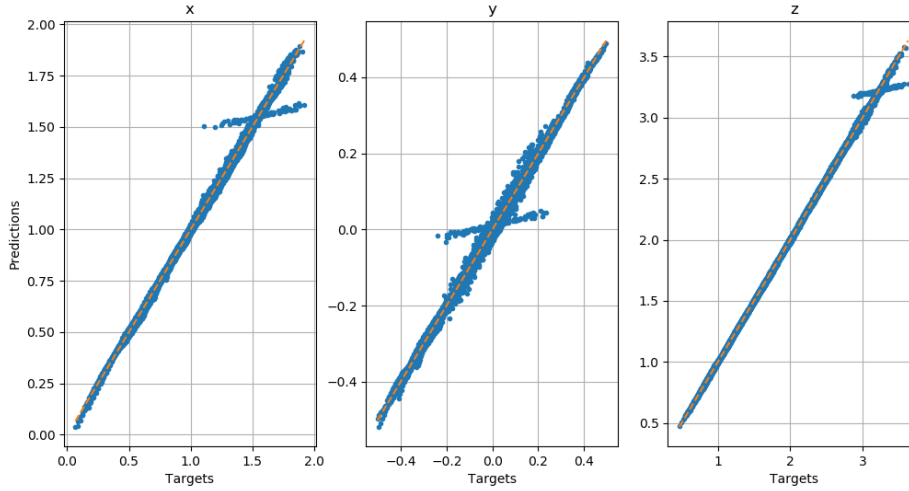


Figure 3.14: EO 3D DNN example of regression curve

The performances of DNN over the DAgger iterations are reported in Table 3.5, and compared to classical supervised learning in Fig. 3.15. Since there are three dimensions, the average final position and velocity are defined as following:

$$\text{Pos: } \sqrt{x_f^2 + y_f^2 + z_f^2}$$

$$\text{Vel: } \sqrt{v_{x_f}^2 + v_{y_f}^2 + v_{z_f}^2}$$

Iter	Val loss	Pos [m]	Vel [m/s]	D_{CM}	New data	Data
0	$5.5 \cdot 10^{-2}$	2.03	1.39	$9.5 \cdot 10^{-4}$	-	30,000
1	$5.2 \cdot 10^{-2}$	0.60	1.05	$3.1 \cdot 10^{-4}$	1674	31,674
2	$5.7 \cdot 10^{-2}$	0.85	0.64	$1.5 \cdot 10^{-4}$	1602	33,276
3	$1.1 \cdot 10^{-1}$	0.79	0.75	$1.1 \cdot 10^{-3}$	3695	36,971

Table 3.5: EO 3D DNN results DAgger

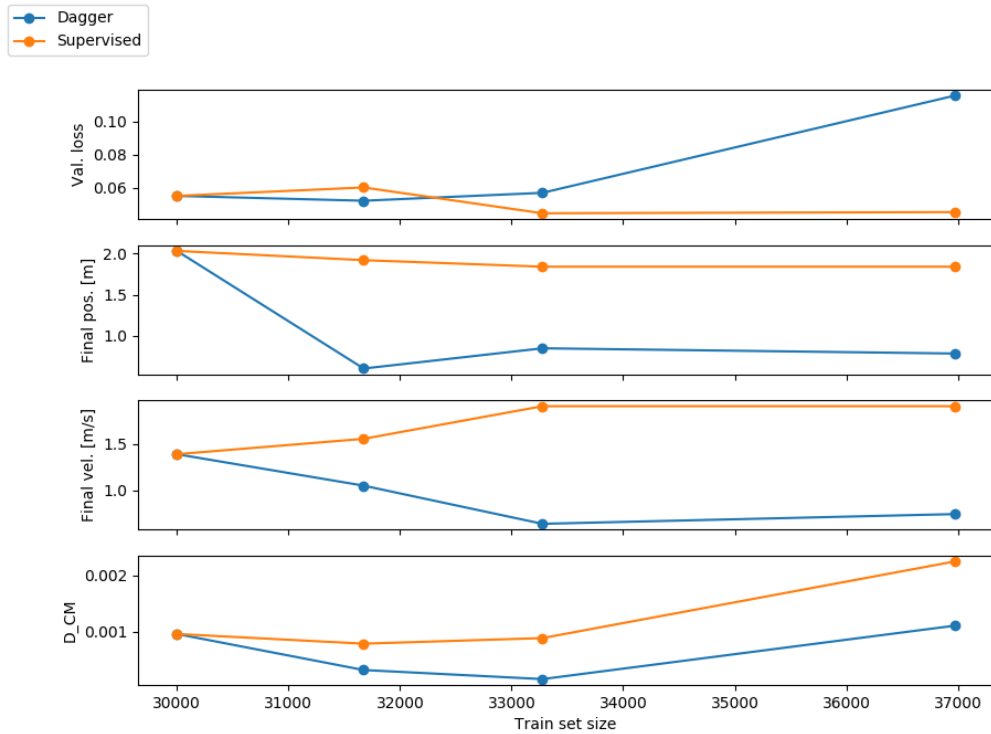


Figure 3.15: EO 3D DNN results DAGger vs Supervised. The subplots represent the validation loss, the average final position, the average final velocity and the Cramer-Von Mises distance.

3.4.7 Extreme Learning Machine

For the ELM it is chosen to use an initial train set with just 10.000 states. In order to choose the number of neurons for each iteration, it is used the same procedure of the 1D case. The input features are rescaled in a range $[0, 1]$ with the MinMaxScaler of Keras.

Fig. 3.16 shows that the optimal number of neurons (where the validation loss reaches a minimum) increases along the DAGger iterations. This is the same phenomena encountered with the DNN: augmenting the train size with DAGger, a more complex network is required.

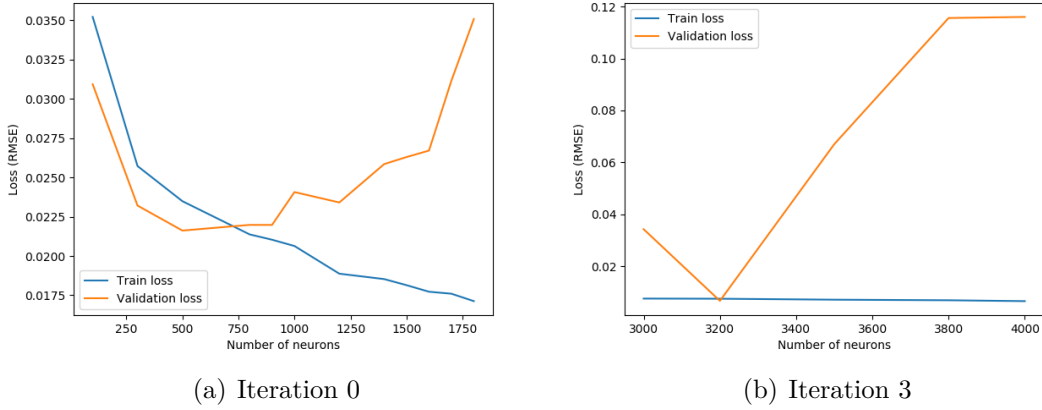


Figure 3.16: EO 3D ELM architecture

It is interesting to observe the presence of outliers in the regression curves if Fig. 3.17 even in this case, even though the ELM proved to be very powerful on the regression problem and achieved better results than the DNN on the Monte Carlo simulation.

