CRANFIELD UNIVERSITY

THOMAS RETRAINT

PHYSICS-INFORMED DEEP LEARNING FOR AEROSPACE SYSTEMS

SCHOOL OF AEROSPACE, TRANSPORT AND MANUFACTURING
Applied Artificial Intelligence

MSc
Academic Year: 2020 - 2021

Supervisor: Professor Gokhan Inalhan
Associate Supervisor: Professor Antonios Tsourdos
August 2021

CRANFIELD UNIVERSITY

SCHOOL OF AEROSPACE, TRANSPORT AND MANUFACTURING
Applied Artificial Intelligence

MSc

Applied Artificial Intelligence

Academic Year 2020 - 2021

THOMAS RETRAINT

Physics-informed Deep Learning for Aerospace Systems

Supervisor: Professor Gokhan Inalhan
Associate Supervisor: Professor Antonios Tsourdos
August 2021

This thesis is submitted in partial fulfilment of the requirements for the degree of MSc.

ABSTRACT

Embedded Artificial Intelligence (AI) models are now widely deployed in all types of transportation means. In this thesis, an approach has been proposed to automate and optimise the braking phase of aircraft landings using the latest Machine Learning (ML) technology, namely Deep Reinforcement Learning (DRL). For this purpose, a dynamic aircraft landing simulator has been developed from scratch, which can reproduce landing conditions according to different weathers and runways. This simulator is the basis of a DRL Agent environment based on the Proximal Policy Optimisation (PPO) algorithm being able to take the best braking decisions according to the mechanical status of the aircraft to ensure an optimised landing. The set of actions available to the agent consists of the four modes of the Auto-brake system that is already implemented in a large number of aircraft and widely used by pilots. The agent achieves a landing success rate of more than 95% in several flight conditions, and thus proves that this method is promising for this kind of application.

Keywords:

Machine Learning, Deep Reinforcement Learning, Embedded Artificial Intelligence Model, Aircraft Landings, Proximal Policy Optimization, Braking Systems, Physics Simulations

ACKNOWLEDGEMENTS

First of all, I would like to thank my supervisor Prof. Gokhan Inalhan for his support and helpful guidance throughout the thesis.

I would also like to thank the Airbus team with whom I had very interesting discussions and gave relevant feedbacks on my work during the last few months.

I would also like to give a special thanks to my Group Project classmates as well as to my housemates without whom this year would not have been the same.

Finally, all this would not have been possible without the help and love of all my family, my father, my mother and my girlfriend, Arooshi, whom I would really like to thank.

TABLE OF CONTENTS

ABSTRACT i
ACKNOWLEDGEMENTS ii
LIST OF FIGURES v
LIST OF TABLES viii
LIST OF EQUATIONS ix
LIST OF ABBREVIATIONS xi
1 INTRODUCTION 12
1.1 Introduction to Deep Learning 13
1.1.1 Basic Neural Network Structure 13
1.1.2 Hidden Layer 14
1.1.3 Activation Function 15
1.1.4 NN training 16
1.2 Overfitting in ML 17
1.3 Problem Statement & Objectives 18
1.4 Structure of the report 18
2 LITERATURE REVIEW 19
2.1 Landing systems of aircrafts 19
2.1.1 Landing phases 19
2.1.2 Braking system of an airplane 22
2.2 Methods to counter overfitting 25
2.2.1 Constraining Model Complexity 25
2.2.2 Regularization Methods 26
2.3 Innovative alternatives 28
3 METHODOLOGY 30
3.1 Introduction to Reinforcement Learning 30
3.1.1 Functional Principles of RL 30
3.1.2 Elements of a RL model 30
3.2 Landing Simulator 32
3.2.1 Conception of the simulator 32
3.2.2 Implementation of the Simulator 36
3.2.3 Outputs of the simulator 38
3.3 Implementation of RL model 41
3.3.1 TensorForce Implementation 42
3.3.2 Agent Implementation 46
4 RESULTS & DISCUSSION 52
4.1 Hyperparameters tuning 52
4.1.1 Non-Network Parameters 53
4.1.2 Network Parameters 58
4.1.3 Others Parameters 59
4.2 Policy of the Model 61

4.2.1 Perfect Condition Agent 61
4.2.2 Rainy Condition Model 66
4.3 Discussion 69
5 CONCLUSION 72
6 FUTURE WORK 74
REFERENCES 77
APPENDICES 80

LIST OF FIGURES

Figure 1.1-A Structure of a neuron 13

Figure 1.1-B Fully Connected NN 14

Figure 1.1-C ReLU Function 15

Figure 1.2-A Example of (a) underfitting, (b) good fit and (c) overfitting [5] 17

Figure 2.1-A Phases of the flight of an airplane [6] 19

Figure 2.1-B Illustration of an ILS [7] 20

Figure 2.1-C Landing distances of an airplane [8] 21

Figure 2.1-D Spoiler on an A320 [9] 22

Figure 2.1-E Illustration of wheel Brakes [10] 23

Figure 2.1-F Illustration of thrust-reverser [12] 24

Figure 2.2-A Illustration of early stopping 27

Figure 3.1-A Elements of a RL model 31

Figure 3.2-A Forces applied to the aircraft 33

Figure 3.2-B Declaration of the different constants 37

Figure 3.2-C Landing trajectory of the landing simulator, perfect conditions and Auto-Brake = High 38

Figure 3.2-D Grounded trajectory of the landing simulator, perfect conditions and Auto-Brake = High 38

Figure 3.2-E Landing trajectory of the landing simulator, rainy conditions and Auto-Brake = High 39

Figure 3.2-F Grounded trajectory of the landing simulator, rainy conditions and Auto-Brake = High 39

Figure 3.2-G Landing trajectory of an A320 plane for the different Auto-brake modes 40

Figure 3.2-H Grounded trajectory of an A320 plane for the different Auto-brake modes 40

Figure 3.3-A RL model with the Landing simulator 41

Figure 3.3-B Implementation of the state's function 42

Figure 3.3-C Implementation of the action function 43

Figure 3.3-D Implementation of the terminal function of an episode 43

Figure 3.3-E Implementation of the execute function 45
Figure 3.3-F Implementation of the reset function. 45
Figure 3.3-G Implementation of the elements in TensoForce 46
Figure 3.3-H Operating principal of the MM algorithm [28] 47
Figure 3.3-I Exemple of KL Divergence on 2 Gaussian distributions [29] 47
Figure 3.3-J Shape of the lower limit $\boldsymbol{M}$ [30] 48
Figure 3.3-K Illustration of the clipped method [27] 50
Figure 3.3-L Example of the implementation of a PPO agent 51
Figure 4.1-A Results of the Subsampling Fraction parameter 54
Figure 4.1-B Results of the Likelihood Ratio Clipping parameter 55
Figure 4.1-C Results of the L2 Regularization parameter 56
Figure 4.1-D Results of the Entropy Regularization parameter 57
Figure 4.1-E Results of the Network Size parameter 58
Figure 4.1-F Results of the Network Depth parameter 58
Figure 4.1-G Results of the Discount parameter 59
Figure 4.1-H Results of the Exploration parameter 60
Figure 4.2-A Reward against the number of batches for the optimized model . 61
Figure 4.2-B Policies after 1000 & 2000 episodes of training (Perfect Conditions) 62
Figure 4.2-C Policies after 3000 & 4000 episodes of training (Perfect Conditions) 63
Figure 4.2-D Policies after 5000 & 6000 episodes of training (Perfect Conditions) 63
Figure 4.2-E Policies after 7000 & 8000 episodes of training (Perfect Conditions) 64
Figure 4.2-F Policies after 9000 & 10000 episodes of training (Perfect Conditions) 64
Figure 4.2-G Policies after 1000 & 2000 episodes of training (Rainy Conditions) 66
Figure 4.2-H Policies after 3000 & 4000 episodes of training (Rainy Conditions) 67
Figure 4.2-I Policies after 5000 & 6000 episodes of training (Rainy Conditions) 67

Figure 4.2-J Policies after 7000 & 8000 episodes of training (Rainy Conditions) 68
Figure 4.2-K Policies after 9000 & 10000 episodes of training (Rainy Conditions) 68

LIST OF TABLES

Table 2.1-A Auto-brake Modes correlated to the braking values 24

Table 4.1-A Non-Network Parameter Selection 57

Table 4.2-A Success Rate of the Perfect Condition Agent 65

Table 4.2-B Success Rate of the Rainy Condition Agent 69

Table 4.3-A Success Rate of the Perfect Conditions Agent on Rainy Conditions 70

LIST OF EQUATIONS

(1-1)..... 15
(1-2)..... 15
(1-3)..... 16
(1-4)..... 17
(1-5)..... 17
(2-1)..... 26
(2-2)..... 26
(3-1)..... 32
(3-2)..... 32
(3-3)..... 34
(3-4)..... 34
(3-5)..... 34
(3-6)..... 34
(3-7)..... 34
(3-8)..... 35
(3-9)..... 35
(3-10)..... 35
(3-11)..... 36
(3-12)..... 36
(3-13)..... 42
(3-14)..... 44
(3-15)..... 47
(3-16)..... 48
(3-17)..... 49
(3-18)..... 49
(3-19)..... 50
(3-20)..... 50
(3-21)..... 50

(4-1)..... 52
(4-2)..... 52

LIST OF ABBREVIATIONS

PBDL	Physics-Based Deep Learning
DL	Deep Learning
ML	Machine Learning
NLP	Natural Language Processing
NN	Neural Network
FC	Fully Connected
ReLU	Rectified Linear Activation Function
ILS	Instrument Landing System
FAF	Final Approach Fix
LDA	Landing Distance Available
LDR	Landing Distance Required
PCA	Principal Component Analysis
AI	Artificial Intelligence
RL	Reinforcement Learning
PPO	Proximal Policy Optimization
MM	Minorize-Maximization
KL	Kullback-Leibler
TRPO	Trust Region Policy Optimization
DRL	Deep Reinforcement Learning

1 INTRODUCTION

The main topic of this thesis is the development of physics-based Deep Learning models. This field exploits the power of Deep Learning and neural networks to solve complex physics-based problems. The field of Physics-Based Deep Learning (PBDL) is growing rapidly and is very active in view of the latest published research.

Within this discipline, it is possible to differentiate between the approaches to the problems encountered: a) to focus on perfecting the model design, or b) combine all sorts of new methods available. More specifically, the different approaches can be categorised into two parts: *forward* simulation, which aims to predict the states or evolutions of a physical system [1], and *inverse* simulation, which seeks to obtain the fundamental parameters and variables of the physical system from the observations that are made [2]. To solve the forward and inverse problems, we can identify different integration methods which can be categorised into:

- **Data-driven:** the data is extracted and produced by the physical system. This system can be real or simulated.
- **Loss-driven**: the physical mechanisms are programmed into a *loss function* in the form of different operations. Learning is then done by evaluating this *loss function* and often uses gradient descent to be optimised.
- **Interleaved:** the simulations are combined with the output of a Machine Learning (ML) model which is very often a neural network. This usually requires a completely separate simulator and therefore allows a link between the physical system and the learning model itself. This technique is very often used in time-dependent physical systems where these models can take full advantage of their predictive power on the future behaviour of the system mechanics.

1.1 Introduction to Deep Learning

1.1.1 Basic Neural Network Structure

Deep Learning (DL) is the ML technique that has been the most talked about in the last decade because of its performance in certain applications that produced unexpected results. This technique is used in image processing, for example, or in Natural Language Processing (NLP) which aims at the analysis of text by the computer. DL models are based on the so-called neural networks (NN) which are a set of layers of neurons. A neuron is a block consisting of parameters, weights and bias terms that are updated as the model trained.

Let's take an input data $x \in \mathbb{R}^n$, the output of a neuron is in the form of $\Psi(Wx + b)$, where Ψ is a non-linear function, $W \in \mathbb{R}^{n*m}$ is the linear operation and $b \in \mathbb{R}^m$ is the bias [3]. Below is the structure of a neuron :

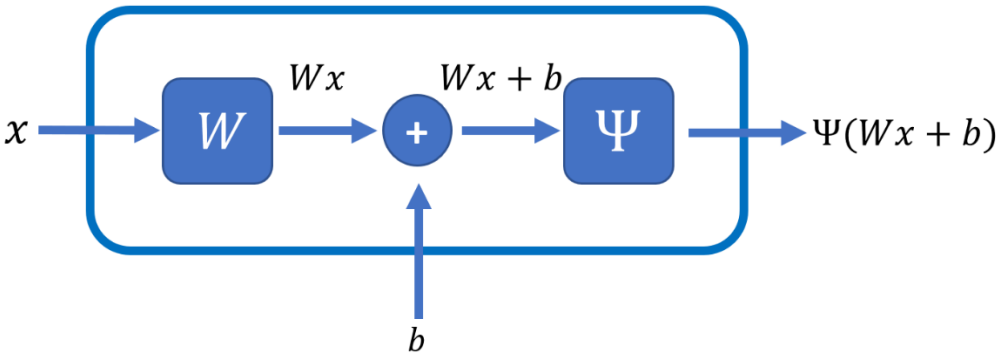


Figure 1.1-A Structure of a neuron

The function of a single neuron is not complex, but that does not guarantee the comprehension of complex physical phenomena. Physical behaviours can only be appreciated if different neurons are put together. Thus, it is required to create different types of neural network architectures. This arrangement of neurons is called hidden layers.

1.1.2 Hidden Layer

The neurons of a NN are arranged in layers comprising of a defined number of neurons. Depending on the type of task to be performed, the architecture of each layer is modified. The most common and simple layer is the Fully Connected (FC) layer. In this layer each neuron is directly connected to the other neurons of the next layer, as depicted in the figure below:

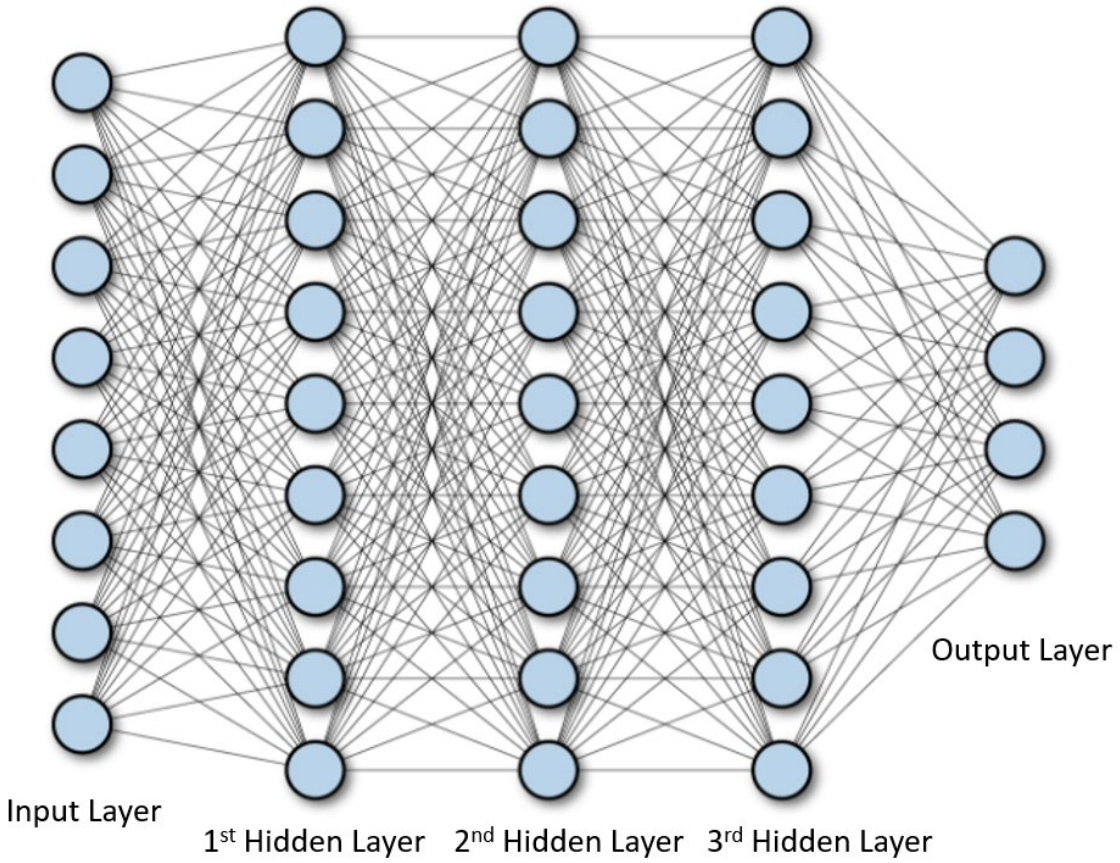


Figure 1.1-B Fully Connected NN

In this case the NN has 3 hidden layers of 9 neurons which are all connected to each other. This type of architecture makes it possible to propagate the result of a neuron and the information to all the others of the following layer. A coefficient varying between 0 and 1 is assigned to each link. This coefficient is called weight and represents the importance of the result of a neuron in relation to another in the network.

1.1.3 Activation Function

Until now, the operations performed are simply linear combinations of coefficients found within each neuron. However, to understand the functioning of complex systems, non-linear features are essential. To tackle this, the layers of neurons apply a function to their output, which is called the activation function. In addition to introducing nonlinearity to the model, activation functions allow, for some, to normalise the range of neuron outputs which increases the efficiency of a model. The following are examples of the most common activation functions:

$$ReLU(x) = \begin{cases} 0, & \text{for } x < 0 \\ x, & \text{for } x \geq 0 \end{cases} \quad \textbf{(1-1)}$$

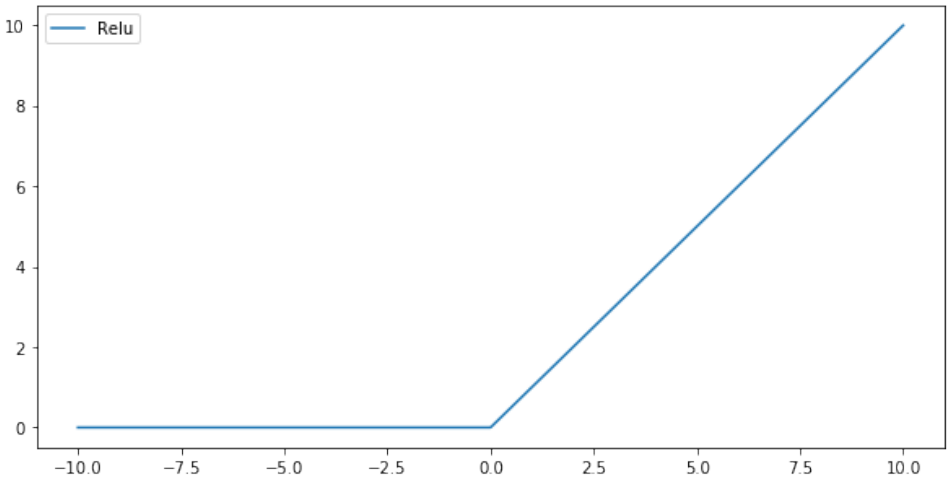


Figure 1.1-C ReLU Function

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \quad \textbf{(1-2)}$$

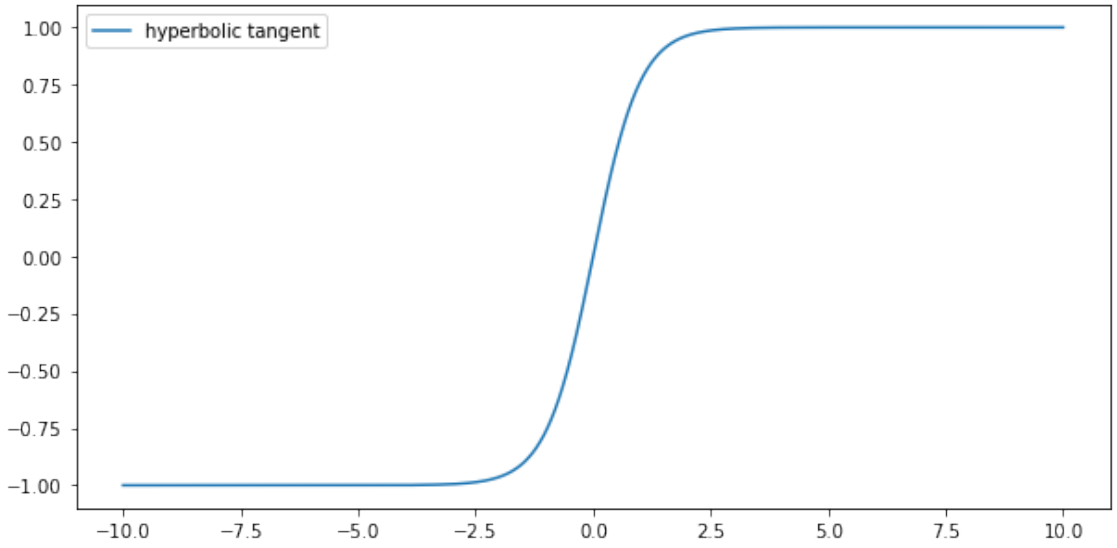


Figure 1.1-D Hyperbolic Tangent Function

1.1.4 NN training

In order for a DL model to train, it is necessary to evaluate its performance. To do this, the models use a *loss function* noted $\mathcal{L}$. Depending on the task to be solved with the NN, different types of functions are possible. For a classification task, one of the most used functions is the *cross-entropy function*:

$$\mathcal{L}_{CE} = -\sum (y_i \log(p_i) + (1 - y_i)\log(1 - p_i)) \quad \textbf{(1-3)}$$

With y_i, the ground-truth label correlated to the input and p_i represents the predicted probability of the respective class as the output of the model.

