Politecnico di Milano

SCHOOL OF INDUSTRIAL AND INFORMATION ENGINEERING
Master of Science in Space Engineering

A DEEP LEARNING APPROACH TO AUTONOMOUS LUNAR LANDING

Advisor
Prof. Pierluigi Di Lizia

Co-advisors
Prof. Roberto Furfaro
Prof. Francesco Topputo

Candidates
Ilaria Bloise
Matr.858676

Marcello Orlandelli
Matr.863314

Academic Year 2017–2018

POLITECNICO
MILANO 1863

THE UNIVERSITY
OF ARIZONA

Ilaria Bloise, Marcello Orlandelli: *A deep learning approach to autonomous landing*
| Master of Science in Space Engineering, Politecnico di Milano.
| Research in collaboration with University of Arizona, Tucson (Az).

Politecnico di Milano:
www.polimi.it

School of Industrial and Information Engineering:
www.ingindinf.polimi.it

University of Arizona:
www.arizona.edu

Ringraziamenti

Questa ricerca è stata sviluppata presso l'Università dell'Arizona, a Tucson. È stata una grande esperienza e vogliamo ringraziare il Prof. Francesco Topputo e il Prof. Pierluigi Di Lizia per averci dato questa possibilità, per il supporto che ci hanno dato e per la fiducia riposta in noi. Durante la permanenza in Arizona, la presenza, l'amicizia e i consigli del Prof. Roberto Furfaro sono stati un punto di riferimento decisivo e per questo motivo vogliamo ringraziarlo moltissimo. Vorremmo ringraziare anche il Prof. Mauro Massari e Tanner Campbell per l'importante aiuto e l'interesse per la nostra ricerca. Da ultimo vogliamo ringraziare Chris Reidy for l'essenziale supporto tecnico.

I. B.
M. O.

Ringrazio la mia famiglia, presenza e sostegno certo sopratutto in questi mesi lontana da casa. E' bello sapere che c'è sempre qualcuno che mi aspetta. Grazie a Luca, perché anche nel mezzo del deserto, con te, tutto fiorisce. E grazie a tutti i miei amici per la compagnia continua anche durante i mesi a Tucson.
Milano, July 2018 I. B.

Colgo l'occasione della fine di questo percorso per ringraziare persone che non ringrazio mai abbastanza. Voglio esprimere sincera gratitudine a mio padre per il sostegno concreto che non è mai mancato, (questa laurea è un po' anche sua) e alle donne della mia vita, mia madre e Irene, perchè senza di loro non sarei quello che sono. Infine, voglio ringraziare gli amici che, sparsi nel mondo, mi accompagnano sempre.
Milano, July 2018 M. O.

Acknowledgements

This research was carried out at the University of Arizona, Tucson. It has been a great experience and we want to thank Prof. Francesco Topputo and Prof. Pierluigi Di Lizia for giving us this possibility, for the support they gave us and for the trust placed in us. During the stay in Arizona, the presence, the friendship and the advices of Prof. Roberto Furfaro have been a decisive reference point and for this reason we want to thank him very much. We would like to thank also Prof. Mauro Massari and Tanner Campbell for the important help and the interest in our research. Finally, we want to thank Chris Reidy for the essential technical support.

I. B.
M. O.

I want to thank my family, certain presence and support especially during these months far away from home. It is nice to know that there is always someone waiting for me. Thanks to Luca, beacuse even in the middle of the desert, with you, everything blooms. And thanks to all my friends for the continuous companionship also during my months in Tucson.
Milano, July 2018 I. B.

I take the opportunity of the end of this path to thank people that I never thank enough. I want to express my sincere gratitude to my father for the concrete support that he has never missed, (this degree is also his) and to the women of my life, my mother and Irene, because without them I would not be what I am. Finally, I want to thank the friends who, scattered around the world, always accompany me.
Milano, July 2018 M. O.

Non preoccupatevi, vi preoccupate troppo

Contents

List of Figures

List of Tables

Sommario

Negli ultimi anni, nel vasto ambito dell' Intelligenza Artificiale (AI), le nuove tecniche di Machine Learning stanno ricoprendo un ruolo centrale rivelandosi molto potenti e versatili. Per questo si prevede che possano diventare protagoniste per le applicazioni spaziali e sono già tema di ricerca. Grazie all'enorme disponibilità di dati, le reti neurali sono in grado di elaborare ed estrarre informazioni utili per predire da sequenze di dati o classificarne di nuovi. L'obiettivo primario di questa tesi è dimostrare come sia possibile alleggerire la navigazione spaziale, sostituendo algoritmi ingenti dal punto di vista computazionale con una rete neurale allenata opportunamente. Questo risulta essere uno strumento molto vantaggioso per eseguire operazioni in prossimità di un corpo celeste. Nel dettaglio, questo lavoro si incentra sullo sviluppo di reti neurali capaci di eseguire un atterraggio lunare basandosi su una navigazione ottica, cioè immagini del suolo della Luna. Sono state sviluppate Recurrent Neural Networks (RNN), il cui input è lo stato della traiettoria di discesa e l'output è l'azione di controllo corrispondente. Considerati gli ottimi risultati conseguiti, l'analisi è stata portata avanti nello studio di Convolutional Neural Network (CNN) i cui input sono le immagini del suolo lunare durante la fase di atterraggio e gli output sono le rispettive azioni di controllo. Sono state studiati due tipologie di atterragio: la prima puramente verticale (1D) e affrontata sfruttando le sole immagini; la seconda incentrata su un atterraggio planare (2D). In questo ultimo caso sono state usate entrambe le tipologia di reti (RNN e CNN). I risultati sono particolarmente soddisfacenti anche se sono emersi i limiti che l'approccio adottato in questo lavoro presenta. Nella parte finale sono infatti proposte nuove soluzioni da considerare per sviluppi futuri.

Abstract

Over the past few years, in the huge field of Artificial Intelligence (AI), new Machine Learning techniques are playing a central role, proving to be very powerful and versatile. For this reason, it is expected that they could become protagonist of space applications and they are already under study. Thanks to the large availability of data, neural networks are able to elaborate and extract useful information to predict from sequences of data or to classify new ones. The goal of this work is to demonstrate how it is possible to lighten space navigation, replacing computational heavy algorithms with a well trained neural network. This proves to be a very useful tool in proximity operations with celestial objects. More in detail, this research is focused on the development of neural networks able to perform a lunar landing, exploiting an optic navigation, i.e. Moon surface images. Recurrent Neural Networks (RNN) have been developed, in which the descent trajectory state is the input and the relative control action is the output. Considering the optimal achieved results, the analysis has been carried forward with the study of Convolutional Neural Networks (CNN), where the lunar surface images are the inputs and the relative control actions are the outputs. Two kinds of landings have been considered: the first one is a pure vertical (1D) landing, faced exploting only the images; the second one is a planar (2D) landing. In this last case, both neural networks have been used (RNN and CNN). The results are particularly satisfying even if the limits of the adopted approach emerged. In fact, in the final part of this thesis new solutions for future developments are proposed.

Chapter 1

Introduction

1.1 Work justification and purposes

The Moon has always been recognized as an important destination for space science and exploration. In fact, various landings on the Moon were made during the 1950s and 1960s, like the soft landings performed by the Soviet unmanned missions, known as *Luna program*, the U.S. unmanned lunar mission *Ranger* and *Surveyor* and finally the manned *Apollo* missions. Nowadays, it still represents an interesting challenge for technology and software innovation. Meanwhile, encouraged by advancements in parallel computing technologies (e.g., Graphic Processing Units, GPUs), availability of massive labelled data as well as breakthrough in understanding of deep neural networks, there has been an explosion of machine learning algorithms that can be used to solve space guidance problems. It is expected that deep learning methods will play a critical role in autonomous and intelligent systems. In this framework, a spacecraft able to land on the Moon in an autonomous way, would be a significant step further in the research of new intelligent tools for space exploration. Focusing on a Moon landing, the goal of this work is to design a set of deep neural networks, i.e. Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN) which are able to substitute navigation and guidance classical methods, by predicting the fuel-optimal control actions, using only raw Moon surface images taken by on-board cameras.

1.2 State of the art

During the final few minutes and several meters of a lunar landing, the approach phase begins. Up to now this phase has been led by a guidance system loop, shown in Fig. 1.1, whose working principles are to follow a reference trajectory

defined off-line.

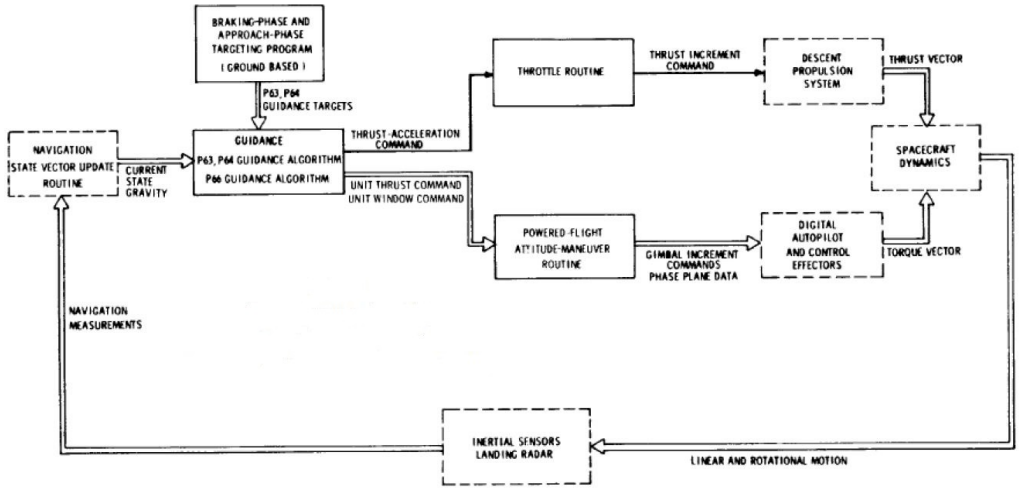


Figure 1.1: Guidance system loop of Apollo mission [1]

In particular, the guidance algorithm is fed by the guidance targets (landing position, constraints) and by the current state. It computes the errors between the reference and the present conditions, which are transformed by a PD controller in acceleration commands. They are sent to the propulsion system and to the digital autopilot. Then, thanks to throttable thrusters, the spacecraft dynamics is modified accordingly and these changes are felt by sensors. Theirs measurements are used to reconstruct and update the spacecraft state, thanks to a navigation routine. This procedure requires a high computational cost on-board for state estimators and a pre-selected optimal (or suboptimal) trajectory for landing, which cannot be determined on-board since there isn't any closed-form solution for this kind of problem. Typically those trajectories are computed numerically on Earth. Nowadays, the autonomous guidance systems algorithm are becoming more and more powerful, because it is possible to avoid state estimator and the relative high computational cost, substituting the classical procedure with Artificial Intelligence tools.

Machine learning algorithms have already been employed in many applications, such as image understanding, language processing and games, where it has been proved that neural networks can substitute human actions with extraordinary similarity [3, 4, 5]. This research wants to prove that also in space guidance problems artificial intelligence, in particular deep neural networks, can be employed, linking images handling to control actions. In the field of control problems, some machine learning tools have already been developed. In [6] robotic arms have been trained to operate several simple actions, employing many layers of CNNs able to extract features from the images taken by cameras installed on the robot.

The remarkable innovation in this paper is the direct link between raw images (processed by CNN) and control actions by using deep neural networks to map pixels into robotic arm motions. Dealing with a different field, like space navigation, the aim of this thesis is to use image features to predict the control action for a lander descent. Another significant literature outcome is the possibility to avoid the complete state estimation by exploiting neural networks in controlling a system. For example, in [7] by Sergey Levine, aerial vehicles, such as quadcopters, were able to avoid collisions with obstacles, within a certain area, without the complete state estimation, thanks to well trained neural networks that directly maps sensor reading to actions. The purpose of this thesis is to demonstrate that also in the space navigation sphere, the state estimation can be substituted by neural networks, which is one of the most relevant aspect of their applications in guidance problems. Finally, in the aerospace framework, deep neural networks have been employed in a real-time control during landing procedures. In [8], it has been unsterstood that deep neural networks can be trained to learn the optimal state feedback in a number of continuous-time deterministic, nonlinear systems of interest in the aerospace domain. The results showed that landings driven by the trained networks are similar to the simulated optimal ones. The approach followed in this thesis is different even if a similar problem is faced. In fact, the step further is the use of images to perform a landing.

Machine learning is still far from building an autonomous agent that is able to solve more or less complex real-world tasks. Imitation learning is one of the possible solution that will make that agents closer to this purpose. These techniques improve the network performances taking advantage from past mistakes. Up to now, imitation learning have been applied to control a terrestrial vehicle on urban streets, which requires human actions during the training phase. Dealing with space environment, a human presence is not available on-line; therefore in this work it has been substituted with a self-developed software tool.

The starting point has been to understand how image processing systems are designed and how they can be trained. MNIST is a canonical and historically significant image classification benchmark and there has been a considerable amount of research published on MNIST image classification ([3]). It is a database of 60000 training images and 10000 test images. Each image represents a black and white handwritten digit with 28×28 pixels. Each image is assigned a single truth label digit from $[0,\ 9]$ making this a supervised multi-class classification problem. MNIST neural network is able to recognize handwritten digits. It has been very useful to understand how to implement a CNNs in *MATLAB*, *Python-Tensorflow* and *Python-Keras*, which are the software tools used in this work. In particular,

Python and the environment *Tensorflow* (in which *Keras* is embedded) are the most commonly exploited tools for machine learning and deep learning. The work here presented applies many of these techniques together, in order to achieve the best results from the process of raw images taken by on-board cameras. For this purpose, CNNs and RNNs with Long-Short Term Memory logic are employed. In this way, it is possible to take a step forward in understanding how deep learning techniques can applied to space guidance problems.

1.3 Proposed approach

Deep neural networks can be employed to directly select actions without the need of direct filters for state estimation. Indeed, the optimal guidance is determined by processing the images only. For this purpose, Supervised Machine Learning algorithms have been implemented. In this framework, deep networks are trained with many example inputs and their desired outputs (labels), given by a supervisor. During the training phase, the goal is to model the unknown functional relationship that links the given inputs with the given outputs. Inputs and labels come from a properly generated dataset. The images associated to each state have been the inputs and the fuel-optimal control actions have been the labels. Here, two possible scenarios have been considered, i.e. 1) a vertical 1D Moon landing and 2) a planar 2D Moon landing. For both cases, fuel-optimal trajectories have been generated by software packages such as the General Pseudospectral Optimal Control Software (GPOPS) considering a set of initial conditions. With this dataset a training phase has been performed. Subsequently, in order to improve the network accuracy a Dataset Aggregation (DAgger) approach has been applied. The net performances have been verified by a testing phase on new trajectories. Moreover, to verify the robustness of the system a Monte Carlo analysis has been carried out. For both test phase and Monte Carlo analysis, trajectories have been generated by perturbing the initial conditions used for the training dataset.

1.4 Thesis structure

This work includes eight chapters. In Chapter 2, Machine Learning techniques are addressed, paying attention to the techniques and the neural networks that are used in this research. In particular, Supervised Learning, Recurrent Neural Network and Convolution Neural Network. In Chapter 3, the problem of a planar 2D Moon landing with Recurrent Neural Network is faced. The problem is formalized by describing the equations of motions and the generation of the dataset. The proposed architecture able to solve the problem is described and

details about the training and test phase are given. After the description of the DAgger approach, which has been used to improve the accuracy of the tool, some final simulations and Monte Carlo analysis are performed. The final results are finally shown. In Chapter 4, the Moon surface images simulator is described. The landing site choice is explained and an algorithm, used to improve the Digital Terrain Model resolution, is illustrated. At the end of the chapter, the simulator working principles and the images characteristics are described. Chapter 5 and Chapter 6 face the cases of a vertical landing and of a planar landing respectively, trying to solve the problems with images. The structure of these chapters replicate the one adopted for the third chapter. In Chapter 7, the conclusions of the work are provided and in the last chapter some considerations for future work are provided.

Chapter 2

Artificial intelligence

Through research of intelligent systems, it is possible to try to understand how the human brain works and model or simulate it on the computer. Artificial Intelligence (AI) involves machines that can perform tasks that are characteristic of human intelligence such as learning, reasoning and making decisions. This chapter explores the AI techniques focusing on the networks used in this work.

2.1 Machine Learning

Machine learning is a way of achieving AI and it is becoming very important to solve problems related to computer vision, robotics, finance, video gaming and space exploration because allows to construct computer programs that automatically improve with the experience and are then accurate in predicting outcomes without being explicitly programmed. Dealing with machine learning means construct algorithms that can learn from data and can make predictions on them. These programming algorithms are called Artificial Neural Networks which are a mathematical model of a biological neuron. In fact, they are able to simulate how the neurons work by transmitting information via synapses between two axons terminal. The artificial model of the biological neuron is shown in Fig. 2.1 and it is called also perceptron. The multiple input signals coming from the external environment are represented by the input set $\{x_1, x_2, ..., x_n\}$. Each input is a numerical value that represents the electrical impulses in the biological neuron. The synaptic junctions of the network are implemented on the artificial neuron as a set of synaptic weights $\{w_1, w_2, ..., w_n\}$ that express the importance of the inputs to the outputs (according to the functionality of neurons). The relevance of the input is computed by multiplying each input by its corresponding synaptic weight ([9], [10], [2]). The *linear aggregator* ($\sum$) gathers all the weighted inputs and thanks to biases the information go straight on to the output. Biases b_j are

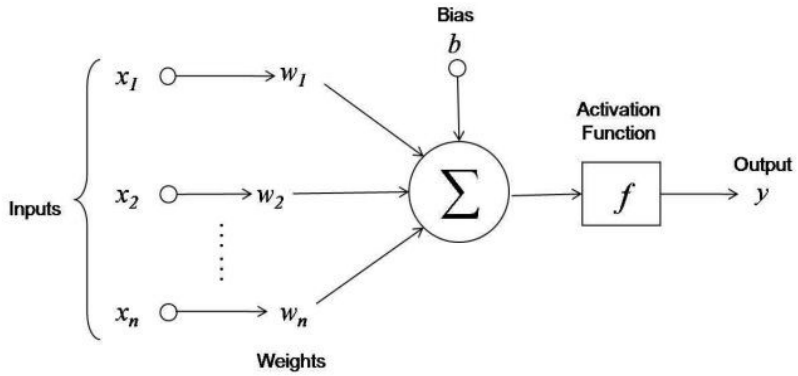


Figure 2.1: Neuron model [2]

variables used to specify a proper threshold that the result, produced by the linear aggregator, should have to generate a requested value. This is essential for the information to go toward the neuron output. The *activation function* f limits the neuron output within a reasonable range of values. Each output is practically the result of the following expression $\sum_j w_j x_j + b_j$. This is repeated for each of the neuron crossed by the inputs. Therefore the output is equal to:

$$output = \sum_j (w_j x_j) + b_j \tag{2.1}$$

A complex system of neurons could lead to solve important problems and a neural network looks like Fig. 2.2. A set of neurons is called *layer* and therefore the

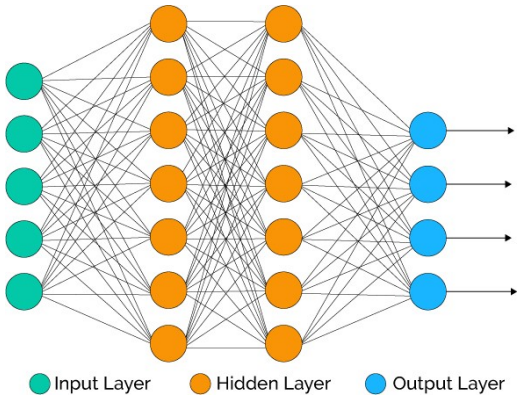


Figure 2.2: Neural Network model

artificial neural network can be divided in three parts :

- *Input layer* which is responsible for receiving information from the external.
- *Hidden layers* which are the net core because perform most of the internal processing and extract the information from the inputs.

– *Output layer* which produces and presents the final network outputs.

If the number of hidden layers is greater than one, the network is a *deep neural network* and the learning procedure is named *deep learning*. Neural Network receives a vector as input and transforms it through a series of hidden layers, each connected to all neurons in the previous layer. Designed networks must be *trained* and then, according to the several fields where the net is used, there are many types of Machine Learning. Before entering the classes of Machine learning it is necessary to clarify what *training a network* means. According to the mathematical model of the neuron, the training process of a neural network consists of applying steps for tuning the synaptic weights and biases of its neurons, in order to generalize the solutions produced by its outputs. This procedure requires the presentation of samples (training set) which express the system behavior and on which the network performs its training. The set of training steps is called learning algorithm. By executing the learning algorithm, the network will be able to extract features about the system inputs. Moreover, each complete presentation of all the samples belonging to the training set is called *epoch*. Depending on the type of learning algorithm the classes of machine learning are:

- **Supervised learning**: The machine is trained with many example inputs and their desired outputs (labels), given by a supervisor. The goal is to find a rule which links the given input with the given output. A typical supervised learning task is classification.