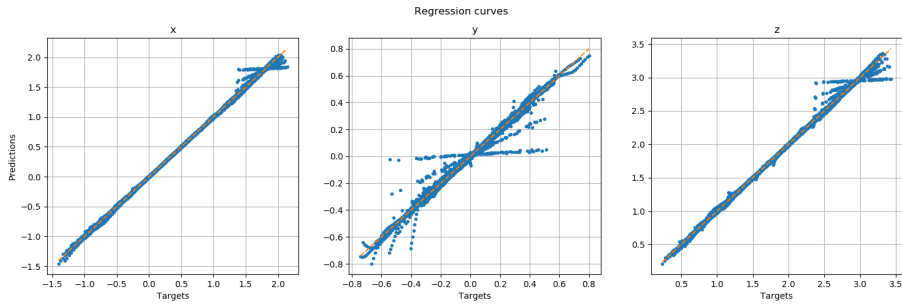


Figure 3.17: EO 3D ELM example of regression curve

The performances of DNN over the DAgger iterations are reported in Table 3.5, and compared to classical supervised learning in Fig. 3.15. Position and velocity are defined as in the DNN 3D case.

Iter	Val loss	Pos [m]	Vel [m/s]	D_{CM}	New data	Data
0	$2.0 \cdot 10^{-3}$	2.10	3.22	$4.9 \cdot 10^{-4}$	-	10,000
1	$6.7 \cdot 10^{-4}$	0.69	2.39	$6.6 \cdot 10^{-4}$	604	10,604
2	$7.9 \cdot 10^{-4}$	0.14	0.41	$3.6 \cdot 10^{-4}$	1107	11,711
3	$6.2 \cdot 10^{-1}$	0.06	0.33	$7.9 \cdot 10^{-3}$	1942	13,653

Table 3.6: EO 3D ELM results DAGger

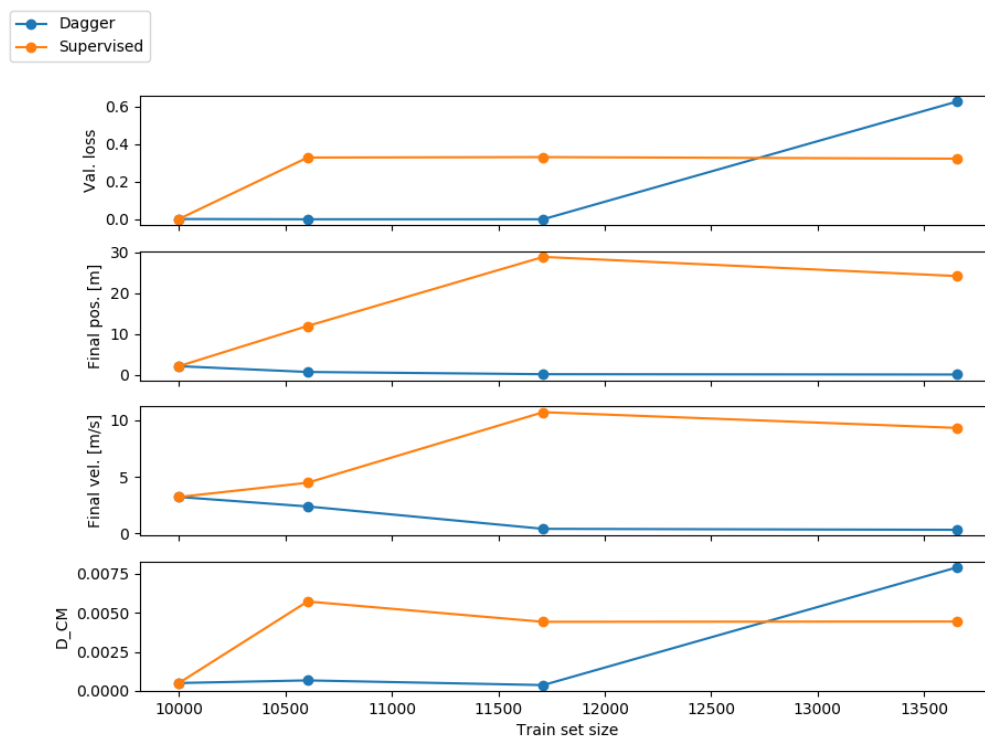


Figure 3.18: EO 3D ELM results DAGger vs Supervised. The subplots represent the validation loss, the average final position, the average final velocity and the Cramer-Von Mises distance.

Chapter 4

Fuel Optimal Landing Problem

While the EO problem was mostly used as a case study to test and understand DAgger, the FO problem is more interesting for practical application, since it represents an optimal landing where the fuel consumption is minimized. Furthermore, the module of the control thrust is constrained between a minimum value T_{min} and a maximum value T_{max} as it would be in a real situation.

Consider a lander in a 3D Cartesian reference system with axes $\{\hat{x}, \hat{y}, \hat{z}\}$, subject to the gravity force \mathbf{g} and control thrust \mathbf{T} . It starts at time t_0 from initial position \mathbf{r}_0 with initial velocity \mathbf{v}_0 , and it has to reach the target state $\mathbf{r}_f, \mathbf{v}_f$ at time t_f . The equations of motion are:

$$\begin{cases} \dot{\mathbf{r}} = \mathbf{v} \\ \dot{\mathbf{v}} = \frac{\mathbf{T}}{m} + \mathbf{g} \\ \dot{m} = -\frac{\|\mathbf{T}\|}{I_{sp} \cdot g_0} \end{cases} \quad BC : \begin{cases} \mathbf{r}(t_0) = \mathbf{r}_0 \\ \mathbf{v}(t_0) = \mathbf{v}_0 \\ m(t_0) = m_0 \end{cases} \quad \begin{cases} \mathbf{r}(t_f) = \mathbf{r}_f \\ \mathbf{v}(t_f) = \mathbf{v}_f \end{cases} \quad (4.1)$$

where $g = 1.62 \text{ N/kg}$ is the gravitational field of the Moon, $g_0 = 9.81 \text{ N/kg}$ the gravitational field of the Earth and $I_{sp} = 200 \text{ s}$ the specific impulse of the lander's

engine. The cost function is:

$$J_{FO} = \min \int_{t_0}^{t_f} \|\mathbf{T}\| dt \quad (4.2)$$

4.1 GPOPS

GPOPS, which stands for *General Pseudospectral Optimal Control Software*, is the software used to generate optimal trajectories for the FO problem. It uses a direct method for solving the two-points-boundary-value-problem (TPBVP), and for this reason the solution is not very accurate, compared to a software using an indirect method. Anyway, it is accurate enough to generate a dataset to train a ML model. GPOPS has in particular one parameter called *mesh tolerance* which influences the accuracy of the solution. Fig. 4.1 shows two examples of GPOPS solutions for a 1D trajectory, one with a mesh tolerance of 10^{-4} and one with a mesh tolerance of 10^{-8} . It is clear that a smaller mesh tolerance gives a more accurate solution (in particular for the output) and also more points.