The model is trained by a method called *backpropagation*. This aims to minimise the value of the *loss function* during the learning iterations. The process is initiated with the data input x across the entire network followed by the computation of the loss function value $\mathcal{L}(x, W)$, where W represents the parameters of the model (weights, bias...). The process is then continued by calculating $\frac{\partial \mathcal{L}}{\partial W}$. In the end, the algorithm updates from the gradient of each layer recursively, from the last to the input layer.

Let's consider the output of the layer l as $z^{(l)}$. It is possible to calculate the gradient of the previous layer $z^{(l-1)}$ which is the input of the layer l . Below are the formulas associated with the layer parameters $W^{(l)}$:

$$\frac{\partial \mathcal{L}}{\partial z^{(l-1)}} = \frac{\partial \mathcal{L}}{\partial z^{(l)}} * \frac{\partial z^{(l)}(W^{(l)}, z^{(l-1)})}{\partial z^{(l-1)}} \quad \textbf{(1-4)}$$

And the gradient with respect to the parameters:

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \frac{\partial \mathcal{L}}{\partial z^{(l)}} * \frac{\partial z^{(l)}(W^{(l)}, z^{(l-1)})}{\partial W^{(l)}} \quad \textbf{(1-5)}$$

These two formulas therefore dictate the calculation of the gradient for each parameter for each layer of the NN. Finally, the optimisation can be done with gradient-based optimizer [4].

1.2 Overfitting in ML

Overfitting is one of the many phenomena that any scientist or statistician seek to avoid in a predictive model. In statistics, overfitting is the act of creating a model that is unique to a data set. As a consequence, the results observed afterwards turn out to be very unreliable. Here is an example of overfitting in the context of a polynomial approximation. See figure below:

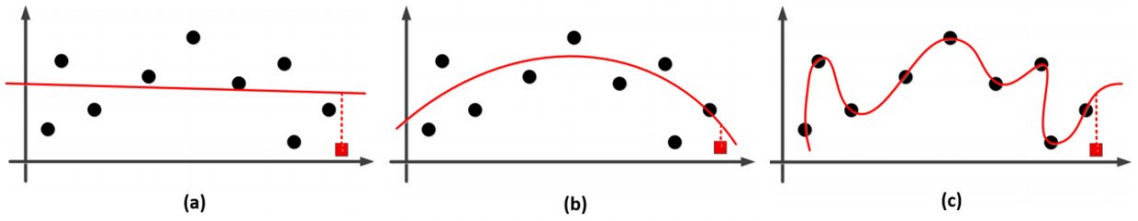


Figure 1.2-A Example of (a) underfitting, (b) good fit and (c) overfitting [5]

The black dots represent the training set and the red square is the test instance. The red curve represents the interpretation of the training set model. One can notice the results from the representation above (a) case of underfitting which occurs when the model is too simple, and (c) case of overfitting, in which the evaluation of the test point by the models is inaccurate. Indeed, the distance of the red square from the curves is greater than in the case of good fit (b). Graph

(c) also demonstrates that the model passes exactly through all the points showing a total adaptation of the model to the data set. Thus, when faced with new data, the model cannot extrapolate and the output is much worse than what the training set would have predicted.

1.3 Problem Statement & Objectives

Overfitting is a generic issue that has continued to persist since the dawn of AI in different transportations, specifically on-board aircrafts AI models, for the purpose of this thesis. Traditional methods, as elaborated previously above, and to be discussed further, have fallen short of tackling the various aspects that lead to overfitting e.g., increased number of parameters, overtraining of the initial data set leading to good performance on training data set and poor performance on unknown data, prevention of generalization etc.

Thus, the objective is to is to provide concrete answers to the problem faced by the engineering teams by proposing one or more innovative methods to optimize the landing phase of an airplane by exerting minimal efforts in the braking phase with maximum efficacy. This process maximizes the life expectancy of the brakes and tyres, as well as the comfort of the passengers.

This thesis is in collaboration with Airbus. Due to security reasons and sensitivity of the information, the company was not in a position to share resources such as data or models already developed.

1.4 Structure of the report

This thesis report is structured as follows: Chapter 1, a quick introduction to the basic concepts in order to set the framework and introduce the problem and its objectives; Chapter 2, a literature review to deepen the field and to evaluate the different applicable methods; Chapter 3, the implementation of this work; Chapter 4, the results of this implementation as well as an interpretation and discussion on the use of the method; Chapter 5, draws conclusions from this work; Chapter 6, proposes potential new ideas to be implemented in order to enlarge and fulfilled the implementation.

2 LITERATURE REVIEW

To solve the on-board aircraft AI model overfitting issue, it is necessary to understand the exercises being tried to solve with their DL model. To do this, it is essential to learn about aircraft landing, their sequence, and their mechanics that govern these events. Therefore, a review of this literature is necessary.

Subsequently, a more detailed understanding of overfitting within ML models is also required. For this purpose, an overview of techniques to counteract overfitting is developed in the second step.

2.1 Landing systems of aircrafts

2.1.1 Landing phases

The flight of an aircraft from start to finish takes place in different stages. See figure below:

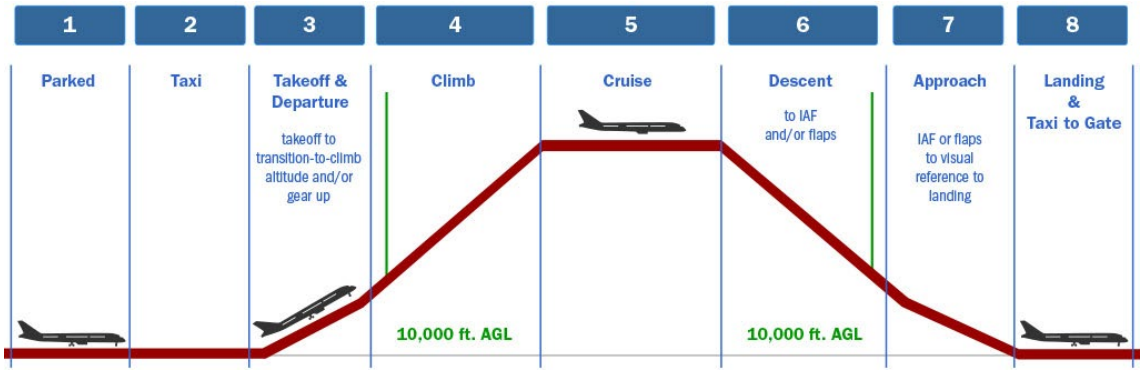


Figure 2.1-A Phases of the flight of an airplane [6]

For the purpose of landing only, the last 3 phases have each a distinct role to play.

2.1.1.1 Descent

The descent in an airplane flight represents the portion where the aircraft drops in altitude. The aircraft in its cruising phase usually reaches an altitude of 10,000m and remains between 300 and 150m altitude in the end. The descent usually takes about 20 minutes. Normally, the descent is supposed to be at a constant airspeed with a constant aircraft angle. However, for various reasons

such as air traffic or weather disturbances the pilot can decide to accelerate or brake the descent depending on the conditions faced. To do this, he can vary the angle of the aircraft by adjusting the thrust of the engines.

In the beginning and during this phase, the passengers hear the sound of the engines gradually decreasing since the aircraft gains speed by descending due to gravity [7]. In the end of the phase, passengers do feel slight accelerations and an increase in noise in preparation for the last phase before touchdown: the final approach.

2.1.1.2 Final Approach

In the final approach the pilot makes final adjustments to the aircraft's position in relation to the runway and its speed. It is during this phase that the landing gear deploys. To assist the crew, the Instrument Landing System (ILS) is used. This system consists of several antennae placed at different locations on the runway in order to locate the aircraft precisely in space provided. Based on this position, the weather conditions, and the runway, the pilot can determine whether he is correctly positioned using the Final Approach Fix (FAF) [8]. See figure below:

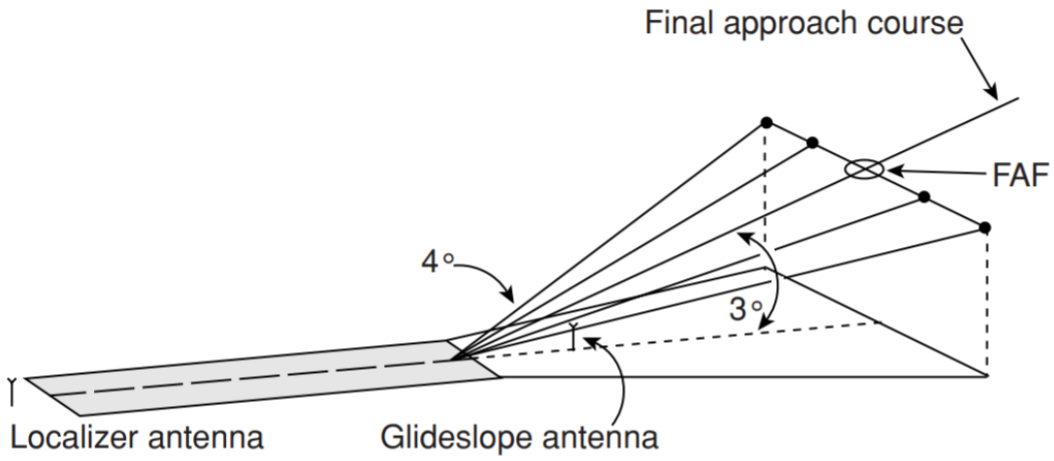


Figure 2.1-B Illustration of an ILS [7]

Hence, if the aircraft is too far from its reference point, the pilot needs to abort the landing and make a new pass to retry the manoeuvre.

2.1.1.3 Landing Phase

Landing is the last phase of flight where the aircraft touches the ground again. The landing phase starts exactly at 15.2 m altitude. Aircrafts normally land on asphalt or concrete runways. This process is carried out by slowly reducing the actual speed of the aircraft while reducing its altitude in order to have the smoothest possible impact on the ground. Once on the ground, the aircraft begins its true braking phase where most of its speed is reduced in a short period of time. The landing braking distance of the aircraft is very controlled and regulated. Depending on the weather conditions or the length of the runway, an aircraft will not have the same landing behaviour. The braking distance is not necessarily the same from one flight to another. See figure below:

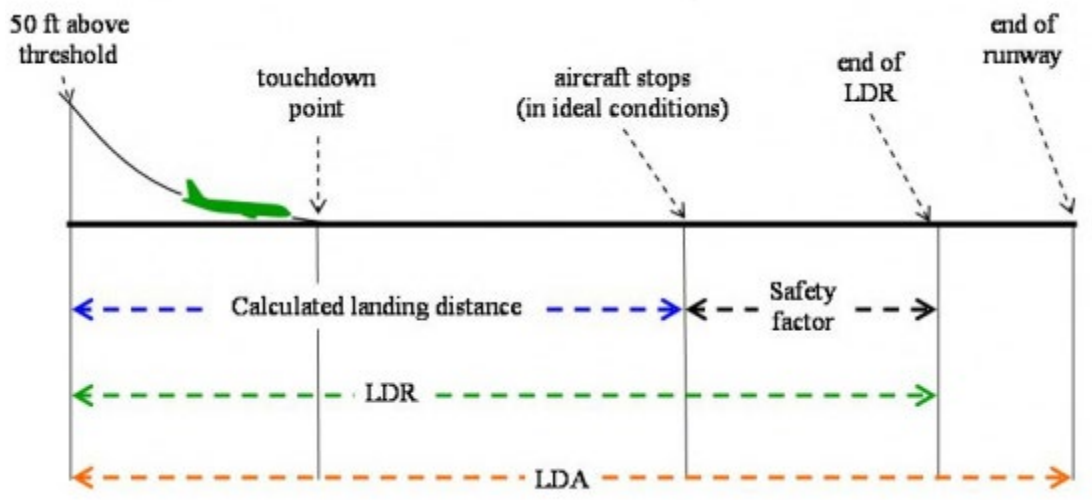


Figure 2.1-C Landing distances of an airplane [8]

The official definition of the braking distance of an aircraft is “the horizontal distance travelled by the aircraft. The calculation of this length is calculated from a predefined altitude until the aircraft comes to a complete stop". [8]. The Landing Distance Available (LDA) represents the length of the runway. To calculate the Landing Distance Required (LDR) it is necessary to define a Safety Factor which will then determine the braking distance that the aircraft must follow.

2.1.2 Braking system of an airplane

An aircraft has different ways of braking and thus landing safely. It is important to study these to understand the landing mechanism.

2.1.2.1 Spoilers/Speed Brakes

Spoilers and speed brakes are surfaces that can be controlled and deployed automatically or manually by the pilot. See figure below:

Figure 2.1-D Spoiler on an A320 [9]

Most times these parts are located on the wings of the aircraft and extend perpendicular to the ground to increase drag and reduce lift.

2.1.2.2 Wheel Brakes

Wheel brakes are present on the aircraft's landing gear. Like in a car, they are pressed disks that transform the mechanical energy of the aircraft into heat. See figure below:

Figure 2.1-E Illustration of wheel Brakes [10]

Most times these brakes are located on the rear landing gear of the aircraft. On larger aircraft, the brakes may also be present in the front. The brakes can be controlled manually with two pedals that the pilot can operate.

The brakes are equipped with an anti-skid aircraft brake system that consists of important hydraulic system ensuring safe take-off and landing. This system prevents the wheels from skidding, thus preventing tyre bursts, aquaplaning when the runway is covered with a film of water, and also to protect the tyres from damage if one of the wheels gets locked or rotates at a speed non-corresponding to the speed of the aircraft [11]. This system is fully automatic and does not require any action from the pilot.

2.1.2.3 Thrust-Reverser

Thrust reversers temporarily help reduce the speed of an aircraft after touch-down, which allows to further reduce the damaging effects on the brakes and allows for shorter landing distances. See figure below:

Figure 2.1-F Illustration of thrust-reverser [12]

It appears that even when idle, aircraft engines continue to produce thrust, which consequently oppose braking. The mechanical brakes found on modern landing gear are theoretically sufficient to stop the aircraft with their simple use. However, in the event of rain or snow, their effectiveness can be questioned. This method of braking is considered as one of the important safety features.

2.1.2.4 Auto-Brake

Auto-brake is an autonomous system that pilots can activate during the landing phase to make their task easier. When activated, the aircraft will automatically decelerate to a certain value [11]. See table below:

Table 2.1-A Auto-brake Modes correlated to the braking values

Auto-brake Modes	Deceleration values (m/s^2)
0	0
1	-0.914
2	-1.524
3	-2.134
4	-3.352

This system consists of 4 different modes which correspond to 4 different decelerations. The modes need to utilise the different braking elements in order to ensure braking as closely possible to these values depicted in the table above. This system is very convenient for pilots' use as it can be activated with just a push of a button in the cockpit. The pilot can also change the mode during landing if necessary.

Now that the way aircraft landing has been described, the next step is to explore the different possible techniques that can aid to solve overfitting of on-board AI models.

2.2 Methods to counter overfitting

Prevention of over-training a ML model, more specifically a DL model to propagate generalization is quit challenging. Multiple parameters can interfere with the quality of a model. Therefore, when one notices that a NN suffers from overfitting, it is possible to act and modify certain variables of the model in order to reach its maximum potential [13]. This section will focus on the development of some of these methods which are normally very effective.

2.2.1 Constraining Model Complexity

The first effective method to counter overfitting is to reduce the complexity of the problem to be solved. A model can overfit on a dataset because it has the means to do so. Thus, by reducing the capabilities of the model it is possible to reduce the probability that a model will overfit on a particular dataset [14]. This can be achieved in two ways:

- Changing the structure of the NN.
- Changing the parameters of the NN.

In the first case, it is possible to use Grid Search algorithms that evaluate the performance of different models according to, the number of neurons per layer, or the number of hidden layers.

In the second case, it is advisable to select only the most relevant variables which carry the greatest weight in the model's decision. This can be done, for example,

by reducing the number of input variables. The use of techniques such as Principal Component Analysis (PCA), allows the importance of the different features of a dataset to be evaluated and only the most important ones to be selected [15]. These methods for modifying the parameters of the DL model are called *Regularization*.

2.2.2 Regularization Methods

2.2.2.1 Weight/Activity Regularization

This technique adds a coefficient to the loss function already implemented in the NN. There are several types of Regularizer. The most common are the L1 and L2 Regularizers [16]. Here is the way to calculate them:

$$L1 = \lambda \sum_{i=1}^{n} |W_i| \quad \textbf{(2-1)}$$

$$L2 = \lambda \sum_{i=1}^{n} |W_i|^2 \quad \textbf{(2-2)}$$

With W_i representing the value of the weights and λ representing a parameter that is set when the layer is created. Due to this coefficient, which is added to the loss function, it is possible to penalise neurons or connections that have much larger outputs than others. The fact that some elements of the NN are over-represented poses a sensitivity problem later on with regard to the input of new data. The addition of a regularisation coefficient increases the sensitivity of the model.

2.2.2.2 Early Stopping

Another method of regularisation consists of training the model on a relatively large number of epochs. During this training, the model will evaluate its performance according to a metric that can be set beforehand on a validation data set. Therefore, it is possible to monitor and plot the evolution of the chosen metric as a function of the number of epochs. See figure below:

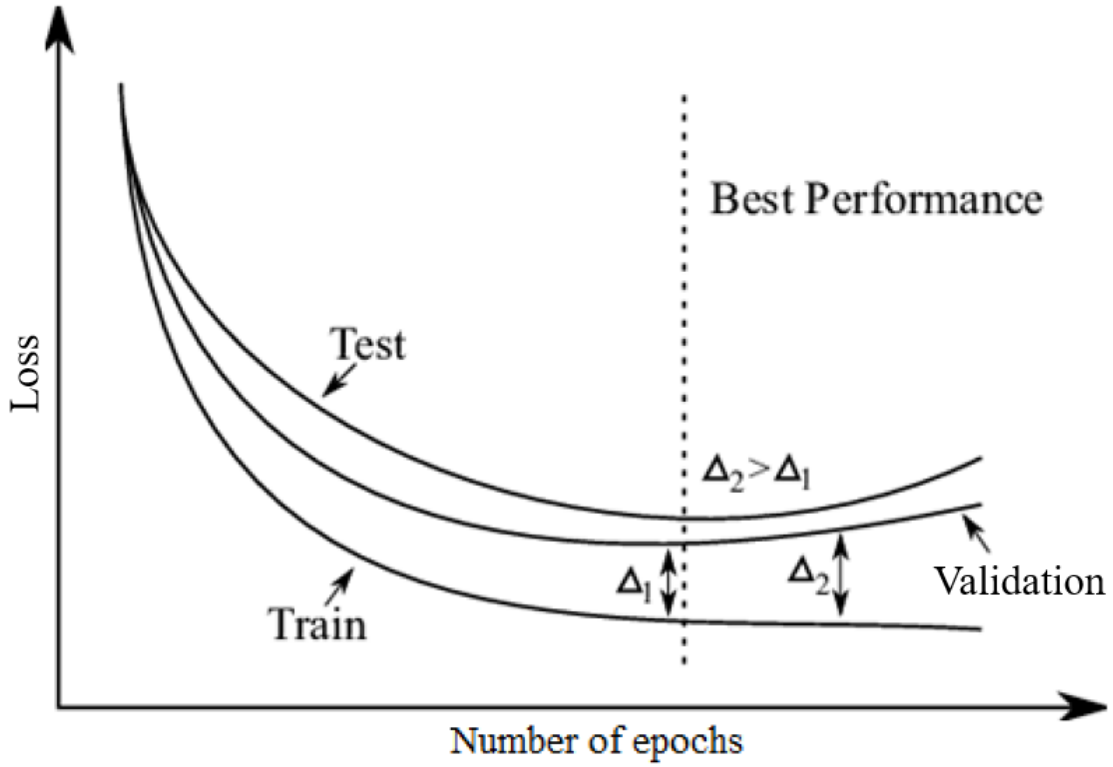


Figure 2.2-A Illustration of early stopping

In this example, the metric used is the evaluation of loss function. When a model is overfitted we often notice that the gap between the loss of the training set and the validation set increases from one iteration to the next [17]. The aim of this method is to find the optimal number of epochs that minimise the difference in the metric used between the training and validation sets.

2.2.2.3 Noise

Another main cause of overfitting is that the model manages to memorise certain inputs from the training set and associate them with certain outputs. This kind of behaviour is very detrimental to the generalisation of the model. To counter this, it is possible to add so-called noise to the input values of the model. This is often done by adding a randomly drawn value to the input values of the NN. This technique can be compared to data augmentation because for each new training iteration, the input will be slightly different. This addition of noise during the training phase of the NN has the same effect as a regularization of the weights and therefore increases the robustness of the model.

2.2.2.4 Dropout

The last technique mentioned to avoid overfitting a NN is to add a dropout layer to the network architecture. The dropout layer affects the learning of the model by randomly ignoring neurons in that layer. This has the effect of creating a whole new layer that is different from the old one as a number of neurons are inactive [18]. This kind of technique adds perturbations during the training of the model forcing some NN nodes to be more or less dependent on their inputs.

Dropout is materialised by a user-defined coefficient varying from 1 to 0. This coefficient represents the probability that a neuron is disabled on the layer. Just like any other parameter, this coefficient can be optimised using Grid Search algorithms [19].