- **Unsupervised learning**: No labels are given to the learning algorithm, therefore the system tries to learn without a teacher. In this case, the machine tries to find structures in the inputs.

- **Reinforcement learning**: The learning system called agent, can observe the environment, select and perform actions. From its action, it gets rewards or penalties in return and improves and learn the best strategy which is called policy. Basically, the machine interacts with the dynamic environment.

For this work Supervised Machine learning has been used.

2.2 Supervised Machine Learning

The idea behind supervised machine learning is simply learning from the examples. The supervised learning strategy consists of having the desired outputs available for a given set of input signals; in other words, each training sample is composed of the input signals and their corresponding outputs (*labels*). The goal

of the network is to develop a rule, a procedure that classifies the given data. This is achieved because the learning algorithm supervises the discrepancy between the produced outputs (label) with respect to the desired ones. The network is considered trained when this discrepancy is within an acceptable value range. The discrepancy is minimized by adjusting continually the synaptic weights and biases of the network according to the definition of training. This discrepancy is defined by a *loss function*; a detailed explanation is given in the following chapters. The agents (the network) can learn using two types of sets of data, a *training set* and a *test set*. The training set is composed of the labeled examples and the test set with unlabeled example. The test set is available to verify the performances of the trained network. There are many types of network and each one is peculiar in the way the data are processed. For the purpose of this work the main ones are presented: Recurrent Neural Network and Convolutional Neural Network.

2.3 Recurrent Neural Network

Recurrent Neural Networks (RNNs) are powerful tools that deal with data sequences. The advantage of an RNN lies in its memory. In fact, it stores information that have been computed few steps so far and uses them to improve the prediction and its accuracy. Fig. 2.3 shows an RNN unrolled into a full network to better understand how this net works. The working principle is as follows:

- x_i is the input at the time step t_i
- h_i is the hidden state at time step t_i. It represents the memory of the network which is computed based on the previous hidden state h_{i-1} and the input at the current step x_i.
- y_i is the output at the time step t_i

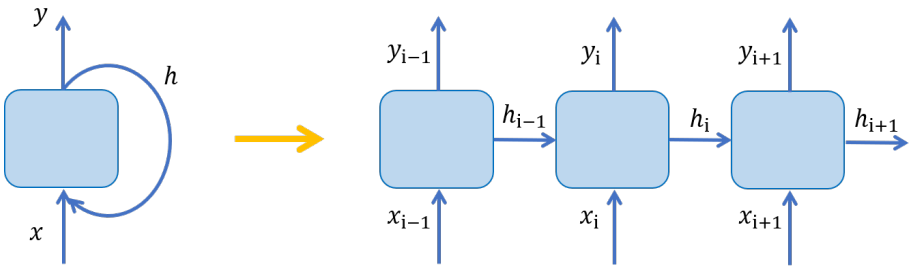


Figure 2.3: Recurrent neural network

RNNs are able to use information coming from a long sequence, but experience has shown that they are limited to looking back only a few steps. Considering

what said before, the most commonly used RNNs are the LSTM (Long-Short-Term-Memory). The difference is just the way in which it computes the hidden state. The memories in LSTMs are called *cells*. Internally these cells learn what to store in a long-term-state, what to erase from memory, and what to read from it. A typical cell is shown in Fig. 2.4. Referring to Fig. 2.4 the hidden state

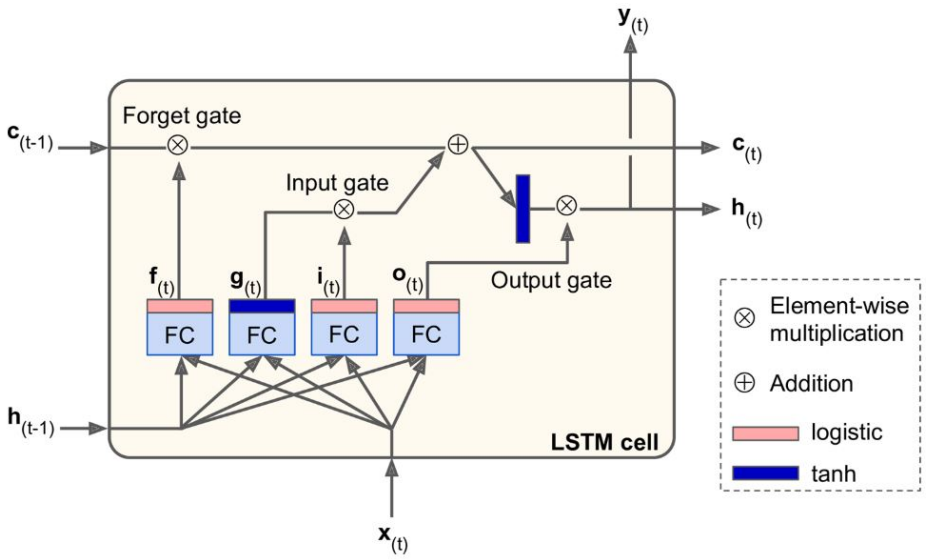


Figure 2.4: LSTM cell

is divided in two vector: the long-term state $\mathbf{c}_{(t)}$ and the short-term state $\mathbf{h}_{(t)}$. The incoming long-term state $\mathbf{c}_{(t-1)}$ goes through a *forget gate*, dropping some memories and then it adds some new memories with an adding operation (the memories to store are selected by the *input gate*). The result is $\mathbf{c}_{(t)}$ which is sent straight on without any other transformation [2]. The long-term state $\mathbf{c}_{(t)}$ is copied and filtered by the *output gate* (which applies a *tanh* function on the input) and this operation produces the short-term state $\mathbf{h}_{(t)}$ (note that it is equal to the cell's output $\mathbf{y}_{(t)}$). Let's look how the current input vector $\mathbf{x}_{(t)}$ is transformed. It is fed with the previous short-term state $\mathbf{h}_{(t-1)}$ to different fully connected layers that have different purposes:

- The main layer is the one with $\mathbf{g}_{(t)}$ as output. It analyzes the current inputs $\mathbf{x}_{(t)}$ and the previous $\mathbf{h}_{(t-1)}$ with a *tanh* function. The output is partially stored in the long-term state.
- The other three layers are called *gate controllers* and use logistic activation function which means that the outputs are in a range from 0 to 1 and, in particular, 0 closes the gate and 1 opens it. In particular:
  - The *forget gate* (controlled by $\mathbf{f}_{(t)}$) controls which parts of long-term state should be erased.

- The *input gate* (controlled by $\mathbf{i}_{(t)}$) controls which parts of $\mathbf{g}_{(t)}$ should be added to the long-term state.
- The *output gate* (controlled by $\mathbf{o}_{(t)}$) controls which parts of the long-term state should be read and sent to the output terms ($\mathbf{h}_{(t)}$ and $\mathbf{y}_{(t)}$)

In conclusion an LSTM cell can learn to recognize an important input, learn to store and preserve it for as long as it is needed and learn to extract information whenever is needed.

2.4 Convolutional Neural Network

Neural networks and deep learning provide the possibility to find solutions for many problems related to image recognition. In particular, the Convolutional Neural Network (CNN) is used to detect what an image is or what it contains by transforming the original image, through layers, to a class scores. The architecture of a CNN is designed to take advantage of the 2D or 3D $[width, height, depth]$ structure of an input image which is processed as pixels values. The basic CNN structure uses many types of layers which are:

- Convolutional Layer
- Pooling Layer
- Fully Connected Layer
- Output Layer (last Fully Connected Layer)

Convolutional Layer: the objective of a Convolutional layer is to extract features from the input volume by applying filters on the image. It is the most demanding layer in terms of computations of a Convolutional Neural Net and the layer′s parameters consist of a set of learnable filters. Each filter is practically a matrix spatially smaller than the image which is scanned along width and height (2D case). In Fig. 2.5 filter convolution is shown. Referring to the Neural Network definition, filters (or kernels) are the weights of this layer. In fact, as the filter is sliding on the input image, it multiplies its values with the original pixel values of the image and these multiplications are all summed up giving only one number as output. Repeating this procedure for all the regions on which filter is applied the input volume is reduced and transformed and then passed to the *Max-pooling layers*. Before going deeper in the analysis of the Max-pooling layer, it is important to specify how filters

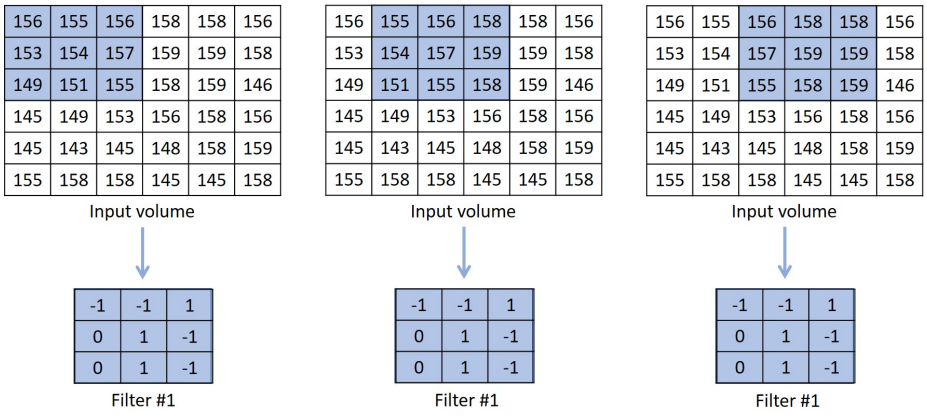


Figure 2.5: Example of filter application

slide. The involved parameter is the*Stride*. In the Fig. 2.5 it is possible to note that the filter moves one pixel at a time. The stride in this case is, in fact, equal to 1. This parameter determines how the input image volume reduces and normally, it is set such that the output volume is an integer and not a fraction. Moreover, in order to control the spatial size of the output volumes, an additional parameter is used: the *Padding* parameter, which pads with zeros the border of the input image. If the padding is not used the information at the borders will be lost after each Conv. layer and this will reduce both the size of the volumes and the performance of the layer. An example of what the Padding does is shown in Fig. 2.6.

0	0	0	0	0	0
0	156	158	158	156	0
0	153	156	158	156	0
0	145	148	158	159	0
0	158	145	145	158	0
0	0	0	0	0	0

Figure 2.6: Example of padding

Resuming the *hyperparameters* to be set for the Convolutional layer are:

- *Number of filters.*
- *Stride* with which the filter slides.
- *Padding* with which the input is padded with zeros around the border.

Fig. 2.7 shows the application of the filter. Basically this operation performs dot products between the filters and local regions of the input.

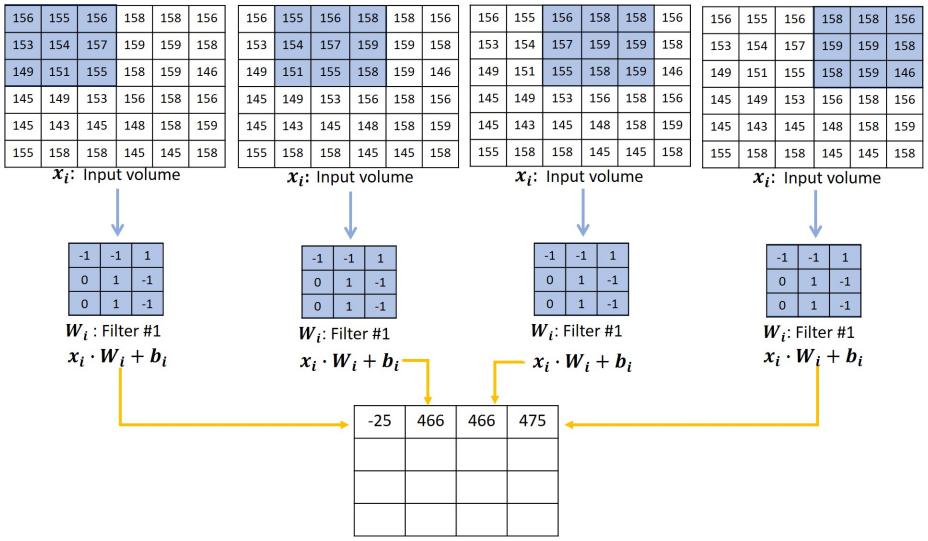


Figure 2.7: Example of filters application on image pixels

An example of image transformed by the application of filters in a convolutional layer is shown in Fig. 2.8.

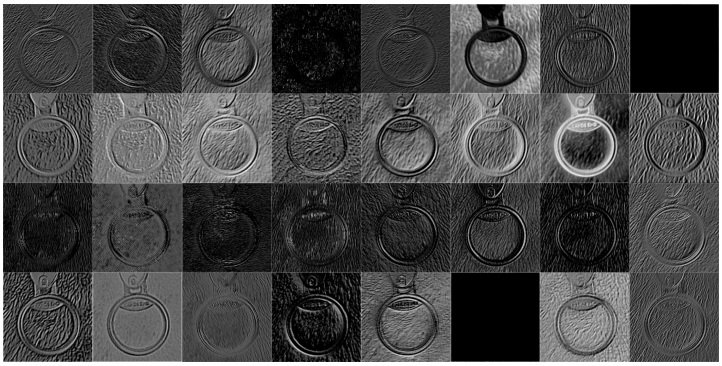


Figure 2.8: Example of filters application result on a image

The spatial size of the output volume can be computed as function of input size as:

$$\frac{(I - F + 2P)}{S} + 1 \tag{2.2}$$

where: I is the input volume, F is the filter size of the Convolutional Layer, S is the stride with which the filter is applied and P is the padding used on the border (if any).

Pooling Layer: it reduces progressively the spatial size of the input volume in order to control overfitting. The Pooling Layer operates independently on every depth slice of the input and there are different functions to be used, the common one is the Max-pooling. It uses the MAX operation taking only the most important part. Also for this layer two hyperparameters have to

be chosen: the filter window (F) and the stride (S). In Fig. 2.9 an example of max-pooling operation is shown with 2×2 filter window and stride equal to 2.

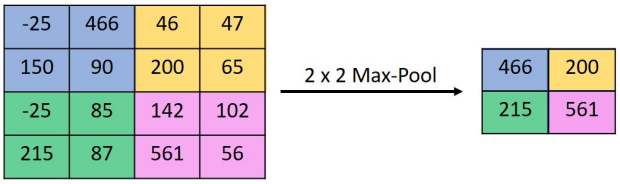


Figure 2.9: Example of Max-pooling layer

Fully Connected Layer: the purpose of the Fully Connected layer is to use the features arriving from the convolutional layers for classifying the input image into various classes (Fig. 2.10). The last fully-connected layer uses a softmax activation function for classification purpose.

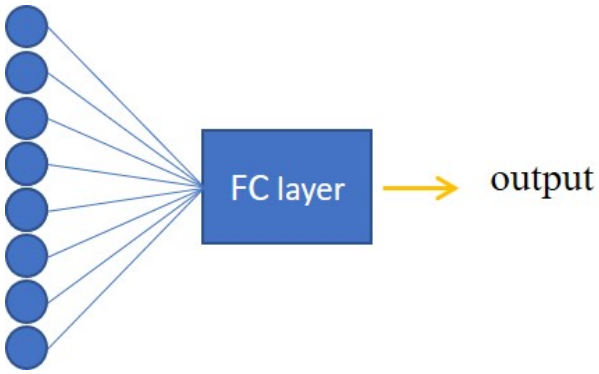


Figure 2.10: Fully connected layer

Chapter 3

From state to control: planar Moon landing

The first problem faced in this work is a planar Moon landing in which an RNN is exploited to predict the control action directly from the knowledge of the state. The network has been trained according to a supervised learning algorithm. It is possible to see how the recurrent networks are able to achieve very good results in this scenario, in which the input is a series of data (the states) and each input belongs to a particular sequence (the trajectory). As already mentiones in the previous chapters, RNNs can take advantage of their memory to keep track of what has entered the network before and use those information to better predict the output. At this stage Moon surface images are not considered. After the problem formalization, in which the equations of motion are described, the procedure to generate the dataset is illustrated. Afterwards, the proposed network architecture is shown and the training and test phases are illustrated. The accuracy is improved according to a DAgger approach. Finally the results of a Monte Carlo simulations are provided.

3.1 Problem formalization

A planar 2D Moon landing has been considered, in which the variables of state are downrange and altitude. The state is composed of five components: the variables of state and mass. The mass of the spacecraft is coupled with the others through the thrust. The control action will have two components as well, one aligned with the vertical direction and one with the horizontal. As it will be explained later, instead of two Cartesian components, the magnitude and the two unit vector components have been used to describe the thrust vector. The

equations of motion are the following:

$$\begin{cases} \dot{\mathbf{x}} = \mathbf{v} \\ \dot{\mathbf{v}} = \mathbf{g} + \dfrac{\mathbf{T}}{m} \\ \dot{m} = -\dfrac{\mathbf{T}}{I_{sp}\, g_0} \end{cases} \tag{3.1}$$

If h is the altitude and d is the downrange, $\mathbf{x} = [d, h]$ and $\mathbf{v} = [v_d, v_h]$. The thrust $\mathbf{T}$ is $\mathbf{T} = [T_d, T_h]$ and $\mathbf{g}$ is the lunar gravity acceleration. I_{sp} is the specific impulse and g_0 is the reference gravity acceleration. Fig. 3.1 shows a simple scheme of the involved variables. These equations have been integrated to minimize the

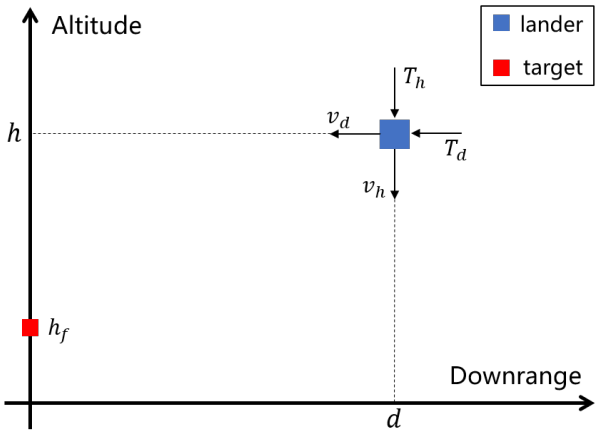


Figure 3.1: Involved variables in the selected reference frame

following cost function:

$$\int_0^{t_f} \|\mathbf{T}\|\, d\tau$$

where $\mathbf{u}$ is the control action and t_f is the final time, which is not fixed. As it can be seen, the optimality condition is achieved when the control actions are minimized, i.e. when the depleted mass is minimum. From now on this optimality condition will be called fuel-optimal condition. As the theory says, the solution of such problem has a bang-bang profile, in which the thrust is alternatively maximum or minimum, or on/off. For this reason, when the neural networks is designed, it is possible to treat the problem like a classification one, in which only two choices are possible: thrust at maximum or thrust at minimum. A different strategy has been considered for what concerns the thrust components, as explained later. Initial conditions are given as initial position $\mathbf{r}_0$ (downrange and altitude) and initial velocity $\mathbf{v}_0$ (horizontal and vertical). The initial mass is set equal to m_0. Final conditions on each variable of state, except the mass, have been imposed. Since the final time is not fixed, only a lower limit for the mass value has been considered and set equal to the dry mass m_{dry}.

3.2 Dataset generation

Dealing with supervised learning, a dataset is needed to train the network. In order to generate a suitable dataset, the problem just formalized had to be solved many times, considering a set of different initial conditions. For this purpose, an optimizer was necessary. The General Pseudospectral Optimal Control Software (GPOPS) has been chosen. GPOPS is a *MATLAB* software package intended to solve general non-linear optimal control problems, where systems are described bydifferential or algebraic equations. GPOPS has been chosen for its simplicity in problem implementation and for the ease in visualizing and handling states and control actions, which have been used to create the datasets. In the followings, the most important steps for the implementation are illustrated. To use GPOPS the user has to define the system of equations of motion and the cost function. This equations are written in two separate scripts, while in the main code the user defines initial conditions, final conditions and the bound constraints. This bounds are defined in terms of maximum and minimum values that each state and each control action can assume during the integration and optimization. Their value should be properly selected to avoid being too tight. General constraints can also be imposed by defining them together with the equations of motion. GPOPS needs a first guess solution to start the optimization. Since the problem has been solved many times, the free-fall solution has been selected to be the initial guess solution for only the first set of initial conditions; subsequently each GPOPS solution has been the initial guess for the next set of initial conditions. This helped the software to converge faster and more accurately.