It is known from optimal control theory for a fuel optimal landing problem that the thrust profile is bang-bang. For this reason, the output is categorical: 0 means that the thrust is minimum (1000 N), 1 means that it is maximum (3400 N). The thrust is then computed as $T = 1000 + 2400 \cdot T_c$ [N]. As Fig. 4.1 (a) and (b) show, in the output solution there is in both cases a middle point during the switching time: that is a collateral effect of the direct method used to solve the TPBVP. Different parameters, other than the mesh tolerance, have been tweaked in order to try to remove it, but without success. As a solution, the middle points are just removed from the train set.

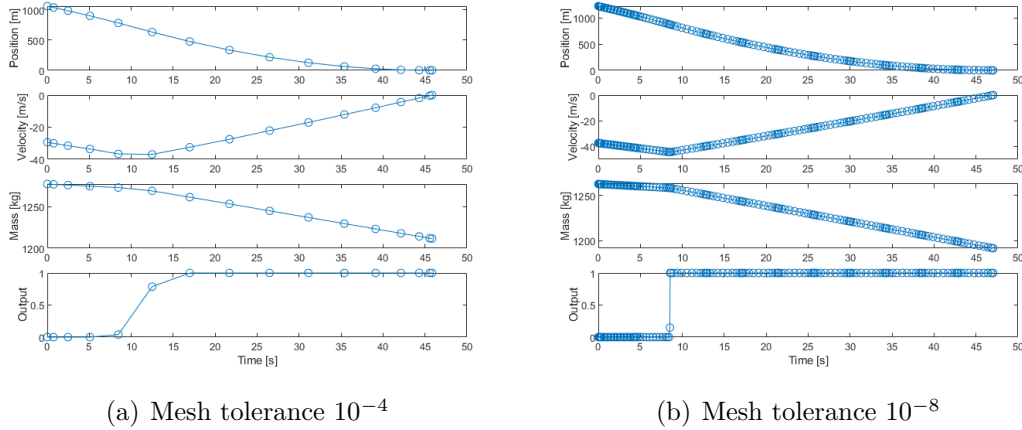


Figure 4.1: Examples of GPOPS solution for 1D case

4.2 Choice of the input features

It is worth to explain the choice of using the mass as input feature. In fact, Izzo [1] used a DNN for a FO problem, and he assumed complete knowledge of position and velocity, but did not use the mass as input feature. However, the state of the dynamics includes the mass, which indeed is a variable in the equations of motions 4.1. The DNNs and ELMs, unlike RNNs which consider the observation's history, only take one observation at a time to predict the output: this means that it is needed to respect the Markov Property discussed in section 2.3, otherwise the model lacks information about the spacecraft's state, and may make wrong predictions because of it. For example, consider the following initial conditions for a 1D problem:

- $X_1 = [450 \text{ m}, -40 \text{ m/s}, 1000 \text{ kg}]$
- $X_2 = [450 \text{ m}, -40 \text{ m/s}, 950 \text{ kg}]$

After generating a trajectory from these initial conditions with GPOPS, the first state is labeled with a maximum thrust $y_1 = 1$, while the second state is labeled

with a minimum thrust $y_2 = 0$. This explains why the mass needs to be used as input feature.

Note that this matter was not present in the EO problem, because the control acceleration was not constrained and so the dynamics was ultimately not dependent on the mass, as eqs. 3.2 show.

Finally, it is pointed out that the mass variation is usually not measured on-board by the spacecrafts, but it actually could be easily calculated in real time from the initial mass and the control thrust history:

$$m(t_k) = m(t_0) - \sum_{i=1}^k \frac{\|\mathbf{T}_{i-1}\|}{I_{sp} \cdot g_0} (t_k - t_{k-1}) \quad (4.3)$$

4.3 1D case

4.3.1 Problem formulation

Consider a lander subject to the gravity force and the control force. It starts at time $t_0 = 0$ s from the initial altitude h_0 with the initial velocity v_0 and the initial mass m_0 , and it has to reach the final state $h_f = 0$ m and $v_f = 0$ m/s at time t_f . The equations of motion are:

$$\begin{cases} \dot{h} = v \\ \dot{v} = \frac{T}{m} - g \\ \dot{m} = -\frac{T}{I_{sp} \cdot g_0} \end{cases} \quad BC : \begin{cases} h(0) = h_0 \\ v(0) = v_0 \\ m(0) = m_0 \end{cases} \quad \begin{cases} h(t_f) = 0 \\ v(t_f) = 0 \end{cases} \quad (4.4)$$

where h is the altitude, v the vertical velocity, T the control thrust. Unlike the EO problem, now the thrust is constrained to be between 1000 N and 3400 N. As will be explained in section 4.3.4, this causes practical issues during the DAgger procedure that need to be solved with some trick.

The goal is to train a network able to give for each state of the trajectory a control thrust T , such that the lander reaches the final state minimizing the cost function:

$$J_{FO} = \min \int_0^{t_f} \|T\| dt \quad (4.5)$$

In order to train the model, supervised learning techniques are used. The altitude h , velocity v and mass m are used as input features. The target t_c is categorical (0-1).

- Input features:

$$X = [h, v, m] \quad X \in \mathbb{R}^{3 \times 1}$$

- Target:

$$y = t_c \quad y \in \mathbb{R}^{1 \times 1}$$

4.3.2 Train and test dataset generation

For the FO 1D landing problem, 500 optimal trajectory are generated with GPOPS, setting the mesh tolerance of GPOPS to 10^{-5} , obtaining a total of approximately 10.000 states. The initial conditions for these trajectories are sampled uniformly randomly within the following ranges:

Variable	Min. value	Max. value
h_0	1000 m	1500 m
v_0	-40 m/s	-20 m/s
m_0	1200 kg	1400 kg

Table 4.1: FO 1D train set initial condition

The train set is visualized in Fig. 4.2, both from a 2D and 3D perspective. The points in red represent states in which the optimal thrust is maximum, while the points in blue represent states where the optimal thrust is minimum.