2.3 Innovative alternatives

The literature is full of methods and techniques to limit the impact of overfitting on a model. However, most engineering teams have already implemented these ideas in the hope of solving their problem. Unfortunately, they have not been successful. Traditional techniques do not always work.

It is therefore necessary to propose new and innovative alternative methods that many have not implemented that could solve overfitting.

As witnessed, a simple neural network cannot effectively solve this problem, even with maximum optimisation and a thorough study of its overfitting. Looking for a completely new approach to the problem therefore seems to be the key. For this reason, it was necessary to look at the latest advances in AI.

The latest research and news on AI in recent years revolves around Reinforcement Learning (RL) models, such as the success of the Alpha Go algorithm developed by the company Deep Mind, which is one of the greatest feats of AI, beating the reigning world champion at Go [20].

Beyond the achievements that have been in the news in recent years, RL is well suited to solve the problem of creating an AI model to optimise the landing of its aircraft. As explained in the introduction, *Interleaved* physics simulation models combine a ML model with a simulator that governs the learning of the algorithm.

Here, the aircraft must learn a behaviour, namely landing, to succeed in its task. The development of RL models requires a large volume of training hours on the attached simulator. This is why most of the models known today are developed and used in video games or traditional games such as chess. These simulators, more commonly known as environments, are very easy to reproduce and to control. The computation time of the environment must also be relatively fast so that the learning phase of the model is shortened.

To achieve this, implementation of a RL model was required to solve the problem. Furthermore, it was necessary to create the appropriate "environment" of the model, particularly, recreation of a landing simulator. To “teach” the model to evolve and learn, it is essential to recreate and/or mimic the physics and the mechanics of aircraft landings. This simulator then allows the implementation and development of the RL model itself.

3 METHODOLOGY

3.1 Introduction to Reinforcement Learning

3.1.1 Functional Principles of RL

RL differs from other ML techniques such as supervised and unsupervised learning in its design. Reinforcement Learning algorithms learn by making mistakes and therefore correcting them. In human psychology, for example, reinforcement is used to increase the likelihood of execution of a particular behaviour or to increase the frequency of a particular behaviour in the future, by adding or withdrawing a stimulus. A small child, unbeknownst to the dangers of fire, intrigued by the flame, will touch it, feel pain, and learn that fire is not to be played with. Humans learn from mistakes, rectify them in the future. This is exactly how RL works.

Reinforcement Learning models can solve very complex problems that cannot be solved by traditional methods such as Deep Learning. The algorithm will strive for perfection in its environment because of past errors. One of the main problems of this technique is that it requires a lot of computation time and an environment that can provide training episodes very quickly and in very large quantities. This is why most of the RL algorithms that are known to date are mainly applied to video games, as OpenAI has done with the Atari games [21], because it is very easy to simulate a very large number of games.

This technique is therefore perfect for this use case, as the landing simulator will be able to provide a large number of simulations in a short time. It is also very easy and quick to instantiate a new simulation. All these elements make the application of RL in this case very interesting.

For a Deep RL model to work, it is necessary to implement all the elements.

3.1.2 Elements of a RL model

In technical terms, Reinforcement Learning models are composed of an agent making observations based on the states it receives from the environment in which it evolves. Due to these observations, the agent will be able to make

decisions that will have an impact on a reward/punishment that it will perceive. The agent's main objective is therefore to maximise the expectation of this reward in the long term. Here is a graph summarising the different elements and showing the architecture of the model:

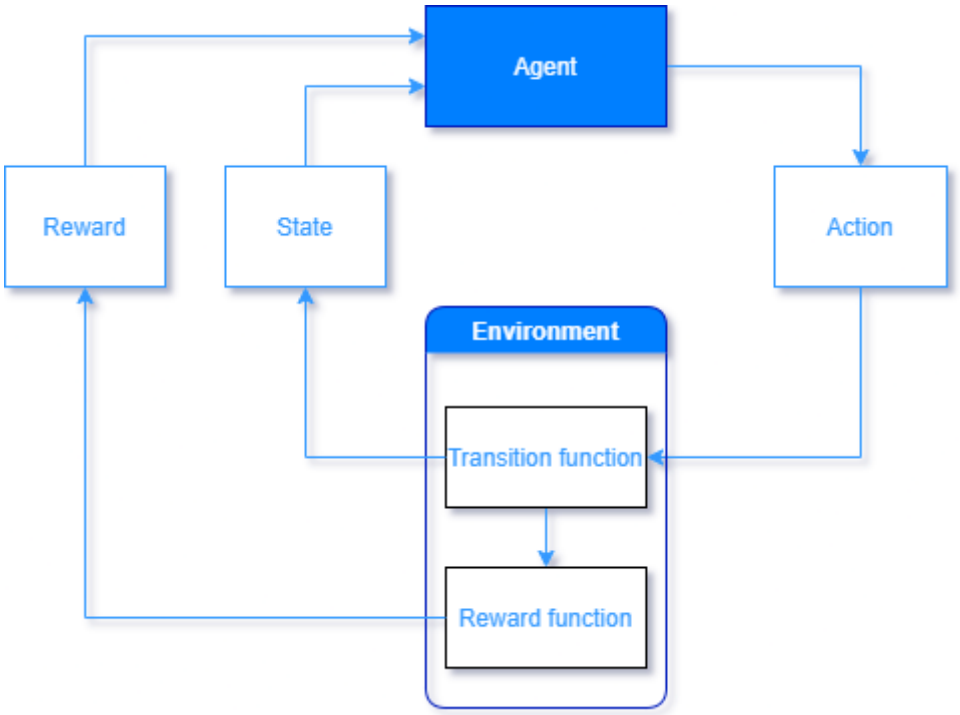


Figure 3.1-A Elements of a RL model

The central element of a Reinforcement Learning model is the agent itself. An agent is a NN which is represented at the start of the learning process. The agent has what is called a *policy*, often noted as π. A policy defines how the agent will behave at that particular time. We can define a policy as a mapping from perceived states of the environment to actions [22]. The behaviour of this *policy* is thus defined with the rewards and the different states it will receive from the environment. The goal is to maximise the reward value at the end of each episode. An episode is a simulation instance until the environment reaches an end state. In this case, the episode corresponds to a landing.

The other important element of RL models is the environment. The environment defines all the rules of the agent's world. The agent created will only be able to

learn from this environment. The environment is made up of a transition function which, depending on the actions taken by the agent, updates the agent's evolutionary environment and provides it with the updated state of its "world". It also consists of a reward function. The reward function is the function that governs whether the agent's actions taken at a certain time should be encouraged or punished. It is therefore essential if we want to make our agent adopt and learn the desired behaviour to designate the reward function accordingly.

As mentioned earlier, it is necessary to recreate an aircraft landing environment as close to reality in order to train a model as precisely to the physics in virtual earth and make it behave in a way that can be replicated in the real world.

3.2 Landing Simulator

3.2.1 Conception of the simulator

The aircraft landing simulator represents the environment of our model. The simulator will update the state of the system according to its actions. In order to develop this model, literature on avionics was referred to.

To begin with, the basics of physics and Newtons' laws of motion was applied to the motion of the aircraft [23]. Here are the equations used:

$$m\frac{dV_x}{dt} + m\left(\omega_y V_z - \omega_z V_y\right) = \sum F_x \quad \textbf{(3-1)}$$

$$m\frac{dV_y}{dt} + m(\omega_z V_x - \omega_x V_z) = \sum F_y \quad \textbf{(3-2)}$$

Where:

m weight of the plane

V_x, V_y velocity of the plane along the different axes

$\omega_x, \omega_y, \omega_z$ moments acting on the plane

F_x, F_y forces along the different axes

Below is the diagram showing all the forces taken into account applied to the aircraft:

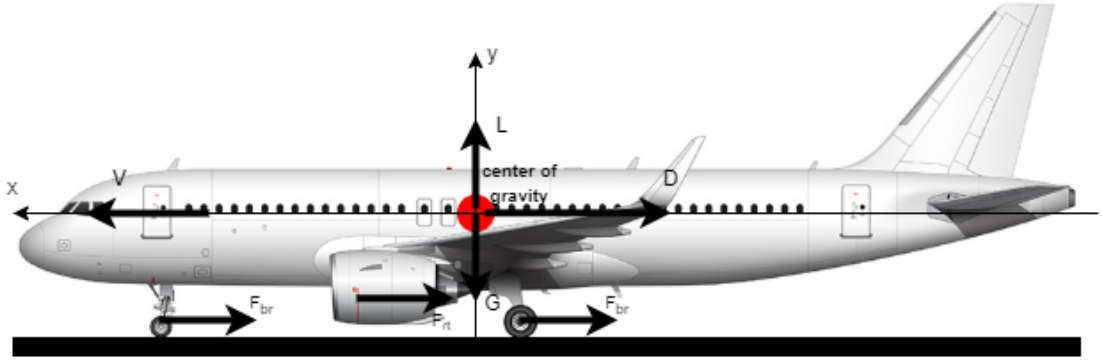


Figure 3.2-A Forces applied to the aircraft

The first force to be considered is the action of the gravity on the aircraft:

$$G = mg \sin(\theta)$$

Where

m weight of the plane

g standard gravity

θ angle of the runaway

The next forces considered were the lift forces L and drag forces D. Lift is the component of the force experienced by a moving body in a fluid that is perpendicular to the direction of motion (to the relative wind). Drag is the force that opposes the movement of a body in a liquid or gas and acts as friction. Mathematically, it is the component of the forces exerted on the body in the direction opposite to the relative velocity of the body with respect to the fluid.

$$L = C_L \frac{1}{2} \rho V^2 S \quad \textbf{(3-3)}$$

$$D = C_D \frac{1}{2} \rho V^2 S \quad \textbf{(3-4)}$$

Where

C_L, C_D	aerodynamic force coefficients
S	wing area
ρ	air density
V	air speed

The aerodynamic force coefficients can be calculated from this equation:

$$C_D = C_{D0} + \frac{C_L^2}{\pi A e} \quad \textbf{(3-5)}$$

Also written in this form:

$$C_D = C_{D0} + k C_L^2 \quad \textbf{(3-6)}$$

With

$$k = \frac{1}{\pi A e} \quad \textbf{(3-7)}$$

The two parameters C_{D0} and k are the zero-lift coefficients and the lift-induced drag coefficient factor. The values of both parameters are considered as constants under a specific aerodynamic configuration of the aircraft.

The next force to consider is the friction force of the landing gear on the runway.

$$F_{fr} = k_{fr} G \quad \textbf{(3-8)}$$

Where

k_{fr} rolling resistance coefficient

G vehicle weight

The rolling resistance coefficient depends on the speed of the aircraft and the condition of the runway which is represented by the coefficient C_{st}. So, depending on the weather, if it rains for example, the coefficient should change.

$$k_{fr} = (0.0041 + 0.000041V)C_{st} \quad \textbf{(3-9)}$$

Thus, the value of the coefficient is likely to change greatly depending on the conditions. For example, the value of the coefficient is 1.0 on dry asphalt.

As the lift of the aircraft changes with speed, the weight applied to the landing gear also changes during the landing phase. For this we obtain the equation :

$$F_{fr} = k_{fr}(G - L) \quad \textbf{(3-10)}$$

Lastly, the force considered is the actions of the braking elements that the aircraft consists of. The aircraft has three braking systems, namely the landing gear disc brakes, the wing spoilers and the engine thrust reverser. The spoilers braking is already taken into account in the drag and lift forces calculation. To simplify the study, the braking of the aircraft is simulated with the values of the different modes of the aircraft's Auto-brake (Table 2.1-A). This deceleration value also depends on the quality of the aircraft's tyres during braking. Indeed, as most of the braking is done thanks to the disc brakes of the landing gear, it was decided to apply a coefficient varying from 1 to 0 in order to simulate the state of the tyres. Thus, if the tyres are worn, braking will be affected.

For future reference, this braking force will be referred to as F_{br}.

To simplify the study and because only the ground phase is of interest for this study, there will be no moment applied to the aircraft. Consequently, there is no angular motion because the aircraft is on the ground:

So, $\omega_x = 0, \omega_y = 0, \omega_z = 0$

The equation of motion governing the physics of our model can therefore be summarised as:

$$m\frac{dV}{dt} = -D - G - F_{fr} - F_{br} \quad \textbf{(3-11)}$$

$$L_x(t) = Vt \quad \textbf{(3-12)}$$

Where $L_x(t)$ aircraft landing braking distance on the runway

3.2.2 Implementation of the Simulator

For the implementation of the simulator, it was decided to use the Python language. The strength of Python lies essentially in its very active community and therefore contributes to the development and implementation of numerous libraries and technologies. Python is particularly used in the field of AI with libraries such as Scikit Learn, TensorFlow or PyTorch

The simulator consists of 2 files: *the forces.py* file where the constants are defined and where the forces are calculated.

```
AIR_DENSITY = 1.225 #air density in kg/m^3
g = 9.81    #gravity vector in m/s²
### A320 values ###
Cd0_land = 0.120  #zero-lift drag coefficient
k = 0.0334    #lift-induced drag coefficient factor
S = 122.6   #wing surface m²
weight = 60000  #mass in kg
Cst = 0.85  #friction coefficient
lift_coefficient = 0.1  #lift coefficient
angle_runaway = 0   #angle runway in °
```

Figure 3.2-B Declaration of the different constants

The values of the constants correspond to those of the A320 [24-25]. The advantage of the A320 is that it is widely used and well documented in the literature. It is therefore very easy to find relevant information.

The other file, *trajectory_generator.py*, is used to simulate the landing of the aircraft from start to finish. By default, the simulator will calculate the state of the plane with a time step fixed at $\Delta t = 0.1s$. In the simulator the approach phase, i.e., the moment just before the aircraft touches down, is simulated by a linear descent until its altitude is equal to 0. Once it touches down, the simulator will calculate the acceleration at each user-defined time step to update the aircraft's status. The simulation ends when the plane's speed reaches 0. Once the simulation is finished, the script proposes to plot different graphs in order to visualise the plane's trajectories.

3.2.3 Outputs of the simulator

It is therefore possible to, for instance, perform aircraft landing simulations for different weather conditions and to observe the impact on braking distance.

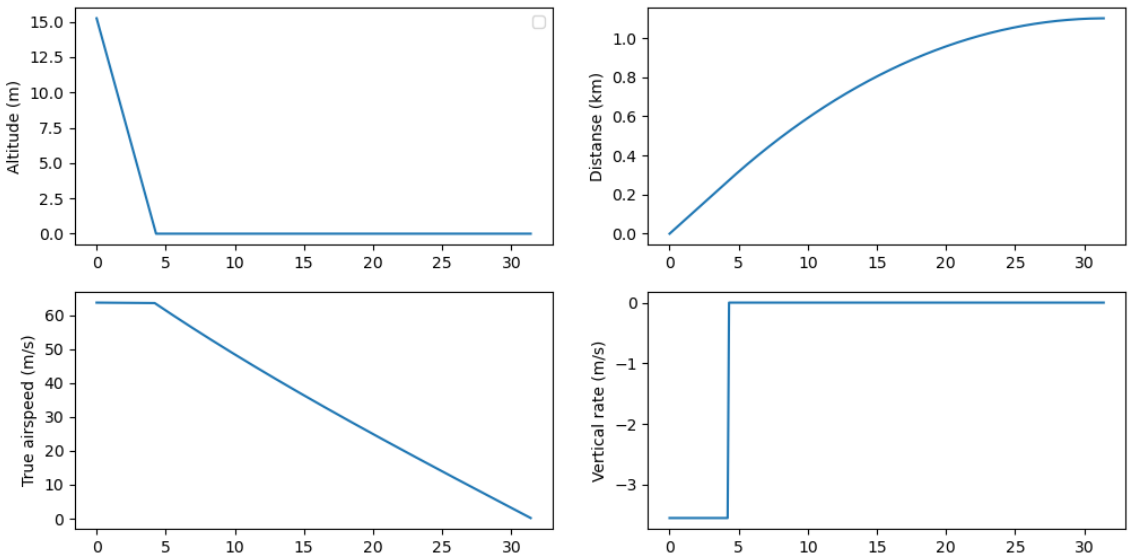


Figure 3.2-C Landing trajectory of the landing simulator, perfect conditions and Auto-Brake = High

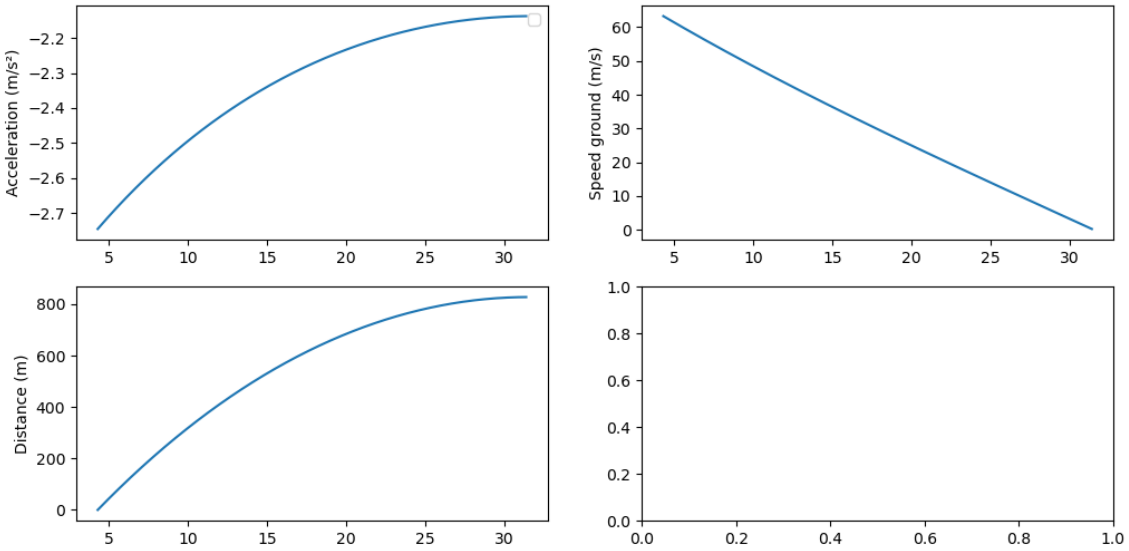


Figure 3.2-D Grounded trajectory of the landing simulator, perfect conditions and Auto-Brake = High

Here is another example in rainy conditions:

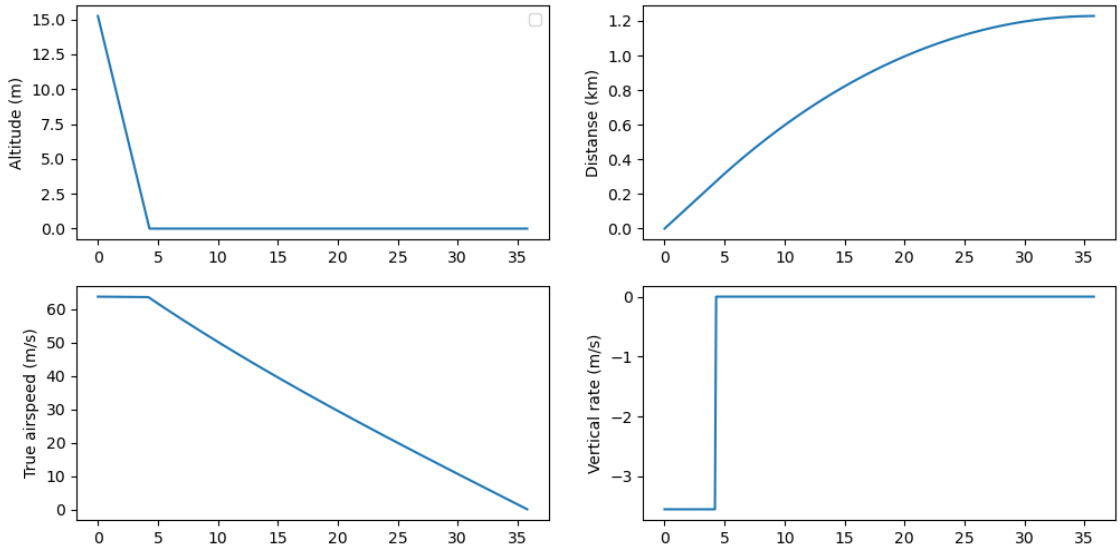


Figure 3.2-E Landing trajectory of the landing simulator, rainy conditions and Auto-Brake = High