As said before, many initial conditions have been considered. The initial mass of the spacecraft has been set equal to 1300 kg. The downrange has been initialized between 1500 and 2000 meters, while the altitude between 1000 and 1500 meters. Fig. 3.2 shows a scheme of the initial positions in the selected reference frame. The initial downrange velocity v_{d_0} changes according to the downrange: when the downrange is maximum, the velocity is maximum in modulus (-15 m/s), vice versa when downrange is minimum, the velocity also is minimum in modulus (-11 m/s). The same reasoning has been applied for the initial vertical velocity v_{h_0}, that goes from -6 to -10 m/s. Unlike the initial conditions, the final ones have been kept constant for all trajectories. In particular, the final downrange d_f has been set equal to 0, the final altitude h_f equal to 50 m and finally, both components of the final velocity (vertical and horizontal) equal to -0.5 m/s. As it can be seen, the final condition is such that the spacecraft has not touched the ground and has a very small downward velocity. Throttlable thrusters have been considered, in which the magnitude of the control action is bounded following the

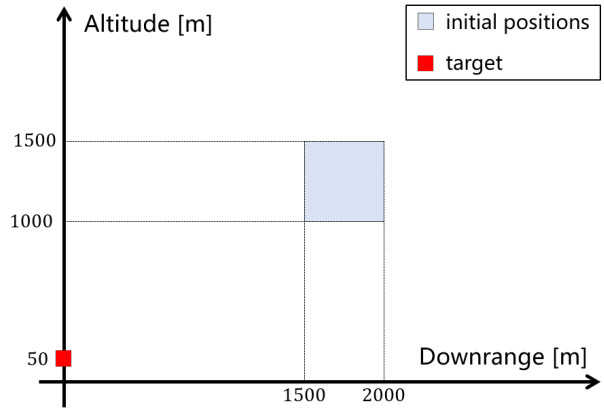


Figure 3.2: Initial positions in the selected reference frame

same strategy applied in [11] and in [12]. The nominal thrust T_{nom} is equal to 4000 N. The maximum thrust and minimum thrust have been fixed equal to 85% and 25% of the nominal thrust respectively (i.e., 3400 N and 1000 N respectively). According to the above assumption, in the classification problem there will be only two alternatives: minimum thrust, or maximum thrust. It is clear that these values refer only to the magnitude of the control action. To understand how the thrusters direction has been taken into account, it is useful to see how the equations of motion have been implemented componentwise in GPOPS:

$$\begin{cases} \dot{d} = v_d \\ \dot{h} = v_h \\ \dot{v_d} = \dfrac{T}{m} \cdot u_d \\ \dot{v_h} = -g + \dfrac{T}{m} \cdot u_h \\ \dot{m} = -\dfrac{T}{I_{sp}\, g_0} \end{cases} \tag{3.2}$$

where T is the magnitude of the control action and u_d and u_h are the components of the thrust direction unit vector. It has been imposed that $u_d^2 + u_h^2 = 1$ at any time. It has been useful to extract from these two components the angle that the thruster forms with the horizontal direction, according to Eq. 3.3.

$$\vartheta = \arctan \frac{u_d}{u_h} \tag{3.3}$$

It is possible to understand now that the neural network will have to solve a classification problem for what concerns the magnitude of the thrust and a regression problem for the direction of the control action. In this way, 2601 trajectories have been generated within the selected two-dimensional portion of space. The parameters of the problem are shown in Tab. 3.1a. A wrap-up of the

initial and the final conditions is shown in Tab. 3.1b and in Tab. 3.1c.

Once the working principles of the optimizer have been described, it is important

Table 3.1: Parameters and state values for the 2D problem

(a) Problems parameters

	value	unit
m_{dry}	500	kg
g	-1.622	m/s^2
Isp	200	s
T_{nom}	4.0	kN
T_{max}	3.4	kN
T_{min}	1.0	kN

(b) 2D initial conditions

	value	unit
d_0	$[1.5,\ 2.0]$	km
h_0	$[1.0,\ 1.5]$	km
v_{d_0}	$[-11,\ -15]$	m/s
v_{h_0}	$[-6, -10]$	m/s
m_0	1.3	ton

(c) 2D final conditions

	value	unit
d_f	0	m
h_f	50	m
v_{d_f}	-0.5	m/s
v_{h_f}	-0.5	m/s

to understand how the dataset has been generated. Each trajectory coming from GPOPS has 61 points, so 61 states and 61 control actions. Each state has five elements (downrange, altitude, velocities and mass); each control action has three components (magnitude, u_d and u_h). As said before, the angle ϑ has been extracted for each couple of unit vector components, so 61 angles have been computed per each trajectory. In this way, instead of having three elements in each control action, only two elements have been considered: the magnitude and the angle. The final dataset has been built by associating each state to each control action. This dataset has been divided in training-set and test-set, the first set comprehends 2409 trajectories, the second one 192.

In Fig. 3.3 it is possible to see an example of a trajectory in the dataset. The initial downrange is equal to 1750 meters and the initial altitude is 1250 meters. In Fig. 3.4 it is possible to see the bang-bang profile of the thrust magnitude. Fig. 3.5 shows the behaviour of the angle ϑ, that the thrust forms with the horizontal direction. Finally, in Fig. 3.6 all the trajectories are plotted, with the intention of showing that they are all in the region of space above a cone of 20°.

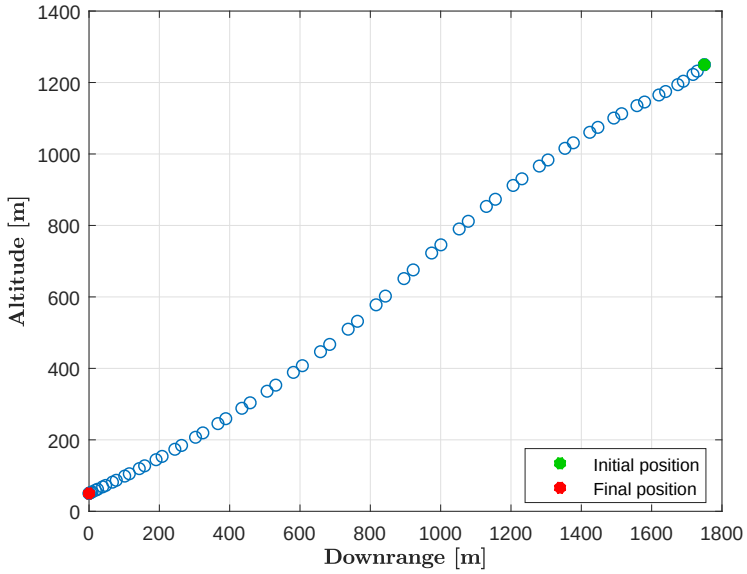


Figure 3.3: 2D trajectory

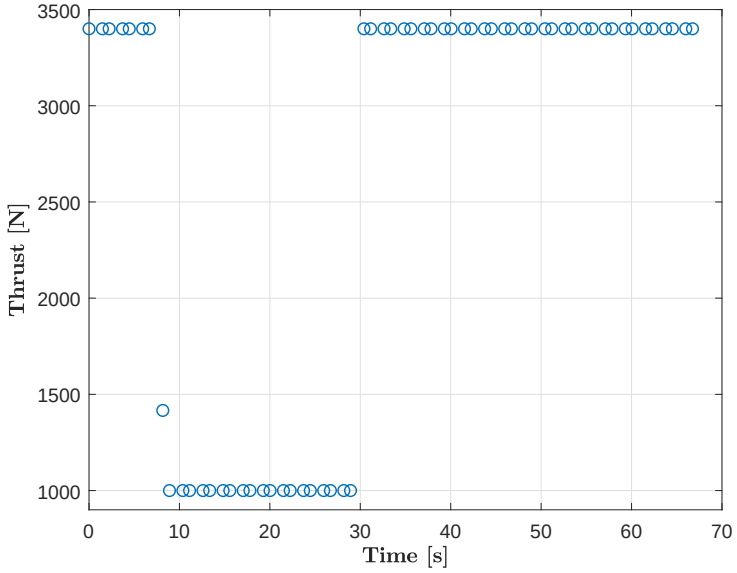


Figure 3.4: 2D thrust magnitude

3.3 Proposed network architecture

The aim of the proposed net is, as said, to exploit the states of the dynamic system (spacecraft) and have a prediction on the control action to perform an optimal planar landing on the Moon. Dealing with a 2D problem, the label associated to each state is composed by two components: one associated to the magnitude of the thrust and the second one to the angle and so to the direction in which the thrust is applied. Since the problem has a bang-bang control action

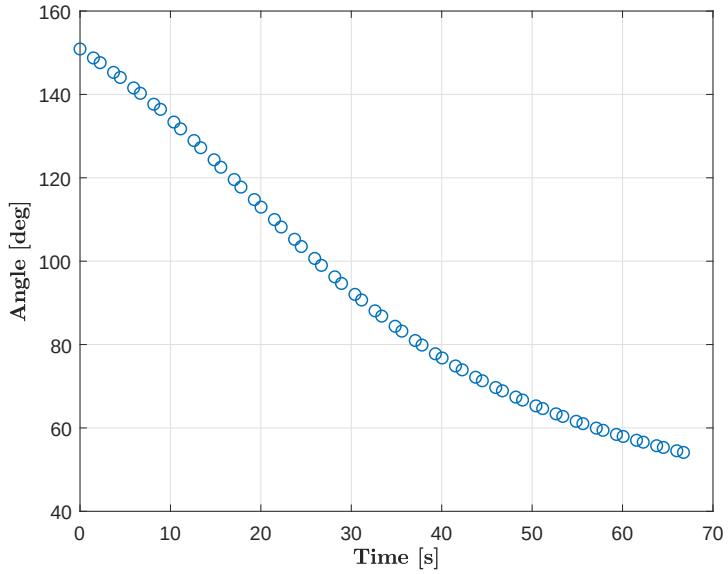


Figure 3.5: 2D thrust angle

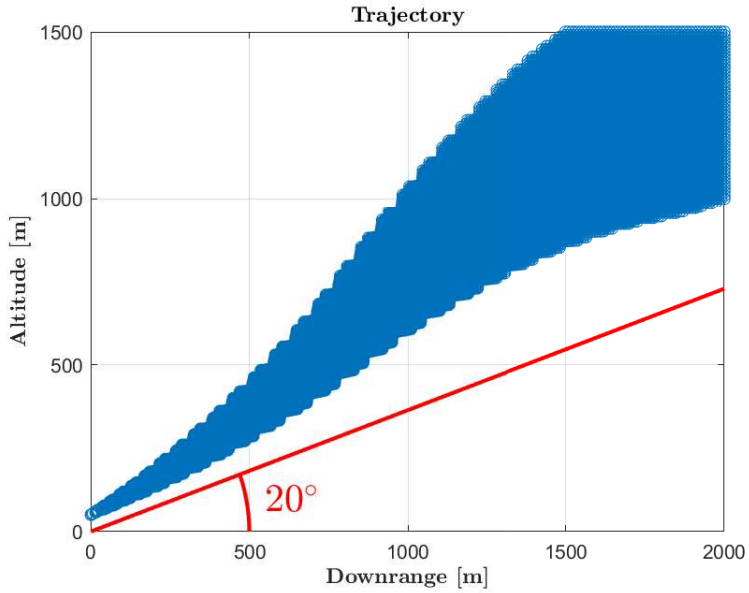


Figure 3.6: All 2D trajectories

profile, the thrust can take its maximum or minimum value only. According to this, the network should deal with a classification problem. Moreover, as shown in Section 3.2., the thrust angle profile is smooth and continuous and therefore, the network should perform a regression on the thrust angle value. Based on all these considerations the proposed architecture is shown in Fig. 3.7. The input enters the LSTM block and the output is sent to two branches, one whose intent is the classification and the other the regression. Some fully connected layers

link LSTM to the final output levels. The network has been implemented in

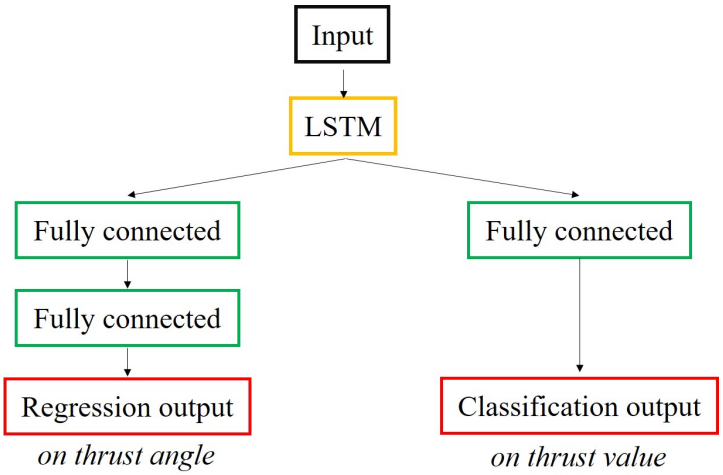


Figure 3.7: Proposed LSTM architecture

Python-Keras and the architecture is detailed in Fig. 3.8. It is useful to analyze

Layer (type)	Output Shape	Param #	Connected to
main_input (InputLayer)	(None, 3, 4)	0	
lstm_1 (LSTM)	(None, 100)	42000	main_input[0][0]
dense_2 (Dense)	(None, 50)	5050	lstm_1[0][0]
dense_1 (Dense)	(None, 2)	202	lstm_1[0][0]
dense_3 (Dense)	(None, 1)	51	dense_2[0][0]
clas_output (Activation)	(None, 2)	0	dense_1[0][0]
regr_output (Activation)	(None, 1)	0	dense_3[0][0]

Total params: 47,303
Trainable params: 47,303
Non-trainable params: 0

Figure 3.8: LSTM architecture in *Python-Keras*

how each component of the LSTM-net by going through the elements reported in Fig 3.8.

- LSTM cell is composed of 100 neurons (line `lstm_1` in Fig. 3.8) and is directly connected to the input layer.
- The branch of the neural network designed to perform the classification on the thrust value (line `dense_1` and `clas_output` in Fig. 3.8) is composed by a dense layer (fully connected layer) and an output layer, both of 2 neurons because the problem has two output classes. The fully connected layer is connected with the LSTM cell.
- For the regression on the thrust angle (line `dense_2`, `dense_3` and `regr_output` in Fig. 3.8) there are two fully connected layers (with 50 and 1 neurons)

and the output layer. Also here the first fully connected layer is directly linked to the LSTM cell.

This selected architecture and in particular the number of neurons lead to a number of 47.303 trainable parameters (resulting from the weights and biases). The next section is focused on the presentation of the inputs and training phase for this RNN−LSTM neural network.

3.4 Training and test phase

3.4.1 Training

Much effort has been put into the network input shape. How could the LSTM learn better from the input information? After a proper tuning, the chosen number of inputs is 3 and this value has been fixed for all the nets developed in this work in both the 1D and the 2D case. After several simulations, it has been understood that three sequential states gave the best results. This means that each trajectory (which is composed by 61 states) has been divided in sets of 3 sequential states and this division has been developed by following two strategies.

The first idea has been to fed the net with sequential states and without separating different trajectories as shown in Fig. 3.9. Note that in this case after one trajectory ends the new one starts. However, another strategy has been

d	h	v_d	v_h
5.5881	52.6972	-4.37533	-1.87406
0.884849	50.5477	-1.8053	-0.950612
0	50	-0.5	-0.5

(a) 1^{st} input

0.884849	50.5477	-1.8053	-0.950612
0	50	-0.5	-0.5
1500	1000	-11	-6

(b) 2^{nd} input

0	50	-0.5	-0.5
1500	1000	-11	-6
1482.47	991.246	-14.2392	-6.60116

(c) 3^{rd} input

Figure 3.9: Example of inputs shape for the first strategy

adopted as it outperformed the above one. Here the idea has been to fed the

net by separating the trajectories one with respect to the other. An example is reported in Fig. 3.10. This second approach avoids confusion between trajectories

d	h	v_d	v_h
9.40714	54.2987	-5.64012	-2.34685
5.5881	52.6972	-4.37533	-1.87406
0.884849	50.5477	-1.8053	-0.950612

(a) 1^{st} input

5.5881	52.6972	-4.37533	-1.87406
0.884849	50.5477	-1.8053	-0.950612
0	50	-0.5	-0.5

(b) 2^{nd} input

1500	1000	-11	-6
1482.47	991.246	-14.2392	-6.60116
1472.01	986.556	-15.8366	-6.8527

(c) 3^{rd} input

Figure 3.10: Example of inputs shape in second strategy

and in fact, the results of the net training are in this case better because the RNN−LSTM is able to learn what is the final state of the optimal trajectory (i.e. $[0, 50, -0.5, -0.5]$).

Since a supervised machine learning has been used in this work, it is important to explain how the labels have been utilized. In each input of 3 states (the 3×4 matrices reported in Fig. 3.9 and Fig. 3.10) the associated label corresponds to the 3^{rd} row. During the training phase, at time t, the net is fed with an input composed by the state at time t, at time $t-1$ and $t-2$ and the control action at time t. Based on the above considerations, the states per trajectory are 61 but, considering how they are divided, the first 2 states enter the network only once. Thus the input packages are 59.

Now, taking a step further, machine learning theory states that input data can be given to the network sequentially, either individually or by small groups called batches. The *batch size* is, in fact, the number of samples (inputs) that are propagated through the network at each training epoch. It has been shown in the literature that a multiple input (so a reasonable value of batch size) has some advantages in the training phase even if there are no golden rules to choose it [2]. In this work, the chosen approach is to have a batch of 59 inputs; in this way a complete trajectory is taken. In order to go deeper in the analysis, a

distinction needs to be made between the classification and regression part. The main difference between them is that the classification task is predicting a discrete class label; whereas regression predicts a continuous quantity. For this reason, the network classification output (thrust magnitude), as well as the input labels, have the format of a binary vector of length 2. For example, when the thrusters are at their maximum value, the correct label is [0; 1] and when they are at their minimum value, the label is [1; 0].

As reported in Section 2.2, the comparison of the predicted net outputs and the correct outputs is carried out through the computation of a *loss function* and the purpose of the training phase is to minimize it. In fact, weights and biases are initialized as random values and as the training phase proceeds, they are updated according to a minimization algorithm. The standard learning algorithm for neural networks is backpropagation with gradient descent (or its variant stochastic gradient descent). In this thesis, the used optimization algorithm (to minimize the loss function) is an *Adam optimizer*. The name is derived from ADAptive Moment estimation. It is well suited for problems that are large in terms of data and parameters and it can be used instead of the classical stochastic gradient descent procedure [13], which maintains a single learning rate (termed alpha) for all weight updates and the learning rate does not change during training. Instead Adam optimizer adapts the learning rate as training unfolds thanks to the *decay* parameter. Note that the learning rate is a hyper-parameter that controls how much the algorithm is adjusting the weights of the network with respect the loss gradient. The lower the value, the slower the travel along the downward slope. While this might be a good idea (using a low learning rate) in terms of making sure that any local minima is missed, it could also mean that the time for convergence will be longer. Furthermore, the decay has an expression given by Eq. 3.4

$$lr_{new} = lr_{old} \cdot decay_{rate}^{\left(\frac{epoch}{decay_{step}}\right)} \tag{3.4}$$

Since the study here presented has been developed in a *Python-Keras* and *Pyhton-Tensorflow* environment, where Adam method is already implemented and can be directly called and used for the training phase, there was no need of a self-made implementation of the algorithm and for this reason the pseudo code is not presented. Moreover, *Python-Keras* offers an optional list of scalar coefficients called *loss_weights* to weight the loss contributions of different model outputs. The loss value that will be minimized by the model will then be the weighted sum of all individual losses, weighted by the *loss_weights* coefficients. In Tab. 3.2 the chosen hyper-parameters are resumed.

Classification predictions can be evaluated using accuracy, whereas regression

Table 3.2: Selected hyper-parameters of the RNN−LSTM

Hyper-parameters	
Batch size	59
Initial learning rate	0.0001
Decay rate	0.0001
Regression loss weight	10
Classification loss weight	60

predictions can be evaluated using root mean squared error (RMSE). Concerning the accuracy evaluation, the loss function associated to this task has been the *Cross-entropy loss*.