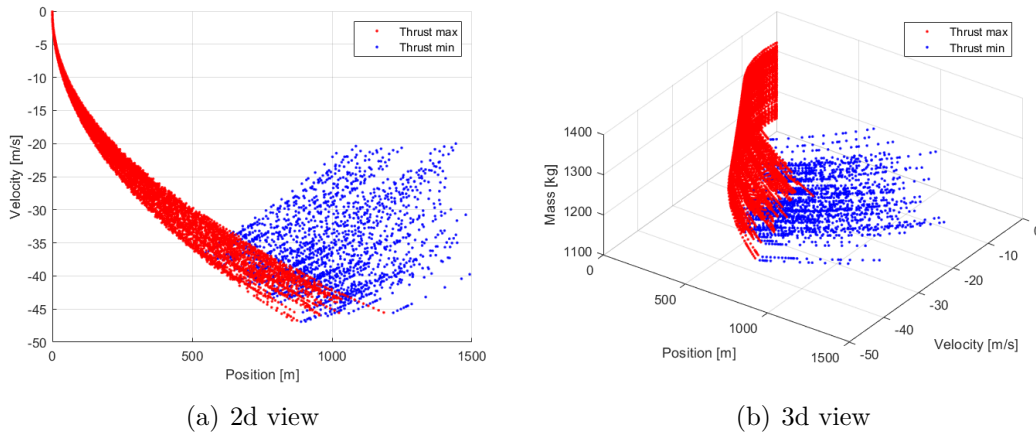


Figure 4.2: FO 1D train set

One may wonder why the train set is generated varying the initial condition of the mass; after all, it is reasonable to assume that the lander will have a fixed known mass when it begins the landing. Before explaining the reason of this choice, it is introduced a plot called *partial dependence plot* (PDP), which is a tool used in ML to get an insight of the model's behavior. Partial dependence plots show how each variable affects the model's predictions. This is done making one variable to vary, while fixing all other variables, and looking how the predictions of the model depend on that variable (like a partial derivative). Figure 4.3 shows an example of PDP: there are three plots, one where the position is varied, one for the velocity and one for the mass. When they do not vary, position is fixed to 700 m , velocity to -40 m/s and mass to 1300 kg . The first two plots look correct, in fact with parity of velocity and mass, it is intuitive that the thrust is maximum if the position is lower; and it is intuitive that, with parity of position and mass, the thrust is maximum if the negative velocity is higher in absolute value. What looks wrong is the fact that, with parity of position and velocity, the thrust is predicted maximum if the mass is lower.

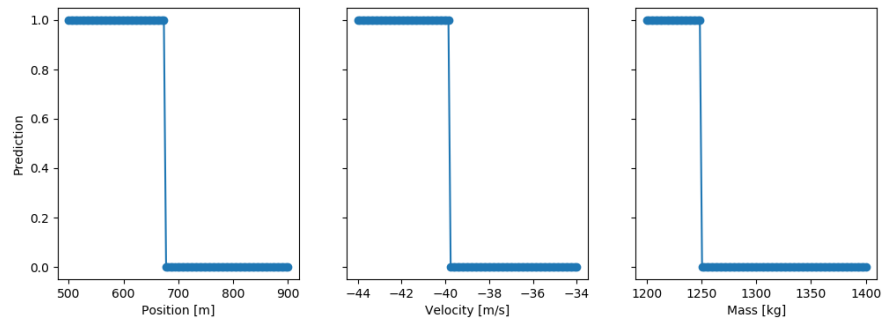


Figure 4.3: Partial dependence plot 1

This happens because if all the training trajectories begin with the same initial mass, the model learns a wrong correlation between the mass and the output. From an intuitive point of view, this is because, if there is a limited space of initial conditions, what the model sees is just a pattern that links low mass (i.e. when the lander is in an advanced phase of the landing) to maximum thrust. If, instead, the training trajectories are generated by varying the initial mass, the PDP obtained is the one represented in Fig. 4.4, and now the third plot shows the behavior that is expected.

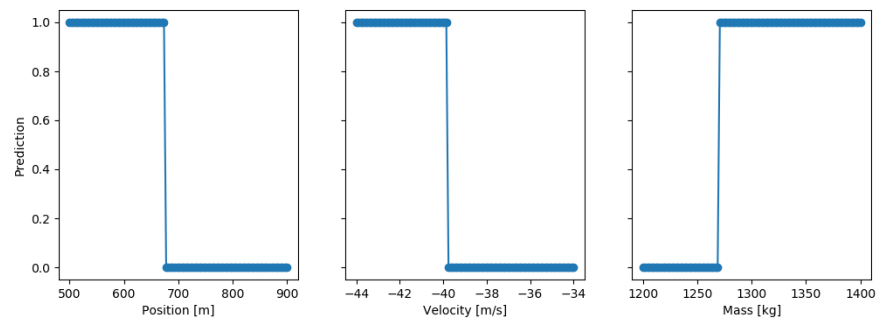


Figure 4.4: Partial dependence plot 2

4.3.3 Dynamics simulator

The dynamics simulator works like the one implemented for the EO problem, with a few changes.

For each initial condition of the test trajectories:

1. Initialize position $h = h_0$, velocity $v = v_0$ and mass $m = m_0$
2. Apply MinMaxScaler to input features $X = [h, v, m]$, and predict thrust $y_{pred} = t_c$
3. Compute the thrust as $T = 1000 + 2400 \cdot t_c$ [N]; integrate equation of motion 4.4 with a time step of 0.1 s (10 Hz)
4. If the altitude $h > 0$ m and the velocity $v < 0$ m/s go back to point 2, otherwise interrupt integration

4.3.4 DAgger procedure

Here it is explained how the DAgger algorithm already described in section 2.4 is implemented for the FO one-dimensional problem:

1. The train set is generated with GPOPS and the ML model is trained.
2. A Monte Carlo simulation is performed: the current policy is run with the test trajectories to obtain the predicted trajectories.
3. The predicted trajectories are labeled with GPOPS. The states where the predicted thrust is wrong ($y_{pred} \neq y_{opt}$) are collected.
4. The train set is augmented with the states collected in the previous point. The procedure is repeated going back to point 1 and training a new model on the augmented train set.

As anticipated in section 4.3.1, the constraint on the thrust levels causes some problems in the DAgger procedure that was not present in the EO problem, where the control acceleration was not bounded. In particular, point 3 of the previous list is not trivial as it seems: it may happen that the predicted trajectories deviate that much from the optimal trajectories that some states are out of the reachability point for the optimal solution, and GPOPS is not able to solve the FO problem from the those states because the thrust is limited (e.g. the altitude is too low and the downward velocity is too high). A few solutions to this problem have been considered:

- Increase the mesh tolerance of GPOPS (e.g. to $10^{-3} - 10^{-2}$). A bigger mesh tolerance reduces the accuracy of the solution, but makes GPOPS more flexible in finding a solution; this is acceptable, since only the first state of the trajectory is of interest for DAgger.
- Just remove the states from where GPOPS is not able to solve the FO problem. This is feasible only for a few states, because it has to be done manually each time that GPOPS is stuck in trying to solve the FO problem. If a lot of states are out of the reachability zone, then consider the third solution.
- Start the first iteration correcting the predicted trajectories only until a certain point (e.g. half of each trajectory); then, gradually increase the stop point with the iterations. In fact, the trained policy is supposed to improve along the iterations and therefore it should be able to reach states more and more near to the target state without deviating too much from the optimal trajectory.

A mix of all the three solutions proposed has been applied.

4.3.5 Visualize DAgger benefits

As for the EO 1D case, it is possible to visualize how applying DAgger makes the predicted trajectories to converge better to the target state. Figure 4.5 and 4.6, respectively from a 2D and 3D perspective, show how augmenting the train dataset helps the predicted trajectory to converge to the target state $[0 \text{ m}, 0 \text{ m/s}]$.