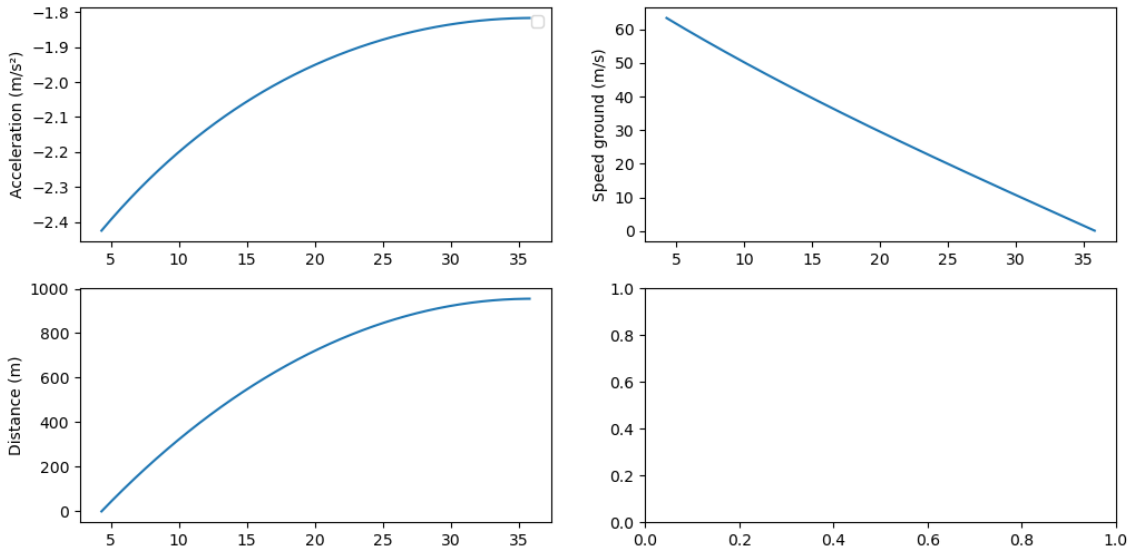


Figure 3.2-F Grounded trajectory of the landing simulator, rainy conditions and Auto-Brake = High

It can be seen that braking is slower and therefore the braking distance is 15% higher. ($C_{st} = 0.85$)

Here is another example of the results that can be obtained by this simulator for the 4 different Auto-brake modes in perfect conditions:

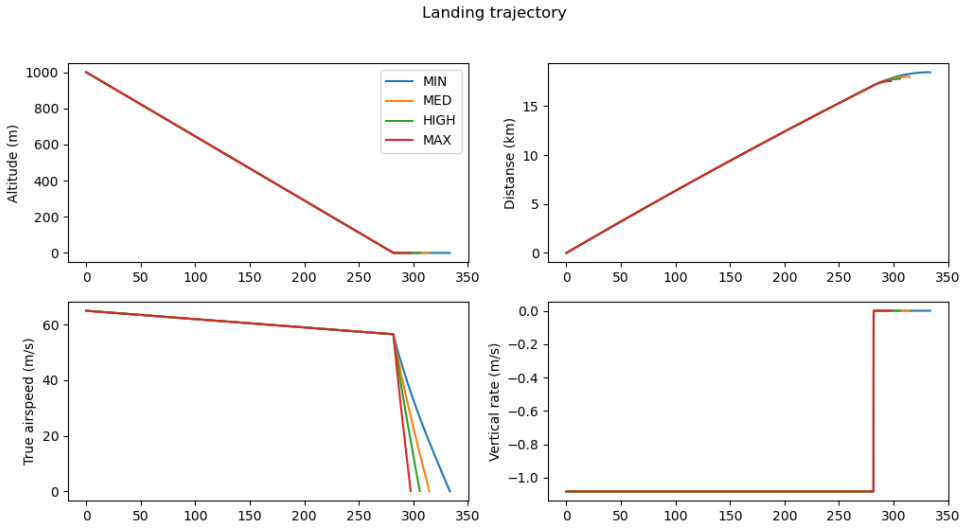


Figure 3.2-G Landing trajectory of an A320 plane for the different Auto-brake modes

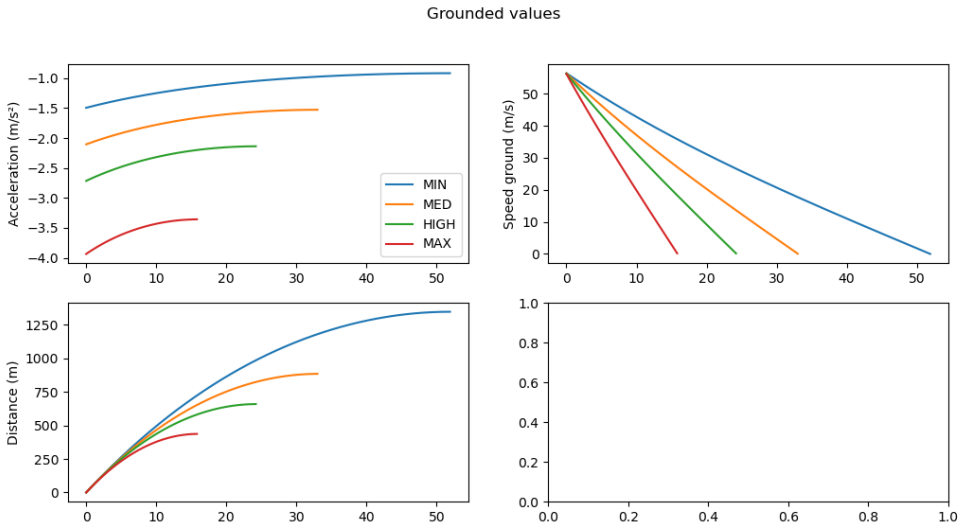


Figure 3.2-H Grounded trajectory of an A320 plane for the different Auto-brake modes

3.3 Implementation of RL model

The model is also implemented in Python. As explained in the previous section, Python relies on its community and in particular its very active scientific community on ML. Thus, by combining libraries such as TensorFlow and TensorForce developed by Google teams and OpenAI Gym developed by the start-up founded by Elon Musk, it is relatively easy to access the cutting edge of the field.

This is why I chose to use mainly TensorForce [26] which is relatively well documented even if still in Beta test phase. It is now necessary to implement all the components essential to the proper functioning of the algorithm.

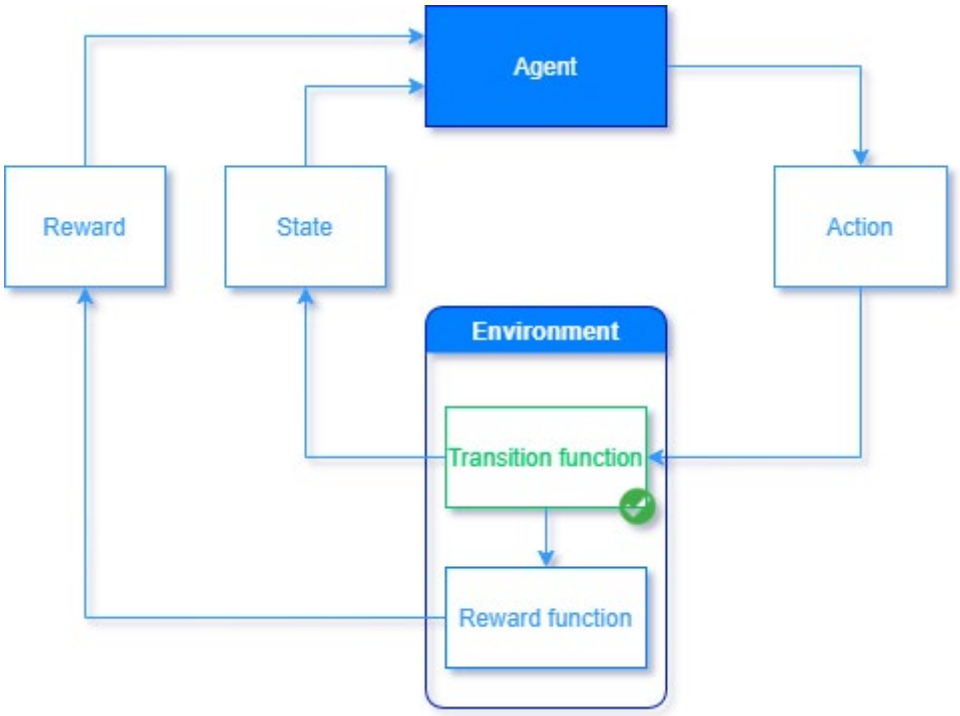


Figure 3.3-A RL model with the Landing simulator

The Landing simulator is therefore the transition function of the model environment.

3.3.1 TensorForce Implementation

The first thing to do is to translate the landing simulator code into an environment that is understandable for the library. For an environment to work, TensorForce requires several basic functions [26]:

-Init: initializes the values needed to generate the environment

-States: defines the shape and the type of variables use as states

-Actions: defines the shape and the type of variables use as actions

-Terminal: defines the events that terminate an episode

-Reset: reset the environment back to initial state in order to start a new episode.

-Execute: processes the action forwarded by the agent and calculate the new state, the reward and check if the episode reach a terminal state.

To be able to land the aircraft, the agent needs to know the acceleration of the aircraft at time t, its speed and its position. The set of states also includes the safety braking distance. The states are therefore made up of 4 elements:

```
def states(self):
    return dict(type="float", shape=(self.STATES_SIZE,))
```

Figure 3.3-B Implementation of the state's function

It was decided to limit the number of input states because the higher this dimension is, the more the necessary calculation time increases considerably. Thus, to keep an acceptable computation time, it was necessary to limit the dimension of the input states. This constraint will be largely developed in the next section.

The agent has a set of actions to stop the aircraft. For this purpose, it was imagined to use the values of the Auto-brake as possible actions for the NN:

$$Action = \{a_{nothing},\ a_{min},\ a_{med},\ a_{high},\ a_{max}\} \qquad \textbf{(3-13)}$$

The set of actions is thus composed of 5 distinct values that can be implemented in this way:

```
def actions(self):
    return dict(type="int",num_values=self.NUM_ACTIONS)
```

Figure 3.3-C Implementation of the action function

Let's move on the conditions for stopping an episode. There are 2 conditions for stopping an episode:

- The speed of the aircraft is less than or equal to 0
- Aircraft exceeds maximum braking distance

```
def terminal(self):
    self.finished = self.LandingModel.V_vec[-1] <= 0
    self.episode_end = self.LandingModel.Pos_vec[-1] >= self.LandingModel.max_braking_distance
    return self.finished or self.episode_end
```

Figure 3.3-D Implementation of the terminal function of an episode

In the first case, the landing is a success, so a different value is returned in case of failure, i.e., if the aircraft exceeds the permitted distance. The aircraft goes beyond the expected distance multiplied by a safety coefficient. The safety coefficient is set at 15% which is the standard value for most airports. It is necessary to distinguish between the two events as the reward of the model will be different.

The next step in the implementation is perhaps the most sensitive and is the reward function. This function will be the basis for learning the model. It must be designed to encourage the agent to perform the task and must be sufficiently precise to ensure that the model learns in the right way. The aim of this model is to encourage the model to use as little of the aircraft's brakes as possible in order to preserve the tyres and brakes as much as possible while ensuring that the aircraft lands safely without endangering the aircraft. A much more evenly distributed braking action also maximises passenger comfort.

Here is the reward function used:

$$r_t = -(v_t - v_{t-1})\alpha$$
$$-\mathbf{1}(S_t = stop)[(bd_{opt} - bd_t)^2\beta - \gamma] \quad \textbf{(3-14)}$$
$$-\mathbf{1}(S_t = run\ off)[v_t\delta + \theta]$$

Where

α coefficient applied according to the action taken by the model

$\beta, \gamma, \delta, \theta > 0$

This reward function has been designed to ensure coverage of all events that may occur during the landing of the aircraft in the environment. Thus, the reward at a time t is defined such that it will encourage the plane to brake. Indeed, the first line of equation (3-14) encourages the plane to lose speed because the agent perceives a positive reward ($v_t > v_{t-1}$). This loss of speed is multiplied by a coefficient which is correlated to the action taken by the model. We want to avoid using the latest auto-brake modes as they cause a lot of damage to the aircraft parts. Thus, α admits 5 different values for as many possible actions. This α value will obviously decrease the higher the braking mode. The aim is to avoid the most important braking modes.

The statement $\mathbf{1}(S_t = stop)$ is equal to 1 if and only if the state is equal to stop, i.e., the plane has successfully stopped during the episode. The model will receive a constant reward γ for the successful landing. However, in order to encourage the model to converge towards the optimal stopping distance, the model will be penalised for each metre more or less after this optimal distance. To do this, the stopping distance of the episode is subtracted from the optimal distance. The result is squared in order to have a positive value and finally a

coefficient β is applied. This value will be subtracted to get the end of episode reward.

The 3rd line deals with the case where the aircraft exceeds the maximum braking distance. The model will then be punished with the value θ. In addition, to indicate to the model that it should converge to a velocity equal to 0 at the end of the episode, the value of the velocity at the end of the episode multiplied by the δ coefficient is withdrawn from it. The value of all coefficients will be determined in the following section.

The implementation of the reward function can be found in Appendix A.

The execute function computes the new states of the environment at time $t + 1$ by taking the actions of the model at time t as parameters. This function also updates the value of the model's reward. Finally, it also checks if an end state has been reached. Here is its implementation:

```
def execute(self, actions,reward):
    next_state = self.LandingModel.compute_timestep(action=actions)
    terminal = self.terminal()
    reward = reward + self.reward(actions)
    return next_state, terminal, reward
```

Figure 3.3-E Implementation of the execute function

Finally, when an end state is reached, the model must be told how to move on to the next episode and under what conditions. To do this, the new episode must be provided with the initial state list. Here is an example of implementation:

```
def reset(self):
    state = np.array([-0.73263,np.random.normal(69.37,4.61),267.2,self.LandingModel.safe_breaking_dist])
    self.LandingModel = LandingModel()
    return state
```

Figure 3.3-F Implementation of the reset function.

The different elements are now implemented in Python in functions that can be understood by TensorForce.

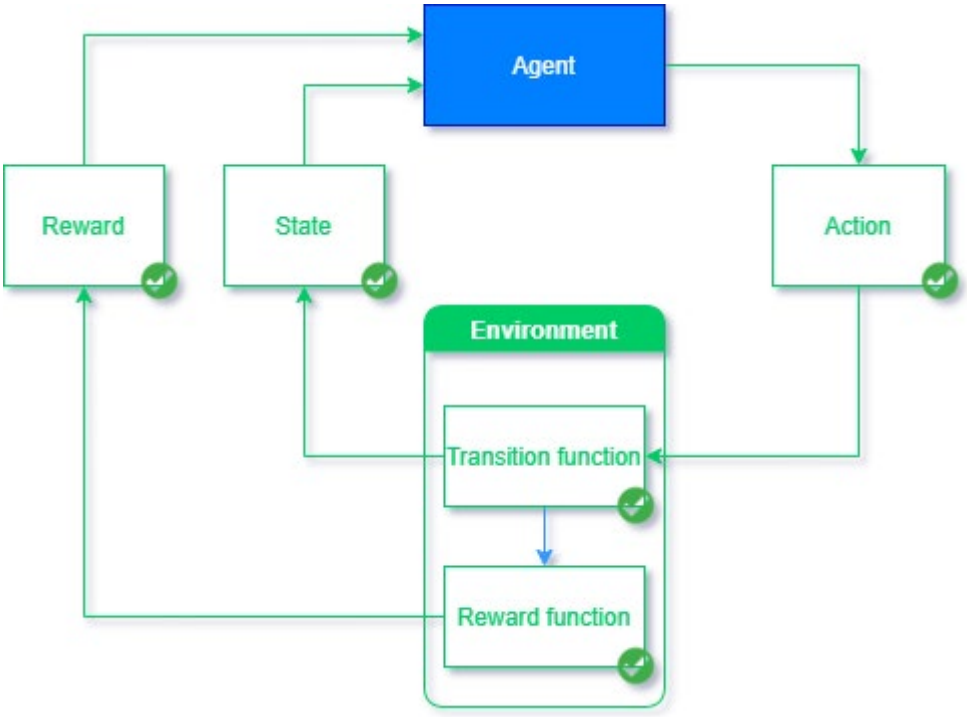


Figure 3.3-G Implementation of the elements in TensoForce

The environment is therefore complete, the model is now able to receive actions from the agent, count them in order to provide the new updated states of the physical system while providing positive or negative feedback according to its actions. All that remains now is the implementation the agent.

3.3.2 Agent Implementation

There are many types of agents in RL. However, one of them stands out from the rest because of its performance and especially its relatively short learning time compared to the others. This agent is called Proximal Policy Optimization (PPO) and was developed by the OpenAI teams in 2017 [27].

3.3.2.1 Proximal Policy Optimization

The goal of the PPO agent is to find the best optimal policy in order to maximize the reward. To do this, the algorithm uses the Minorize-Maximization (MM) algorithm [28], which iteratively maximizes a lower bound function that is denoted

by M while locally approaching the expected reward denoted here by η. See figure below:

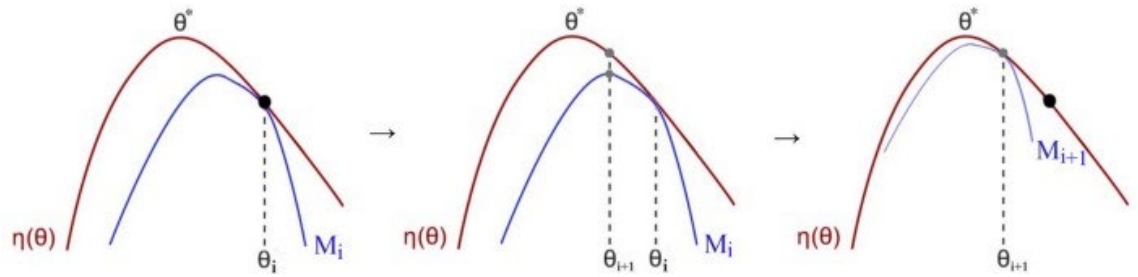


Figure 3.3-H Operating principal of the MM algorithm [28]

First, it starts with an initial default policy and find a lower bound M for η. It then uses the new policy as a new starting point until the policy converges. In order for this to work, it need to find a lower bound M that is easily optimised.

The PPO algorithm monitors the changes it makes to the policy at each iteration through the Kullback-Leibler (KL) divergence. KL-divergence measures the difference between 2 distributions of data P and Q.

$$D_{KL}(P||Q) = \mathbb{E}_x \log\left(\frac{P(x)}{Q(x)}\right) \quad \textbf{(3-15)}$$

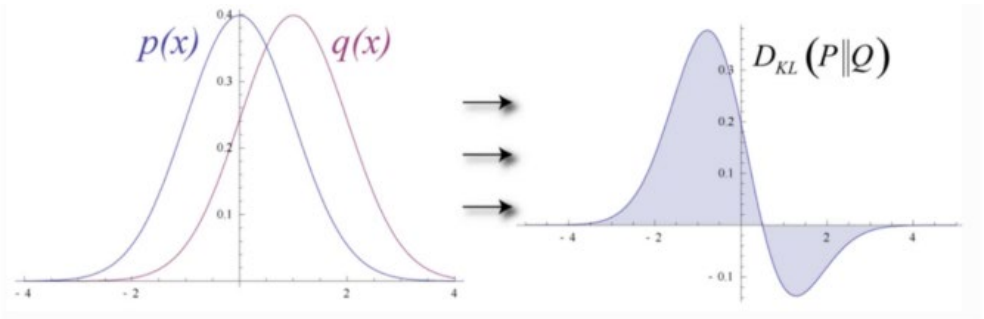


Figure 3.3-I Exemple of KL Divergence on 2 Gaussian distributions [29]

This allows us to measure how different the new policy is from the old one. However, to control the learning of the algorithm we do not want the new policy to be too different from the old one.

It is also necessary to limit the number of policy changes to ensure that it does not make the wrong decision. It turns out that it is possible to find a lower bound M of the following form:

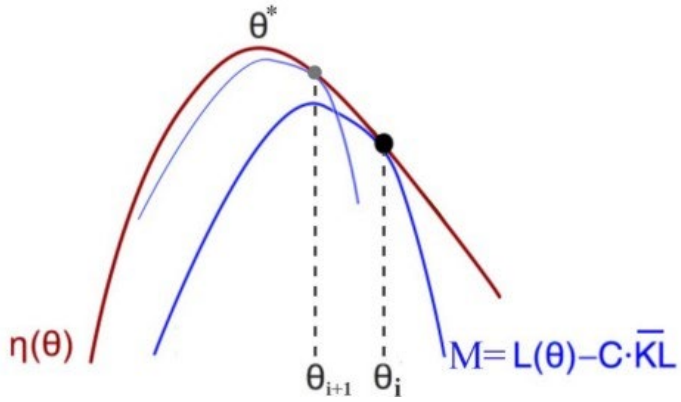


Figure 3.3-J Shape of the lower limit M [30]

With

$$L(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t \right] \quad \textbf{(3-16)}$$

Where

$\pi_\theta, \pi_{\theta_{old}}$ 2 policies involved

$\hat{A}_t$ Estimator of the advantage function (Appendix B)

And the second term is the KL-divergence multiplied by a coefficient C.

L is the expected advantage function for the new policy. It is estimated by the old policy and recalibrated using the probability ratio between the new and old policy. Here, it uses the advantage function and not the expected reward to reduce the variance of the estimate.

Moving on to the 2nd term of M. The PPO algorithm is derived from another algorithm that uses the same method to converge the model policy. This algorithm is called Trust Region Policy Optimization (TRPO) [31]. In this paper, the researchers proved that the second term of M requires the computation of

second order derivatives and their inverses. However, these operations are very computationally demanding and greatly reduce the efficiency of the algorithm.

The researchers also proved that this problem could be reformulated as an optimisation problem:

$$\underset{\theta}{maximize}\ \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t\right] \qquad \textbf{(3-17)}$$

$$subject\ to\ \hat{\mathbb{E}}_t\left[KL[\pi_{\theta_{old}}(a_t|s_t), \pi_\theta(a_t|s_t)]\right] \leq \delta$$

Or

$$\underset{\theta}{maximize}\ \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t\right] - \beta\hat{\mathbb{E}}_t\left[KL[\pi_{\theta_{old}}(a_t|s_t), \pi_\theta(a_t|s_t)]\right] \qquad \textbf{(3-18)}$$

To solve this problem in the given time frame there are 2 approaches:

- Approximate some calculations of second-order derivatives and their inverses to reduce calculation time
- Try to solve the problem using first-order derivatives, such as gradient descent, which are much shorter in computation time, by bringing them closer to second-order derivatives by adding softer constraints.