***Cross-entropy loss*:** it measures the performance of a classification model whose output is a probability value between 0 and 1. In particular it indicates the distance between the model prediction and the true value of the output. The cross entropy error is computed as:

$$y_{cross-entropy} = -\sum_{j} t_j \log(y_{softmax_j}) \tag{3.5}$$

***Softmax function*:** Softmax function outputs a probability distribution suitable for probabilistic interpretation in classification tasks. This function takes a vector of real numbers and transforms it into a vector of real numbers in range [0; 1] which represents the probabilities and will be used for determining the class for the given inputs. The softmax function computes the ratio between the exponential of the input value and the sum of the exponentials of all the input values. The analytic formula is:

$$y_{softmax} = \frac{e^{z_k}}{\sum_{j=1}^{J} e^{z_j}} \tag{3.6}$$

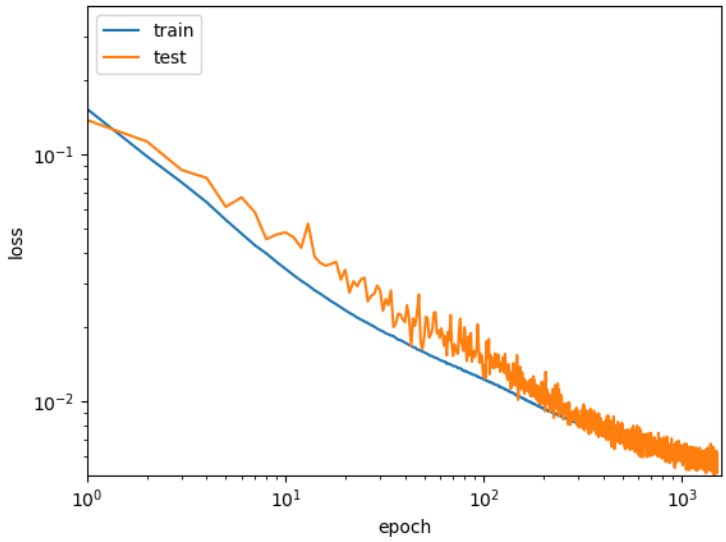
The RMSE (Eq. 3.7) for regression, on the other hand, represents the sample standard deviation of the differences between predicted values and real values:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{I}(\bar{y_i} - y_i)^2}{n}} \tag{3.7}$$

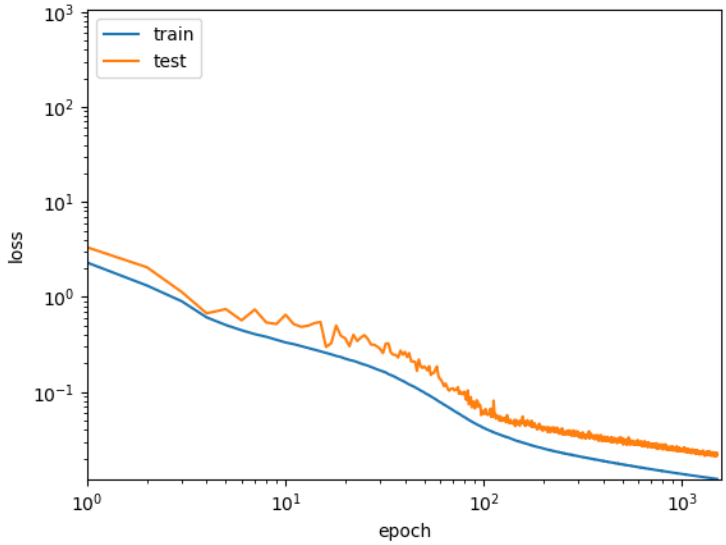
where y_i is the real value and $\bar{y_i}$ the predicted one.

The results are now illustrated. The training has been run for 1500 epochs over 2409 trajectories. In order to have a feedback during the training phase on the net performances the training set has been split such that 5% of the dataset

has been used for validation. Validation datasets can be used for regularization by early stopping: the training is stopped when the error on the validation dataset increases, as this is a sign of overfitting to the training dataset. In Fig. 3.11 it



(a) Classification loss



(b) Regression loss

Figure 3.11: Classification and regression losses evolution during the RNN−LSTM training phase

is possible to note how the loss function has been minimized along all epochs. The orange curve corresponds to the model validation which proves that no over fitting of data is present and the RNN−LSTM is well trained.

3.4.2 Test

The test phase has been applied to validate the final model after training by estimating model properties. As already mentioned, the test set consists of 192 trajectories. To visualize the trained net performances on the classification (accuracy) a *confusion matrix* has been computed. The number of correct and incorrect predictions has been summarized by counting and breaking them by each class: 0 is associated to minimum thrust, 1 to maximum. As shown in Fig. 3.12

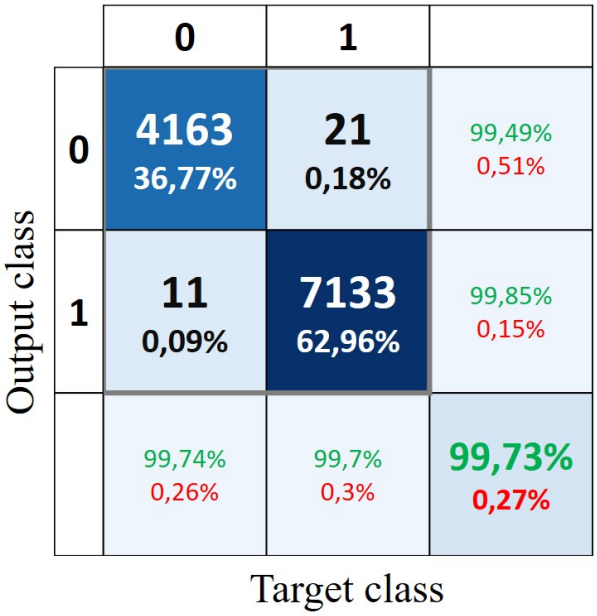


Figure 3.12: Confusion matrix RNN-LSTM

the RNN-LSTM trained model has an accuracy of $99,73\%$. To evaluate if the classification accuracy is consistently high, MNIST images dataset ([3]) has been taken as a reference, even if no images are involved in an RNN−LSTM. According to the state of art ([16]), tests on MNIST (which is a trivial dataset) can at most reach a precision of 99.79%. To obtain a good result in a more complex problem, a threshold of 99% on the accuracy is chosen as minimum target. On the other hand to evaluate the performances on regression the plot on the predicted regression angles has been considered. The result is shown in Fig. 3.13. The results can be considered to be very good, as the final RMSE is $0.2°$.

3.5 Accuracy improvement: DAgger approach

An approach has been investigated to obtain further improvements of the net performances. Several algorithms have been proposed and are available in literature. The most promising is the Imitation Learning thanks to which the

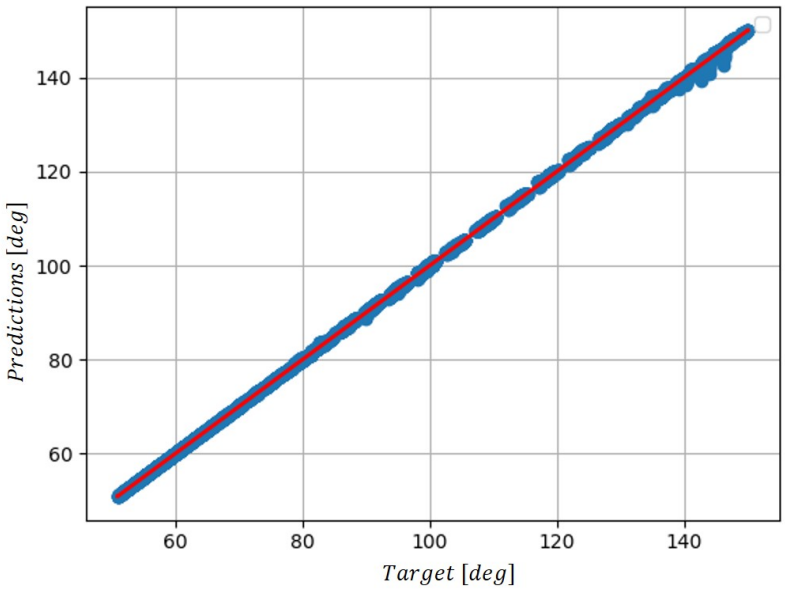


Figure 3.13: Regression on angles predicted with the trained RNN−LSTM

learner tries to imitate an external expert action in order to achieve the best performance. In particular in this work a DAgger (Data Aggregation) approach has been developed. The main advantage of this type of algorithms is that an expert teaches the learner how to recover from past mistakes. Nowadays, a classical application of DAgger is for autonomous vehicle and it applies, mathematically speaking, the following steps([14]):

1. Train the net (in this case a car) on a dataset $\mathbf{D}$ made of **human** observations and actions.
2. Run the net to get performances and then a new set of observations $\mathbf{D}_\pi$.
3. Ask the **human** expert to label the new dataset with actions.
4. Aggregate all data in a new dataset $\mathbf{D}_{new} = \mathbf{D} \cup \mathbf{D}_\pi$.
5. Re-train the net.

Practically what happens with an autonomous car is that the driver corrects online the errors done by the vehicle. Since it is not possible to exploit an human action/correction in space, the DAgger approach developed for this work is slightly different and it goes throught the following steps:

1. Train the net on a dataset $\mathbf{D}$ generated with GPOPS.
2. Run the net to get performance by using the test set $\mathbf{D}_{test}$.
3. Check for which trajectories the trained model error is not acceptable in terms of classification accuracy and RMSE.

4. Pick up the wrong trajectories in $\mathbf{D}_{wrong}$.

5. Use $\mathbf{D}_{wrong}$ to re-train the net and to improve the performances.

As said in Section 3.4.2 the global accuracy in the test phase was 99.73% and the global RMSE = 0.2°. The wrong trajectories have been picked up from the test dataset applying the criteria shown in Tab. 3.3 on each prediction. With the

Table 3.3: Criteria used to evaluate wrong predicted trajectories

Criteria	
Accuracy	$< 99\%$
RMSE	$> 0.3°$

trained RNN-LSTM and these criteria the wrong trajectories turned out to be 38 on the 192 belonging to the test set. Then, the DAgger has been implemented following two strategies:

- First strategy has been designed to train only on the trajectories with wrong predictions using the trained net model (Fig. 3.14).

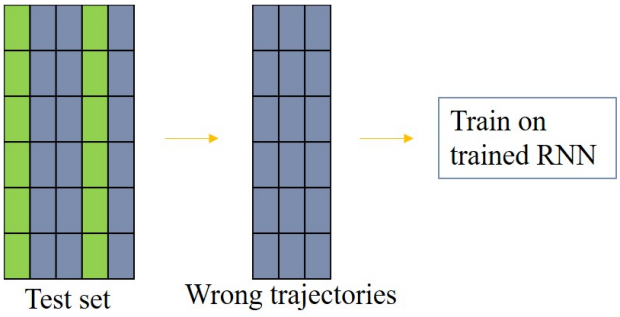


Figure 3.14: 1° strategy for the DAgger approach

- Second strategy has been conceived to enlarge the training test with the wrong trajectories and then re-train the net model (Fig. 3.15).

Afterwards, in order to evaluate the DAgger model, a new dataset of 100 trajectories, that had never been used by the model, has been created. The results are here reported and explained.

The net without DAgger achieves an accuracy of 99.23% and RMSE = 0.42° (Fig. 3.16). The model trained using the first DAgger strategy reaches an accuracy of 97.89% and RMSE = 0.44° (Fig. 3.17). Finally the results for the second

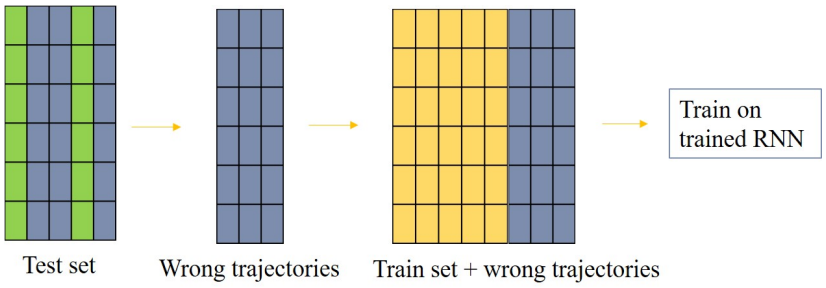


Figure 3.15: 2° strategy for the DAgger approach

DAgger strategy provide an accuracy of 99.25% and a RMSE = 0.40° (Fig. 3.18). All the results are summarized in Tab. 3.4.

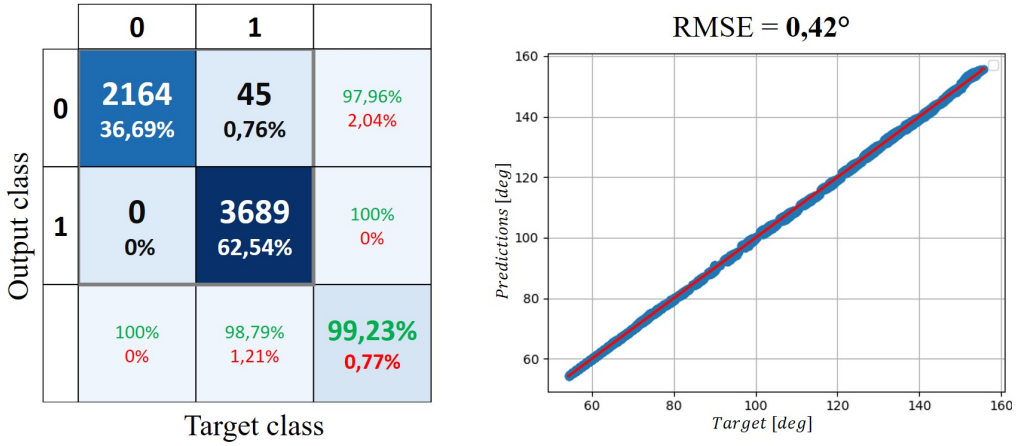


Figure 3.16: Accuracy and regression using the trained RNN−LSTM on 100 new trajectories

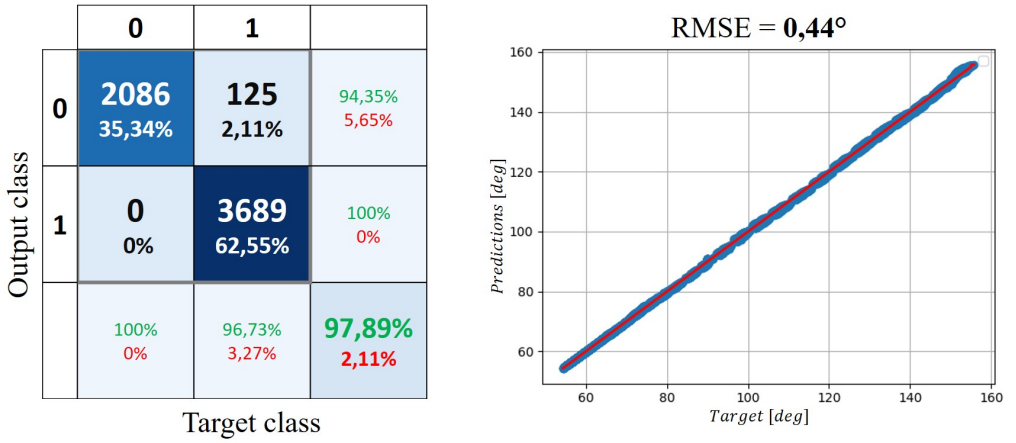


Figure 3.17: Accuracy and regression using the trained RNN−LSTM according to first DAgger strategy

In conclusion, it has been discovered that re-training the model only on wrong

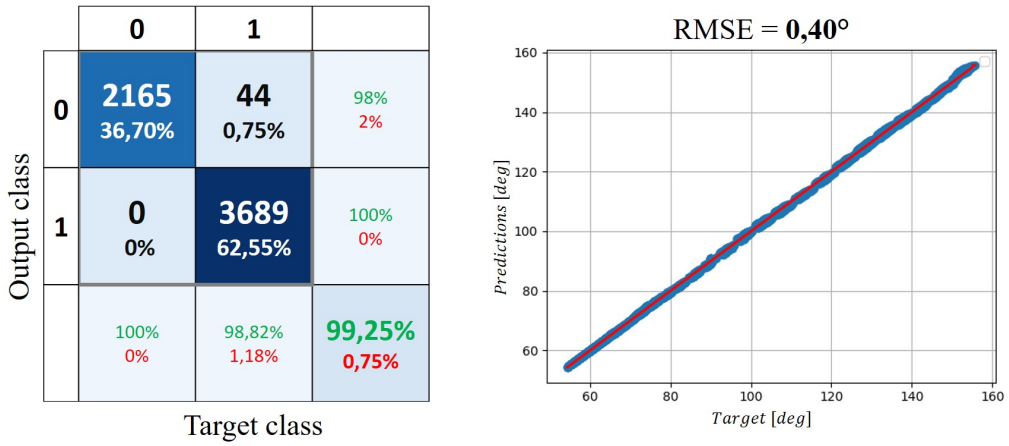


Figure 3.18: Accuracy and regression using trained RNN−LSTM according to second DAgger strategy

Table 3.4: Summary of the performances of the DAgger approaches

Model	Accuracy	RMSE
Trained net	99,23%	0.42°
1° *strategy*	97,89%	0.44°
2° *strategy*	99,25%	0.40°

predicted trajectories means that the weights and biases are updated focusing strictly on the wrong predictions. Therefore, testing the new model on a new dataset shows a loss of generality. Instead, re-training on the enlarged set allows to achieve the best performance.

3.6 Performance on a dynamics simulator

As it has been shown, the RNN−LSTM is able to achieve very good results in terms of accuracy on the classification and precision in the regression. Another important test that has been performed is a simulation of the network performances once it is installed on-board. The aim is, to verify if the spacecraft can be controlled only by a well trained neural network, to perform a landing. For this purpose, a suitable dynamics simulator has been developed, in which the trained RNN−LSTM has been used. In the simulation the spacecraft is controlled only by the neural network, starting from a known initial condition. In addition, the state at any instant is supposed to be known by the spacecraft and it is fed into the network. Since the network input is composed of three consecutive states, as described before, three initial consecutive states have to be known to initiate the simulation.

The network takes this first input to predict the first control action, with which the dynamics are propagated throughout the duration of the time step, keeping the control constant. The propagation is performed by integrating the equations of motion with an ordinary differential equation solver, that applies a Runge-Kutta method. Once the integration is finished, the new generated state is aggregated with the last two states of the previous input, in order to prepare the new input for the network. A new prediction and a new dynamics propagation follow. The loop is repeated until the final time or until some criteria are satisfied. The first criterion imposes the loop to stop when an altitude of 50 meters (which is the lower limit of the training trajectories) is reached. The second criterion applies when the spacecraft starts to rise up in altitude. In fact, it has been discovered that, after the 50 m have been reached, the neural network tends to control the spacecraft in such a way that it increases its altitude, moving it away from the ground. In the followings, the results of the simulations are shown. The simulator used to generate the followings figures has been developed in *Python*.

A simulation has been performed considering the initial conditions of a trajectory taken from the testing set and the results are shown in the following. In Tab 3.5 it is possible to see a comparison between the theoretical optimal final state and the final state of a spacecraft completely controlled by the neural network. As it is shown, they are very similar. In Fig. 3.19 the optimal trajectory and the predicted one are shown. It is possible to see how the predicted one is close to the optimal one. Finally, in Fig. 3.20 it is possible to see an example of what happends when the stopping criteria are not applied. The network has learnt to reach the lowest limit of 50 meters. It is also true that, since during the training all the trajectories considered end at 50 meters, states with altitudes below this limit are outside the range of the dataset. Consequently, the attention must be focused on what happens within the range covered by the dataset: what happens outside this range should be discarded.