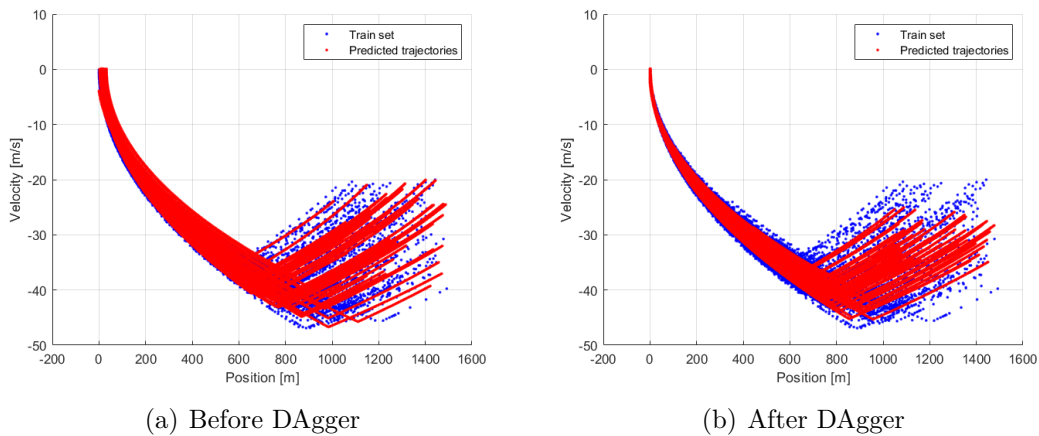


Figure 4.5: Predicted trajectories over train set, before and after DAgger, 2D view

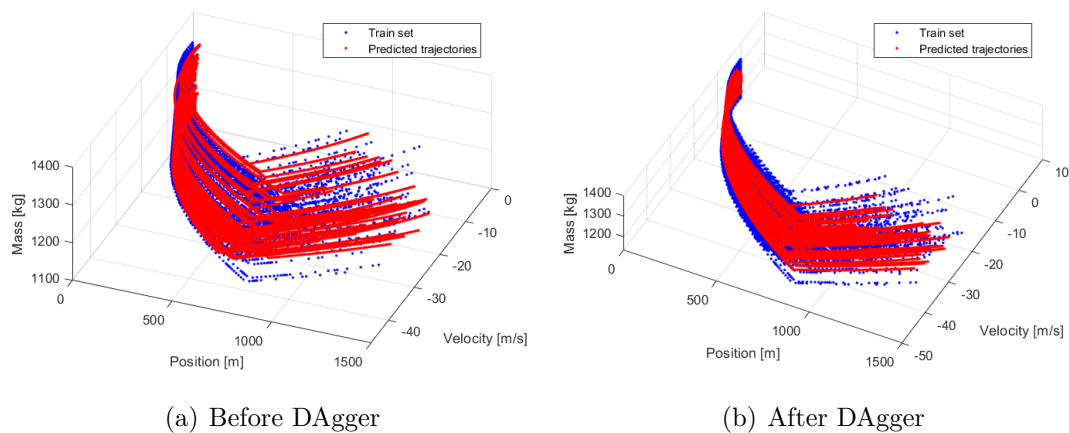


Figure 4.6: Predicted trajectories over train set, before and after DAgger, 3D view

4.3.6 Deep Neural Network

The original size of the 1D train set with 500 trajectories is approximately 10.000 states. Like in the other cases, the network complexity increases over the DAgger iterations, as Fig. 4.7 shows.

Layer (type)	Output Shape	Param #
main_input (InputLayer)	(None, 3)	0
dense_1 (Dense)	(None, 32)	128
dense_2 (Dense)	(None, 32)	1056
clas_output (Dense)	(None, 2)	66
Total params: 1,250 Trainable params: 1,250 Non-trainable params: 0		

(a) Iteration 0

Layer (type)	Output Shape	Param #
main_input (InputLayer)	(None, 3)	0
dense_1 (Dense)	(None, 64)	256
dense_2 (Dense)	(None, 64)	4160
dense_3 (Dense)	(None, 64)	4160
clas_output (Dense)	(None, 2)	130
Total params: 8,706 Trainable params: 8,706 Non-trainable params: 0		

(b) Iteration 4

Figure 4.7: FO 1D DNN architecture

The following hyperparameters are common to all the architectures:

- Learning rate: 10^{-3} , decreases by a factor of 0.9 each time the validation loss does not decreases for 5 consecutive epochs.
- Mini-batch size: 8, same used by Izzo in [1].
- Activation function: ReLU, same used by Izzo in [1].
- Normalization: input features are rescaled in a range $[0, 100]$ using the Min-MaxScaler of Keras.

The following hyperparameters, instead, are searched again for each iteration:

- Number of hidden layers.
- Number of neurons per layer.

An example of confusion matrix is represented in Fig. 4.10. It shows that over a total of 5000 test states:

- 3547 are correctly predicted as max thrust (100% of the states with max thrust)
- 1442 are correctly predicted as min thrust (99.2% of the states with min thrust)
- 11 are wrongly predicted as max thrust (0.8% of the states with min thrust)
- 0 are wrongly predicted as min thrust (0% of the states with max thrust)

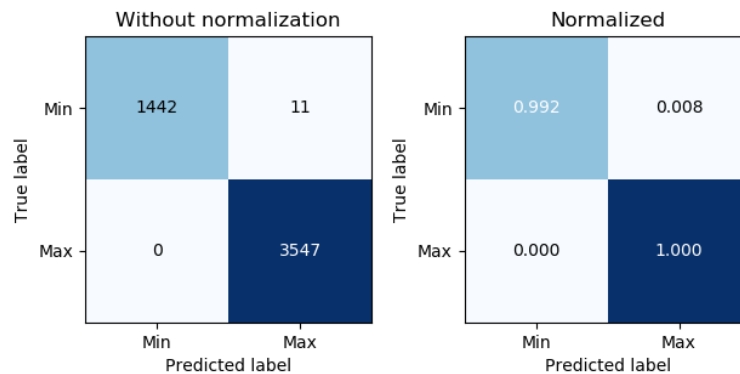


Figure 4.8: FO 1D DNN Confusion matrix

The performances of DNN over the DAgger iterations are reported in Table 4.2, and compared to classical supervised learning in Fig. 4.9.

Iter	Val acc	Pos [m]	Vel [m/s]	D_{CM}	New data	Data
0	0.99	13.6	3.91	$1.9 \cdot 10^{-3}$	-	9,117
1	0.99	3.72	1.94	$1.8 \cdot 10^{-3}$	397	9,514
2	0.99	2.60	4.27	$1.7 \cdot 10^{-3}$	43	9,556
3	0.99	1.54	0.81	$1.3 \cdot 10^{-3}$	293	9,849
4	0.99	0.48	1.02	$8.4 \cdot 10^{-4}$	905	10,754