The TRPO algorithm takes the first approach. PPO is closer to the second option. It will of course be possible for the algorithm to go wrong because it calculates only first order derivatives.

To overcome this problem, researchers have proposed a method. This method is called PPO with Clipped Objective. In its implementation, the method requires 2 policy networks. The first one is the current policy that we want to optimise $\pi_\theta(a_t|s_t)$ is called the actor network. The second to collect samples $\pi_{\theta_k}(a_t|s_t)$ is called the critic network. This sampling idea allows the new policy to be evaluated with samples collected from the old policy.

But with each iteration, the gap between the optimised policy and the old one grows because the old policy remain frozen as the actor policy is getting optimized. The variance of the estimate will therefore logically increase and bad decisions will be made because of the inaccuracies. To overcome this, the algorithm will synchronise the two policies every x iterations.

$$\pi_{\theta_{k+1}}(a_t|s_t) \leftarrow \pi_\theta(a_t|s_t) \tag{3-19}$$

The idea of clipped comes from the fact that we will calculate the ratio between the new policy and the old one:

$$r_t(\theta) = {\pi_\theta(a_t|s_t)}/{\pi_{\theta_k}(a_t|s_t)} \tag{3-20}$$

This ratio therefore measures the difference between the two policies. We then construct a new objective function which will then clip the estimated advantage function if the new policy is too far from the old one. We then have :

$$\mathcal{L}^{CLIP}_{\theta_k}(\theta) = \hat{\mathbb{E}}_t\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right] \tag{3-21}$$

If the ratio between the new policy and the old one is outside the set $(1-\epsilon)$ and $(1+\epsilon)$, the advantage function will be clipped. Here is an illustration of this function where ϵ is set to 0.2:

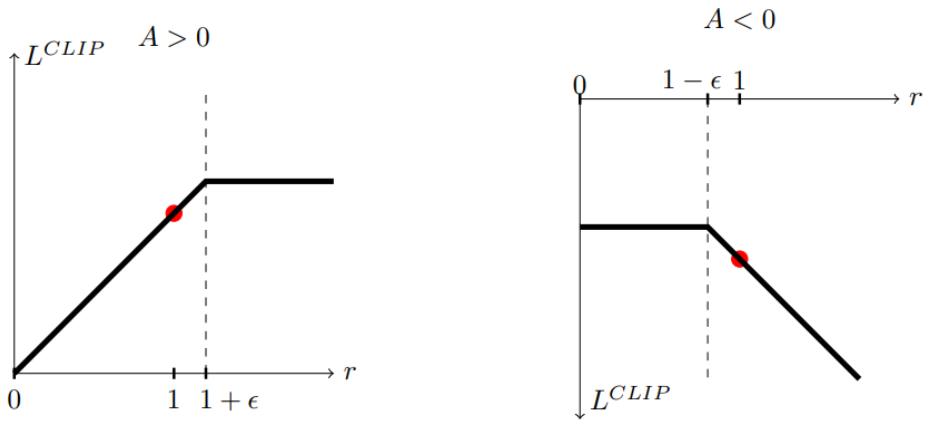


Figure 3.3-K Illustration of the clipped method [27]

This method therefore admits a much simpler approach and uses gradient descent-based optimisation algorithms like Adam for example. PPO adds a soft constraint that can be optimised with first order optimisers. Bad decisions can be made sometimes but the algorithm has found a good balance between computation time and accuracy. Experimentally, the results show that the algorithm achieves excellent performance with unmatched simplicity.

3.3.2.2 Implementation of the agent

The strength of TensorForce is that the library already includes a good number of agents. Thus, there is no need to redevelop the algorithm and the methods it uses. Here is an example of an implementation with all its parameters.

```
agent = Agent.create(
        agent='ppo', environment=environment,
        # Automatically configured network
        network='auto',
        # Optimization
        batch_size=10, update_frequency=2, learning_rate=1e-3, subsampling_fraction=0.2,
        optimization_steps=5,
        # Reward estimation
        likelihood_ratio_clipping=0.2, discount=0.99, estimate_terminal=False,
        # Critic
        critic_network='auto',
        critic_optimizer=dict(optimizer='adam', multi_step=10, learning_rate=1e-3),
        # Preprocessing
        preprocessing=None,
        # Exploration
        exploration=0.0, variable_noise=0.0,
        # Regularization
        l2_regularization=0.0, entropy_regularization=0.0,
        # TensorFlow etc
        name='agent', device=None, parallel_interactions=1, seed=None, execution=None, saver=None,
        summarizer=None, recorder=None
    )
```

Figure 3.3-L Example of the implementation of a PPO agent

As can be seen in the previous figure, the agent includes many different parameters that need to be optimised in order to create a model that manages to converge to a near perfect behaviour.

4 RESULTS & DISCUSSION

Now that all the elements have been implemented, it is now possible to launch the training of the algorithm.

4.1 Hyperparameters tuning

Before being able to interpret the concrete results of the model, it is necessary to find the parameters allowing to define the agent that will theoretically perform the best. Thus, to determine the best parameters of the agent a Grid Search method has been used. To do this, the algorithm will run several model trainings with different parameter combinations.

To distinguish which model is the best, it was decided to use the values of the rewards. The value of the rewards of the episodes is in this case very significant for the performance of one model compared to another.

It is also necessary to define the coefficients of the reward function. The aim of the coefficients is to balance the value of the reward function in order to strongly punish the model when it exceeds the safety distances but also to progressively reward it in order to have a convergence towards the optimal policy. For that, it was after various tests to choose its coefficient values:

$$\alpha = [10,8,5,2,1]$$

$$Action = \{a_{nothing},\ a_{min},\ a_{med},\ a_{high},\ a_{max}\} \quad \textbf{(4-1)}$$

The values of the coefficients for the actions $a_{high},\ a_{max}$ were initially negative. However, the model was reluctant to use these modes to the point where it preferred to miss an episode rather than use them. To rebalance this, these modes now have positive coefficients. They are however largely reduced for the first 2 modes and the mode where the plane does not brake. Here are the values of the other coefficients :

$$\beta = 0.01, \gamma = 500, \delta = 1, \theta = -1000 \quad \textbf{(4-2)}$$

The tuning of the parameters was carried out in a landing environment in the best conditions (no rain, no wind...). The safety landing distance is set at 1200m with a safety coefficient initialized at 15%. Thus, the maximum distance the aircraft can cover is 1380m. Each combination is evaluated over 100 batches of 100 episodes, i.e., 10,000 landings. This number gives the model time to converge in the best case. To add difficulty and to avoid having the same landing conditions for each episode, the approach speed of the aircraft is a random value according to a normal distribution of parameter (69.37,4.32) which corresponds to the distribution of aircraft speeds at the moment of contact with the ground [24].

4.1.1 Non-Network Parameters

Several types of parameters can be distinguished. The first category under consideration here concerns the parameters that do not concern the actor's NN.

The first parameter is the Subsampling Fraction. This parameter controls how many steps of an episode will be used to compute the backpropagation in the NN.

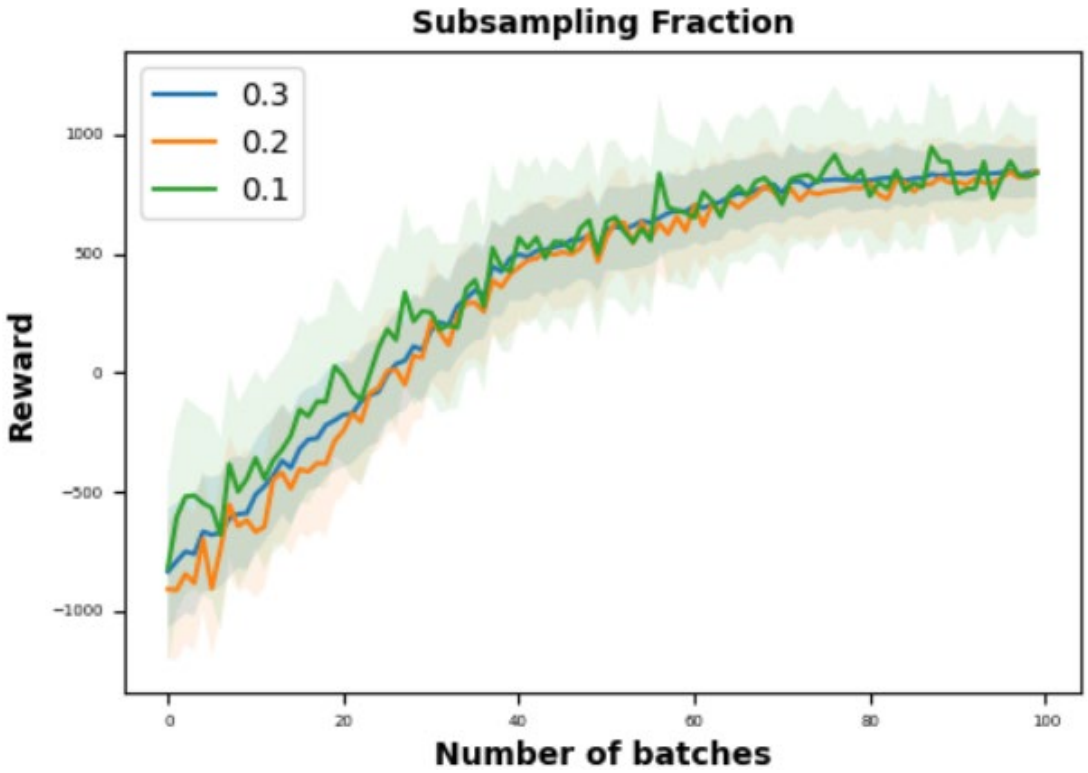


Figure 4.1-A Results of the Subsampling Fraction parameter

The graph below represents the average rewards as a function of the number of batches. Thus, the curves are the average reward values of all combinations that were tested with the value of the coefficient in question. This graph also shows the standard deviation of the reward values which are represented in the coloured area around the curves.

Here, this parameter does not really have an impact on the efficiency of the model. Indeed, whatever the value of the parameter the curves all converge towards the same reward values. Only in terms of the variance of the results, the value 0.3 is more accurate.

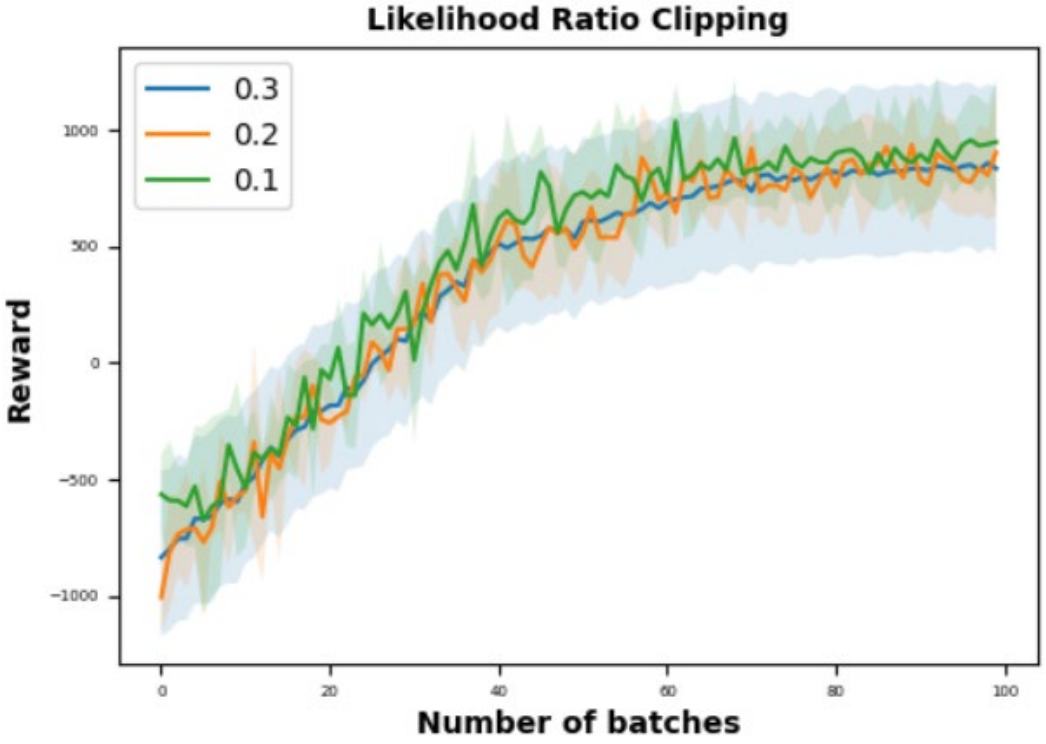


Figure 4.1-B Results of the Likelihood Ratio Clipping parameter

This parameter corresponds to the value of ϵ which allows to define a range of value indicating if the policy should be clipped or not. As before, the parameter does not influence the results of the model in view of the convergence of the curves. Here, the parameter 0.1 is the one with the best result.

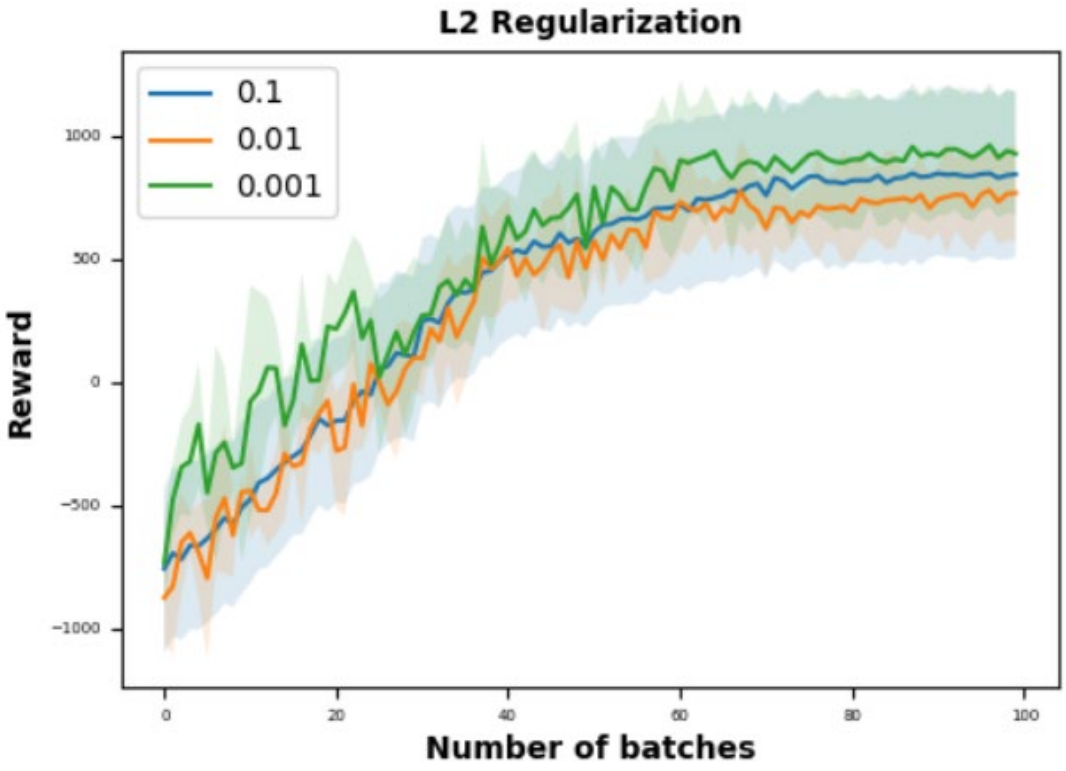


Figure 4.1-C Results of the L2 Regularization parameter

This parameter controls, as explained in part 2, the amount of random noise that the algorithm adds to the agent's loss function to avoid overfitting. Here, the 3 curves converge, only the parameter with the value 0.001 gets better results.

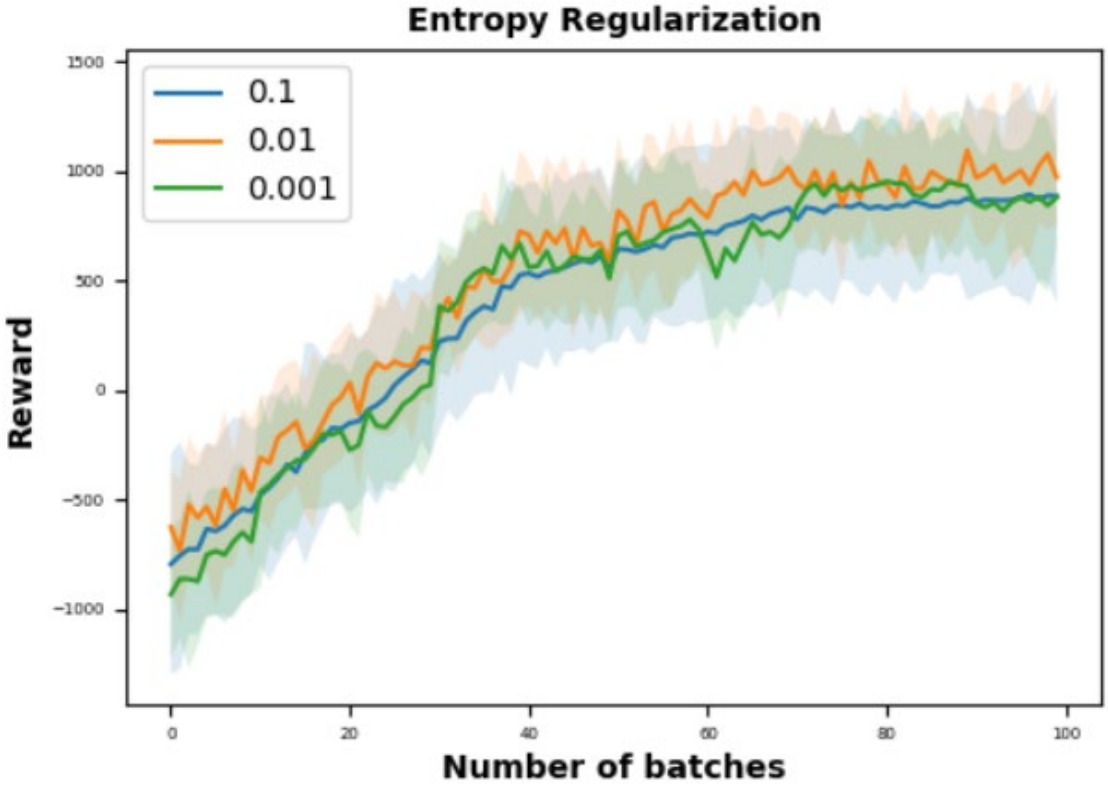


Figure 4.1-D Results of the Entropy Regularization parameter

The entropy regularization parameter allows the policy to not be "overconfident" and thus to be "stuck" on a performance improvement. Here, the curves also converge. However, the parameter with the value equal to 0.01 has better results in the end.

Here are the values of the selected parameters:

Table 4.1-A Non-Network Parameter Selection

Parameter	Selected Value
Subsampling Fraction	0.3
Likelihood Ratio Clipping	0.1
L2 Regularization	0.001
Entropy Regularization	0.01

4.1.2 Network Parameters

The same technique was applied to the parameters of the agent's NN. The first parameter is the number of neurons present on each layer of the Net-work. In this case, the layers are all FC layers whose function is to activate a tanh. Indeed, the network does not require the use of special layers to perform this task.

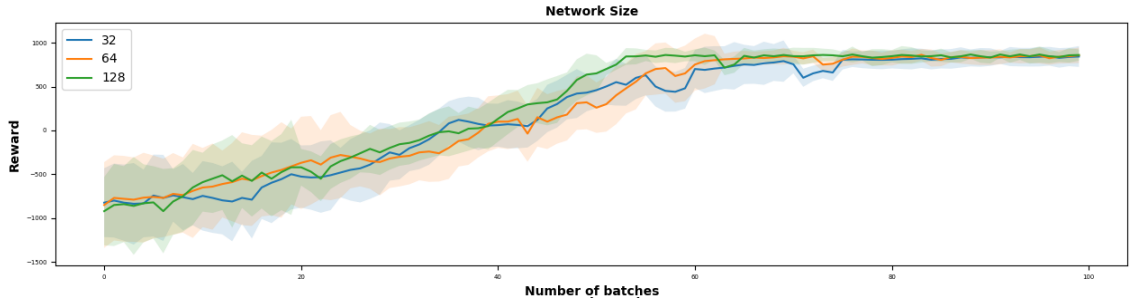


Figure 4.1-E Results of the Network Size parameter

As for the previous graphs, the different models with the 3 parameter values converge. We notice that the networks with layers of 128 neurons converge slightly faster. However, these models are much slower in the learning phase. Thus, it is recommended to reduce the size of the layers to only 64 neurons for better computing time.

The second parameter is the number of hidden layers in the network.