Table 3.5: Comparison between optimal and predicted final state using RNN−LSTM

	optimal	predicted	unit
downrange	0	-0.9	$[m]$
altitude	50	50.3	$[m]$
downrange velocity	-0.5	-0.015	$[m/s]$
altitude velocity	-0.5	-0.6	$[m/s]$

The simulations have shown that the network's performances are very sensitive

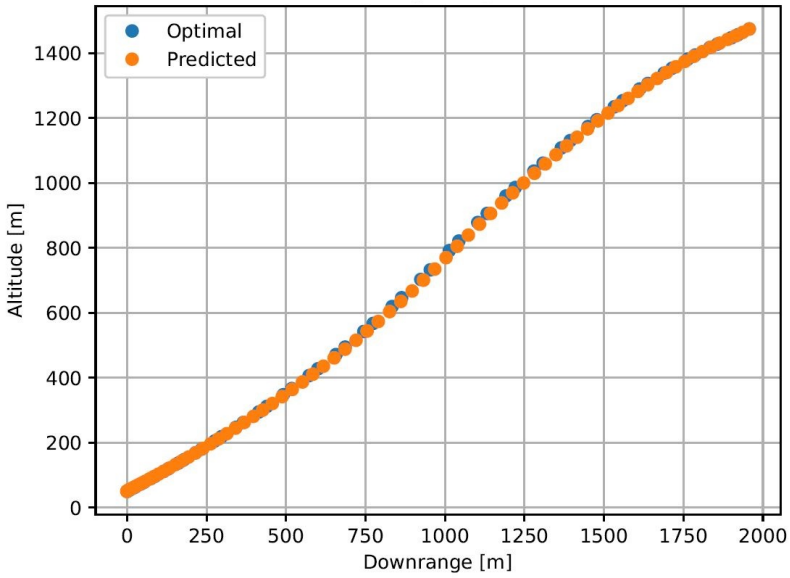


Figure 3.19: Comparison between optimal and predicted 2D trajectory

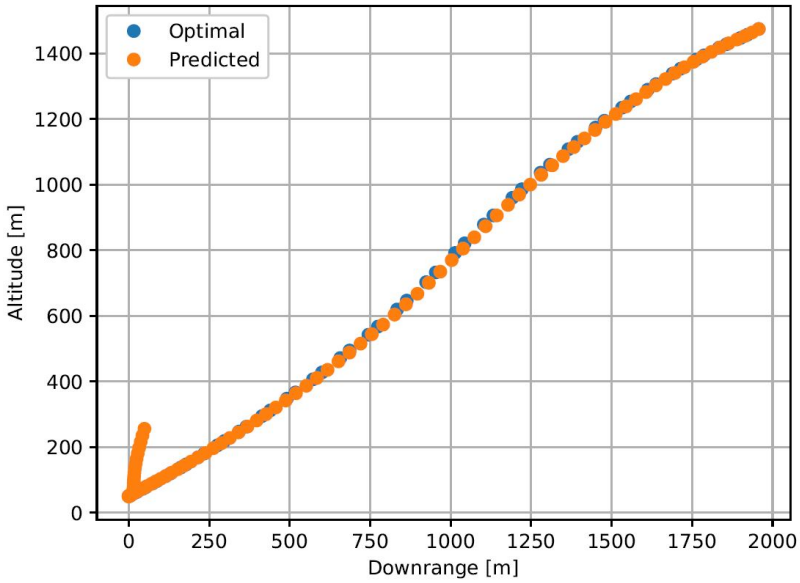


Figure 3.20: Predicted trajectory after the lowest altitude has been reached

to the time step extension. The time step is the only parameter that can be modified by the user. If the time step is reduced, the network will be required to process the input and give a control action more frequently, while if the time step is increased, a smaller number of predictions will be requested. In the first case, the control would be more accurate, but the network may not be trained to provide high frequency predictions. In fact, asking the network to predict the control actions more frequently than it was designed for, may lead to big errors due to the fact that the inputs would be very similar one to each other.

In the second case, the accuracy of the prediction may drop, which could cause the generation of an inaccurate control. A good balance, in which the network is able to give correct predictions and the control is quite accurate despite the approximations, must be found between these two cases. For the example shown, the time step has been set equal to 0.95 s.

3.7 Monte Carlo analysis

A Monte Carlo simulation has been performed to better characterize the errors of the neural network, in terms of landing position and landing velocity. For this purpose a new set of initial conditions has been generated, keeping fixed the initial altitude at 1250 meters and perturbing the initial downrange according to a Gaussian distribution centred at 1750 meters. In this way, a Gaussian distribution of initial conditions, centred in the middle of the region covered by the dataset (Tab. 3.1b), has been considered. In Fig. 3.21 the distribution of initial downrange and their probability is shown. Note that, in so doing, it is assumed that the initial conditions in the middle of the dataset are more probable than the peripheral ones. These initial conditions have been propagated using the dynamics simulator, described in the previous section. For each initial condition, the propagation, has been stopped once the spacecraft reached 50 meters of altitude. The output of this simulation is a series of plots, in which the final downrange, the final vertical velocity and the final horizontal one are shown.

As said regarding the dynamics simulator, an important issue is the selection of the time step. The first Monte Carlo simulation has been performed keeping the time step constant for each trajectory of the set. In this way, each initial condition has been propagated with the same number of control actions predicted by the network. The results have proved that this is not a good way to proceed. As it is shown in Fig. 3.22 the final state of the propagated trajectories are far from the optimal ones. In particular the final velocities are too high (more than -10 m/s for the horizontal component and more than -8 m/s for the vertical one).

Thanks to these results it has been possible to understand that the time step must be tuned for each initial condition. There is not a unique value of time step for which the network provides good results, whatever the initial conditions are. For this reason, a second Monte Carlo simulation has been completed trying to find for each initial conditions the best value of the time step. This has been done in an automatic way using a *for* loop in which many time steps are tried and the one that provides the best results (in terms of final velocities and downrange

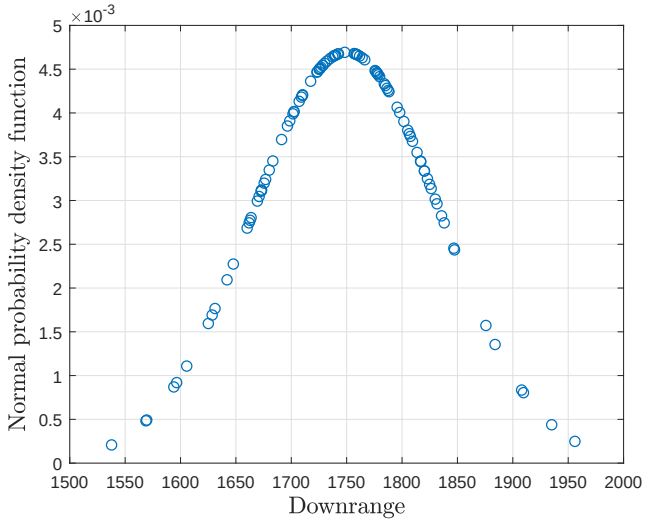
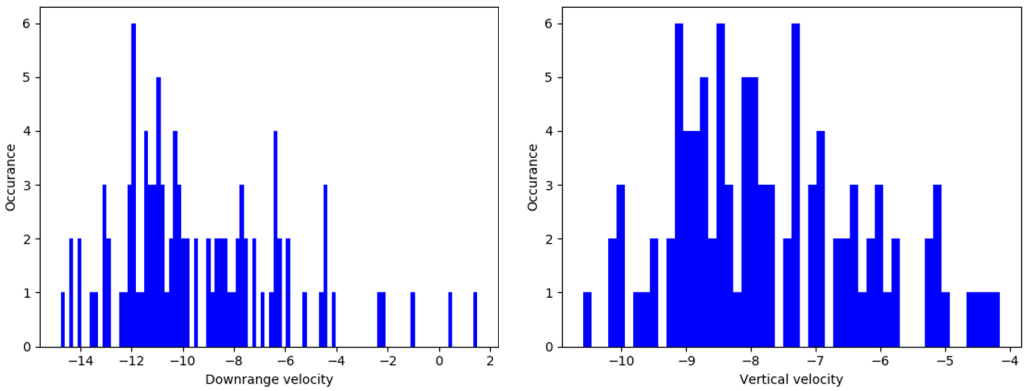


Figure 3.21: Downrange initial conditions for the Monte Carlo simulation



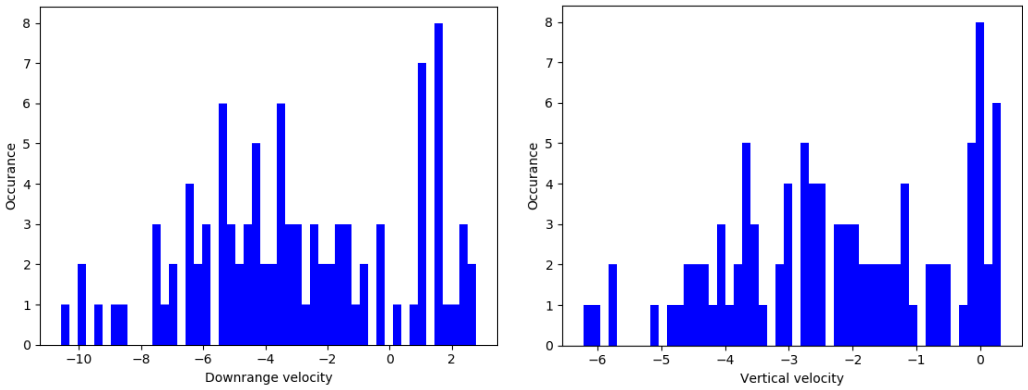
(a) Final downrange velocity **(b)** Final vertical velocity

Figure 3.22: Results of the first Monte Carlo simulations

position) is selected. The results of this second simulation are shown in Fig. 3.23.

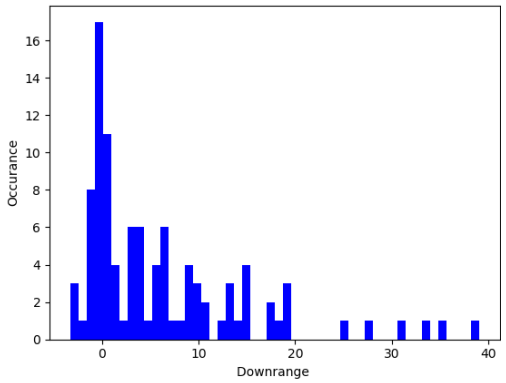
The mean of the final downrange velocity is -2.97 m/s, while the mean of the final vertical one is -2.2 m/s. Finally, the mean of the final downrange is 6.23 m. They are much lower than the results of the previous Monte Carlo simulation, but are still quite far from the optimal ones (-0.5 m/s for both velocities and 0 meters of downrange). In Fig. 3.24 the final downrange are plotted. As it is possible to note, most of the final points shows a final downrange around zero (which is the target).

This result is due to the fact that the control applied during the propagation is not very accurate, since during each integration the control is kept constant. This problem, as mentioned in the previous section, could be solved by asking



(a) Final downrange velocity

(b) Final vertical velocity



(c) Final downrange

Figure 3.23: Monte Carlo second simulation results

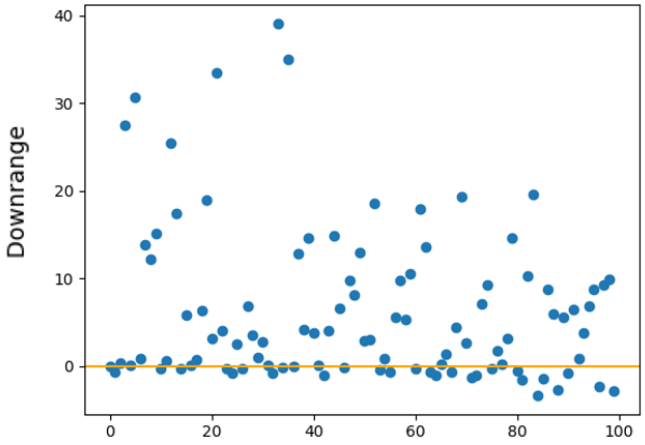


Figure 3.24: Final downrange distribution

the net to process the inputs more frequently, so as to have a discrete but more accurate control during the landing. The limit of this approach is represented by the fact that the neural network is not trained to take inputs at high frequency, because the trajectories within the training set do not have an adequate number

of points. The considered trajectories, in fact, are made of only 61 states, that means one every 0.9 seconds. One solution would be to train the neural network on trajectories with many more points.

Chapter 4

Moon surface images simulator

According to the project requirements, the spacecraft takes pictures of the Moon surface during the landing. This behaviour had to be simulated for both 2D and 1D cases, in order to generate a dataset of images necessary to train the network. For this reason, an image simulator has been developed and implemented by means of a software tool. In this chapter, after an explanation of the landing site choice, the simulator working principles and the images characteristics are described.

4.1 Landing site

The landing site location has been the first choice to make. Considering previous lunar missions, Apollo 16 landing site has been selected. The altitude in this region goes from -160 to 10 meters since some hills and craters are present; nevertheless, this location does not show particular hazards for a hypothetical landing. Moreover, a completely flat and smooth region would not have been a good choice from the CNN point of view because of its poor features: the network works by means of features detection in order to understand how to predict the right output. Craters and hills are objects that can be easily found and identified by the net and the selected landing site presents a suitable surface roughness and irregularity. To generate a realistic image of the surface of a planet or of a satellite, a Digital Terrain Model or (DTM) is needed. The DTM of the selected region of the Moon surface has been taken from The Lunar Reconnaissance Orbiter Camera (LROC) web site [13], where a lot of different kind of images and information about the Moon can be found. The area covered by the DTM has a Western-most longitude of 15.26° and an Eastern-most longitude of 15.42°, while maximum latitude is $-8.71°$ and minimum latitude is $-9.65°$, with a resolution of 2 meters/pixel. In this way, more than 139 km^2 of Moon surface are modelled.

Since there was no sense in using such a large DTM, only a smaller portion of it has been selected and used; in particular, in the 2D case 64 km^2 (4000×4000 pixel) have been considered and only 16 km^2 (2049×2049 pixel) for the 1D case. In fact, considering the vertical 1D landing, the area seen by the on-board camera is affected only by the variation in altitude and what is in the middle of the first image (when the altitude is maximum), will be in the middle of each picture. Instead in the planar case, downrange variation makes the objects seen by the camera change during the manoeuvre and for this reason, a wider region must be considered.

As already said, the resolution of the available DTM is 2 meters/pixel. It is obviously not sufficient when the images are taken from the lowest altitude considered, which is 50 meters. An example of some images generated with the original DTM is shown in Fig. 4.1. As it can be seen, while the altitude decreases starting from the left picture toward the right one, the quality of the images degrades. To improve the resolution, the Diamond-Square algorithm, which is illustrated in the following, has been applied on the DTM.

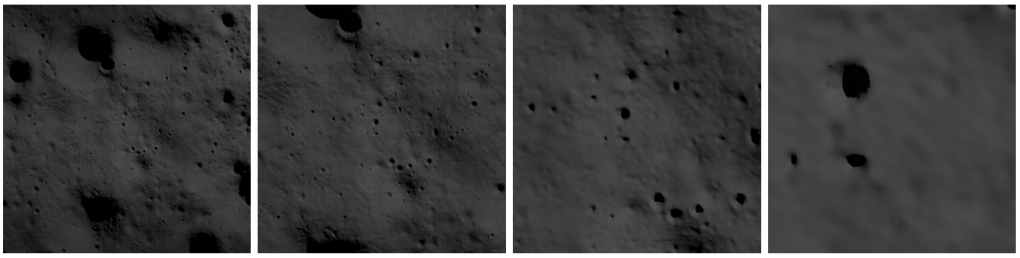


Figure 4.1: Moon surface images taken from different altitudes, generated with original DTM.

4.1.1 Diamond-Square algorithm

The Diamond-Square algorithm is used to generate heightmaps and it has been introduced in 1982 at the annual conference on computer graphics [15]. Its implementation is very simple and much material can be found online([14], [15]). The algorithm performs alternately the diamond and the square steps. In the first one, after the four corner points of the image array are set to initial values, the midpoint of this square is set to be the average of the four corners points plus a random value. In the square step, for each diamond in the array, the midpoint is set to be the average of the four diamond corner plus a random value. At each iteration, the magnitude of the random value is reduced. The algorithm is repeated until all the array values have been set. It is clear that no new information are added to the original image, but the output image will

have much more points than the input one. Fig. 4.2 shows the steps involved in running the diamond-square algorithm on a 5×5 array.

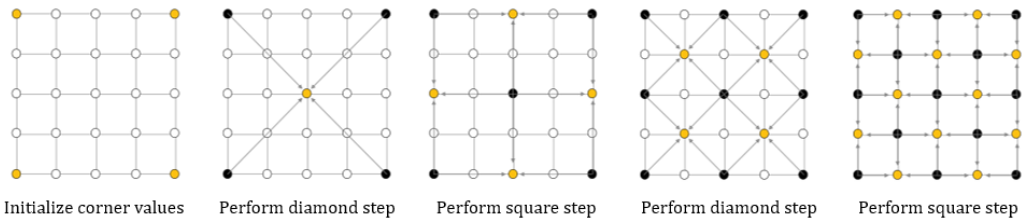


Figure 4.2: Diamond-square algorithm

The portions of the original DTM has been used as inputs and the procedure has been applied. In this way the 1D case DTM passed from a pixel dimension of 2049×2049 to a dimension of 16385×16385 and the 2D case one from 4000×4000 to 16000×16000. Since the portions of Moon surface modelled by the DTMs remain the same, the resolution has been increased: there are much more points to represents the same spatial extent. But it is important to remember that all the added pixels are only averaged on the original ones. As it will be shown in the next section, the images quality has been enhanced. In a similar way, the texture has been generated by applying the diamond-square algorithm without giving an image as input, but only repeating the procedure and adding random values at each iteration. The number of iterations has been selected in order to obtain a texture with a resolution equal to the DTM one.

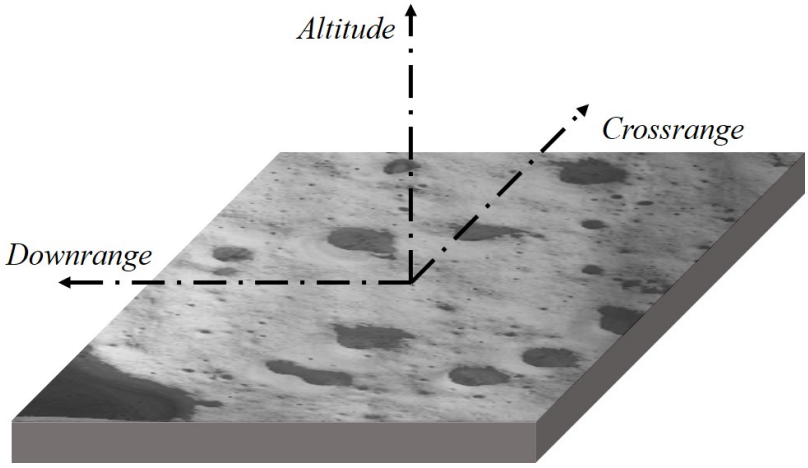
4.2 Images simulation: *POV-Ray*

In order to generate a dataset of images taken during the Moon landing, a simulator has been developed in which each image is associated to the state of the spacecraft at each time step. It is useful to remind now that the states have been computed by GPOPS when the fuel-optimal trajectories have been generated. Regarding the 2D planar landing, a state is composed of downrange, altitude, velocities and mass, while in the 1D vertical case, only altitude, vertical velocity and mass are present. Two simulators have been developed, one for the 1D case and one for 2D landing. In the first one, only the altitude is necessary to generate an image, while in the second one downrange and altitude are used. The simulator has been written in *MATLAB* and the software which has been used to generate every single image is Persistence of Vision Ray Tracer or *POV-Ray*, which is a ray tracing program which generates images from a text-based description of a scene. In a *MATLAB* script, the trajectories coming from GPOPS are loaded and each state is read in a for loop where *POV-Ray* is called to render each

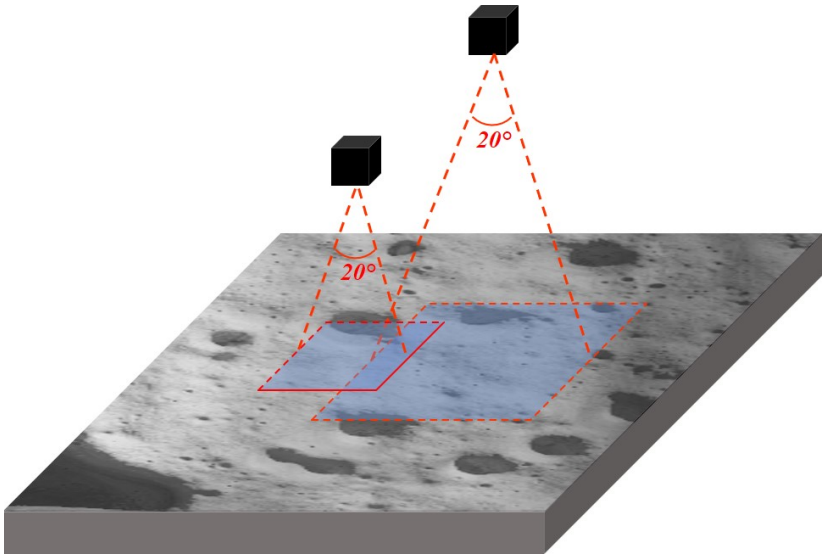
image. In order to render an image, *POV-Ray* needs three important objects: the light source position, the camera position (and properties) and the objects to render. The light source, in this case, is the Sun and its position with respect to the Moon has been considered fixed during the entire simulation because the duration of each landing is so short (about one minute) that the variation of the Sun position is not relevant. The camera position, instead, changes at each time step according to the current state generated by GPOPS. In *POV-Ray* the camera position is expressed through a vector of three coordinates. In the 1D case, only the out-of-plane component changes according to the altitude, while the other two components are kept constant equal to 0. In 2D case, the out-of-plane component is conditioned on the altitude and one of the planar components changes according to the downrange, while the third one is kept equal to 0 (Fig. 4.3). In this way, an image is associated to each state. Since in GPOPS, at each state corresponds a control action, once all the images are generated it is possible to make a dataset in which each image is correlated with the control action. Such dataset is suitable for the training phase of the network. Datasets of more than 6000 images have been generated for both cases under study. The objects to render are described in a *POV-Ray* script in which the DTM and the texture are loaded and scaled according to the image size and to the real altitude range. In this script, the radiosity model and other options regarding the light and the camera properties are implemented. A very important aspect is the size of the image. The size in *POV-Ray* is expressed in pixel and both 1024×1024 and 256×256 images have been generated. First simulations proved that the set made up of the largest images requires excessive computational cost and allocation memory. For this reason, smaller images have been used for the successive simulations. They are quite small but the simulations have revealed that they are sufficiently large to obtain good results. The images are greyscaled and not RGB: this helps a lot during the render of each image and during the training and test phase of the network, reducing the computational cost and the allocation memory. Moreover, there is no need to have RGB images considering the texture of the Moon surface. *POV-Ray* let the user set up some camera properties. The most important one is the angular field of view. For the research purposes, the camera angular field of view has been fixed at $20°$, which is a reasonable value for on-board navigational cameras. A lower value would have involved a too narrow field of view and a too small portion of the surface would have been seen, especially for the images taken from lower altitudes.