Table 4.2: FO 1D DNN results DAgger

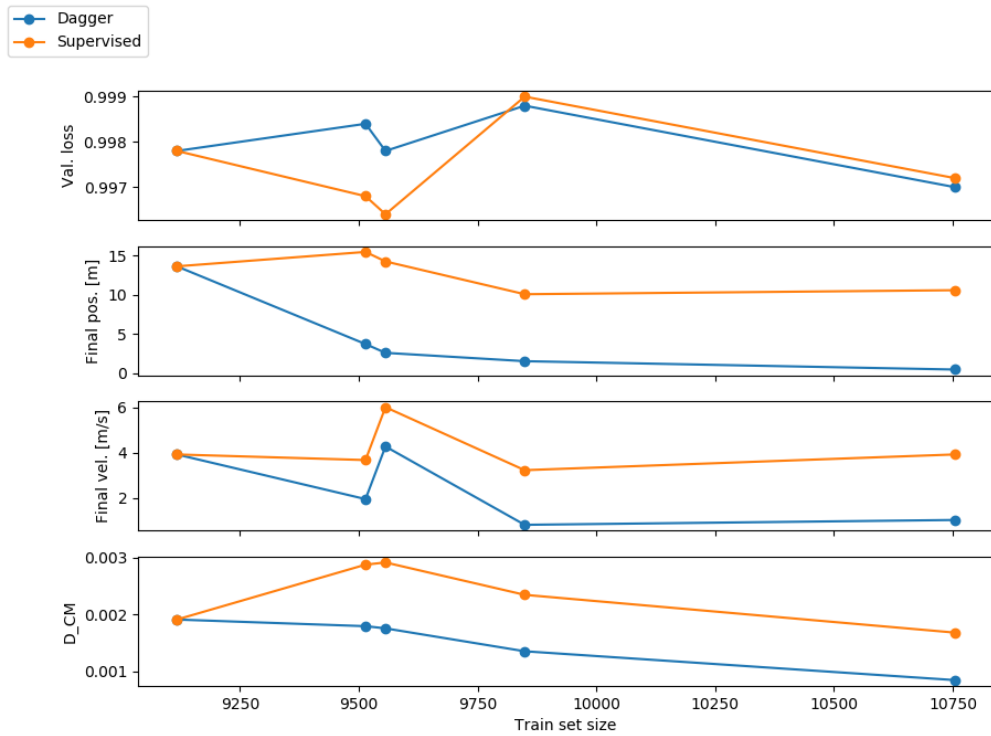


Figure 4.9: FO 1D DNN results DAgger vs Supervised. The subplots represent the validation loss, the average final position, the average final velocity and the Cramer-Von Mises distance.

4.3.7 Extreme Learning Machine

While the ELM proved to be extremely powerful for the regression problem, it showed poor performances on the classification problem. It is not clear this kind of behavior, because in general ELMs are supposed to work well also on classification and multi-classification problems [4]. Figure 4.10 shows an example of classification matrix, while Fig. 4.11 shows an example of distribution of states. Application of DAgger did not bring any improvement on the performances of ELM, for this reason ELM is not used for the FO problem where classification is required.

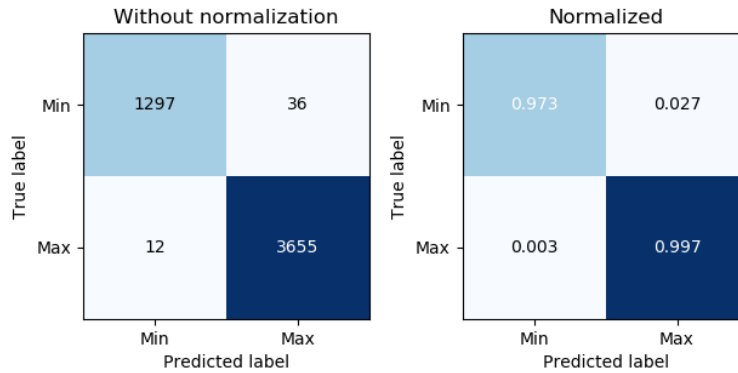


Figure 4.10: FO 1D ELM Confusion Matrix

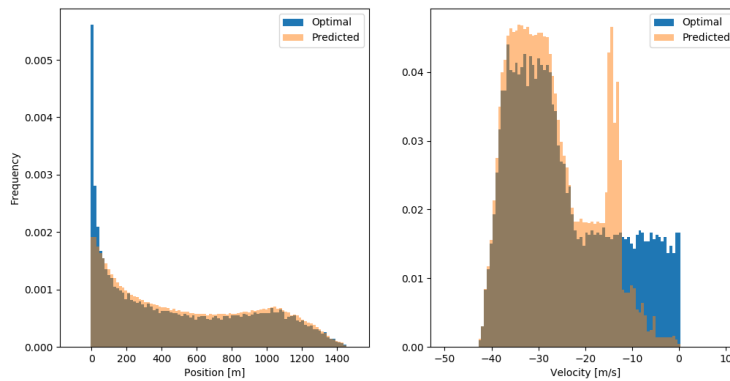


Figure 4.11: FO 1D ELM State Distribution

4.4 3D case

Once DAgger has been successfully applied to the 1D case, it is now time to tackle the more complete and realistic 3D case.

4.4.1 Problem formulation

Consider a lander subject to the downward gravity force and a 3D vector control force. It starts at time $t_0 = 0$ at initial position $[x_0, y_0, z_0]$ with initial velocity $[v_{x0}, v_{y0}, v_{z0}]$ and initial mass m_0 , and it has to reach the final state $x_f = y_f = z_f = 0$ m and $v_{xf} = v_{yf} = v_{zf} = 0$ m/s at time t_f . The equations of motion are:

$$\begin{cases} \dot{x} = v_x \\ \dot{y} = v_y \\ \dot{z} = v_z \\ \dot{v}_x = \frac{T_x}{m} \\ \dot{v}_y = \frac{T_y}{m} \\ \dot{v}_z = \frac{T_z}{m} - g \\ \dot{m} = -\frac{\|\mathbf{T}\|}{I_{sp} \cdot g_0} \end{cases} \quad BC : \begin{cases} x(0) = x_0 \\ y(0) = y_0 \\ z(0) = z_0 \\ v_x(0) = v_{x0} \\ v_y(0) = v_{y0} \\ v_z(0) = v_{z0} \\ m(0) = m_0 \end{cases} \quad \begin{cases} x(t_f) = 0 \\ y(t_f) = 0 \\ z(t_f) = 0 \\ v_x(t_f) = 0 \\ v_y(t_f) = 0 \\ v_z(t_f) = 0 \end{cases} \quad (4.6)$$

The goal is to train a network able to give for each state of the trajectory a control thrust \mathbf{T} , such that the lander reaches the final state minimizing the cost function:

$$J_{FO} = \min \int_0^{t_f} \|\mathbf{T}\| dt \quad (4.7)$$

where $\mathbf{T} = T_x \hat{\mathbf{x}} + T_y \hat{\mathbf{y}} + T_z \hat{\mathbf{z}}$. The module of the thrust vector $\|\mathbf{T}\|$ is constrained to be between 1000 N and 3400 N.

The input features are again position, velocity and mass. The target is a vector of 4 components: the first three $[t_x, t_y, t_z]$ are the unity components of the thrust and are numerical, while the last t_c is categorical (0-1): 0 indicates minimum thrust and 1 maximum thrust.

- Input features:

$$X = [x, y, z, v_x, v_y, v_z, m] \quad X \in \mathbb{R}^{7 \times 1}$$

- Target:

$$y = [t_x, t_y, t_z, t_c] \quad y \in \mathbb{R}^{4 \times 1}$$

4.4.2 Train and test dataset generation

For the FO 3D landing problem, 1000 optimal trajectory are generated with GPOPS for the train set, for a total of approximately 250.000 states (even in this case, it is not necessary to use all these states for the train set). The initial conditions for these trajectories are sampled uniformly randomly within the following ranges:

Variable	Min. value	Max. value
x_0	1500 m	2000 m
y_0	-100 m	100 m
z_0	1000 m	1500 m
v_{x0}	-60 m/s	-50 m/s
v_{y0}	-10 m/s	10 m/s
v_{z0}	-30 m/s	-20 m/s
m	1200 kg	1400 kg

Table 4.3: EO 3D train set initial condition

The train set is represented in Fig. 4.12, where the red volume indicates the zone where the initial conditions are sampled.