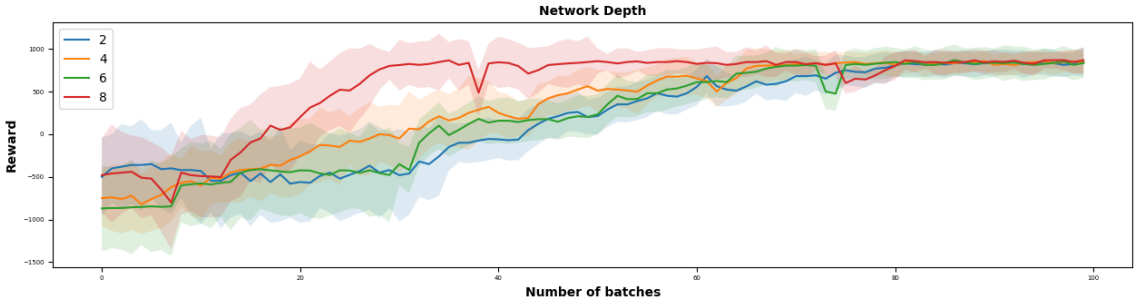


Figure 4.1-F Results of the Network Depth parameter

Here, the Grid Search algorithm compares the results of 4 different NN types with hidden layers ranging from 2 to 8. We notice here, as before, that the models with 8 layers converge faster than the others. Unfortunately, the learning time is rather unreasonable. The models with 4 layers also converge very well and have a good

compromise in terms of learning speed. The NNs will therefore be equipped with 4 hidden layers of 64 neurons each.

4.1.3 Others Parameters

The PPO agent also includes two other parameters that can be modified, which are the Discount Factor and the Exploration. The Discount Factor describes how much the agent will try to predict the endings of episodes rather than focusing on the rewards of the next timestep.

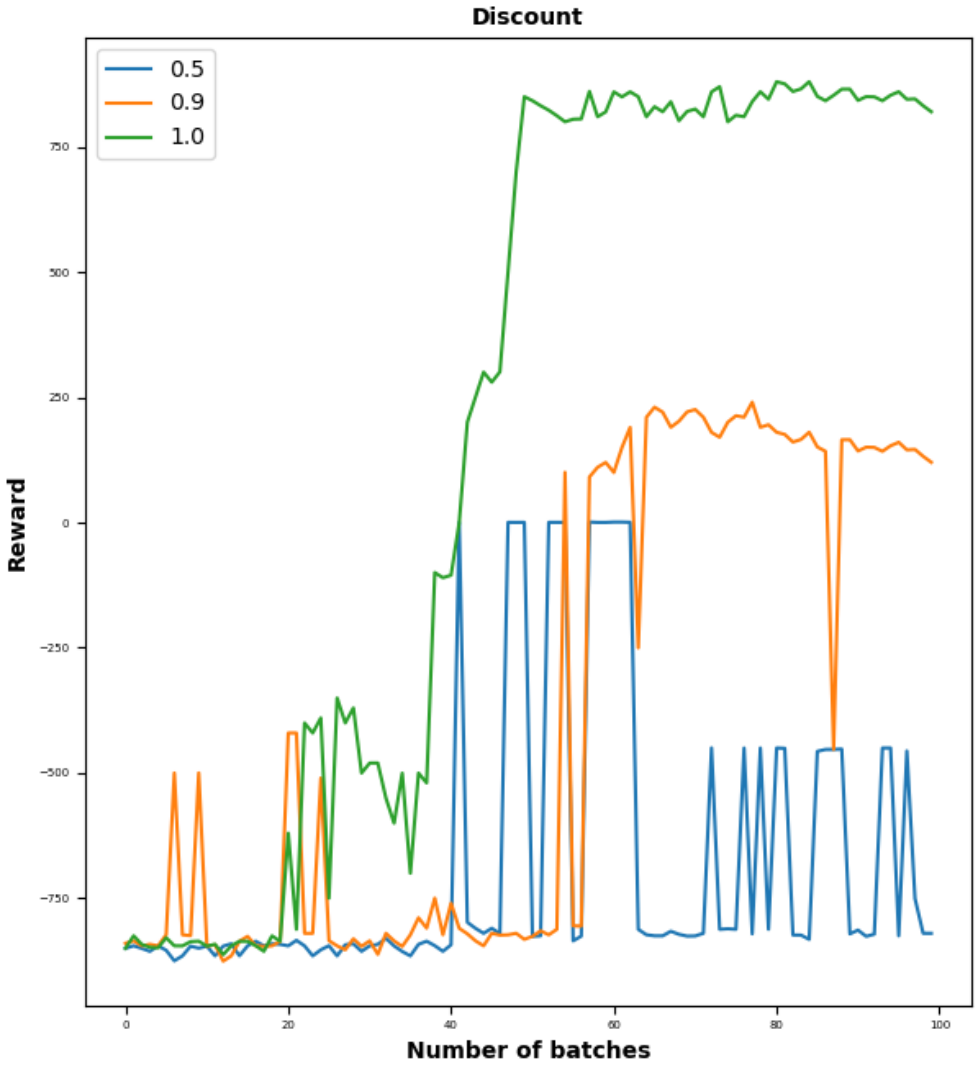


Figure 4.1-G Results of the Discount parameter

Here it is easy to section off the discount value. The value should be 1. The other curves do not converge to an optimal policy in 100 batches.

The second parameter is the Exploration. This parameter represents the probability that the agent will take a random action rather than respect the policy. This allows the agent to try actions that it would not normally take and thus discover new behaviours.

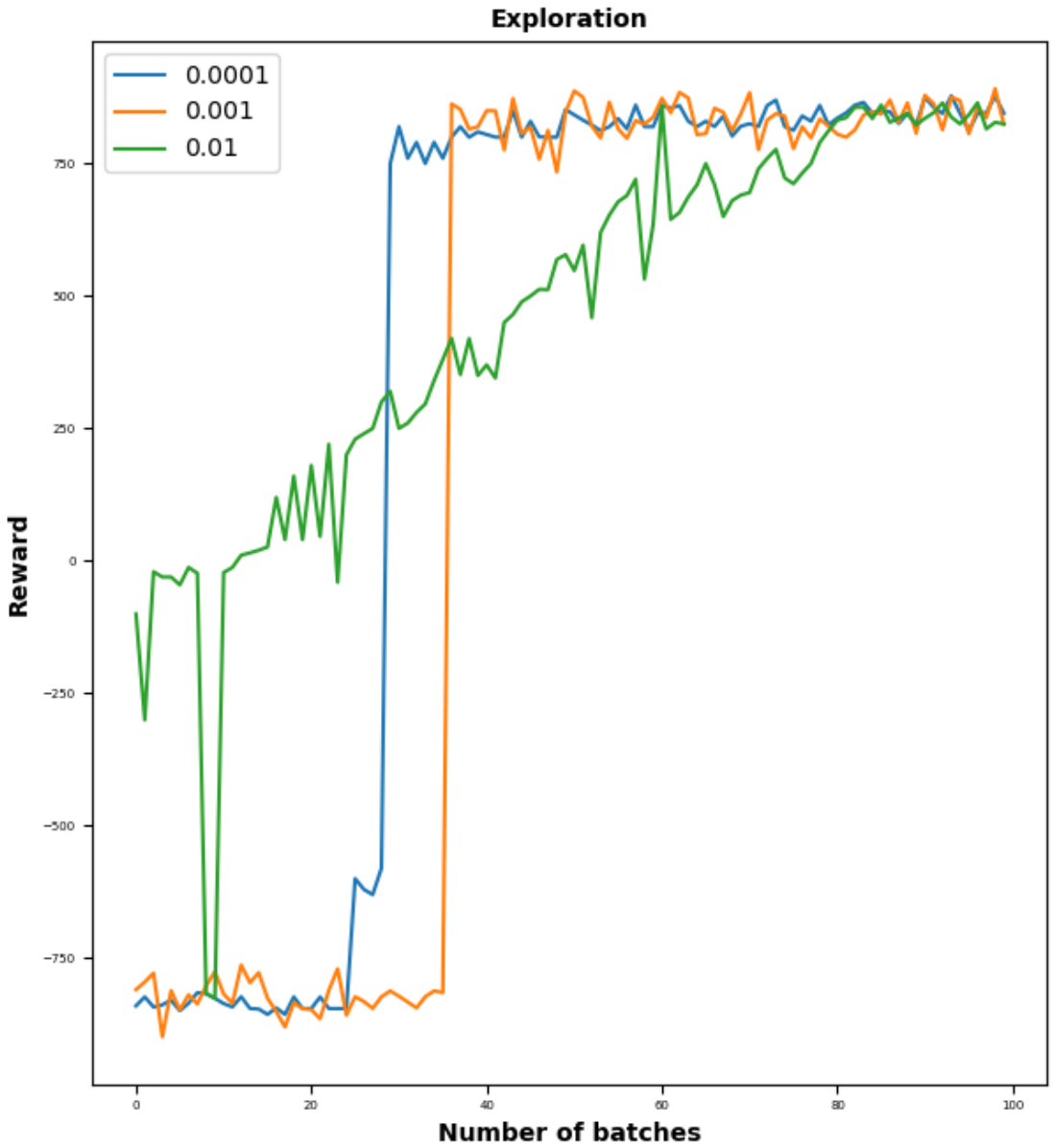


Figure 4.1-H Results of the Exploration parameter

The curves here all converge. It can be seen that in this use case, randomness is not a good option and does not help the model during learning. On the contrary,

it seems to penalise it. Here, we have chosen to take the value 0.0001 for the parameter value.

The agent is now optimised and ready to generate the first exploitable results.

4.2 Policy of the Model

Now that the agent is optimised and all the elements are implemented it is possible to have the first results. To illustrate the performance of the agent, it was decided to take two use cases. An analysis of the agent under perfect landing conditions and another analysis under rainy conditions with a runway covered by a film of water.

4.2.1 Perfect Condition Agent

Here are the results of the model during its training. The graph below plots the reward function against the number of batches.

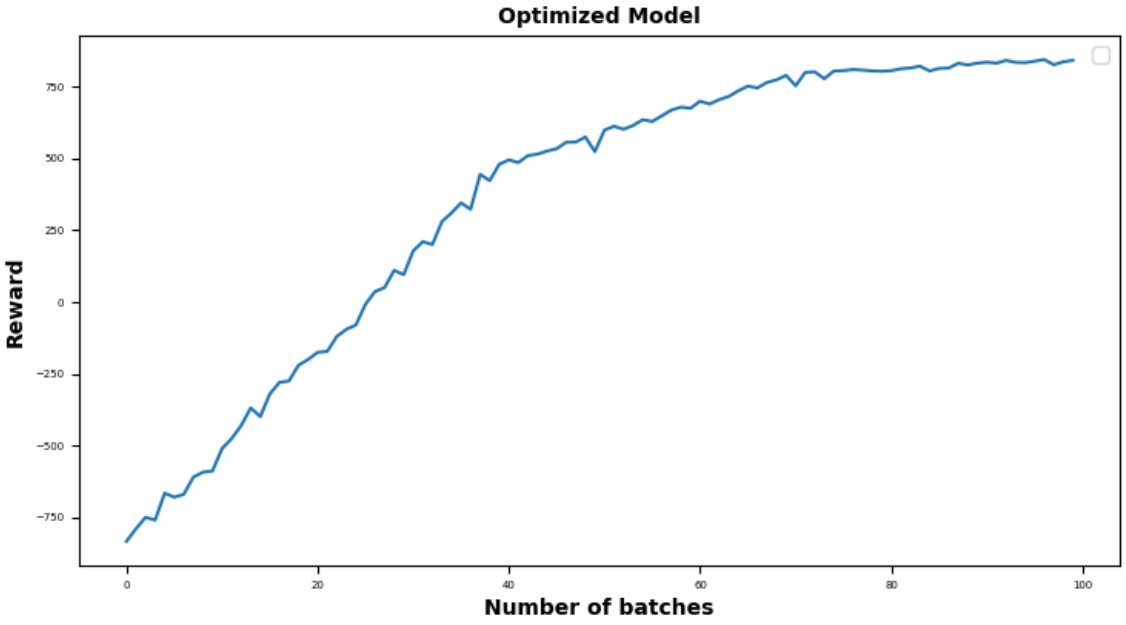


Figure 4.2-A Reward against the number of batches for the optimized model

It can be seen that the model converges almost to the optimal policy after about 70 training batches. Before that, the progression seems linear and continuous for

the first 4000 episodes. Thereafter, the model progresses more slowly over the next 3000 episodes before almost reaching the optimal policy the last 30 batches.

To try to understand how the agent learns it is interesting to plot his actions as he trains. To do this, the actions taken by the agent are plotted every 1000 episodes in the following graphs:

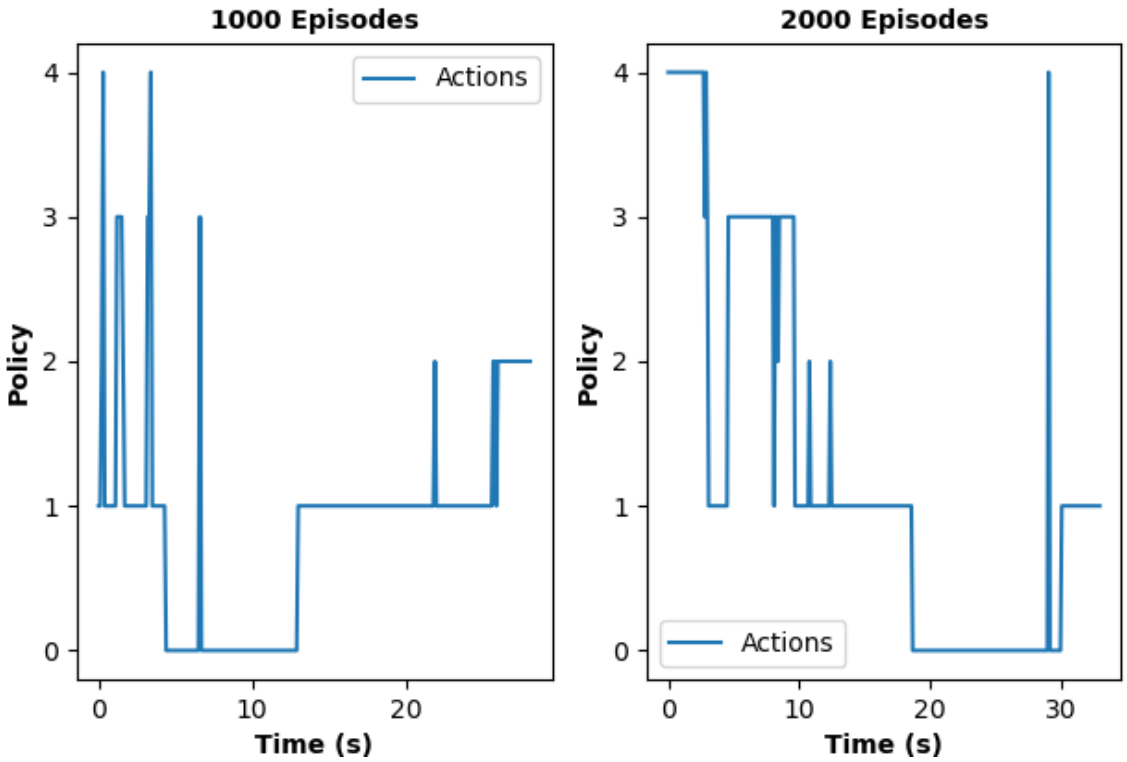


Figure 4.2-B Policies after 1000 & 2000 episodes of training (Perfect Conditions)

The graphs show the policies of the episodes every 1000 training episodes. On the x-axis is the time of the episode. On the ordinate are the actions taken by the model from 0 to 4. 0 being no braking applied by the device up to 4 which represents the maximum braking.

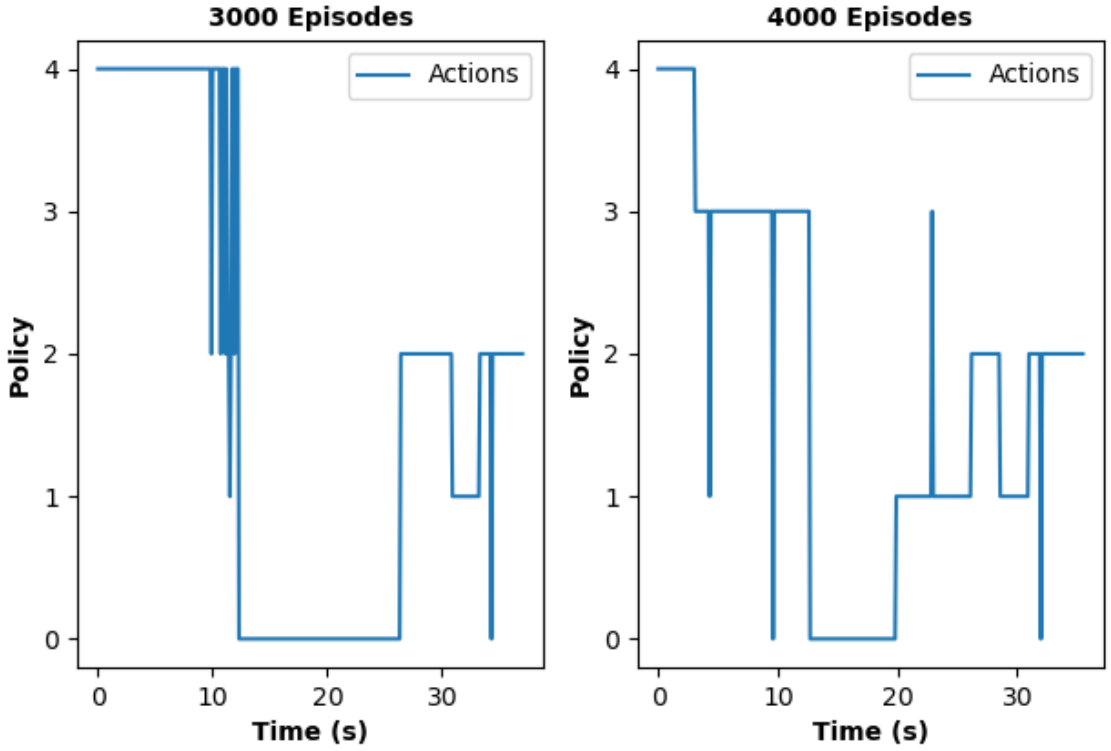


Figure 4.2-C Policies after 3000 & 4000 episodes of training (Perfect Conditions)

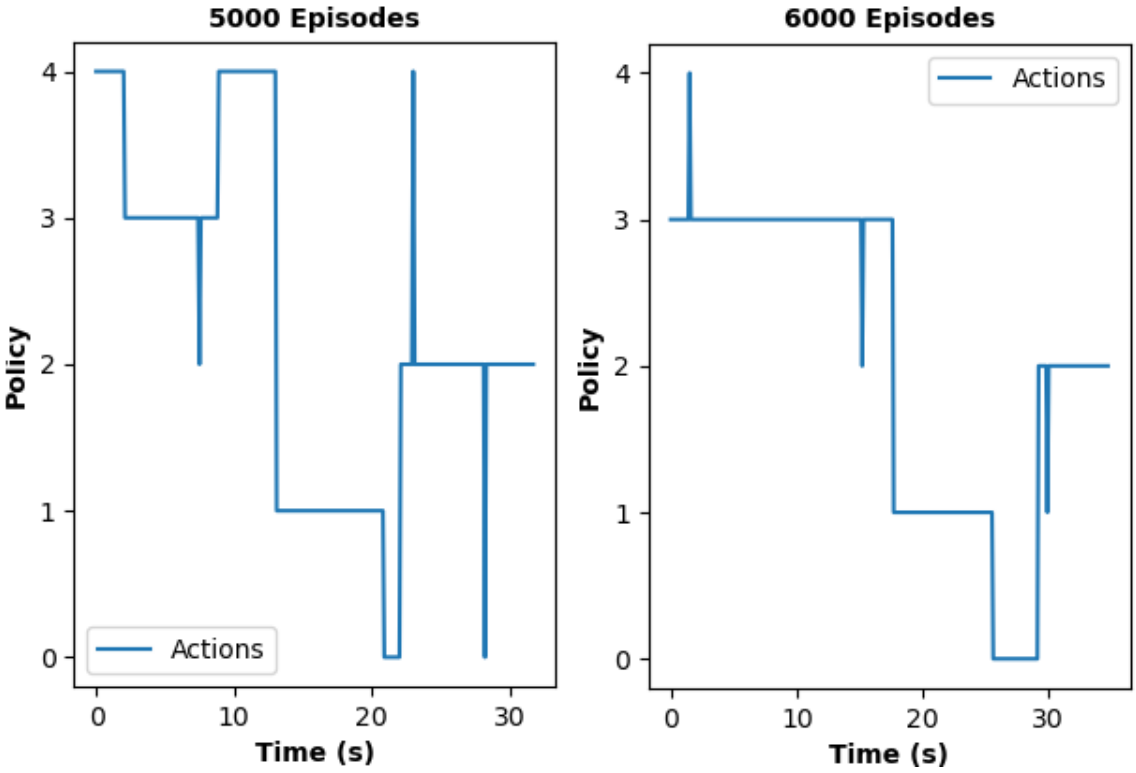


Figure 4.2-D Policies after 5000 & 6000 episodes of training (Perfect Conditions)