It is worth stressing that this simulator works having all the trajectories completed by the optimizer. Once the neural network has been trained, a slightly

different version of the simulator can be employed to test the network and to verify, in closed-loop, if the control action predicted by the CNN is able to land the system with an acceptable amount of error, both in position and velocities. This type of simulator will be described in the next chapter. In Fig. 4.4 some examples of the images related to the 1D vertical landing are shown and in Fig. 4.5 related to the 2D planar landing case.

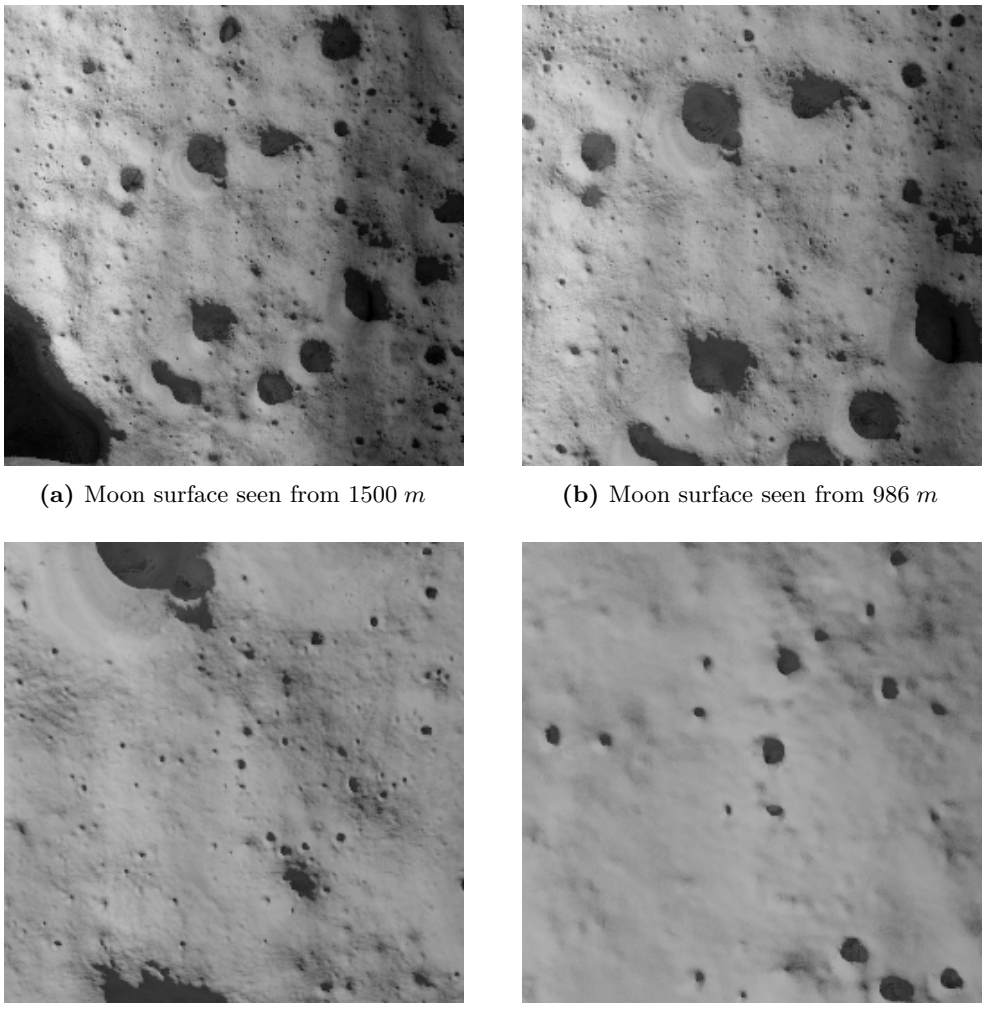


(a) Image simulator axis



(b) Image simulator procedure example

Figure 4.3: Moon image simulator scheme



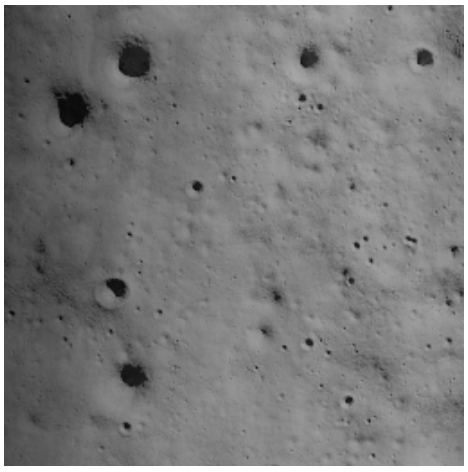
(a) Moon surface seen from 1500 m

(b) Moon surface seen from 986 m

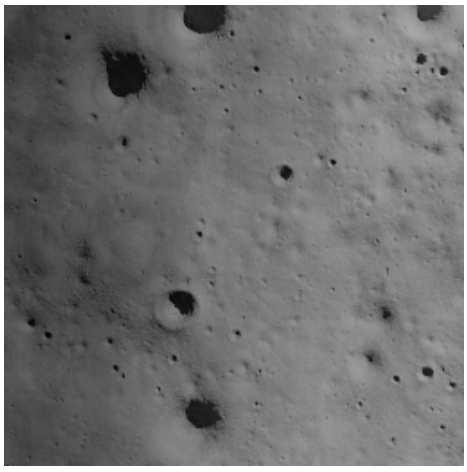
(c) Moon surface seen from 464 m

(d) Moon surface seen from 140 m

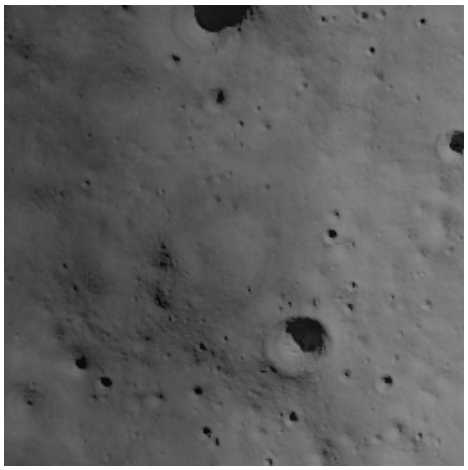
Figure 4.4: Moon surface images taken from different altitudes



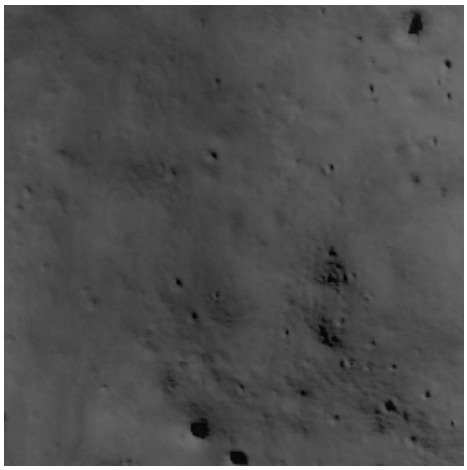
(a) Moon surface seen from 922 m of altitude and 1300 m of downrange



(b) Moon surface seen from 790 m of altitude and 1060 m of downrange



(c) Moon surface seen from 594 m of altitude and 790 m of downrange



(d) Moon surface seen from 365 m of altitude and 520 m of downrange

Figure 4.5: Moon surface images taken from different positions

Chapter 5

From image to control: vertical Moon landing

The problem faced in this chapter is a vertical Moon landing in which a CNN is exploited to predict the control action directly from the Moon surface images. The network has been trained according to supervised learning algorithm. In this first application of CNN, it is possible to see how the convolutional neural network is able to extract information directly from raw images and predict the correct control action. A 1D vertical Moon landing has a dynamics that is simpler than the previous 2D landing. After the problem formalization, in which the equations of motions are described, the procedure to generate the dataset is illustrated. Afterwards, the proposed network architecture is shown and the training and test phase are illustrated. Finally the results coming from test simulations are given.

5.1 Problem formalization

A vertical 1D Moon landing is considered, in which the degree of freedom is the altitude. The state is composed of three components: the altitude, its derivative and the mass. The latter is coupled with the spacecraft dynamics only through the thrust. The control action will have only one components as well, aligned with the vertical direction. It is clear that this problem is only a simplification of the problem shown in Section 3.1, in which instead of two variables, only one is present.
The equation of motion are the following:

$$\begin{cases} \dot{h} = v_h \\ \dot{v_h} = -g + \dfrac{T}{m} \\ \dot{m} = -\dfrac{T}{I_{sp}\, g_0} \end{cases} \tag{5.1}$$

where h is the altitude, v_h is the vertical velocity and m is the mass. The thrust is T and g is the lunar gravity acceleration. I_{sp} is the specific impulse and g_0 is the reference gravity acceleration. These equations have been integrated to minimize the following cost function:

$$\int_0^{t_f} T \, d\tau$$

where u is the control action and t_f is the final time, which is free. As it can be seen, the optimality condition is achieved when the control actions are minimized, in other words, when the depleted mass is minimum. Like before, this condition can be called fuel-optimal condition. It is important to remember that the solution of such problem has a bang-bang profile, in which the thrust is alternatively maximum or minimum, or on/off. For this reason, when the neural networks is designed, it is possible to consider the problem like a classification one, in which only two choices are possible: maximum or minimum thrust. Initial conditions are given as initial altitude h_0 and initial vertical velocity v_{h_0}. The initial mass is set equal to m_0. Final conditions on each variable of state, except the mass, are imposed. Since the final time is free, only a lower limit for the mass is considered and set equal to the dry mass m_{dry}.

5.2 Dataset generation

A dataset is needed to train the network, dealing with supervised learning. In order to generate a suitable dataset, GPOPS has been used as before, solving the equations many times starting from different initial conditions. For more information about GPOPS, see Section 3.2.

The initial mass of the spacecraft has been set equal to 1300 kg. The altitude has been initialized between 1000 and 1500 meters. The initial vertical velocity v_{h_0} changes according to the altitude: when h is maximum, the velocity is maximum in modulus (−11 m/s), vice versa when h is minimum, the velocity also is minimum in modulus (−6 m/s). Unlike the initial conditions, the final ones have been kept constant for all trajectories. In particular, the final altitude h_f is equal to 50 meters and the final velocity is equal to −0.5 m/s. As it can be seen, the final condition is such that the spacecraft has not touched yet the ground and has a very small velocity, directed toward the ground. Throttlable thrusters have been considered, and modelled following the same strategy of Section 3.2. The nominal thrust T_{nom} is equal to 4000 N. The maximum (allowable) thrust has been fixed equal to the 85% (3400 N) of the nominal thrust and the minimum (allowable) equal to 25% (1000 N). It is easy to understand that in this case, the network will have to perform only a classification on the thrust magnitude.

In this way, 101 trajectories have been generated within the selected range of altitude. The parameters of the problem are shown in Tab. 5.1a. A wrap-up of the initial conditions and one of the final ones are shown in Tab. 5.1b and in Tab. 5.1c.

Table 5.1: Parameters and state values for 1D problem

(a) Problems parameters

	value	unit
m_{dry}	500	kg
g	-1.622	s^2
I_{sp}	200	s
T_{nom}	4.0	kN
T_{max}	3.4	kN
T_{min}	1.0	kN

(b) 2D initial conditions

	value	unit
h_0	$[1.0,\ 1.5]$	km
v_{h_0}	$[-6, -10]$	m/s
m_0	1.3	ton

(c) 2D final conditions

	value	unit
h_f	50	m
v_{h_f}	-0.5	m/s

Each trajectory coming from GPOPS has 61 points, so 61 states and 61 control actions. Each states has three elements, altitude, velocity and mass; each control action has only one component, the magnitude. The final dataset has been built by associating each state to each control action. This dataset has been divided in training set and test set; the first one comprehends 81 trajectories, the second one 20.
In Fig. 5.1 it is possible to see an example of a trajectory in the dataset. For the example shown, the initial altitude was equal to 1250 meters. Fig. 5.2 shows the plot of the thrust magnitude bang-bang profile, between the two conditions (minimum or maximum), as the theory predicts.

5.3 Proposed network architectures

To fulfil the described vertical Moon landing, different architectures have been tried and three main kind of CNNs have been implemented:

1. **First architecture**: a classical CNNs has been exploited and therefore, both convolutional layers and max-pooling layers are present (Fig.5.3).
   The hyper-parameters for each layer have been set as:

   - First Convolutional layer, [36 *filters*, *size* 4×4, *stride* 2].
   - Max-pooling layer, [*size* 2×2, *stride* 2].

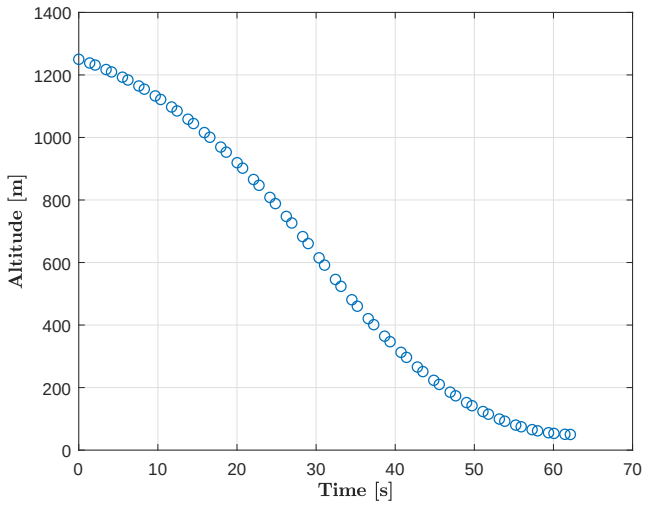


Figure 5.1: 1D altitude profile

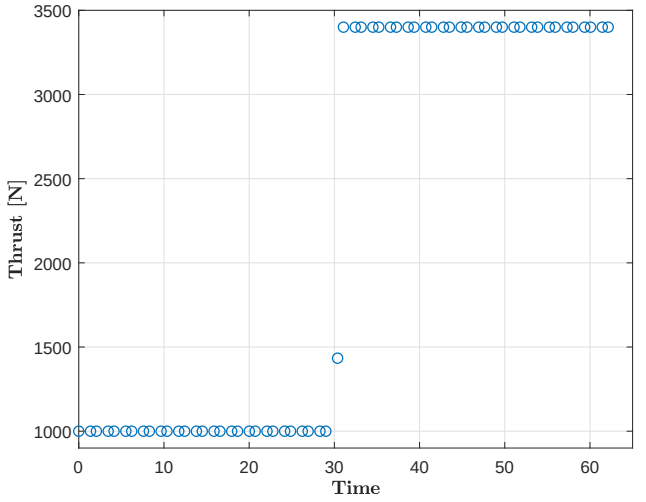


Figure 5.2: 1D thrust magnitude

Figure 5.3: First CNN architecture

- Second Convolutional layer, [36 *filters*, *size* 4×4, *stride* 4].
- Max-pooling layer, [*size* 2×2, *stride* 2].
- Third Convolutional layer, [72 *filters*, *size* 2×2, *stride* 1].
- First Fully connected layer, [256 neurons].
- Second Fully connected layer, [128 neurons].

- Output layer, [2 classes].

2. **Second architecture**: in this network max-pooling layers have been removed. The net looks like the one in Fig. 5.4 and its hyper-parameters have been set as:

   - First Convolutional layer, [36 *filters*, *size* 4×4, *stride* 2].
   - Second Convolutional layer, [36 *filters*, *size* 4×4, *stride* 4].
   - Third Convolutional layer, [72 *filters*, *size* 2×2, *stride* 1].
   - First Fully connected layer, [256 neurons].
   - Second Fully connected layer, [128 neurons].
   - Output layer, [2 classes].

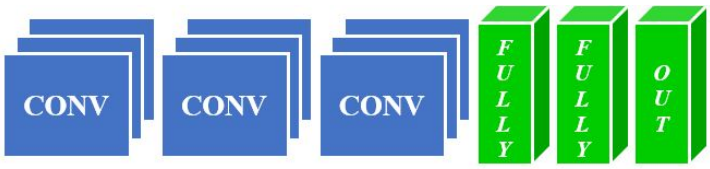


Figure 5.4: Second CNN architecture

3. **Third architecture**: this last architecture (Fig. 5.5) is made up of 5 convolutional layers with following parameters:

   - First Convolutional layer, [36 *filters*, *size* 3×3, *stride* 2].
   - Second Convolutional layer, [36 *filters*, *size* 3×3, *stride* 4].
   - Third Convolutional layer, [72 *filters*, *size* 2×2, *stride* 2].
   - Fourth Convolutional layer, [72 *filters*, *size* 2×2, *stride* 2].
   - Fifth Convolutional layer, [72 *filters*, *size* 2×2, *stride* 1].
   - First Fully connected layer, [256 neurons].
   - Second Fully connected layer, [128 neurons].
   - Output layer, [2 classes].

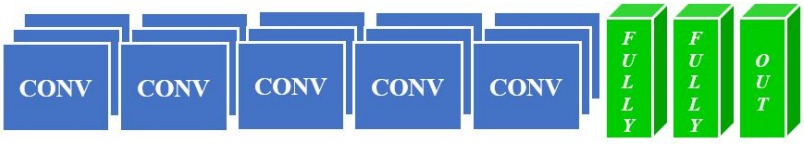


Figure 5.5: Third CNN architecture

The tests have showed that better results can be achieved with network deprived of max-pooling layers. In fact, this kind of layer eliminates many information (as shown in Section 2.4) which instead the network needs to be properly trained and to extract features. When the results revealed themselves promising, it has been decided to increment the number of convolutional layers up to five. In this way, the total amount of data is processed and gradually reduced, but no data are directly thrown away. A rough test accuracy on these CNNs leaded to the accuracies shown in Tab. 5.2. Training phase and more details about third CNN results are described in following the section.

Table 5.2: Accuracy for each implemented CNN

Model	Accuracy
First architecture	90%
Second architecture	95%
Third architecture	97%

5.4 Training and test phase

5.4.1 Training

Differently from the case previously treated, the neural network has been trained to understand from raw images the correct control action to apply to perform a fuel-optimal vertical 1D landing on the Moon. For this purpose, the states, coming from the dataset described in Section 5.2, have been used as input for the 1D Moon images simulator (described in Chapter 4). In this way, a new dataset has been created associating all the images to the relative control actions. As was done in the case of 2D landing with RNN, here too, each input is composed of three consecutive superimposed images. This has an even more important meaning than the previous case studied, because each image is like a static picture of the Moon surface and with one only image it would be impossible to have any information about the velocity of the lander. In this way instead, the net has the possibility to keep track of velocity, improving the accuracy of the response. The sequence of images, belonging to each trajectory (61 states), has been divided in sets of three images following the second strategy explained in Section 3.4.1, according to which each trajectory, is separated from the other. In fact, as it has already been shown, this technique has proved to give better results. The input packages are 59, because eve if images per trajectory are 61, the first two enter

the net only once. As already said, the solution of a fuel-optimal problem has a bang-bang profile and therefore, the control action can have only two possible values. Moreover, in the case of a vertical landing, the control direction is always the vertical one. For these reasons, the network has to solve only a classification problem: from all the information that can be extracted from the inputs, the net has to decide which is the correct thrust magnitude, between the allowed two. Binary vectors of length two have been used to represent the two classes. When the thrusters are at their maximum, the correct label is [0, 1] and when they are at minimum thrust, the label is [1, 0]. Each input package of three sequential images has been associated to only one output label. In this way, from a control system point of view, at time t the control action is predicted taking the image of the current state and those of state $t-1$ and $t-2$. The network has all the information contained in three consecutive frames to predict the correct label.