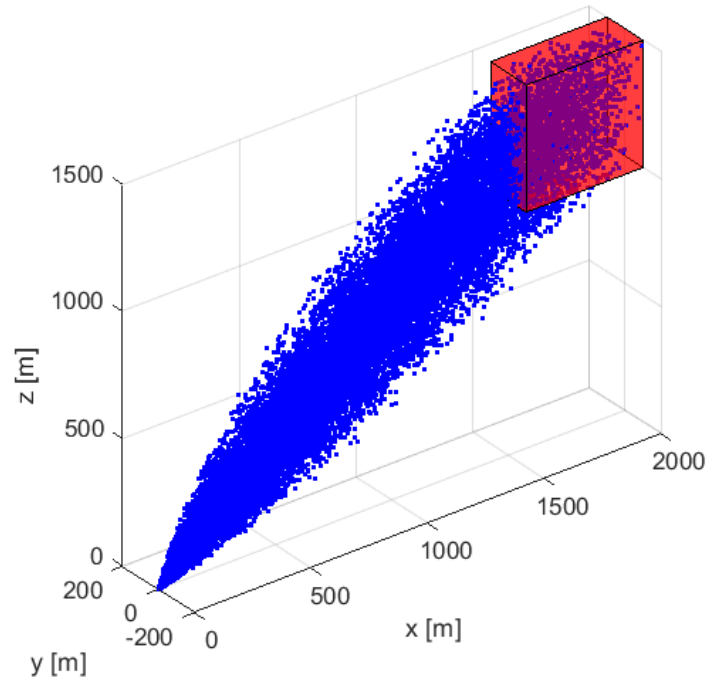


Figure 4.12: FO 3D train set

4.4.3 Dynamics simulator

For each initial condition of the test trajectories:

1. Initialize position $x = x_0$, $y = y_0$, $z = z_0$, velocity $v_x = v_{x0}$, $v_y = v_{y0}$, $v_z = v_{z0}$ and mass $m = m_0$
2. Apply MinMaxScaler to input features $X = [x_0, y_0, z_0, v_{x0}, v_{y0}, v_{z0}, m_0]$, and predict $y_{pred} = [t_x, t_y, t_z, t_c]$
3. Compute the thrust as $\mathbf{T} = [t_x \hat{\mathbf{x}}, t_y \hat{\mathbf{y}}, t_z \hat{\mathbf{z}}] \cdot T$, where $T = 1000 + 2400 \cdot t_c$ [N]; integrate equation of motion 4.6 with a time step of 0.1 s
4. If the altitude $z > 0$ m and the vertical velocity $v_z < 0$ m/s go back to point 2, otherwise interrupt integration

4.4.4 DAgger procedure

1. The train set is generated with GPOPS and the ML model is trained.
2. A Monte Carlo simulation is performed: the current policy is run with the test trajectories to obtain the predicted trajectories.
3. The predicted trajectories are labeled with GPOPS. Since both regression and classification are implied, in order to collect the states for DAgger it is used the following criteria: the predicted states where the prediction error on the regression $y_{err} = ||[t_x, t_y, t_z]_{pred} - [t_x, t_y, t_z]_{opt}||$ is bigger than a threshold ϵ_{min} and below a threshold ϵ_{max} are collected; also, the predicted states where the classification is wrong ($[t_c]_{pred} \neq [t_c]_{opt}$) are collected.
4. The train set is augmented with the states collected in the previous point. The procedure is repeated going back to point 1 and training a new model on the augmented train set.

4.4.5 Deep Neural Network

The original size of the 1D train set with 1000 trajectories is approximately 30.000 states. Like in the other cases, the network complexity increases over the DAgger iterations, as Fig. 4.7 shows.

Layer (type)	Output Shape	Param #	Connected to
main_input (InputLayer)	(None, 7)	0	
dense_1 (Dense)	(None, 32)	256	main_input[0][0]
dense_2 (Dense)	(None, 32)	1056	dense_1[0][0]
dense_3 (Dense)	(None, 32)	1056	dense_2[0][0]
dense_4 (Dense)	(None, 32)	1056	dense_3[0][0]
dense_5 (Dense)	(None, 32)	1056	dense_4[0][0]
dense_6 (Dense)	(None, 32)	1056	dense_5[0][0]
x (Dense)	(None, 1)	33	dense_6[0][0]
y (Dense)	(None, 1)	33	dense_6[0][0]
z (Dense)	(None, 1)	33	dense_6[0][0]
c (Dense)	(None, 2)	66	dense_6[0][0]

Total params: 5,781
Trainable params: 5,781
Non-trainable params: 0

(a) Iteration 0

(b) Iteration 4

Figure 4.13: FO 3D DNN architecture

The following hyperparameters are common to all the architectures:

- Learning rate: 10^{-3} , decreases by a factor of 0.9 each time the validation loss does not decrease for 5 consecutive epochs.
- Mini-batch size: 8, same used by Izzo in [1].
- Activation function: ReLU for all layers and outputs, except tanh for y output.
- Normalization: input features are rescaled in a range $[0, 100]$ using the Min-MaxScaler of Keras.

The following hyperparameters, instead, are searched again for each iteration:

- Number of hidden layers.
- Number of neurons per layer.

The performances of DNN over the DAgger iterations are reported in Table 4.4, and compared to classical supervised learning in Fig. 4.14. The average final

position and velocity are defined as following:

$$\text{Pos: } \sqrt{x_f^2 + y_f^2 + z_f^2}$$

$$\text{Vel: } \sqrt{v_{x_f}^2 + v_{y_f}^2 + v_{z_f}^2}$$

Iter	Val loss	Pos [m]	Vel [m/s]	D_{CM}	New data	Data
0	0.08	8.03	3.91	$1.9 \cdot 10^{-3}$	-	9,117
1	0.05	6.74	1.94	$1.8 \cdot 10^{-3}$	397	9,514
2	0.06	6.54	4.27	$1.7 \cdot 10^{-3}$	43	9,556
3	0.05	1.54	0.81	$1.3 \cdot 10^{-3}$	293	9,849
4	0.09	0.48	1.02	$8.4 \cdot 10^{-4}$	905	10,754