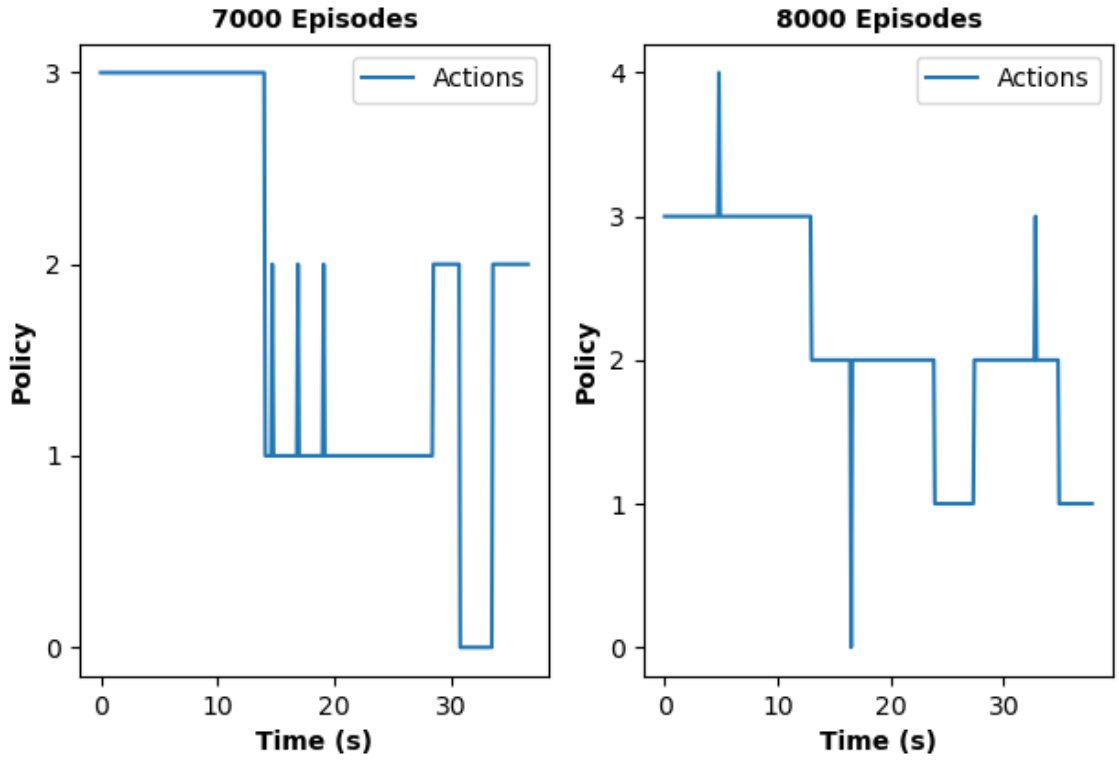


Figure 4.2-E Policies after 7000 & 8000 episodes of training (Perfect Conditions)

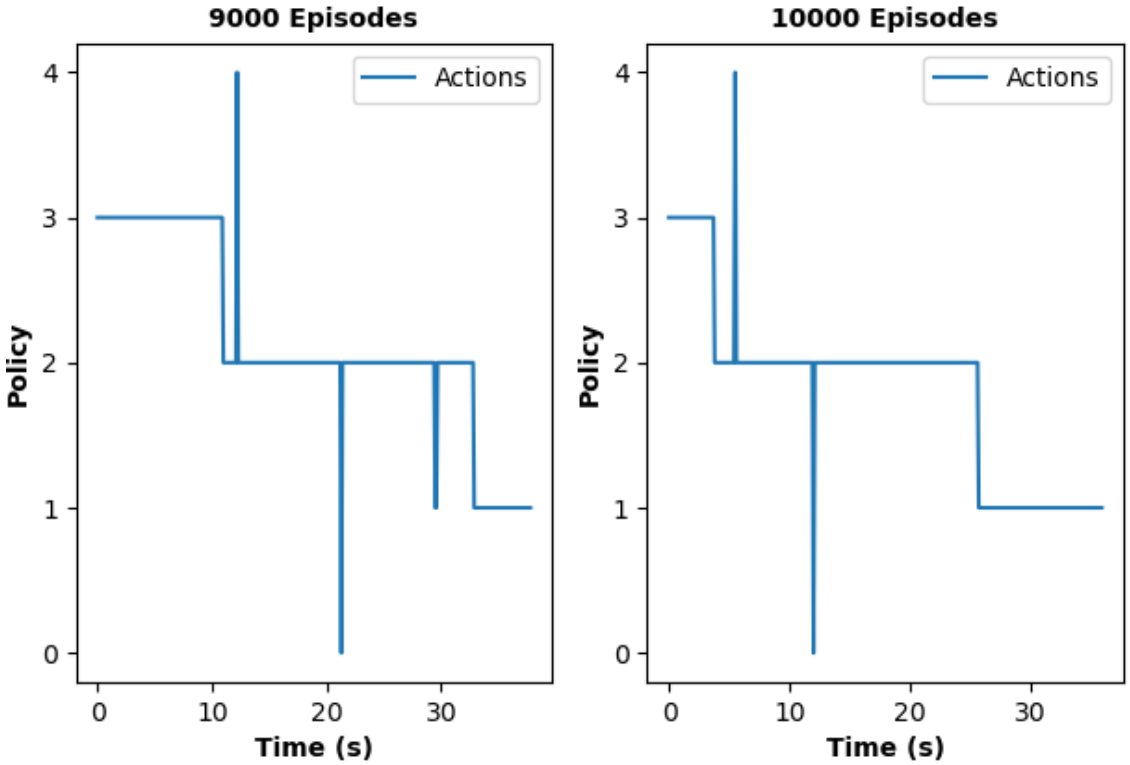


Figure 4.2-F Policies after 9000 & 10000 episodes of training (Perfect Conditions)

As observed in Figure 4.2-A, the result is well justified by the behaviour of the model during its learning process. It can be seen that during the first 40 Batches the agent is in an exploratory phase where no real trend emerges. The agent seems to try combinations of actions in order to increase its reward over the episodes. Then, a pattern of behaviour begins to emerge. For example, after 5000 episodes, the agent no longer opts for action 4 which is the highest Auto-brake mode. The agent will also find out for himself who is better to brake more widely at the beginning of an episode and then adjust the braking later on by using softer modes. This phase is observed from episode 6000 to 8000. After the model is in an optimisation phase, the agent's behaviour does not change fundamentally. Indeed, we still observe the degressive braking. Only the phases start to balance each other in order to have the best compromise in terms of braking and thus to optimise the reward.

It is now possible to test the model with test episodes. For this, the agent will have 1000 landings, the compiled results will give the performance of the model.

Table 4.2-A Success Rate of the Perfect Condition Agent

Model	**Landing Success Rate (Perfect Condition)**
Perfect Conditions Agent	96.2%

The model is relatively efficient with more than 96% successful landings. However, 38 episodes still failed, which is unacceptable in aeronautics. These failed landings are due to the speeds that were randomly drawn before the episode was initiated, which was too high. The agent was therefore unable to manage his events. To overcome this problem, it would be necessary to increase the safety coefficient of the episodes or to train the agent on a shorter optimal braking distance, so that in case of high speeds the model has the margin to manage these events.

4.2.2 Rainy Condition Model

To show that the agent is not only efficient in perfect conditions, it was decided to train a new agent with rain landing conditions.

It is therefore interesting to see if the model performs as well and if the behaviour is the same as with perfect landing conditions. As before, the agent's policies are plotted every 1000 episodes to observe the behaviour.

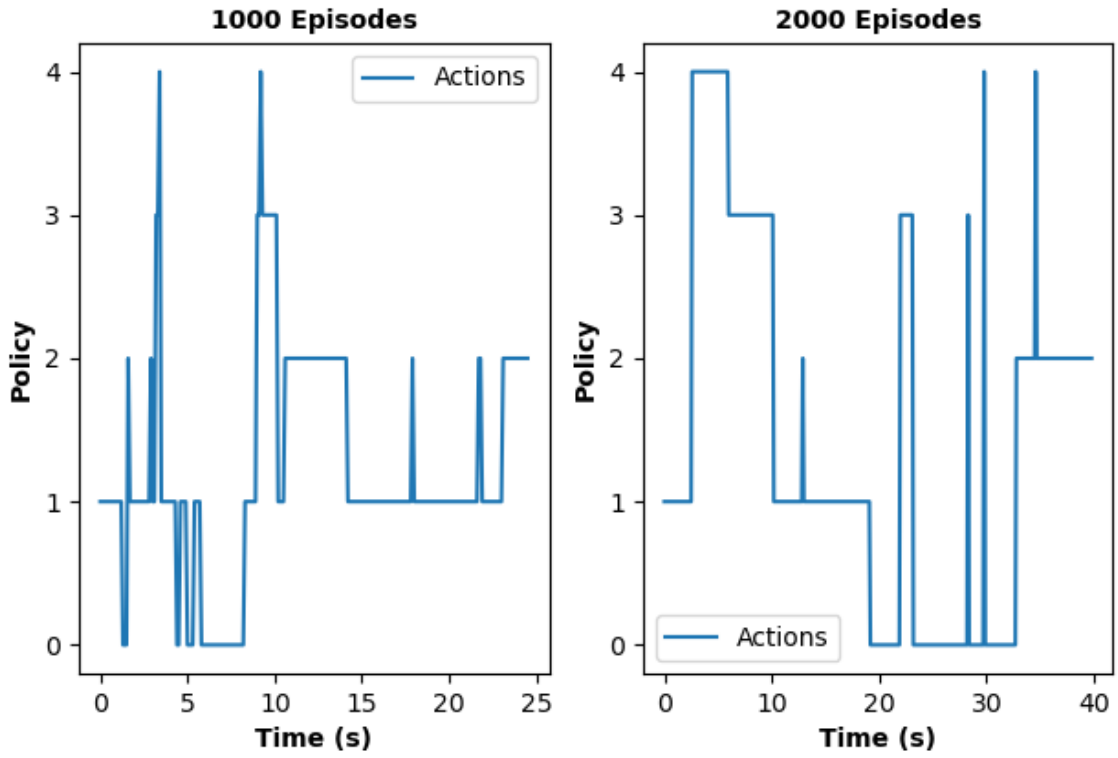


Figure 4.2-G Policies after 1000 & 2000 episodes of training (Rainy Conditions)

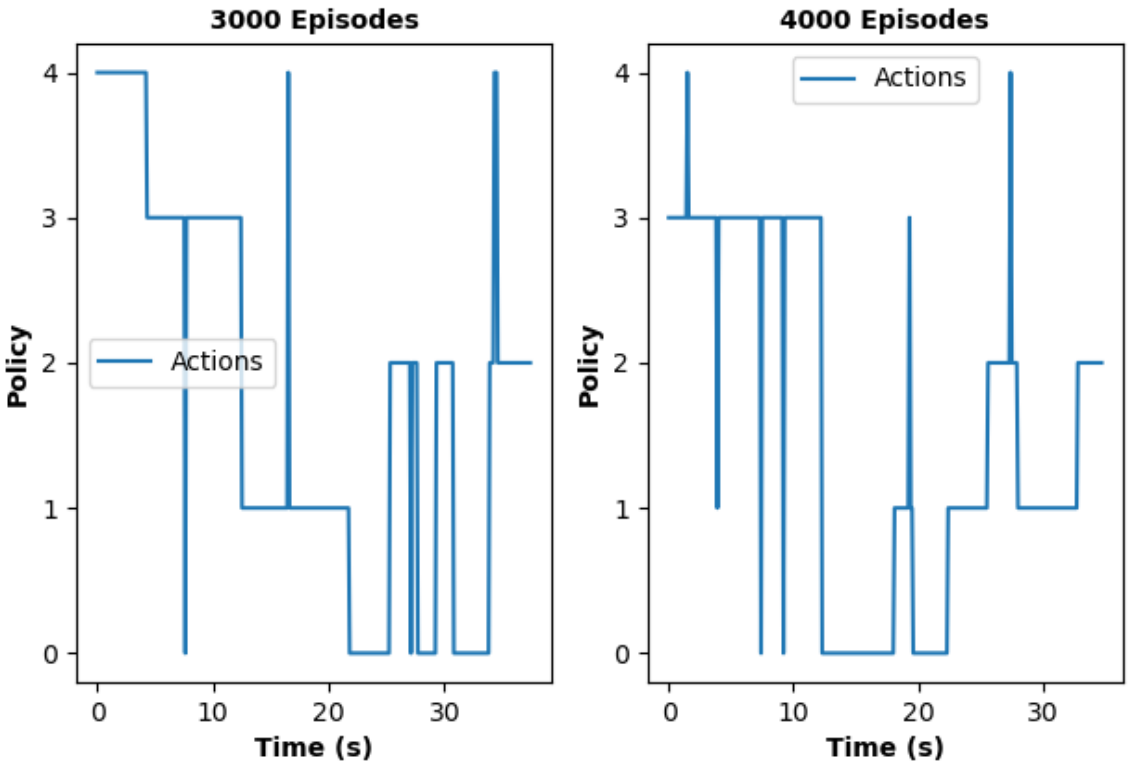


Figure 4.2-H Policies after 3000 & 4000 episodes of training (Rainy Conditions)

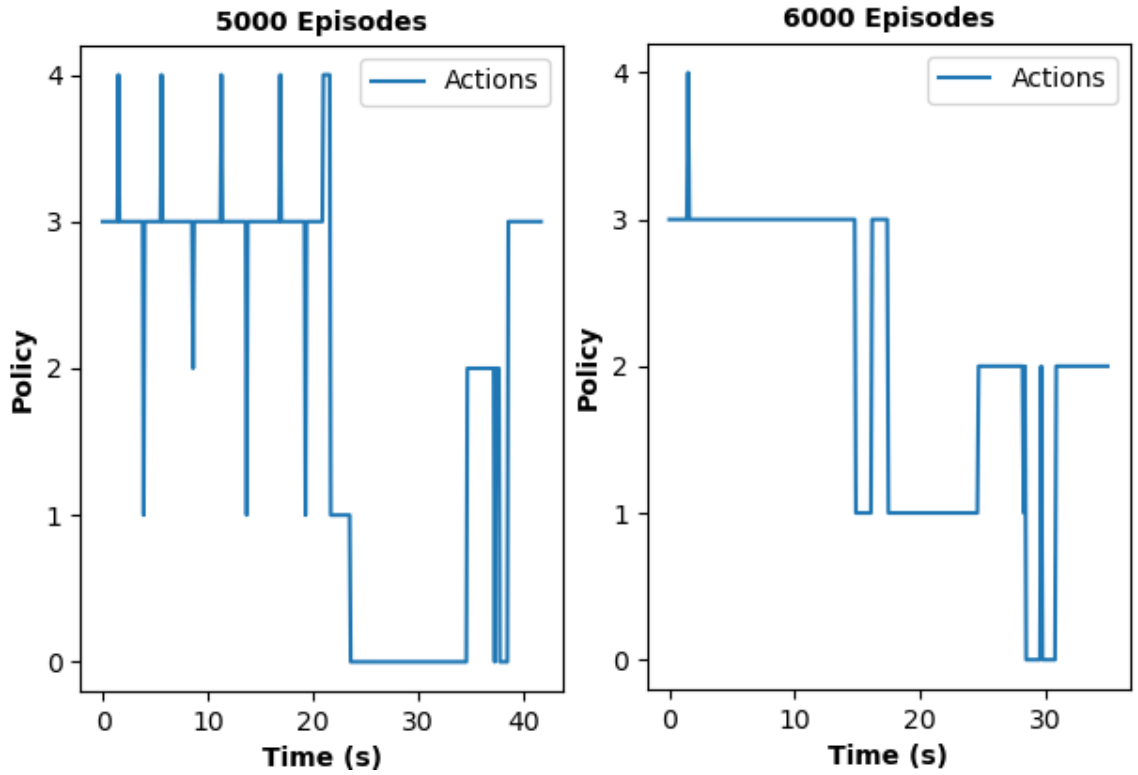


Figure 4.2-I Policies after 5000 & 6000 episodes of training (Rainy Conditions)

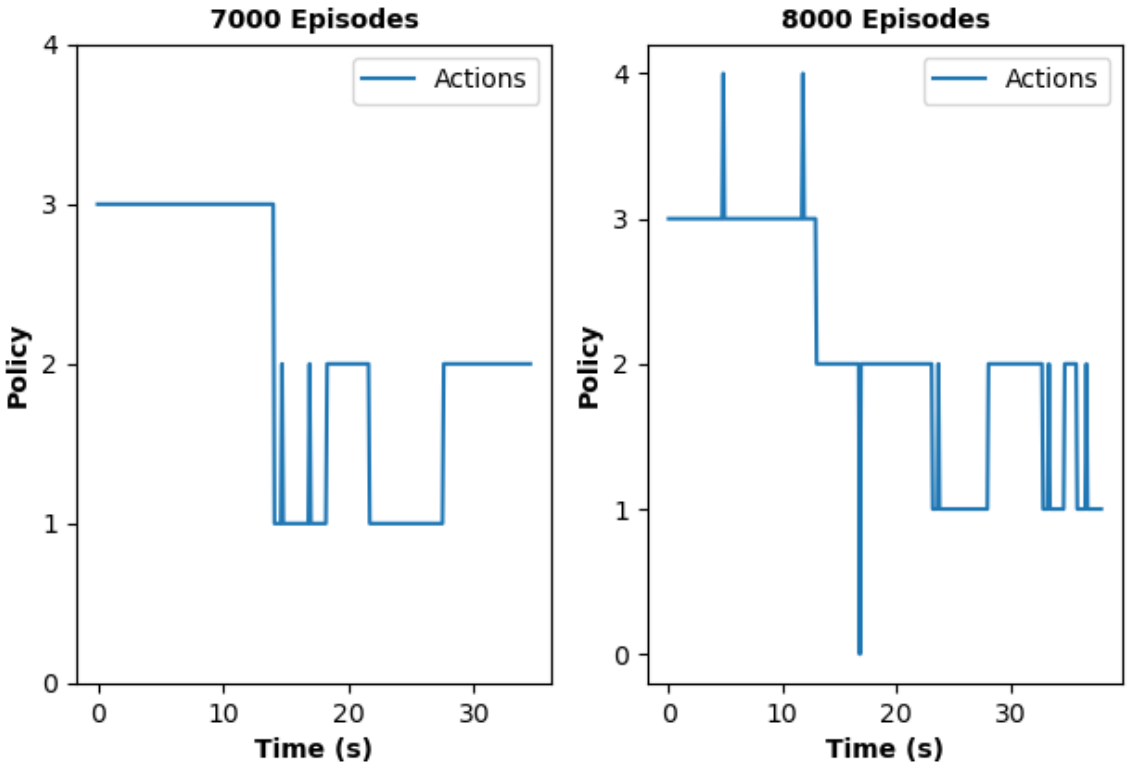


Figure 4.2-J Policies after 7000 & 8000 episodes of training (Rainy Conditions)

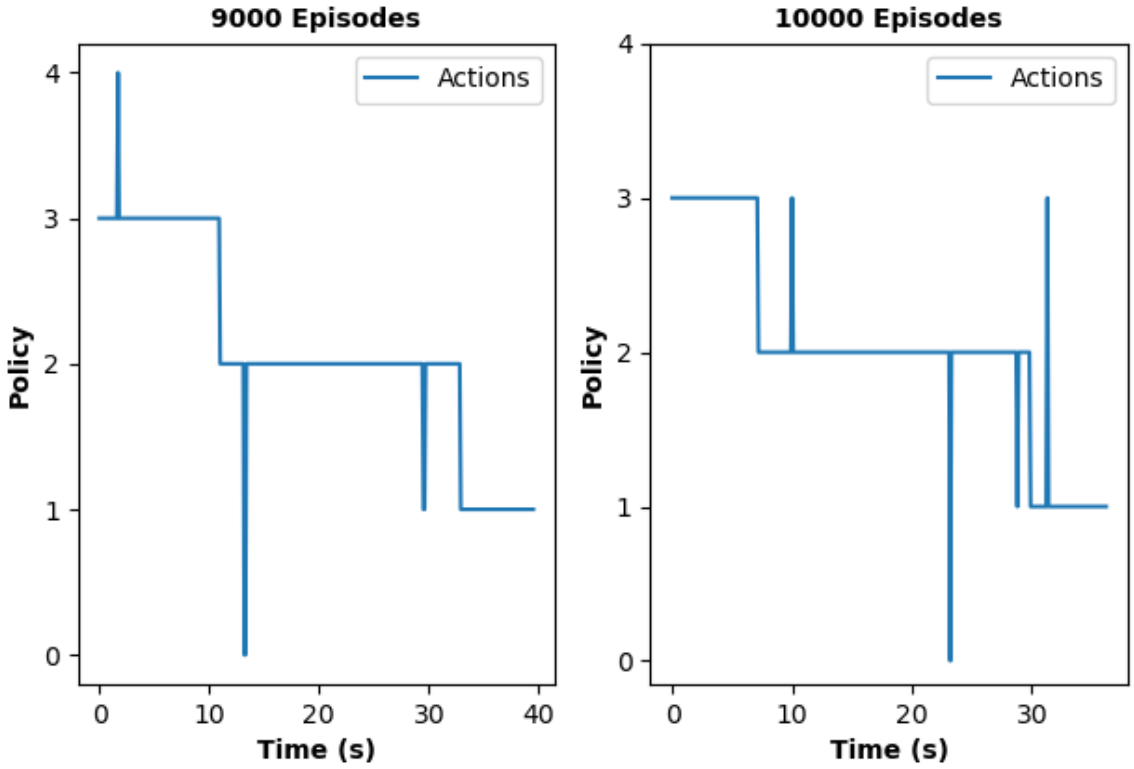


Figure 4.2-K Policies after 9000 & 10000 episodes of training (Rainy Conditions)

Like the previous agent, it is difficult to distinguish behavioural trends within the first 40 batches. The behaviour seems rather random. Thereafter, we can also begin to observe the arrival of the behaviour which is adopted by the agent from episode 6000. We observe the strong braking at the beginning of the episode and then a progressive decrease of it the more the episode advances. It seems that in these conditions the model converges slightly faster to the optimal policy and thus to the right behaviour. As in the previous model, even if rain is present on the runway, thus lowering the braking capacity of the aircraft, the agent no longer uses mode 4 of the Auto-brake. It is also noticeable that the duration of the episodes is slightly longer as it takes longer to stop the aircraft.