The labels have been associated to the images following two different logics.

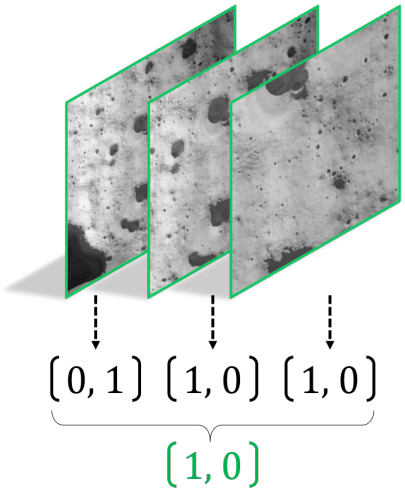


Figure 5.6: CNN input according to 1° logic

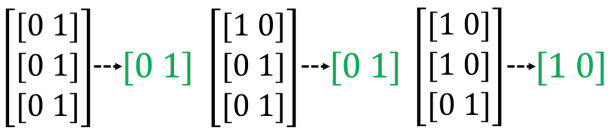


Figure 5.7: Mean of thrust vectors

According to the first technique, the idea has been to exploit all the information available on thrust related to three input images. Therefore the labels to consider, in this case, are three vectors. The thrust entering as label in the system for training has been computed by applying the mean of the input thrust vectors (Fig. 5.6). Some examples of mean procedure are described in Fig. 5.7. The resulting thrust is then linked with the third image, this thinking about the way

to proceed of a control system (predict the thrust of the current state by looking the previous states). This approach has proved to be inaccurate because it is not capable to catch the thrust change, which is not continuous. In particular, the thrust changes are seen with delay. To better understand this consideration, Fig. 5.8 shows how the network works using the approach explained above. In conclusion it is not possible to achieve high accuracy because the net will always make a mistake in the regions where jump conditions are present, even if it works properly far from them.

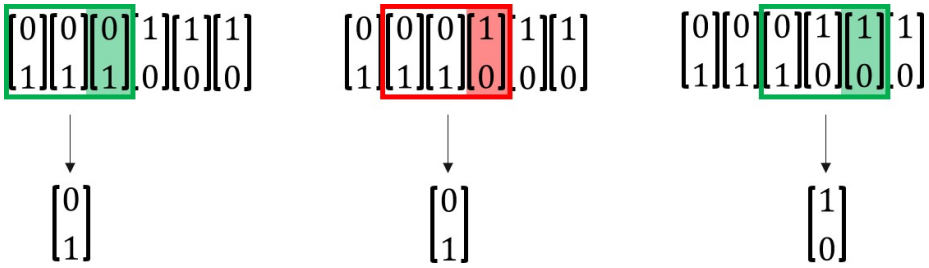


Figure 5.8: Example of wrong input

According to the second approach, the network has been fed with only one label relative to the optimal thrust associated to the third image in input, as it is shown in Fig. 5.9. This logic has proved to be better.

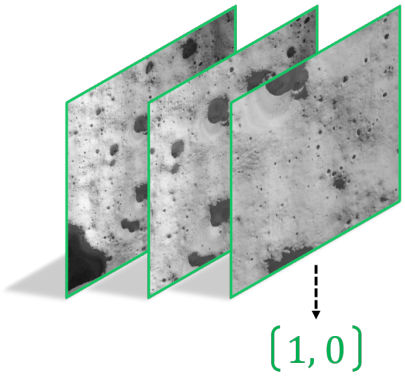


Figure 5.9: CNN input according to 2° logic

Before entering the net, the images are normalized between 0 and 1 considering the maximum value among all the image dataset. In this way each image is normalized with the same logic. Each image enters the net as a square matrix 256×256. Therefore, the input packages are three-dimensional matrices $3 \times 256 \times 256$ [number of images $\times$ image size $\times$ image size]. As explained in Section 3.4.1, it is possible to feed the net with a batch of inputs and this technique can result in an important accuracy improvement. In order to train the network on an entire trajectory at each iteration, a batch of 59 inputs has been considered. In this way,

59 sequential input packages and 59 correct labels enter the net at each iteration during the training phase.

As said in Section 2.2, the comparison of the predicted net outputs and the right outputs is carried out through the computation of a loss function and the purpose of the training phase is to minimize it. In Section 3.4.1, Adam optimization algorithm, used to minimize the loss function, has already been presented. Also in the case of a vertical Moon landing with images, it has been exploited during the training phase. Remember that it can adapt the learning rate as training unfolds thanks to the decay parameter (Eq. 3.4). Also the study here presented has been developed in a *Python-Keras* and *Pyhton-Tensorflow* environment, where Adam method is already implemented. Since the network has to solve only a classification problem, there was no need to weight the loss function values associated to the model outputs. In Tab. 5.3 the chosen hyper-parameters are resumed. Classification predictions can be evaluated using accuracy and the

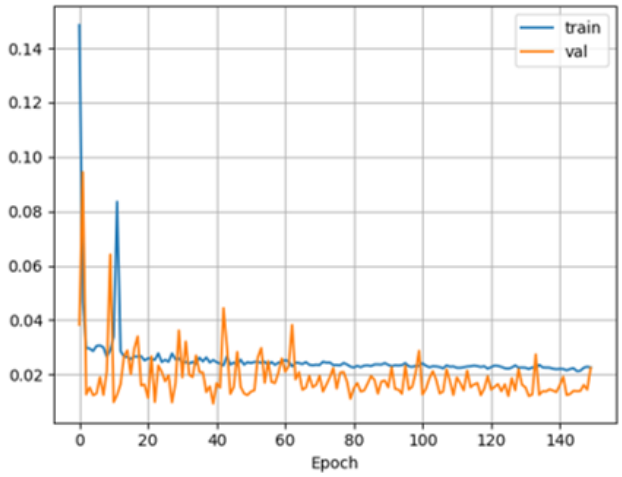
Table 5.3: Resuming of hyper-parameters of CNN 1D

Hyper-parameters	
Batch size	59
Initial learning rate	0.001
Decay rate	0.0001

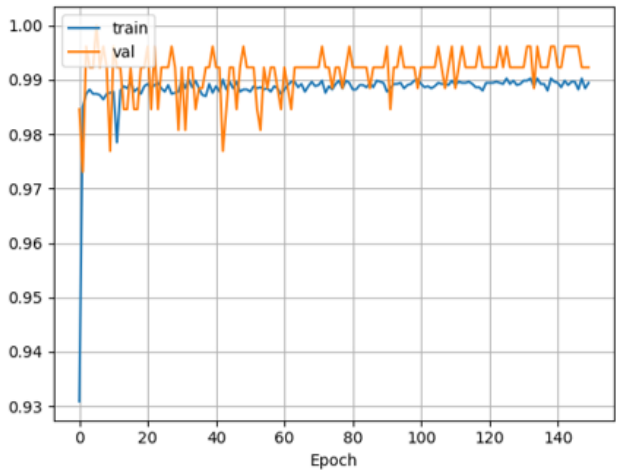
loss function associated is the *Cross-entropy loss* (Eq. 3.5) which indicates the distance between the model prediction and the true value of the output and where softmax function outputs a probability distribution (Eq. 3.6).

The results are now illustrated. The training has been performed using the training set with 4941 images (61 state $\times$ 81 trajectories). The network has been trained in 200 epochs, using in each one a batch of 59 inputs. In order to have a feedback during training phase on net performances the training set has been split, also for this network, in order to have a 5% of the dataset available for the validation model. In Fig. 5.10a the loss trend during the training phase is shown, while in Fig. 5.10b, it is possible to see the CNN accuracy trend. The training phase has been performed with the help of a High Performance Computing (HPC) systems of the University of Arizona and it has taken about 12 hours to complete the train. Looking at the training results, it is clear that some parameters can be changed in order to improve the minimization process of the loss function. In fact, as it can be seen, the loss function, as well as the accuracy, are still a bit noisy after 200 iterations. This could mean that the minima has not been already

reached or that hyper-parameters must be properly tuned. Good results have been anyway obtained even if some future work has to be done.



(a) Loss function trend



(b) Accuracy trend

Figure 5.10: Loss function and accuracy results during CNN 1D training phase

5.4.2 Test

As said before, the test set is made by 20 trajectories. To visualize the trained net performances on the classification (accuracy) a confusion matrix has been computed.

As shown in Fig. 5.11, the CNN trained model has an accuracy of $97,63\%$ (remember that in the confusion matrix 0 is associated to minimum thrust, 1 to maximum) which is lower than the minimum target of 99%, as mentioned in Section 3.4.2. In the following section, a method to improve the precision is illustrated.

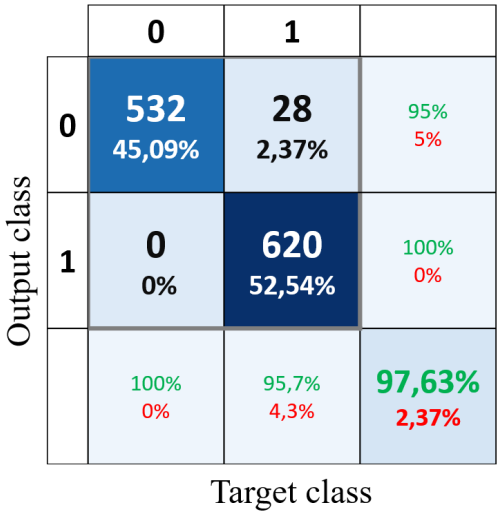


Figure 5.11: Confusion matrix CNN 1D

5.5 Accuracy improvement: DAgger approach

In Section 3.5 the DAgger approach has already been described. After the training and test phase of the CNN a DAgger method has been implemented to enhance the accuracy on the predictions. It is important to remember how the DAgger approach has been implemented in this work:

1. Train the net on a dataset $\mathbf{D}$ generated with GPOPS.
2. Run the net to get performances by using the test set $\mathbf{D}_{test}$.
3. Check for which trajectories the trained model error is not acceptable in terms of classification accuracy and RMSE (if any).
4. Pick up the wrong trajectories in $\mathbf{D}_{wrong}$.
5. Use $\mathbf{D}_{wrong}$ to improve net performances.

As said in Section 5.4.2 the global accuracy on test set has been 97.63%. The trajectories predicted with an accuracy $< 98\%$ have been picked up from the test dataset. With the trained CNN and this criteria the wrong trajectories turned out to be 8 of the 20 belonging to the test set. Then, the DAgger has been implemented following the second strategy explained in Section 3.5: aggregate the training test with the wrong trajectories and then re-train the net model on the new enlarged dataset (Fig. 5.12).

Finally the result for the applied strategy reaches an accuracy of 99.15% (Fig. 5.13). As it is possible to see, thanks to the DAgger technique, the accuracy of the CNN on the predictions has been significantly improved.

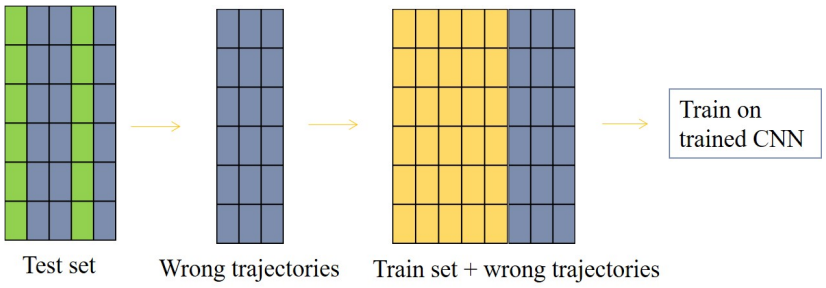


Figure 5.12: Applied DAgger strategy for CNN 1D

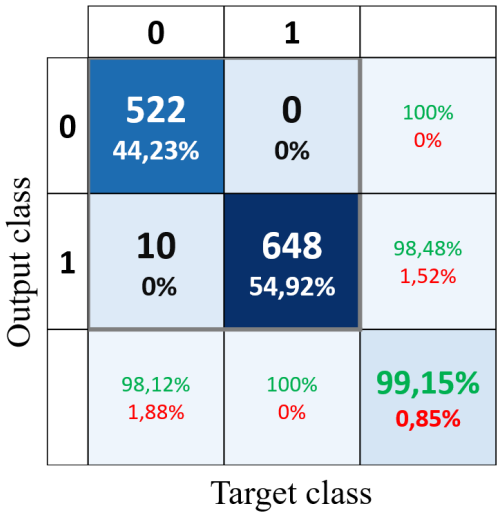


Figure 5.13: Confusion matrix after DAgger approach for CNN 1D

5.6 Performance on a dynamics simulator

Once it has been proven that the CNN is able to achieve good results, it has been interesting to verify if the spacecraft can be controlled only by the well trained neural network, to perform a vertical Moon landing. For this purpose, the dynamics simulator, described in Section 3.6, has been modified and adapted to simulate the behaviour of a lander controlled by a CNN processing raw images. In particular, Moon images instead of the states are the simulator's input. For this reason, the Moon image simulator is embedded in the dynamics simulator, with the aim of transforming initial (and non-initial) states into images. Since the network input is composed of three consecutive images, as described before, three initial consecutive images have to be known to initiate the simulation. The network takes this first input to predict the first control action, with which the dynamics are propagated throughout the duration of the time step, maintaining the control constant. The propagation is performed in the same way as described in Section 3.6. Once the integration is completed, the new generated state is fed

into the image simulator and the new rendered image is aggregated with the last two of the previous input, in order to prepare the new one for the network. A new prediction and a new dynamics propagation follow. The loop is repeated until all the given time interval is covered or until the stopping criteria are satisfied. They force the loop to stop when an altitude of 50 meters (which is the lower limit of the training trajectories) is reached and when the spacecraft starts to rise in altitude. In the following the results of the simulation, performed considering the initial conditions of a trajectory taken from the testing set, are shown. The simulator used to generate the following figures has been developed in *Python*.

In Fig. 5.14 the optimal altitude and the propagated one are shown. In Fig. 5.15 the optimal vertical velocity and the propagated one are compared. The propagated final vertical velocity is equal to -7 m/s, while the optimal one is -0.5 m/s. In Fig. 5.16a the control actions predicted by the CNN and the correct ones are shown. Finally, in Fig. 5.16b it is possible to see what happen when the same CNN is used off-line (instead of on-line), starting from the same initial conditions. It is interesting to notice that, when the network is on-board, some wrong predictions are performed, while there are no errors when the CNN is used off-line to process the same entire fuel-optimal trajectory.

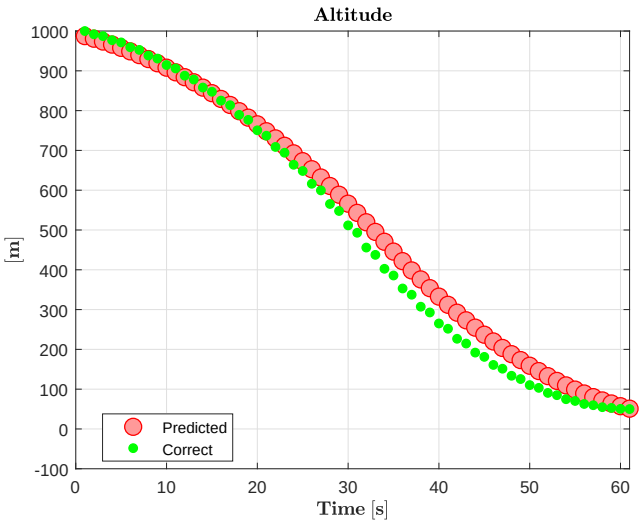


Figure 5.14: Comparison between optimal and predicted altitude

Many simulations like this have shown, also in this case, that the network's performances are very sensitive to the time step extension. A good balance, in which the network is able to give right predictions and the control is quite accurate despite the approximations, must be found. For the example shown, the time step has been set equal to 0.82 s. The error made on the vertical velocity is due to the fact that, as shown in Fig. 5.16, the network makes two wrong predictions when

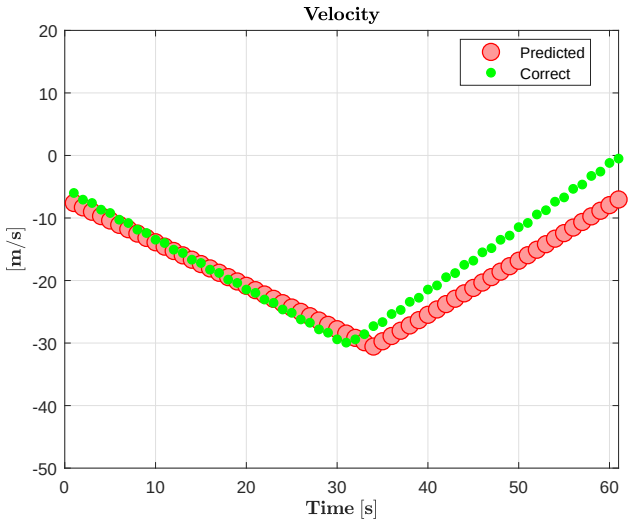
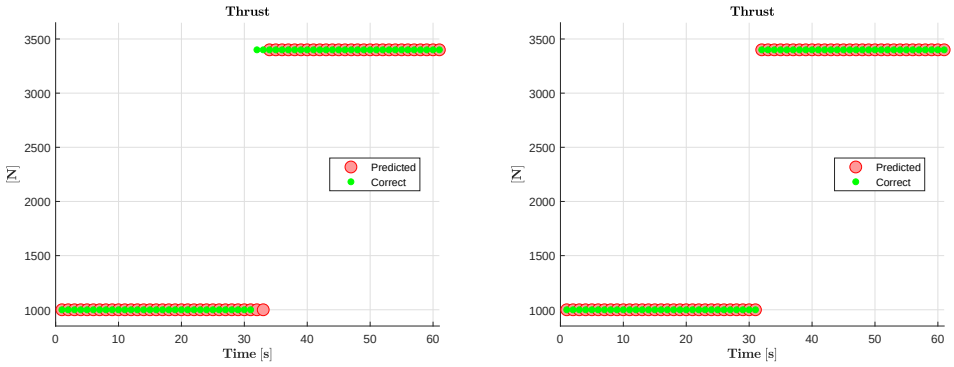


Figure 5.15: Comparison between optimal and predicted vertical velocity



(a) Optimal and predicted thrust on-line **(b)** Optimal and predicted thrust off-line

Figure 5.16: Comparison between on-line and off-line thrust predictions using CNN 1D

it is employed on-line. This is due to the fact that during the on-line simulation, errors on the states are propagated and brought toward the end of the landing. Wrong states (that is wrong images) enter the net which can make some prediction mistakes.

Chapter 6

From image to control: planar Moon landing

The last step of this work has been to perform the planar landing exploiting the surface images of the Moon. This has been reached by melting the RNN−LSTM neural network with the CNN, therefore this chapter is focused on describing the Deep RNN designed for this purpose. The approach is similar to the one explained in Chapter 3 for the vertical landing; in fact, after the training phase of the proposed network, a test has been carried out followed by the DAgger approach.

6.1 Proposed network architecture

The aim of this designed network is to exploit the images made up by the on-board cameras to have a prediction directly on the control action. Dealing with 2D problem, the label associated to each image has two components: one for the thrust magnitude and one for the thrust angle as explained in Section 3.2. The images dataset have been created by using the states belonging to the 2D dataset (Section 3.2). Basically, the trajectories are the same but in the planar problem, instead of the states as input, the net is fed with the corresponding surface images. The architecture designed is shown in Fig. 6.1. The incoming input (package of images) is processed in succession first by a CNN and then by a RNN−LSTM which are linked by a fully connected layer. The processed output of RNN−LSTM goes into two different branches: one which perfom the classification and one aimed for regression as explained in Section 3.3. The network is implemented in *Python-Keras* and it has been composed by:

- Input layer which takes, as mentioned in Section 5.4.1, three consecutive images at a time.