Table 4.4: FO 3D DNN results DAGger

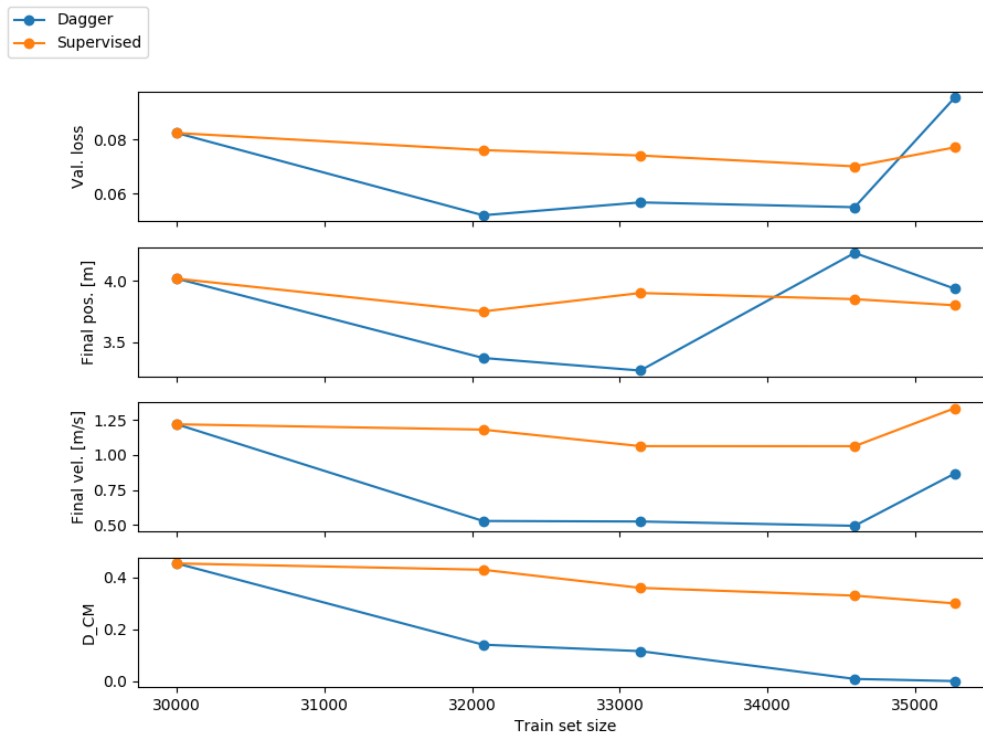


Figure 4.14: FO 3D DNN results DAGger vs Supervised. The subplots represent the validation loss, the average final position, the average final velocity and the Cramer-Von Mises distance.

Chapter 5

Conclusions and future work

5.1 Conclusions

In this work it has been demonstrated that the DAgger approach can be applied to the optimal landing problem, and that it effectively improves the performance of the Machine Learning model at prediction time. The energy optimal and fuel optimal landing problem have been analyzed, both in 1D and 3D cases, using different kind of ML models such as Deep Neural Netowrk and Extreme Learning Machine. Supervised imitation learning has been applied for the first time to EO landing problems, using the time to go as input feature. It has been shown how ELMs are a powerful tool for regression landing problems, while they are a poor choice for classification problems.

A great number of experiments have been performed to choose the train set size, the model's architecture, the input features and how to implement DAgger. It has been proven that the application of DAgger outperformes classic supervised imitation learning, and allows to obtain good performance in sequence prediction problems even with very small datasets (in [1], more than 13.000.000 points were used for the train set for a 2D FO problem, still reaching a final position in the

order of meters).

It has been explained the choice of the input features for each problem, in particular in the FO problem where the choice of using the mass as input features is explained by the Markov property.

In conclusion, this work put the basis for a DAgger approach for supervised imitation learning applied to autonomous lunar landing.

5.2 Future work

In a real mission, the complete knowledge of the state is not easily available, and the Markov property is not trivial. Bloise and Orlandelli showed in [2] that a combination of CNN and RNN can be used to bypass the need of the knowledge of the state, and the model can be trained instead with sequences of image-action pairs to perform an autonomous lunar landing. A natural continuation of this work is the application of the DAgger procedure to an image-guided landing scenario. This would require adapting the DAgger algorithm to sequences of inputs.

Another possible work is the application of the DAgger approach to navigation, such as interplanetary orbit transfers. In this case the main difference with respect to the lunar landing is the dimension of the train set, which would be much bigger because the trajectories would cover hundreds of thousands/millions of kilometers instead of just a few kilometers. A DAgger approach would help to achieve better performance avoiding a training set too large which may affect the feasibility of the training.

Bibliography

- [1] C. Sanchez, D. Izzo, "*Real-time optimal control via Deep Neural Networks: study on landing problems*", Journal of Guidance Control and Dynamics, 2018
- [2] I. Bloise, M. Orlandelli, "*A deep learning approach to autonomous lunar landing*", Politecnico di Milano, 2018
- [3] S. Ross, G. Gordon, J. Bagnell, "*A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning*"
- [4] G. B. Huang, H. Zhou, X. Ding, R. Zhang, "*Extreme Learning Machine for Regression and Multiclass Classification*", IEEE, 2012
- [5] Y. Guo, M. Hawkins, B. Wie, "*Optimal feedback guidance algorithms for planetary landing and asteroid intercept*", 2011
- [6] R. Barocco, "*Applications and optimizations of Extreme Learning Machines for the realization of on-board, real-time guidance algorithms*", Politecnico di Milano, 2018
- [7] A. Géron, "*Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools and techniques to build intelligent systems.*", O'Reilly Media, 2017
- [8] I. Goodfellow, Y. Bengio, A. Courville, "*Deep Learning*", MIT Press, 2016

- [9] C. Zhang, F. Topputo, F. Bernelli-Zazzera, Y. S. Zhao, "Low-thrust minimum-fuel optimization in the circular restricted three-body problem", Journal of Guidance, Control and Dynamics, 2015
- [10] Y. Pan, C. Cheng, K. Saigol, K. Lee, X. Yan, E. A. Theodorou, B. Boots, "Imitation Learning for Agile Autonomous Driving", 2016

Consulted websites:

- [11] Exploration of the Moon, https://en.wikipedia.org/wiki/Exploration_of_the_Moon
- [12] Apollo 11, https://en.wikipedia.org/wiki/Apollo_11
- [13] Kaguya (SELENE), JAXA, http://www.selene.jaxa.jp/index_e.htm
- [14] SpaceX, Mars Colonization, <https://www.spacex.com/mars>
- [15] ESA, Future Moon Base, https://www.esa.int/spaceinimages/Images/2018/11/Future_Moon_base
- [16] The Mars Curse, <https://www.universetoday.com/13267/the-mars-curse-why-have-so-many-missions-failed/>
- [17] S. Levine, Imitation Learning, https://www.youtube.com/watch?v=kl_G95uKTHw&index=3&list=PLkFD6_40KJIwTmSbCv90VJB3Ya04sFwkX&t=0s
- [18] Recurrent Neural Network Detail Guide With Example And Applications, <https://techgrabyte.com/recurrent-neural-network-example-applications/>
- [19] The Evolution and Core Concepts of Deep Learning & Neural Networks, <https://www.analyticsvidhya.com/blog/2016/08/evolution-core-concepts-deep-learning-neural-networks/>
- [20] Understanding Confusion Matrix, <https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62>

-
- [21] Tensorflow, <https://www.tensorflow.org/>
 - [22] Tensorboard, https://www.tensorflow.org/guide/summaries_and_tensorboard
 - [23] Keras, <https://keras.io/>
 - [24] Extreme Learning Machines, http://www.ntu.edu.sg/home/egbhuang/elm_codes.html
 - [25] Stanford CS231n, <http://cs231n.stanford.edu/>
 - [26] Berkeley CS294-112, <http://rail.eecs.berkeley.edu/deeprcourse/>
 - [27] Comparison of imitation learning approaches on Super Tux Kart, 2010, <https://www.youtube.com/watch?v=V00npNnWzSU>
 - [28] Comparison of Supervised Learning and SMILe for Imitation Learning in Mario Bros, 2009, <https://www.youtube.com/watch?v=ld17TQJxE5U>