The model can now also be evaluated over 1000 landings:

Table 4.2-B Success Rate of the Rainy Condition Agent

Model	**Landing Success Rate (Rainy Condition)**
Rainy Conditions Agent	97.5%

Here too the model does not achieve 100% successful landings. However, it does get better results on the test set. The 25 failed landings are also due to the speed values being too high for the model to manage.

4.3 Discussion

The performances of the two models that have been created seem very interesting and are therefore a good option to solve this kind of problem. Indeed, with more adaptation of the models and with more training it would surely be possible to reach the 100% success rate. The RL is therefore a real contribution to the field.

However, these results have to be put into perspective with the task that has been accomplished. First of all, the environment in which the agent evolves is greatly simplified. In this use case, the braking is only done on 2 dimensions and does not manage the lateral movements that the plane can undergo during the flights.

The environment is also very simplified from a physical and mechanical point of view.

Secondly, to achieve such high performance, two different agents are needed. If, for example, the agent trained on perfect conditions is tested on 1000 landings in rainy conditions, very poor results are obtained:

Table 4.3-A Success Rate of the Perfect Conditions Agent on Rainy Conditions

Model	**Landing Success Rate (Rainy Condition)**
Perfect Conditions Agent	52.3%

It is possible to solve this problem by having an agent that can be trained in both perfect and rainy conditions. However, this requires much more learning time as the resulting behaviour is much more complex.

This is also why the model states have only 4 input values. The greater the number of input values, the greater the learning time before a model converges to a policy. The original idea was to make an agent with all the aircraft constants as input (weight, wing size, aerodynamic coefficient...) so that the agent could be generalised to several aircraft and also to include the environment constants (wind speed, weather, runway inclination...). Unfortunately, this learning time was totally underestimated. As a result, no model could converge to a meaningful policy and the results were very poor. In order to counter this problem, it would be necessary to increase the complexity of the NNs used and to carry out the training on dedicated and extremely powerful machines.

Finally, these results should be put into perspective with the braking mode adopted by the agent itself. It is assumed that the aircraft can instantaneously change its deceleration in a fraction of a second, which is totally improbable and absurd for an object weighing several tens of tons. Auto-brake was chosen mainly because it is a system that is already implemented and that most pilots use. Thus, the implementation of an automatic landing system on aircraft is possible with this

option. However, important constraints on both the mode of braking and its use will be necessary before any integration into the aircraft.

5 CONCLUSION

This project stems in part from Airbus' motivation to perfect and optimise an AI model capable of controlling the aircraft's braking system, particularly during the landing phase, based on the aircraft's physics and mechanics. Unfortunately, after development, it has been observed that the Deep Learning model was likely to be overfitted and that even traditional methods of solving this problem were ineffective. It was therefore necessary to approach the problem from another angle and propose an innovative solution using the latest advances in AI.

After a thorough understanding of the problem and the literature, the implementation of a Deep Reinforcement Learning model seemed to be the best option. Indeed, the methods of learning models of RL coupled with the power of a NN give very convincing results on complex problems of physics.

Thus, it was necessary to first create the environment in which the aircraft would evolve. For this, a landing simulator was developed to obtain the trajectories of the aircraft. The simulator includes a panel of parameters allowing the reproduction of different landing conditions.

The simulator was then adapted to a Python RL library, thus allowing the system environment to be completed by incorporating a state transfer function, the set of actions that can be taken by the plane and a reward function allowing the agent to be rewarded or punished according to its behaviour.

Additionally, a thorough optimisation of the agent's NN hyperparameters as well as of the PPO algorithm controlling the policy adopted by the agent allowed to obtain interesting results. In particular, the emergence of a decreasing braking distance over time allows a better accuracy of the model regarding the optimal braking distance. Thus, in perfect condition, the model is able to land the aircraft successfully in more than 96% of cases. Another model trained in rainy conditions achieved over 97% successful landing rates. The use of Deep RL models thus appears to be an excellent alternative to a simple NN within physics-based systems.

However, the results must be seen in the context of the complexity of the actual system. Indeed, the environment and the models created remain very simplified and require further work and research in order to develop an agent capable of operating on a real runway.

6 FUTURE WORK

In order to study this project further, it is necessary to perform more in-depth tasks. In this section, proposals will be developed that could be interesting to implement.

The complexity of the landing simulator should be increased. Two-dimensional simulations cannot lead to results that can be used in reality. Thus, the addition of object mechanics rather than point mechanics is essential. It would also be possible to introduce and train a model on an environment that takes into account the 3 dimensions of space.

Then, if the complexity of the simulator increases, the complexity of the agent should also follow. An agent evolving in a 3-dimensional space will require much more computing power and many more episode batches to be able to observe complex and optimised behaviour. This will also allow us to participate in the design of a more general model that will be able to take in many more values to cover all the subtleties of its environment, something that could not be done on this project due to lack of time.

Currently the trained agent can only act according to 4 Auto-brake modes which are very restrictive and leave few degrees of freedom in the actions of the model. Thus, it could be interesting to increase the size of the set of possible actions. This could also lead to even more complex and successful behaviours. One of the biggest problems with the agent in this project is that it is based on the assumption that the aircraft can vary its deceleration value instantaneously. This assumption is completely unreal. One proposal would be to move from a discrete set of actions to a continuous set. Indeed, this would allow to be much more faithful to the behaviour of an aircraft during a landing. The movement of the aircraft is completely subject to physical constraints.

Finally, to increase the efficiency of the learning of the model it would be interesting to implement a last method.

It would consist in breaking the independence of the episodes between them. Currently, there is no link from one landing to another, and it happens that the

model adopts a risky behaviour which would have led it to miss an episode in order to optimise its reward. The idea here would be to retain the results of past landings so that they have an impact on the reward of future episodes. For example, it would be possible to reward the model more if it manages to land the plane 5 times in a row. This will then lead the model not to optimise the result of an episode but to favour behaviours that are more focused on the safety of the aircraft.

REFERENCES

[1] T. Matsubara, A. Ishikawa, T. Yaguchi, “Deep Energy-Based Modeling of Discrete-Time Physics,” Oct 2020.

[2] E. Franz, B. Solenthaler, N. Thuerey, “Global Transport for Fluid Recon struction with Learned Self-Supervision,” Apr 2021.

[3] L. Shiloh-Perl, R. Giryes, “Introduction to deep learning,” Feb 2020.

[4] K. Chandra, E. Meijer, S. Andow, “Gradient Descent: The Ultimate Opti mizer,” Sep 2019.

[5] B. Ghojogh, M. Crowley, “The Theory Behind Overfitting, Cross Valida tion, Regularization, Bagging, and Boosting” May 2019.

[6] https://i.stack.imgur.com/YfgTX.jpg. Accessed on 28 July 2021

[7] Wang, Guoqing, “The Principles of Integrated Technology in Avion ics Systems || Background introduction,” 1–40, 2020

[8] Ribbens, W.B, “Encyclopedia of Physical Science and Technology || Air craft Instruments”, (), 337–364., 2003

[9] ICAO Annex 8 Part IIIA Paragraph 2.2.3.3. and Part IIIB Sub-part B

[10] https://en.wikipedia.org/wiki/Spoiler_(aeronautics) Accessed on 10 July 2021

[11] https://www.skybrary.aero/index.php/Brakes Accessed on 10 July 2021

[12] https://en.wikipedia.org/wiki/Thrust_reversal Accessed on 10 July 2021

[13] G. Federico, J. Michael, P. Tomaso, “Regularization Theory and Neural Networks Architectures. Neural Computation”, 7(2), 219–269, 1995.

[14] C. M. Bishop, “Neural Networks for Pattern Recognition”, Jan 1996.

[15] G. Hacohen, D. Weinshall, “Principal Components Bias in Deep Neural Networks”, Jun 2021.

[16] T van Laarhoven, “L2 Regularization versus Batch and Weight Normali zation”, Jun 2017.

[17] M. Li, M. Soltanolkotabi, S. Oymak, “Gradient Descent with Early Stop ping is Provably Robust to Label Noise for Overparameterized Neural Networks”, Jul 2019.

[18] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”, 15(56):1929−1958, 2014.

[19] S. Cai, Y. Shu, G. Chen, Beng C. Ooi, W. Wang, M. Zhang, “Efficient and Effective Dropout for Deep Convolutional Neural Networks”, Jul 2020.

[20] D. Silver, A. Huang, C. J. Maddison, “Mastering the game of Go with deep neural networks and tree search”, Jan 2016.

[21] V. Mnih, K. Kavukcuoglu, D. Silver, “Playing Atari with Deep Reinforce ment Learning”, Dec 2013.

[22] R. S. Sutton, A.G. Barto, “Reinforcement Learning: An Introduction”, 2014-2015.

[23] A. Suharevs, P. Trifonovs-Bogdanovs, V. Šestakovs, “Dynamic Model of Aircraft Landing”, Dec 2016.

[24] J. Sun, J. M. Hoekstra, J. Ellerbroek, “OpenAP: An Open-Source Aircraft Performance Model for Air Transportation Studies and Simulations”, Jul 2020.

[25] J. Sun, J. M. Hoekstra, J. Ellerbroek, “Aircraft Drag Polar Estimation Based on a Stochastic Hierarchical Model”, Dec 2018.

[26] https://tensorforce.readthedocs.io/en/0.5.4/index.html

[27] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, “Proximal Policy Optimization Algorithms”, Aug 2017.

[28] K. Lange, “The MM Algorithm”, UCLA, Apr 2007.

[29] https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence Accessed on 10 July 2021.

[30] J. Schulman, “Advanced Policy Gradient Methods: Natural Gradient, TRPO, and More”, Aug 2017.

[31] J. Schulman, S. Levine, P. Moritz, M. Jordan, P. Abbee, “Trust Region Policy Optimization”, Apr 2017.

[32] H. Chae, C. Kang, B. Kim, J. Kim, C. Chung, J. Won Choi “Autonomous Braking System via Deep Reinforcement Learning”, Apr 2017.

APPENDICES

Appendix A Code Implementation

A.1 Environment Implementation

```python
import numpy as np
from tensorforce.environments import Environment
from LandingModel import LandingModel #import the Landing Simulator

#Class representing the environnement
class LandingEnvironment(Environment):
    def __init__(self):
        super().__init__()
        self.LandingModel = LandingModel()  #create an instance of the Landing Simulator
        self.NUM_ACTIONS = len(self.LandingModel.action_vec)    #size of the vector of actions
        self.STATES_SIZE = len(self.LandingModel.states_vec)    #seize of the vector of states
        self.max_step_per_episode = 1000    #represent the number of maximum time step of an episode
        self.finished = False   #variable saying if the plane manage to successfully land
        self.episode_end = False    #variable saying if the plane went above the runway

    #States function
    def states(self):
        return dict(type="float", shape=(self.STATES_SIZE,))

    #Actions function
    def actions(self):
        return dict(type="int",num_values=self.NUM_ACTIONS)

    #return the max timesteps of an episode
    def max_episode_timesteps(self):
        return self.max_step_per_episode

    #close the environnement at then end of the learning process
    def close(self):
        return super().close()

    #reset function, gives the original state of an episode
    def reset(self):
        state = np.array([-0.73263,np.random.normal(69.37,4.32),267.2,self.LandingModel.safe_breaking_dist])    #original state
        self.LandingModel = LandingModel()  #new instance of the landing model
        return state    #return the original state

    #Compute at each timestep the actions of the model, update the state, and calculate the reward
    def execute(self, actions,reward):
        next_state = self.LandingModel.compute_timestep(action=actions) #compute the new states based on the action taken
        terminal = self.terminal()  #check if an episode is done
        reward = reward + self.reward(actions)  #compute the reward
        return next_state, terminal, reward
```

```python
def reward(self,actions):
    const_rew = 500     #constant reward if the plane manage to land (gamma coefficient)
    coef_rew_corr = 0.01    #Beta coefficient

    const_pun = -1000   #constant punishement (theta coefficient)
    coef_pun_corr = 1.0 #delta coefficient

    #determination of the alpha coefficient
    if actions == 0:
        coef = 10
    elif actions == 1:
        coef = 8
    elif actions == 2:
        coef = 5
    elif actions == 3:
        coef = 2
    elif actions == 4:
        coef = 1

    #compute the reward
    reward = -(self.LandingModel.V_vec[-1] - self.LandingModel.V_vec[-2])*coef  #first line

    if self.finished:
        reward += const_rew - ((self.LandingModel.safe_breaking_dist - self.LandingModel.Pos_vec[-1])**2)*coef_rew_corr #2nd line

    if self.episode_end:
        reward += const_pun - self.LandingModel.V_vec[-1]*coef_pun_corr #3rd line

    return reward
```

A.2 Landing Simulator Implementation

```python
import numpy as np
import math

class LandingModel:
    def __init__(self):
        ###  A320 Values ###
        self.g = 9.81 #gravity vector in m/s²
        self.m = 60000 #mass in kg
        self.S = 122.6 #wings surface in m²
        self.RHO = 1.225 #air density in kg/m^3
        self.CD0_LAND = 0.120 #zero-lift drag coefficient
        self.k = 0.0334 #lift-induced drag coefficient factor
        self.Cst = 1 #friction coefficient
        self.lift_coef = 0.1 #lift coefficient
        self.slope = 0 #runaway slope in °
        self.tyre_condtition = 1 #coefficient represneting tyres conditions
        self.safe_breaking_dist = 1500 #safe braking distance parameter
        self.security_coefficient = 1.15 #coefficient to determine the lenght of the maximal acceptable braking distance
        self.max_braking_distance = self.security_coefficient*self.safe_breaking_dist
        self.wind_speed = 0

        self.timestep = 0.1 #value of the time between each states (0.1s)

        self.action_vec = np.array([0,1,2,3,4]) #Action Ensemble that the agent can take

        self.Acc_vec = [-0.73263]   #initilize the value of the deceleration at t=0
        self.V_vec = [np.random.normal(69.37,4.32)]    #initillize the speed of the plane at t=0
        self.Pos_vec = [267.2]  #initilize the x position of the plane at t=0 (landing start when the plane reach 50ft altitude, the compute only concerne the ground phase)

        self.states_vec = np.array([self.Acc_vec[-1],self.V_vec[-1],self.Pos_vec[-1],self.safe_breaking_dist])  #State of the environnement


    def calc_lift(self,v): #Calculation of the lift of the plane
        return self.lift_coef*self.S*(self.RHO*(v+self.wind_speed)**2/2)

    def calc_drag(self,v):  #Calculation of the drage of the plane
        return self.calc_drag_coefficient()*self.S*(self.RHO*(v*self.wind_speed)**2/2)

    def calc_drag_coefficient(self):    #calculation of the drag coefficient
        return self.CD0_LAND + self.k*(self.lift_coef**2)

    def rolling_resistance_coeffcient(self,v):  #calculation of the rolling resistance coefficient
        return (0.0041+0.000041*v)*self.Cst

    def calc_rolling_resistance_force(self,v,m):    #Calculation of the rolling resistance force
        return m*self.rolling_resistance_coeffcient(v)
```

```python
    def calc_rolling_resistance_force(self,v,m):    #Calculation of the rolling resistance force
        return m*self.rolling_resistance_coeffcient(v)

    def calc_slope_runaway_force(self,m):   #Calculation of the action of the runway on the plane
        return m*self.g*math.sin(self.slope)


    def calc_acc(self,v):   #Compute all the forces to calculate the acceleration of the plane
        acc = -self.calc_drag(v) - self.calc_rolling_resistance_force(v,self.m-self.calc_lift(v)) - self.calc_slope_runaway_force(self.m - self.calc_lift(v))
        acc = acc/self.m
        return acc

    def compute_timestep(self,action):  #Compute the action of the model and update the state of the plane for the next timestep
        acc_sum = 0
        #translate the action into the value of the deceleration
        if action == 0:
            acc_sum = 0
        elif action == 1:
            acc_sum = -0.914411
        elif action == 2:
            acc_sum = -1.524018
        elif action == 3:
            acc_sum = -2.1336259
        else:
            acc_sum = -3.3528407

        acc_sum = acc_sum*self.tyre_condtition*self.Cst #apply the coefficients of friction and tyre condition the the deceleration

        x = self.Pos_vec[-1] + self.V_vec[-1]*self.timestep #compute the new x position of the plane
        a = self.calc_acc(self.V_vec[-1]) + acc_sum #compute the new acceleration
        v = self.V_vec[-1] + a*self.timestep    #compute the new velocity of the plane

        #add the value to the respective lists
        self.Pos_vec.append(x)
        self.Acc_vec.append(a)
        self.V_vec.append(v)

        return np.array([self.Acc_vec[-1],self.V_vec[-1],self.Pos_vec[-1],self.safe_breaking_dist]) #return the new state
```

Appendix B Advantage Function

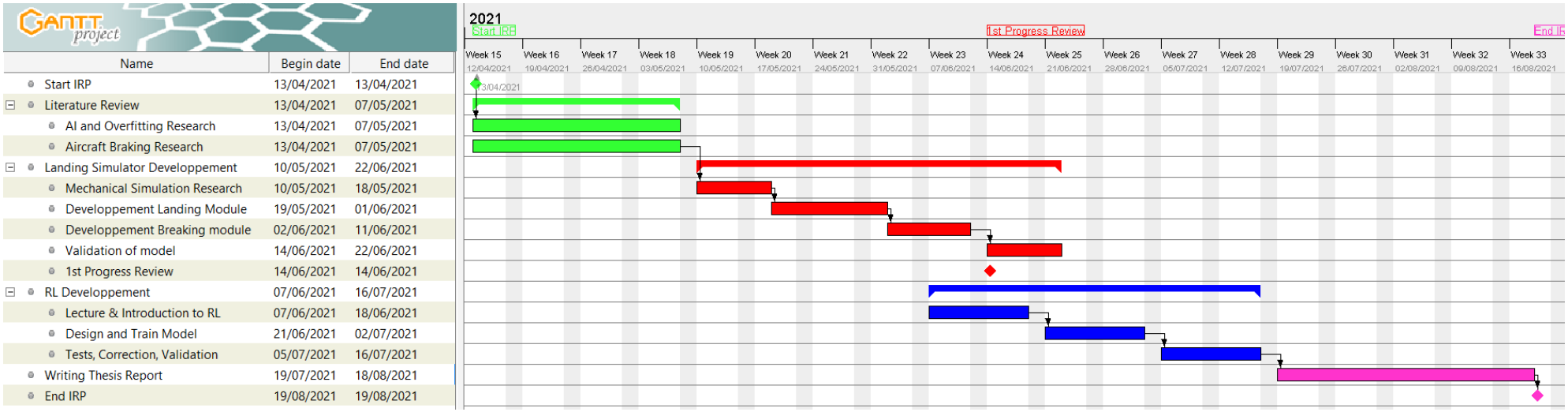
DRL models are mapping input features to future rewards that are called Q values. This map of input features and the variety of q values at a given states allow allows the reinforcement learning agent to obtain a global picture of the environment, which helps it to choose the optimal behaviour.

But this mapping can be hard to evaluate since it has a high variance. To tackle this issue, it is possible to use the Advantage Function.

The Advantage Function is the difference between the Q value at a given state minus the value function of the state:

$$A(s,a) = Q(s,a) - V(s)$$

Appendix C Gantt Chart



Name	Begin date	End date
Start IRP	13/04/2021	13/04/2021
Literature Review	13/04/2021	07/05/2021
AI and Overfitting Research	13/04/2021	07/05/2021
Aircraft Braking Research	13/04/2021	07/05/2021
Landing Simulator Developpement	10/05/2021	22/06/2021
Mechanical Simulation Research	10/05/2021	18/05/2021
Developpement Landing Module	19/05/2021	01/06/2021
Developpement Breaking module	02/06/2021	11/06/2021
Validation of model	14/06/2021	22/06/2021
1st Progress Review	14/06/2021	14/06/2021
RL Developpement	07/06/2021	16/07/2021
Lecture & Introduction to RL	07/06/2021	18/06/2021
Design and Train Model	21/06/2021	02/07/2021
Tests, Correction, Validation	05/07/2021	16/07/2021
Writing Thesis Report	19/07/2021	18/08/2021
End IRP	19/08/2021	19/08/2021

Appendix D Risk Analysis

Type	Event/Description	Probability (1-5)	Impact (1-5)	Risk (1-25)
Organizational	Delay in the original schedule	2	4	10
Technical	The simulator is not complete enough	3	4	12
Technical	Reinforcement learning is not effective for this use case.	1	5	5
Technical	The Deep learning model is overfitted	2	3	6
Regulatory	Airbus is not allowed to share any data or model	4	4	16
Technical	The model performs badly	2	5	10
Technical	The model requires a lot of computing power and time to train	4	3	12