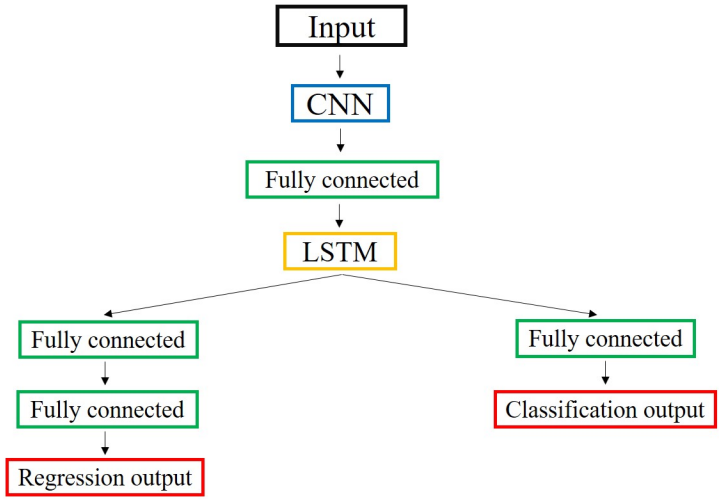


Figure 6.1: Proposed deep RNN for planar landing

- Convolutional neural network (Fig. 6.2). Note that, the hyper-parameters

conv2d_5 (Conv2D)	(None, 2, 32768, 36)	612	main_input[0][0]
conv2d_6 (Conv2D)	(None, 1, 16384, 36)	5220	conv2d_5[0][0]
conv2d_7 (Conv2D)	(None, 1, 8192, 72)	10440	conv2d_6[0][0]
conv2d_8 (Conv2D)	(None, 1, 4096, 72)	20808	conv2d_7[0][0]
conv_flatten (Flatten)	(None, 294912)	0	conv2d_8[0][0]

Figure 6.2: CNN melted in deep RNN network in *Pyhton-Keras*

for the Conv. layers have been tuned differently from the Conv. layers in 1D case detailed in Section 5.3. The convolutional layers have been set equal to 4 instead of 5:

  - First Convolutional layer, [36 *filters*, *size* 4×4, *stride* 2].
  - Second Convolutional layer, [36 *filters*, *size* 2×2, *stride* 2].
  - Third Convolutional layer, [72 *filters*, *size* 2×2, *stride* 2].
  - Fourth Convolutional layer, [72 *filters*, *size* 2×2, *stride* 2].

  The last layer (flatten layer) allows to transform the CNN outputs in a row in order to prepare the input for LSTM cell.

- Fully connected layer which is followed by a reshape layer as shown in line `dense_2` and line `reshape_2` in Fig. 6.3 These two layers have been used to

dense_2 (Dense)	(None, 100)	29491300	conv_flatten[0][0]
reshape_2 (Reshape)	(None, 100, 1)	0	dense_2[0][0]

Figure 6.3: Input reshape layer before LSTM cell in *Pyhton-Keras*

reshape the input as requested for LSTM cell.

- Long-short-term-memory cell is the same as described in Section 3.3.

For the designed net the trainable parameters have been resulted 29.574.483 which means a computationally expensive training phase.

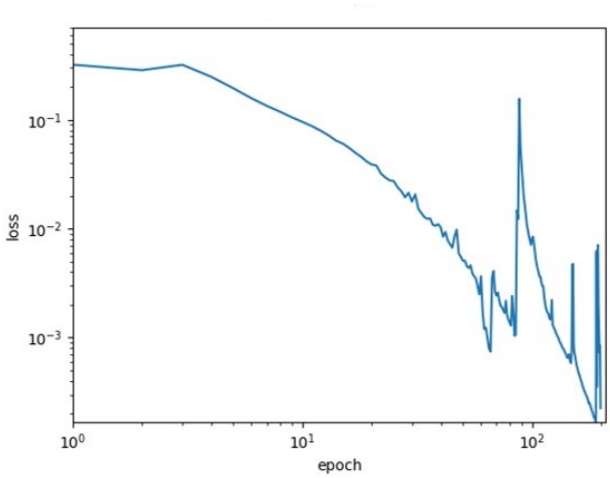
6.2 Training and test phase

6.2.1 Training

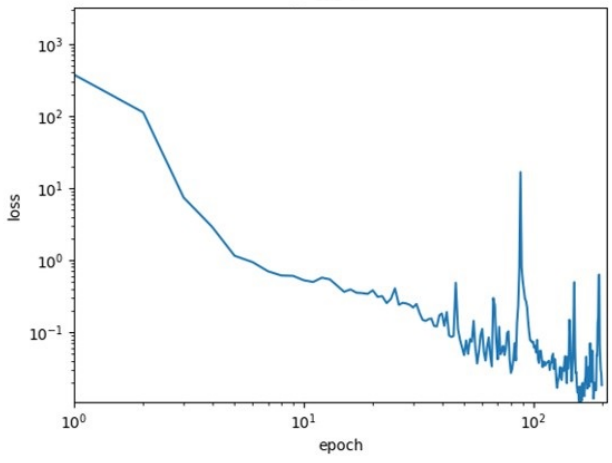
The idea to train the deep RNN has been the same of previous sections; therefore, the inputs have been gathered in packages of 3 images. As already described in Section 5.4.1 this choice improves the net response because allows to keep track of velocities during landing. The batch size has been set to 59 also for this network, in order to fed the input layer with one trajectory at a time. The original images (with a dimension 256×256) have been normalized between 0 and 1 and reshaped such that the input pack is a matrix 3×65.536 [number of images $\times$ pixels per image]. This because each image is transformed in a row. The associated label (which consists of the thrust magnitude and the thrust angle) is the one corresponding to last image of each input matrix (see Section 5.4.1). Since the training phase is computationally demanding, the epochs have been set to only 200 and using only 80 trajectories (4880 images). Given that the number of trajectories for train is lower, no validation split has been done on dataset. The simulation have been performed thanks to HPC (High Performance Computing) systems of University of Arizona, like Extremely LarGe Advanced TechnOlogy (El Gato), which uses specially designed hardware to achieve high performance economically, including NVIDIA K20X GPUs and Intel Xeon Phi 5110p Coprocessors (for more details see [17]). The loss functions are the same used for previous simulation (see Section 5.4.1), cross entropy for classification purpose and RMSE for regression purpose (Tab. 6.1). The results of training phase are shown in Fig. 6.4. As it is possible to note from these graphs, the training phase in noisy enough to affirm that no convergence has been achieved. This is due to the complex problem; in fact, the input image is transformed in a row of normalized numbers that pass through all neurons so many times increasing the required computations. Therefore, a better tuning of hyper-parameters is required to mitigate this phenomenon and achieve better results. This could lead to an increase of number of training epochs, a lower initial learning rate or a more precise choice of loss weights. Moreover, while in the regression a final flat trend can be highlighted, the classification is very far from convergence. Furthermore in

Table 6.1: Resuming of hyper-parameters of Deep RNN

Hyper-parameters	
Batch size	59
Initial learning rate	0.001
Decay rate	0.0001
Regression loss weight	10
Classification loss weight	60



(a) Classification loss for deep RNN during training



(b) Regression loss for deep RNN during training

Figure 6.4: Classification and regression losses evolution during training phase for deep RNN

the graphs (Fig. 6.4) some clear peaks are present and they are due to the change in the learning rate during the training phase. Also these peaks underline the

considerations just made.

6.2.2 Test

According to the threshold chosen to evaluate the reached precision response and justified in Section 3.4.2, the results are considered good. Even if the model should be better trained, the results are very promising. A test have been developed over 30 trajectories and the related confusion matrix is shown in Fig. 6.5 (remember that in the confusion matrix 0 is associated to minimum thrust, 1 to maximum). The accuracy on the thrust classification reaches the $98,51\%$ and the RMSE $= 0.76°$ (Fig. 6.6). Unlike the results obtained for the RNN−LSTM, in Fig. 6.6 there are some points that do not lie on the regression line.

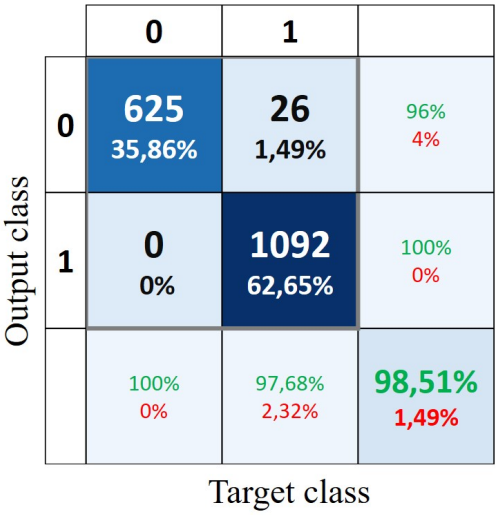


Figure 6.5: Confusion matrix deep RNN

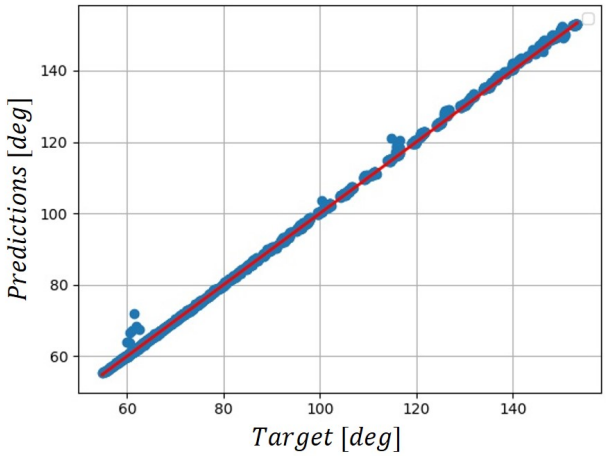


Figure 6.6: Regression with trained deep RNN

6.3 Accuracy improvement: DAgger approach

The DAgger on Deep RNN has highlighted the need of a better tuning of parameters. In fact, according to the same criteria seen before in Tab. 3.3 applied for RNN−LSTM, the wrong predicted trajectories have been 30 over the 30 belonging to the test set (100% of the set). Performing a DAgger some conclusions can be underlined. Looking at Fig. 6.7, it is possible to note that even if the accuracy clearly increase and only $0,57\%$ of the total predictions are wrong, the RMSE has a worse value. It passes from the $0.76°$ (using the trained model) to $1.44°$ after DAgger. This means again that the loss weights have not been properly chosen or the epochs have not been enough to allow a complete training phase. A longer training phase should lead to a convergence in updating the weights

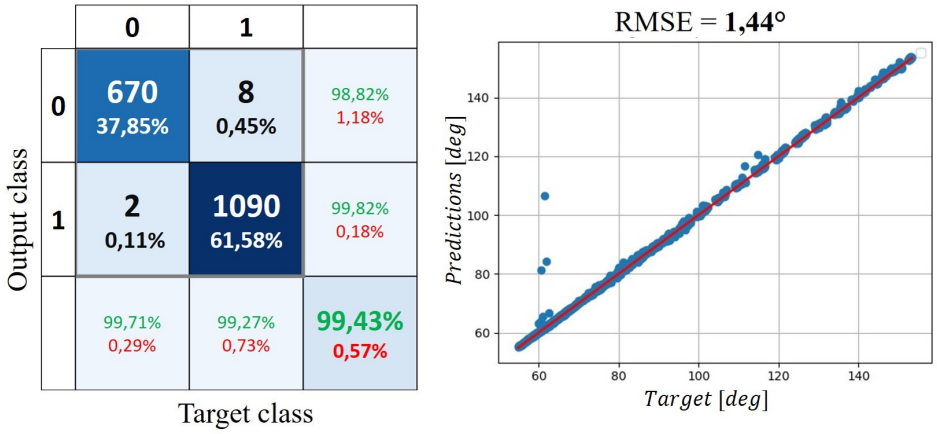


Figure 6.7: Results after DAgger approach applied to Deep RNN

and biases but the training time could result too high since for 200 epochs and employing the HPC systems, the simulation procedure has taken more than 20 hours. Many conclusions can be drawn from the proposed Deep RNN: given the power of the tools used and proved in developing the CNN for the 1D problem and the RNN−LSTM for the 2D problem, a better response in using Depp RNN for a 2D landing was expected. The inaccurate results, particularly on regression, bring out the need for more robust training phase. This means better use of HPC system and, specifically on network training, an appropriate tuning of the parameters (learning rate, decay rate, loss weights, number of epochs).

Despite this, the output is considered valid because the error on angle value is limited to $1.44°$ after DAgger and the error on thrust magnitude is $0,57\%$. This shows that it is possible and convenient to exploit a neural network also with optical navigation for proximity operations. The network in fact, is operatively speaking an advantage because the forecast on the control action is provided in a short time.

Chapter 7

Conclusions and future works

7.1 Conclusions

This thesis has highlighted advantages and limits for machine learning applied to a Moon landing. Surely the great benchmark is the possibility to exploit the Artificial Intelligence dealing with space framework and in particular with the help of images arriving from the on-board instruments. What has been discovered is that deep neural networks can achieve high accuracy in space guidance problems, using full states knowledge or other sensors, like on-board cameras. In the following table a wrap-up of the most important numerical results is shown.

Table 7.1: Results resuming

Networks	Classification accuracy	Regression error
$CNN\ 1D$	99.15%	
$RNN - LSTM$	99.25%	0.4°
$Deep\ RNN$	99.43%	1.44°

Considering the 2D planar landing, the Recurrent neural network described in Chapter 3, is able to determine the fuel-optimal control actions to apply, by processing three consecutive states of the system. An accuracy of 99.43% on the classification and a regression error of 0.4° are achieved during the test phases. It has been shown how Recurrent networks are able to exploit information computed in the past to improve the next prediction. This particular feature has demonstrated to be very useful dealing with a guidance problem, in which long sequences of data are considered and processed. Dealing with a vertical Moon landing, the convolutional neural network illustrated in Chapter 4 has been trained to process three consecutive Moon surfaces images to understand the

thrust profile. Even if the problem is simpler from a dynamical point of view, the obtained results (accuracy of 99.15%) are relavant considering the fact that only raw images are taken as input and therefore no state estimation is necessary. Finally, the Deep Recurrent neural network shown in Chapter 6, has been designed linking the previous two with the aim to solve a problem more similar to reality. The information extracted from the Moon surface raw images are exploited during a 2D landing to control the lander, solving a classification (on thrust magnitude) and a regression (on thrust angle) problem simultaneously. An accuracy of 99.43% and an error of 1.44° on the thrust angle prediction have been achieved. This first results show that deep neural networks can be employed to solve high-fidelity space guidance and navigation problems.

It has been proven that the DAgger approach can enhance the predictions accuracy even if there isn't any human action/correction, which is the classical procedure followed up to now for terrestrial experiments. This represents an important evidence that imitation learning can play a significant role in the future artificial intelligence applications.

7.2 Future work

Despite the results and the applied techniques are very promising, some conclusions must be underlined.

As it has been demonstrated, image processing nets (CNN) need a long time for training phase and according to the accuracy and precision requested by a delicate environment such as the space, it cannot be satisfied with a few hundred of epochs. For this reasons, a future development of this work could be focused on a more accurate use of GPUs and HPC systems in order to perform a better training phase.

Another research should be made in studying the assistance of more than one cameras and using also sensors measurements. Particularly during the last few steps of a landing phase, images alone could be not sufficient for an optimal prediction on the control actions. This is the principal reason whereby the constraint of 50 meters has been imposed. Embedding the information coming from other kind of sensors (i.e. optical and inertial) with those coming from the images, could lead to a performances improvement.

A remark has already been underlined describing the dynamics simulator for both 1D and 2D case. The limits of the generated dataset and of the applied approach have emerged from these on-line simulations. All the networks developed in this thesis have been trained with fuel-optimal trajectories created with GPOPS,

in which each optimal control action is associated to a state. In an on-line simulation (i.e. like the dynamics simulations), the states coming from the integration of the equations of motion could be affected by errors, due to integration inaccuracy. Since the network inputs are conditioned on the states, the errors are then propagated along the predicted trajectories. This problem is evident looking at the result of the Monte Carlo simulation in Section 3.7. As mentioned, a training phase performed with denser trajectories of points may partially solve the problem, making the networks more robust against the propagated errors. In this way, propagated errors would be less effective on the on-line predicted trajectory. On the other hand, these conclusions open the possibility of implementing a Reinforcement Learning algorithm instead of a Supervised one, because the need of a dataset is overcame. In this framework, the network interacts with an environment (i.e. the dynamics of the system) to learn a policy, with which it is possible to apply an optimized control action, at each time step. An interesting method is represented by the Guided Policy Search algorithm presented by Sergey Levine and Vladlen Koltun in [16]. In this paper it is shown how trajectory optimization can direct and guide the policy search away from poor local optima. This different approach would make the deep neural networks more powerful and able to overcome the limit deriving from the possible inaccuracy of the training set.

Bibliography

[1] AR Klumpp. Apollo guidance, navigation and control: Apollo lunar descent guidance. massachusetts institute of technology, charles stark draper lab. Technical report, TR-R695, Cambridge, MA, 1971.

[2] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems.* " O'Reilly Media, Inc.", 2017.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[4] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[5] Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard L Lewis, and Xiaoshi Wang. Deep learning for real-time atari game play using offline monte-carlo tree search planning. In *Advances in neural information processing systems*, pages 3338–3346, 2014.

[6] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373, 2016.

[7] Tianhao Zhang, Gregory Kahn, Sergey Levine, and Pieter Abbeel. Learning deep control policies for autonomous aerial vehicles with mpc-guided policy search. In *Robotics and Automation (ICRA), 2016 IEEE International Conference on*, pages 528–535. IEEE, 2016.

[8] Carlos Sánchez-Sánchez and Dario Izzo. Real-time optimal control via deep neural networks: study on landing problems. *Journal of Guidance, Control, and Dynamics*, 41(5):1122–1135, 2018.

[9] Ivan Nunes Da Silva, Danilo Hernane Spatti, Rogerio Andrade Flauzino, Luisa Helena Bartocci Liboni, and Silas Franco dos Reis Alves. *Artificial Neural Networks*. Springer, 2017.

[10] Anil K Jain, Jianchang Mao, and K Moidin Mohiuddin. Artificial neural networks: A tutorial. *Computer*, 29(3):31–44, 1996.

[11] Roberto Furfaro, Jules Simo, Brian Gaudet, and Daniel Wibben. Neural-based trajectory shaping approach for terminal planetary pinpoint guidance. In *AAS/AIAA Astrodynamics Specialist Conference 2013*, pages Paper–AAS, 2013.

[12] Roberto Furfaro, Dario Cersosimo, and Daniel R Wibben. Asteroid precision landing via multiple sliding surfaces guidance techniques. *Journal of Guidance, Control, and Dynamics*, 36(4):1075–1092, 2013.

[13] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[14] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635, 2011.

[15] Alain Fournier, Don Fussell, and Loren Carpenter. Computer rendering of stochastic models. *Communications of the ACM*, 25(6):371–384, 1982.

[16] Sergey Levine and Vladlen Koltun. Guided policy search. In *International Conference on Machine Learning*, pages 1–9, 2013.

[17] Shai Segal, Avishy Carmi, and Pini Gurfil. Vision-based relative state estimation of non-cooperative spacecraft under modeling uncertainty. In *Aerospace Conference, 2011 IEEE*, pages 1–8. IEEE, 2011.

[18] Son-Goo Kim, John L Crassidis, Yang Cheng, Adam M Fosbury, and John L Junkins. Kalman filtering for relative spacecraft attitude and position estimation. *Journal of Guidance, Control, and Dynamics*, 30(1):133–143, 2007.

[19] Daniel R Wibben and Roberto Furfaro. Optimal sliding guidance algorithm for mars powered descent phase. *Advances in Space Research*, 57(4):948–961, 2016.

[20] Igor Mordatch, Kendall Lowrey, Galen Andrew, Zoran Popovic, and Emanuel V Todorov. Interactive control of diverse complex characters with neural networks. In *Advances in Neural Information Processing Systems*, pages 3132–3140, 2015.

Consulted Web sites

[1] Python. https://www.python.org/.

[2] TensorFlow. https://www.tensorflow.org/.

[3] MNIST on TensorFlow. https://www.tensorflow.org/versions/r1.0/get_started/mnist/beginners.

[4] Deep Learning on MathWorks. https://www.mathworks.com/help/nnet/deep-learning-training-from-scratch.html.

[5] Stanford University course. http://cs231n.stanford.edu/.

[6] Convolutional Networks at Stanford University. http://cs231n.github.io/convolutional-networks/.

[7] RNN at Stanford University. http://cs231n.stanford.edu/slides/2018/cs231n_2018_lecture10.pdf.

[8] Recurrent Networks in Python. https://pythonprogramming.net/recurrent-neural-network-rnn-lstm-machine-learning-tutorial/.

[9] Convolutional Networks in Python. https://pythonprogramming.net/convolutional-neural-network-cnn-machine-learning-tutorial/.

[10] TensorFlow Keras. https://keras.io/.

[11] Deep Learning at Berkeley (Sergey levine). http://rail.eecs.berkeley.edu/deeprlcourse/.

[12] Persistance of Vision Raytracer. http://www.povray.org/.

[13] Lunar Reconnaissance Orbiter Camera. http://lroc.sese.asu.edu/.

[14] Diamond - Square algorithm in Python. http://jmecom.github.io/blog/2015/diamond-square/.

[15] Diamond - Square algorithm in Matlab. https://www.mathworks.com/matlabcentral/fileexchange/?utf8=%E2%9C%93&term=diamond+square.

[16] MNIST accuracy. http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html#4d4e495354.

[17] El Gato Super Computer . http://elgato.arizona.edu/system-configuration.